Scientific Python Lectures

lectures.scientific-python.org

Edited by

K. Jarrod Millman
Stéfan van der Walt
Gaël Varoquaux
Emmanuelle Gouillart
Olaf Vahtras
Pierre de Buyl

creative commons

Gaël Varoquaux • Emmanuelle Gouillart • Olav Vahtras
Pierre de Buyl • K. Jarrod Millman • Stéfan van der Walt
Christopher Burns • Adrian Chauve • Robert Cimrman • Christophe Combelles
Ralf Gommers • André Espaze • Zbigniew Jędrzejewski-Szmek
Valentin Haenel • Michael Hartmann • Gert-Ludwig Ingold • Fabian Pedregosa
Didrik Pinte • Nicolas P. Rougier • Joris Van den Bossche • Pauli Virtanen

and many others...

Contents

I Getting started with Python for science **2**

1 Python scientific computing ecosystem **4**
1.1 Why Python? . . . 4
1.2 The scientific Python ecosystem . . . 6
1.3 Before starting: Installing a working environment . . . 7
1.4 The workflow: interactive environments and text editors . . . 7

2 The Python language **12**
2.1 First steps . . . 13
2.2 Basic types . . . 14
2.3 Control Flow . . . 22
2.4 Defining functions . . . 26
2.5 Reusing code: scripts and modules . . . 32
2.6 Input and Output . . . 41
2.7 Standard Library . . . 42
2.8 Exception handling in Python . . . 49
2.9 Object-oriented programming (OOP) . . . 52

3 NumPy: creating and manipulating numerical data **54**
3.1 The NumPy array object . . . 54
3.2 Numerical operations on arrays . . . 68
3.3 More elaborate arrays . . . 83
3.4 Advanced operations . . . 87
3.5 Some exercises . . . 91
3.6 Full code examples . . . 95

4 Matplotlib: plotting **108**
4.1 Introduction . . . 108
4.2 Simple plot . . . 109
4.3 Figures, Subplots, Axes and Ticks . . . 118
4.4 Other Types of Plots: examples and exercises . . . 121
4.5 Beyond this tutorial . . . 130
4.6 Quick references . . . 131
4.7 Full code examples . . . 133

5 SciPy : high-level scientific computing **211**
5.1 File input/output: `scipy.io` . . . 213
5.2 Special functions: `scipy.special` . . . 213

5.3 Linear algebra operations: `scipy.linalg` . . . 215
5.4 Interpolation: `scipy.interpolate` . . . 216
5.5 Optimization and fit: `scipy.optimize` . . . 218
5.6 Statistics and random numbers: `scipy.stats` . . . 223
5.7 Numerical integration: `scipy.integrate` . . . 224
5.8 Fast Fourier transforms: `scipy.fft` . . . 227
5.9 Signal processing: `scipy.signal` . . . 229
5.10 Image manipulation: `scipy.ndimage` . . . 231
5.11 Summary exercises on scientific computing . . . 237
5.12 Full code examples for the SciPy chapter . . . 250

6 Getting help and finding documentation **291**

II Advanced topics 294

7 Advanced Python Constructs **296**
7.1 Iterators, generator expressions and generators . . . 297
7.2 Decorators . . . 301
7.3 Context managers . . . 310

8 Advanced NumPy **313**
8.1 Life of ndarray . . . 314
8.2 Universal functions . . . 328
8.3 Interoperability features . . . 337
8.4 Array siblings: `chararray`, `maskedarray` . . . 340
8.5 Summary . . . 343
8.6 Contributing to NumPy/SciPy . . . 343

9 Debugging code **346**
9.1 Avoiding bugs . . . 347
9.2 Debugging workflow . . . 349
9.3 Using the Python debugger . . . 349
9.4 Debugging segmentation faults using gdb . . . 355

10 Optimizing code **358**
10.1 Optimization workflow . . . 359
10.2 Profiling Python code . . . 359
10.3 Making code go faster . . . 362
10.4 Writing faster numerical code . . . 363

11 Sparse Arrays in SciPy **366**
11.1 Introduction . . . 366
11.2 Storage Schemes . . . 368
11.3 Linear System Solvers . . . 381
11.4 Other Interesting Packages . . . 386

12 Image manipulation and processing using NumPy and SciPy **387**
12.1 Opening and writing to image files . . . 388
12.2 Displaying images . . . 390
12.3 Basic manipulations . . . 391
12.4 Image filtering . . . 393
12.5 Feature extraction . . . 398
12.6 Measuring objects properties: `scipy.ndimage.measurements` . . . 401
12.7 Full code examples . . . 405
12.8 Examples for the image processing chapter . . . 405

13 Mathematical optimization: finding minima of functions **432**
13.1 Knowing your problem . . . 433
13.2 A review of the different optimizers . . . 435

13.3 Full code examples . . . 442
13.4 Examples for the mathematical optimization chapter . . . 442
13.5 Practical guide to optimization with SciPy . . . 480
13.6 Special case: non-linear least-squares . . . 482
13.7 Optimization with constraints . . . 483
13.8 Full code examples . . . 485
13.9 Examples for the mathematical optimization chapter . . . 485

14 Interfacing with C **486**
14.1 Introduction . . . 487
14.2 Python-C-Api . . . 487
14.3 Ctypes . . . 495
14.4 SWIG . . . 499
14.5 Cython . . . 504
14.6 Summary . . . 509
14.7 Further Reading and References . . . 509
14.8 Exercises . . . 509

III Packages and applications 511

15 Statistics in Python **513**
15.1 Data representation and interaction . . . 514
15.2 Hypothesis testing: comparing two groups . . . 519
15.3 Linear models, multiple factors, and analysis of variance . . . 522
15.4 More visualization: seaborn for statistical exploration . . . 528
15.5 Testing for interactions . . . 532
15.6 Full code for the figures . . . 533
15.7 Solutions to this chapter's exercises . . . 561

16 Sympy : Symbolic Mathematics in Python **564**
16.1 First Steps with SymPy . . . 565
16.2 Algebraic manipulations . . . 566
16.3 Calculus . . . 567
16.4 Equation solving . . . 569
16.5 Linear Algebra . . . 570

17 `scikit-image`: image processing **572**
17.1 Introduction and concepts . . . 573
17.2 Importing . . . 575
17.3 Example data . . . 576
17.4 Input/output, data types and colorspaces . . . 576
17.5 Image preprocessing / enhancement . . . 578
17.6 Image segmentation . . . 581
17.7 Measuring regions' properties . . . 584
17.8 Data visualization and interaction . . . 585
17.9 Feature extraction for computer vision . . . 585
17.10 Full code examples . . . 586
17.11 Examples for the scikit-image chapter . . . 586

18 scikit-learn: machine learning in Python **598**
18.1 Introduction: problem settings . . . 599
18.2 Basic principles of machine learning with scikit-learn . . . 603
18.3 Supervised Learning: Classification of Handwritten Digits . . . 608
18.4 Supervised Learning: Regression of Housing Data . . . 613
18.5 Measuring prediction performance . . . 618
18.6 Unsupervised Learning: Dimensionality Reduction and Visualization . . . 623
18.7 Parameter selection, Validation, and Testing . . . 626
18.8 Examples for the scikit-learn chapter . . . 634

IV About the Scientific Python Lectures **681**

19 About the Scientific Python Lectures **682**
19.1 Authors 682

Index **685**

Part I

Getting started with Python for science

This part of the *Scientific Python Lectures* is a self-contained introduction to everything that is needed to use Python for science, from the language itself, to numerical computing or plotting.

CHAPTER 1

Python scientific computing ecosystem

Authors: *Fernando Perez, Emmanuelle Gouillart, Gaël Varoquaux, Valentin Haenel*

1.1 Why Python?

1.1.1 The scientist's needs

- Get data (simulation, experiment control),
- Manipulate and process data,
- Visualize results, quickly to understand, but also with high quality figures, for reports or publications.

1.1.2 Python's strengths

- **Batteries included** Rich collection of already existing **bricks** of classic numerical methods, plotting or data processing tools. We don't want to re-program the plotting of a curve, a Fourier transform or a fitting algorithm. Don't reinvent the wheel!
- **Easy to learn** Most scientists are not paid as programmers, neither have they been trained so. They need to be able to draw a curve, smooth a signal, do a Fourier transform in a few minutes.
- **Easy communication** To keep code alive within a lab or a company it should be as readable as a book by collaborators, students, or maybe customers. Python syntax is simple, avoiding strange symbols or lengthy routine specifications that would divert the reader from mathematical or scientific understanding of the code.
- **Efficient code** Python numerical modules are computationally efficient. But needless to say that a very fast code becomes useless if too much time is spent writing it. Python aims for quick development times and quick execution times.

- **Universal** Python is a language used for many different problems. Learning Python avoids learning a new software for each new problem.

1.1.3 How does Python compare to other solutions?

Compiled languages: C, C++, Fortran...

Pros

- Very fast. For heavy computations, it's difficult to outperform these languages.

Cons

- Painful usage: no interactivity during development, mandatory compilation steps, verbose syntax, manual memory management. These are **difficult languages** for non programmers.

Matlab scripting language

Pros

- Very rich collection of libraries with numerous algorithms, for many different domains. Fast execution because these libraries are often written in a compiled language.
- Pleasant development environment: comprehensive and help, integrated editor, etc.
- Commercial support is available.

Cons

- Base language is quite poor and can become restrictive for advanced users.
- Not free and not everything is open sourced.

Julia

Pros

- Fast code, yet interactive and simple.
- Easily connects to Python or C.

Cons

- Ecosystem limited to numerical computing.
- Still young.

Other scripting languages: Scilab, Octave, R, IDL, etc.

Pros

- Open-source, free, or at least cheaper than Matlab.
- Some features can be very advanced (statistics in R, etc.)

Cons

- Fewer available algorithms than in Matlab, and the language is not more advanced.
- Some software are dedicated to one domain. Ex: Gnuplot to draw curves. These programs are very powerful, but they are restricted to a single type of usage, such as plotting.

Python

Pros

- Very rich scientific computing libraries
- Well thought out language, allowing to write very readable and well structured code: we "code what we think".
- Many libraries beyond scientific computing (web server, serial port access, etc.)
- Free and open-source software, widely spread, with a vibrant community.
- A variety of powerful environments to work in, such as IPython, Spyder, Jupyter notebooks, Pycharm, Visual Studio Code

Cons

- Not all the algorithms that can be found in more specialized software or toolboxes.

1.2 The scientific Python ecosystem

Unlike Matlab, or R, Python does not come with a pre-bundled set of modules for scientific computing. Below are the basic building blocks that can be combined to obtain a scientific computing environment:

Python, a generic and modern computing language

- The language: flow control, data types (`string`, `int`), data collections (lists, dictionaries), etc.
- Modules of the standard library: string processing, file management, simple network protocols.
- A large number of specialized modules or applications written in Python: web framework, etc. ... and scientific computing.
- Development tools (automatic testing, documentation generation)

> **See also**
>
> *chapter on Python language*

Core numeric libraries

- **NumPy**: numerical computing with powerful **numerical arrays** objects, and routines to manipulate them. https://numpy.org/

  > **See also**
  >
  > *chapter on numpy*

- **SciPy** : high-level numerical routines. Optimization, regression, interpolation, etc https://scipy.org/

  > **See also**
  >
  > *chapter on SciPy*

- **Matplotlib** : 2-D visualization, "publication-ready" plots https://matplotlib.org/

> **See also**
>
> *chapter on matplotlib*

Advanced interactive environments:

- **IPython**, an advanced **Python console** https://ipython.org/
- **Jupyter**, **notebooks** in the browser https://jupyter.org/

Domain-specific packages,

- **pandas, statsmodels, seaborn** for *statistics*
- **sympy** for *symbolic computing*
- **scikit-image** for *image processing*
- **scikit-learn** for *machine learning*

and many more packages not documented in the Scientific Python Lectures.

> **See also**
>
> *chapters on advanced topics*
>
> *chapters on packages and applications*

1.3 Before starting: Installing a working environment

Python comes in many flavors, and there are many ways to install it. However, we recommend to install a scientific-computing distribution, that comes readily with optimized versions of scientific modules.

Under Linux

If you have a recent distribution, most of the tools are probably packaged, and it is recommended to use your package manager.

Other systems

There are several fully-featured scientific Python distributions:

- Anaconda
- WinPython

1.4 The workflow: interactive environments and text editors

Interactive work to test and understand algorithms: In this section, we describe a workflow combining interactive work and consolidation.

Python is a general-purpose language. As such, there is not one blessed environment to work in, and not only one way of using it. Although this makes it harder for beginners to find their way, it makes it possible for Python to be used for programs, in web servers, or embedded devices.

1.4.1 Interactive work

We recommend an interactive work with the IPython console, or its offspring, the Jupyter notebook. They are handy to explore and understand algorithms.

Under the notebook

To execute code, press "shift enter"

Start *ipython*:

```
In [1]: print('Hello world')
Hello world
```

Getting help by using the **?** operator after an object:

```
In [2]: print?
Signature: print(*args, sep=' ', end='\n', file=None, flush=False)
Docstring:
Prints the values to a stream, or to sys.stdout by default.

sep
  string inserted between values, default a space.
end
  string appended after the last value, default a newline.
file
  a file-like object (stream); defaults to the current sys.stdout.
flush
  whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

See also

- IPython user manual: https://ipython.readthedocs.io/en/stable/
- Jupyter Notebook QuickStart: https://docs.jupyter.org/en/latest/start/index.html

1.4.2 Elaboration of the work in an editor

As you move forward, it will be important to not only work interactively, but also to create and reuse Python files. For this, a powerful code editor will get you far. Here are several good easy-to-use editors:

- Spyder: integrates an IPython console, a debugger, a profiler...
- PyCharm: integrates an IPython console, notebooks, a debugger... (freely available, but commercial)
- Visual Studio Code: integrates a Python console, notebooks, a debugger, ...

Some of these are shipped by the various scientific Python distributions, and you can find them in the menus.

As an exercise, create a file *my_file.py* in a code editor, and add the following lines:

```
s = 'Hello world'
print(s)
```

Now, you can run it in IPython console or a notebook and explore the resulting variables:

```
In [3]: %run my_file.py
Hello world

In [4]: s
Out[4]: 'Hello world'

In [5]: %whos
Variable   Type    Data/Info
----------------------------
s          str     Hello world
```

From a script to functions

While it is tempting to work only with scripts, that is a file full of instructions following each other, do plan to progressively evolve the script to a set of functions:

- A script is not reusable, functions are.
- Thinking in terms of functions helps breaking the problem in small blocks.

1.4.3 IPython and Jupyter Tips and Tricks

The user manuals contain a wealth of information. Here we give a quick introduction to four useful features: *history*, *tab completion*, *magic functions*, and *aliases*.

Command history Like a UNIX shell, the IPython console supports command history. Type *up* and *down* to navigate previously typed commands:

```
In [6]: x = 10

In [7]: <UP>

In [8]: x = 10
```

Tab completion Tab completion, is a convenient way to explore the structure of any object you're dealing with. Simply type object_name.<TAB> to view the object's attributes. Besides Python objects and keywords, tab completion also works on file and directory names.*

```
In [9]: x = 10

In [10]: x.<TAB>
         as_integer_ratio() conjugate()       imag              to_bytes()
         bit_count()        denominator       numerator
         bit_length()       from_bytes()      real
```

Magic functions The console and the notebooks support so-called *magic* functions by prefixing a command with the `%` character. For example, the `run` and `whos` functions from the previous section are magic functions. Note that, the setting `automagic`, which is enabled by default, allows you to omit the preceding `%` sign. Thus, you can just type the magic function and it will work.

Other useful magic functions are:

- `%cd` to change the current directory.

```
In [11]: cd /tmp
/tmp
```

- `%cpaste` allows you to paste code, especially code from websites which has been prefixed with the standard Python prompt (e.g. `>>>`) or with an ipython prompt, (e.g. `in [3]`):

```
In [12]: %cpaste
```

- `%timeit` allows you to time the execution of short snippets using the `timeit` module from the standard library:

```
In [12]: %timeit x = 10
11.5 ns +- 0.0165 ns per loop (mean +- std. dev. of 7 runs, 100,000,000 loops␣
↪each)
```

> **See also**
>
> *Chapter on optimizing code*

- `%debug` allows you to enter post-mortem debugging. That is to say, if the code you try to execute, raises an exception, using `%debug` will enter the debugger at the point where the exception was thrown.

```
In [13]: x === 10
  Cell In[13], line 1
    x === 10
        ^
SyntaxError: invalid syntax


In [14]: %debug
> /home/jarrod/.venv/lectures/lib64/python3.11/site-packages/IPython/core/
↪compilerop.py(86)ast_parse()
     84         Arguments are exactly the same as ast.parse (in the standard␣
↪library),
     85         and are passed to the built-in compile function."""
---> 86         return compile(source, filename, symbol, self.flags | PyCF_ONLY_
↪AST, 1)
     87
     88     def reset_compiler_flags(self):
ipdb> locals()
{'self': <IPython.core.compilerop.CachingCompiler object at 0x7f30d02efc10>,
↪'source': 'x === 10\n', 'filename': '<ipython-input-1-8e8bc565444b>', 'symbol
↪': 'exec'}
ipdb>
```

> **See also**

Chapter on debugging

Aliases Furthermore IPython ships with various *aliases* which emulate common UNIX command line tools such as `ls` to list files, `cp` to copy files and `rm` to remove files (a full list of aliases is shown when typing `alias`).

Getting help

- The built-in cheat-sheet is accessible via the `%quickref` magic function.
- A list of all available magic functions is shown when typing `%magic`.

CHAPTER 2

The Python language

Authors: *Chris Burns, Christophe Combelles, Emmanuelle Gouillart, Gaël Varoquaux*

Python for scientific computing

We introduce here the Python language. Only the bare minimum necessary for getting started with NumPy and SciPy is addressed here. To learn more about the language, consider going through the excellent tutorial https://docs.python.org/3/tutorial. Dedicated books are also available, such as Dive into Python 3.

Tip

Python is a **programming language**, as are C, Fortran, BASIC, PHP, etc. Some specific features of Python are as follows:

- an *interpreted* (as opposed to *compiled*) language. Contrary to e.g. C or Fortran, one does not compile Python code before executing it. In addition, Python can be used **interactively**: many Python interpreters are available, from which commands and scripts can be executed.
- a free software released under an **open-source** license: Python can be used and distributed free of charge, even for building commercial software.
- **multi-platform**: Python is available for all major operating systems, Windows, Linux/Unix, MacOS X, most likely your mobile phone OS, etc.

- a very readable language with clear non-verbose syntax
- a language for which a large variety of high-quality packages are available for various applications, from web frameworks to scientific computing.
- a language very easy to interface with other languages, in particular C and C++.
- Some other features of the language are illustrated just below. For example, Python is an object-oriented language, with dynamic typing (the same variable can contain objects of different types during the course of a program).

See https://www.python.org/about/ for more information about distinguishing features of Python.

2.1 First steps

Start the **Ipython** shell (an enhanced interactive Python shell):

- by typing "ipython" from a Linux/Mac terminal, or from the Windows cmd shell,
- **or** by starting the program from a menu, e.g. the Anaconda Navigator, the Python(x,y) menu if you have installed one of these scientific-Python suites.

Tip

If you don't have Ipython installed on your computer, other Python shells are available, such as the plain Python shell started by typing "python" in a terminal, or the Idle interpreter. However, we advise to use the Ipython shell because of its enhanced features, especially for interactive scientific computing.

Once you have started the interpreter, type

```
>>> print("Hello, world!")
Hello, world!
```

Tip

The message "Hello, world!" is then displayed. You just executed your first Python instruction, congratulations!

To get yourself started, type the following stack of instructions

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<class 'int'>
>>> print(b)
6
>>> a*b
18
>>> b = 'hello'
>>> type(b)
<class 'str'>
>>> b + b
'hellohello'
```

(continues on next page)

(continued from previous page)

```
>>> 2*b
'hellohello'
```

Tip

Two variables `a` and `b` have been defined above. Note that one does not declare the type of a variable before assigning its value. In C, conversely, one should write:

```
int a = 3;
```

In addition, the type of a variable may change, in the sense that at one point in time it can be equal to a value of a certain type, and a second point in time, it can be equal to a value of a different type. *b* was first equal to an integer, but it became equal to a string when it was assigned the value *'hello'*. Operations on integers (`b=2*a`) are coded natively in Python, and so are some operations on strings such as additions and multiplications, which amount respectively to concatenation and repetition.

2.2 Basic types

2.2.1 Numerical types

Tip

Python supports the following numerical, scalar types:

Integer

```
>>> 1 + 1
2
>>> a = 4
>>> type(a)
<class 'int'>
```

Floats

```
>>> c = 2.1
>>> type(c)
<class 'float'>
```

Complex

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> type(1. + 0j)
<class 'complex'>
```

Booleans

```
>>> 3 > 4
False
```

(continues on next page)

(continued from previous page)

```
>>> test = (3 > 4)
>>> test
False
>>> type(test)
<class 'bool'>
```

Tip

A Python shell can therefore replace your pocket calculator, with the basic arithmetic operations `+`, `-`, `*`, `/`, `%` (modulo) natively implemented

```
>>> 7 * 3.
21.0
>>> 2**10
1024
>>> 8 % 3
2
```

Type conversion (casting):

```
>>> float(1)
1.0
```

2.2.2 Containers

Tip

Python provides many efficient types of containers, in which collections of objects can be stored.

Lists

Tip

A list is an ordered collection of objects, that may have different types. For example:

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> type(colors)
<class 'list'>
```

Indexing: accessing individual objects contained in the list:

```
>>> colors[2]
'green'
```

Counting from the end with negative indices:

```
>>> colors[-1]
'white'
>>> colors[-2]
'black'
```

Warning

Indexing starts at 0 (as in C), not at 1 (as in Fortran or Matlab)!

Slicing: obtaining sublists of regularly-spaced elements:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[2:4]
['green', 'black']
```

Warning

Note that `colors[start:stop]` contains the elements with indices `i` such as `start<= i < stop` (`i` ranging from `start` to `stop-1`). Therefore, `colors[start:stop]` has `(stop - start)` elements.

Slicing syntax: `colors[start:stop:stride]`

Tip

All slicing parameters are optional:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[3:]
['black', 'white']
>>> colors[:3]
['red', 'blue', 'green']
>>> colors[::2]
['red', 'green', 'white']
```

Lists are *mutable* objects and can be modified:

```
>>> colors[0] = 'yellow'
>>> colors
['yellow', 'blue', 'green', 'black', 'white']
>>> colors[2:4] = ['gray', 'purple']
>>> colors
['yellow', 'blue', 'gray', 'purple', 'white']
```

Note

The elements of a list may have different types:

```
>>> colors = [3, -200, 'hello']
>>> colors
[3, -200, 'hello']
>>> colors[1], colors[2]
(-200, 'hello')
```

Tip

For collections of numerical data that all have the same type, it is often **more efficient** to use the `array` type provided by the `numpy` module. A NumPy array is a chunk of memory containing fixed-sized items. With NumPy arrays, operations on elements can be faster because elements are regularly spaced in memory and more operations are performed through specialized C functions instead of Python loops.

Tip

Python offers a large panel of functions to modify lists, or query them. Here are a few examples; for more details, see https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Add and remove elements:

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> colors.append('pink')
>>> colors
['red', 'blue', 'green', 'black', 'white', 'pink']
>>> colors.pop() # removes and returns the last item
'pink'
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors.extend(['pink', 'purple']) # extend colors, in-place
>>> colors
['red', 'blue', 'green', 'black', 'white', 'pink', 'purple']
>>> colors = colors[:-2]
>>> colors
['red', 'blue', 'green', 'black', 'white']
```

Reverse:

```
>>> rcolors = colors[::-1]
>>> rcolors
['white', 'black', 'green', 'blue', 'red']
>>> rcolors2 = list(colors) # new object that is a copy of colors in a different␣
↪memory area
>>> rcolors2
['red', 'blue', 'green', 'black', 'white']
>>> rcolors2.reverse() # in-place; reversing rcolors2 does not affect colors
>>> rcolors2
['white', 'black', 'green', 'blue', 'red']
```

Concatenate and repeat lists:

```
>>> rcolors + colors
['white', 'black', 'green', 'blue', 'red', 'red', 'blue', 'green', 'black', 'white']
>>> rcolors * 2
['white', 'black', 'green', 'blue', 'red', 'white', 'black', 'green', 'blue', 'red']
```

Tip

Sort:

```
>>> sorted(rcolors) # new object
['black', 'blue', 'green', 'red', 'white']
>>> rcolors
['white', 'black', 'green', 'blue', 'red']
>>> rcolors.sort()  # in-place
>>> rcolors
['black', 'blue', 'green', 'red', 'white']
```

Methods and Object-Oriented Programming

The notation `rcolors.method()` (e.g. `rcolors.append(3)` and `colors.pop()`) is our first example of object-oriented programming (OOP). Being a `list`, the object *rcolors* owns the *method function* that is called using the notation `.`. No further knowledge of OOP than understanding the notation `.` is necessary for going through this tutorial.

Discovering methods:

Reminder: in Ipython: tab-completion (press tab)

```
In [1]: rcolors.<TAB>
                append()  count()   insert()  reverse()
                clear()   extend()  pop()     sort()
                copy()    index()   remove()
```

Strings

Different string syntaxes (simple, double or triple quotes):

```
s = 'Hello, how are you?'
s = "Hi, what's up"
s = '''Hello,
       how are you'''          # tripling the quotes allows the
                               # string to span more than one line
s = """Hi,
what's up?"""
```

```
In [2]: 'Hi, what's up?'
  Cell In[2], line 1
    'Hi, what's up?'
                ^
SyntaxError: unterminated string literal (detected at line 1)
```

This syntax error can be avoided by enclosing the string in double quotes instead of single quotes. Alternatively, one can prepend a backslash to the second single quote. Other uses of the backslash are, e.g., the newline character `\n` and the tab character `\t`.

Tip

Strings are collections like lists. Hence they can be indexed and sliced, using the same syntax and rules.

Indexing:

```
>>> a = "hello"
>>> a[0]
'h'
>>> a[1]
'e'
>>> a[-1]
'o'
```

Tip

(Remember that negative indices correspond to counting from the right end.)

Slicing:

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,'
>>> a[2:10:2] # Syntax: a[start:stop:step]
'lo o'
>>> a[::3] # every three characters, from beginning to end
'hl r!'
```

Tip

Accents and special characters can also be handled as in Python 3 strings consist of Unicode characters.

A string is an **immutable object** and it is not possible to modify its contents. One may however create new strings from the original one.

```
In [3]: a = "hello, world!"

In [4]: a.replace('l', 'z', 1)
Out[4]: 'hezlo, world!'
```

Tip

Strings have many useful methods, such as `a.replace` as seen above. Remember the `a.` object-oriented notation and use tab completion or `help(str)` to search for new methods.

See also

Python offers advanced possibilities for manipulating strings, looking for patterns or formatting. The interested reader is referred to https://docs.python.org/3/library/stdtypes.html#string-methods and https://docs.python.org/3/library/string.html#format-string-syntax

String formatting:

```
>>> 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string') # with␣
↪more values use tuple after %
'An integer: 1; a float: 0.100000; another string: string'
```

(continues on next page)

(continued from previous page)

```
>>> i = 102
>>> filename = 'processing_of_dataset_%d.txt' % i    # no need for tuples with just␣
↪one value after %
>>> filename
'processing_of_dataset_102.txt'
```

Dictionaries

Tip

A dictionary is basically an efficient table that **maps keys to values**.

```
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'emmanuelle': 5752, 'sebastian': 5578, 'francis': 5915}
>>> tel['sebastian']
5578
>>> tel.keys()
dict_keys(['emmanuelle', 'sebastian', 'francis'])
>>> tel.values()
dict_values([5752, 5578, 5915])
>>> 'francis' in tel
True
```

Tip

It can be used to conveniently store and retrieve values associated with a name (a string for a date, a name, etc.). See https://docs.python.org/3/tutorial/datastructures.html#dictionaries for more information.

A dictionary can have keys (resp. values) with different types:

```
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
{'a': 1, 'b': 2, 3: 'hello'}
```

More container types

Tuples

Tuples are basically immutable lists. The elements of a tuple are written between parentheses, or just separated by commas:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

Sets: unordered, unique items:

```
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
{'a', 'b', 'c'}
>>> s.difference(('a', 'b'))
{'c'}
```

2.2.3 Assignment operator

Tip

Python library reference says:

> Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects.

In short, it works as follows (simple assignment):

1. an expression on the right hand side is evaluated, the corresponding object is created/obtained
2. a **name** on the left hand side is assigned, or bound, to the r.h.s. object

Things to note:

- A single object can have several names bound to it:

```
In [5]: a = [1, 2, 3]

In [6]: b = a

In [7]: a
Out[7]: [1, 2, 3]

In [8]: b
Out[8]: [1, 2, 3]

In [9]: a is b
Out[9]: True

In [10]: b[1] = 'hi!'

In [11]: a
Out[11]: [1, 'hi!', 3]
```

- to change a list *in place*, use indexing/slices:

```
In [12]: a = [1, 2, 3]

In [13]: a
Out[13]: [1, 2, 3]

In [14]: a = ['a', 'b', 'c'] # Creates another object.

In [15]: a
Out[15]: ['a', 'b', 'c']

In [16]: id(a)
Out[16]: 139891681694144
```

(continues on next page)

(continued from previous page)

```
In [17]: a[:] = [1, 2, 3] # Modifies object in place.

In [18]: a
Out[18]: [1, 2, 3]

In [19]: id(a)
Out[19]: 139891681694144
```

- the key concept here is **mutable vs. immutable**
  - mutable objects can be changed in place
  - immutable objects cannot be modified once created

See also

A very good and detailed explanation of the above issues can be found in David M. Beazley's article Types and Objects in Python.

2.3 Control Flow

Controls the order in which the code is executed.

2.3.1 if/elif/else

```
>>> if 2**2 == 4:
...     print("Obvious!")
...
Obvious!
```

Blocks are delimited by indentation

Tip

Type the following lines in your Python interpreter, and be careful to **respect the indentation depth**. The Ipython shell automatically increases the indentation depth after a colon : sign; to decrease the indentation depth, go four spaces to the left with the Backspace key. Press the Enter key twice to leave the logical block.

```
>>> a = 10

>>> if a == 1:
...     print(1)
... elif a == 2:
...     print(2)
... else:
...     print("A lot")
...
A lot
```

Indentation is compulsory in scripts as well. As an exercise, re-type the previous lines with the same indentation in a script `condition.py`, and execute the script with `run condition.py` in Ipython.

2.3.2 for/range

Iterating with an index:

```
>>> for i in range(4):
...     print(i)
0
1
2
3
```

But most often, it is more readable to iterate over values:

```
>>> for word in ('cool', 'powerful', 'readable'):
...     print('Python is %s' % word)
Python is cool
Python is powerful
Python is readable
```

2.3.3 while/break/continue

Typical C-style while loop (Mandelbrot problem):

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     z = z**2 + 1
>>> z
(-134+352j)
```

More advanced features

`break` out of enclosing for/while loop:

```
>>> z = 1 + 1j

>>> while abs(z) < 100:
...     if z.imag == 0:
...         break
...     z = z**2 + 1
```

`continue` the next iteration of a loop.:

```
>>> a = [1, 0, 2, 4]
>>> for element in a:
...     if element == 0:
...         continue
...     print(1. / element)
1.0
0.5
0.25
```

2.3.4 Conditional Expressions

`if <OBJECT>`

Evaluates to False:

- any number equal to zero (0, 0.0, 0+0j)
- an empty container (list, tuple, set, dictionary, . . .)

- `False`, `None`

Evaluates to True:

- everything else

`a == b`
: Tests equality, with logics:

```
>>> 1 == 1.
True
```

`a is b`
: Tests identity: both sides are the same object:

```
>>> a = 1
>>> b = 1.
>>> a == b
True
>>> a is b
False

>>> a = 1
>>> b = 1
>>> a is b
True
```

`a in b`
: For any collection `b`: `b` contains `a`

```
>>> b = [1, 2, 3]
>>> 2 in b
True
>>> 5 in b
False
```

If `b` is a dictionary, this tests that `a` is a key of `b`.

2.3.5 Advanced iteration

Iterate over any *sequence*

You can iterate over any sequence (string, list, keys in a dictionary, lines in a file, ...):

```
>>> vowels = 'aeiouy'

>>> for i in 'powerful':
...     if i in vowels:
...         print(i)
o
e
u
```

```
>>> message = "Hello how are you?"
>>> message.split() # returns a list
['Hello', 'how', 'are', 'you?']
>>> for word in message.split():
...     print(word)
...
Hello
```

(continues on next page)

(continued from previous page)

```
how
are
you?
```

Tip

Few languages (in particular, languages for scientific computing) allow to loop over anything but integers/indices. With Python it is possible to loop exactly over the objects of interest without bothering with indices you often don't care about. This feature can often be used to make code more readable.

Warning

Not safe to modify the sequence you are iterating over.

Keeping track of enumeration number

Common task is to iterate over a sequence while keeping track of the item number.

- Could use while loop with a counter as above. Or a for loop:

```
>>> words = ('cool', 'powerful', 'readable')
>>> for i in range(0, len(words)):
...     print((i, words[i]))
(0, 'cool')
(1, 'powerful')
(2, 'readable')
```

- But, Python provides a built-in function - `enumerate` - for this:

```
>>> for index, item in enumerate(words):
...     print((index, item))
(0, 'cool')
(1, 'powerful')
(2, 'readable')
```

Looping over a dictionary

Use **items**:

```
>>> d = {'a': 1, 'b':1.2, 'c':1j}

>>> for key, val in sorted(d.items()):
...     print('Key: %s has value: %s' % (key, val))
Key: a has value: 1
Key: b has value: 1.2
Key: c has value: 1j
```

Note

The ordering of a dictionary is random, thus we use `sorted()` which will sort on the keys.

2.3.6 List Comprehensions

Instead of creating a list by means of a loop, one can make use of a list comprehension with a rather self-explaining syntax.

```
>>> [i**2 for i in range(4)]
[0, 1, 4, 9]
```

Exercise

Compute the decimals of Pi using the Wallis formula:

$$\pi = 2 \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

2.4 Defining functions

2.4.1 Function definition

```
In [1]: def test():
   ...:     print('in test function')
   ...:
   ...:

In [2]: test()
in test function
```

Warning

Function blocks must be indented as other control-flow blocks.

2.4.2 Return statement

Functions can *optionally* return values.

```
In [3]: def disk_area(radius):
   ...:     return 3.14 * radius * radius
   ...:

In [4]: disk_area(1.5)
Out[4]: 7.0649999999999995
```

Note

By default, functions return `None`.

Note

Note the syntax to define a function:

- the `def` keyword;
- is followed by the function's **name**, then
- the arguments of the function are given between parentheses followed by a colon.
- the function body;
- and `return object` for optionally returning values.

2.4.3 Parameters

Mandatory parameters (positional arguments)

```
In [5]: def double_it(x):
   ...:     return x * 2
   ...:

In [6]: double_it(3)
Out[6]: 6

In [7]: double_it()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 double_it()

TypeError: double_it() missing 1 required positional argument: 'x'
```

Optional parameters (keyword or named arguments)

```
In [8]: def double_it(x=2):
   ...:     return x * 2
   ...:

In [9]: double_it()
Out[9]: 4

In [10]: double_it(3)
Out[10]: 6
```

Keyword arguments allow you to specify *default values*.

Warning

Default values are evaluated when the function is defined, not when it is called. This can be problematic when using mutable types (e.g. dictionary or list) and modifying them in the function body, since the modifications will be persistent across invocations of the function.

Using an immutable type in a keyword argument:

```
In [11]: bigx = 10

In [12]: def double_it(x=bigx):
   ....:     return x * 2
   ....:

In [13]: bigx = 1e9  # Now really big

In [14]: double_it()
Out[14]: 20
```

Using an mutable type in a keyword argument (and modifying it inside the function body):

```
In [15]: def add_to_dict(args={'a': 1, 'b': 2}):
   ....:     for i in args.keys():
   ....:         args[i] += 1
   ....:     print(args)
   ....:

In [16]: add_to_dict
Out[16]: <function __main__.add_to_dict(args={'a': 1, 'b': 2})>

In [17]: add_to_dict()
{'a': 2, 'b': 3}

In [18]: add_to_dict()
{'a': 3, 'b': 4}

In [19]: add_to_dict()
{'a': 4, 'b': 5}
```

Tip

More involved example implementing python's slicing:

```
In [20]: def slicer(seq, start=None, stop=None, step=None):
   ....:     """Implement basic python slicing."""
   ....:     return seq[start:stop:step]
   ....:

In [21]: rhyme = 'one fish, two fish, red fish, blue fish'.split()

In [22]: rhyme
Out[22]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [23]: slicer(rhyme)
Out[23]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [24]: slicer(rhyme, step=2)
Out[24]: ['one', 'two', 'red', 'blue']

In [25]: slicer(rhyme, 1, step=2)
Out[25]: ['fish,', 'fish,', 'fish,', 'fish']

In [26]: slicer(rhyme, start=1, stop=4, step=2)
Out[26]: ['fish,', 'fish,']
```

The order of the keyword arguments does not matter:

```
In [27]: slicer(rhyme, step=2, start=1, stop=4)
Out[27]: ['fish,', 'fish,']
```

but it is good practice to use the same ordering as the function's definition.

Keyword arguments are a very convenient feature for defining functions with a variable number of arguments, especially when default values are to be used in most calls to the function.

2.4.4 Passing by value

Tip

Can you modify the value of a variable inside a function? Most languages (C, Java, ...) distinguish "passing by value" and "passing by reference". In Python, such a distinction is somewhat artificial, and it is a bit subtle whether your variables are going to be modified or not. Fortunately, there exist clear rules.

Parameters to functions are references to objects, which are passed by value. When you pass a variable to a function, python passes the reference to the object to which the variable refers (the **value**). Not the variable itself.

If the **value** passed in a function is immutable, the function does not modify the caller's variable. If the **value** is mutable, the function may modify the caller's variable in-place:

```
>>> def try_to_modify(x, y, z):
...     x = 23
...     y.append(42)
...     z = [99] # new reference
...     print(x)
...     print(y)
...     print(z)
...
>>> a = 77    # immutable variable
>>> b = [99]  # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
>>> print(a)
77
>>> print(b)
[99, 42]
>>> print(c)
[28]
```

Functions have a local variable table called a *local namespace*.

The variable `x` only exists within the function `try_to_modify`.

2.4.5 Global variables

Variables declared outside the function can be referenced within the function:

```
In [28]: x = 5

In [29]: def addx(y):
   ....:     return x + y
   ....:

In [30]: addx(10)
Out[30]: 15
```

But these "global" variables cannot be modified within the function, unless declared **global** in the function.

This doesn't work:

```
In [31]: def setx(y):
   ....:     x = y
   ....:     print('x is %d' % x)
   ....:
   ....:

In [32]: setx(10)
x is 10

In [33]: x
Out[33]: 5
```

This works:

```
In [34]: def setx(y):
   ....:     global x
   ....:     x = y
   ....:     print('x is %d' % x)
   ....:
   ....:

In [35]: setx(10)
x is 10

In [36]: x
Out[36]: 10
```

2.4.6 Variable number of parameters

Special forms of parameters:

- `*args`: any number of positional arguments packed into a tuple
- `**kwargs`: any number of keyword arguments packed into a dictionary

```
In [37]: def variable_args(*args, **kwargs):
   ....:     print('args is', args)
   ....:     print('kwargs is', kwargs)
   ....:

In [38]: variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'x': 1, 'y': 2, 'z': 3}
```

2.4.7 Docstrings

Documentation about what the function does and its parameters. General convention:

```
In [39]: def funcname(params):
   ....:     """Concise one-line sentence describing the function.
   ....:
   ....:     Extended summary which can contain multiple paragraphs.
   ....:     """
   ....:     # function body
   ....:     pass
   ....:
```

(continues on next page)

(continued from previous page)

```
In [40]: funcname?
Signature: funcname(params)
Docstring:
Concise one-line sentence describing the function.
Extended summary which can contain multiple paragraphs.
File:      ~/src/scientific-python-lectures/<ipython-input-13-64e466df6d64>
Type:      function
```

Note

Docstring guidelines

For the sake of standardization, the Docstring Conventions webpage documents the semantics and conventions associated with Python docstrings.

Also, the NumPy and SciPy modules have defined a precise standard for documenting scientific functions, that you may want to follow for your own functions, with a `Parameters` section, an `Examples` section, etc. See https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard

2.4.8 Functions are objects

Functions are first-class objects, which means they can be:

- assigned to a variable
- an item in a list (or any collection)
- passed as an argument to another function.

```
In [41]: va = variable_args

In [42]: va('three', x=1, y=2)
args is ('three',)
kwargs is {'x': 1, 'y': 2}
```

2.4.9 Methods

Methods are functions attached to objects. You've seen these in our examples on *lists*, *dictionaries*, *strings*, etc...

2.4.10 Exercises

Exercise: Fibonacci sequence

Write a function that displays the `n` first terms of the Fibonacci sequence, defined by:

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

Exercise: Quicksort

Implement the quicksort algorithm, as defined by wikipedia

```
function quicksort(array)
    var list less, greater
    if length(array) < 2
        return array
    select and remove a pivot value pivot from array
    for each x in array
        if x < pivot + 1 then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

2.5 Reusing code: scripts and modules

For now, we have typed all instructions in the interpreter. For longer sets of instructions we need to change track and write the code in text files (using a text editor), that we will call either *scripts* or *modules*. Use your favorite text editor (provided it offers syntax highlighting for Python), or the editor that comes with the Scientific Python Suite you may be using.

2.5.1 Scripts

Tip

Let us first write a *script*, that is a file with a sequence of instructions that are executed each time the script is called. Instructions may be e.g. copied-and-pasted from the interpreter (but take care to respect indentation rules!).

The extension for Python files is `.py`. Write or copy-and-paste the following lines in a file called `test.py`

```
message = "Hello how are you?"
for word in message.split():
    print(word)
```

Tip

Let us now execute the script interactively, that is inside the Ipython interpreter. This is maybe the most common use of scripts in scientific computing.

Note

in Ipython, the syntax to execute a script is `%run script.py`. For example,

```
In [1]: %run test.py
Hello
how
are
you?

In [2]: message
Out[2]: 'Hello how are you?'
```

The script has been executed. Moreover the variables defined in the script (such as `message`) are now available inside the interpreter's namespace.

Tip

Other interpreters also offer the possibility to execute scripts (e.g., `execfile` in the plain Python interpreter, etc.).

It is also possible In order to execute this script as a *standalone program*, by executing the script inside a shell terminal (Linux/Mac console or cmd Windows console). For example, if we are in the same directory as the test.py file, we can execute this in a console:

```
$ python test.py
Hello
how
are
you?
```

Tip

Standalone scripts may also take command-line arguments

In `file.py`:

```
import sys
print(sys.argv)
```

```
$ python file.py test arguments
['file.py', 'test', 'arguments']
```

Warning

Don't implement option parsing yourself. Use a dedicated module such as `argparse`.

2.5.2 Importing objects from modules

```
In [3]: import os

In [4]: os
Out[4]: <module 'os' (frozen)>

In [5]: os.listdir('.')
Out[5]:
['profile_zpw1zshu',
 'profile_yx2v9j1u',
 'profile_hvadjvo0',
 'profile_3e7b7mnh',
 'profile_u_dkva9k',
 'profile_17z9xbiu',
 'profile_nhsyjqfm',
 'profile_uyk_j767',
 'profile_g6ucuiy9',
 'profile_4anwydli',
 'profile_wej8tpc7',
 'dotnet-diagnostic-831-1101-socket',
 'profile_nnfonejt',
 'clr-debug-pipe-831-1101-out',
 'profile_h8ahy7fq',
```

(continues on next page)

(continued from previous page)

```
'profile_hebci4u3',
'profile_662qdwyj',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-chrony.service-aXXAhc',
'profile_odcbc8pq',
'profile_rkxkqy6r',
'clr-debug-pipe-1824-12120-out',
'profile_un100jp7',
'profile_glp4r62l',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-polkit.service-kkpwfH',
'profile_5vwjni03',
'profile_buspmxtb',
'.ICE-unix',
'clr-debug-pipe-1824-12120-in',
'clr-debug-pipe-831-1101-in',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-ModemManager.service-Empwlt',
'snap-private-tmp',
'profile_sfdld8m4',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-systemd-logind.service-cYg7Es',
'profile_yc76tsby',
'profile_dcwdz0l1',
'profile_99cs1dgd',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-systemd-resolved.service-nnkqv9',
'profile_a3cfjro5',
'profile_xxx9__92',
'profile_1e7iroyb',
'profile_dj6lnbhy',
'clr-debug-pipe-1808-11921-in',
'profile_0p2hwy1o',
'profile_md53_vd8',
'profile_u0zph5is',
'profile_lci7bgu5',
'dotnet-diagnostic-1824-12120-socket',
'profile_fo__gd4r',
'profile_c547dl1i',
'profile_8j1j5lyh',
'profile_e07ps17i',
'profile_bklgzk93',
'profile_ypn7g29s',
'.XIM-unix',
'profile_ebl3yo6f',
'.X11-unix',
'profile_hh8uhz1r',
'hsperfdata_root',
'profile_mn8s6x17',
'dotnet-diagnostic-1808-11921-socket',
'profile_7h7xd5mi',
'profile_3ckq9d03',
'profile_x8bqc_g8',
'profile_c57e_94a',
'profile_b49of1_b',
'clr-debug-pipe-1808-11921-out',
'profile_iohwvmzf',
'profile_6f04cas7',
'profile_daahkszg',
'profile_ogv3klaq',
'profile_y0hi0gcb',
```

(continues on next page)

(continued from previous page)

```
'profile_yyac79ne',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-haveged.service-KLck1c',
'.font-unix',
'profile_lu41b8dd',
'profile_1t3sdp4r',
'profile_hbu89y3v',
'profile_wmjz7lc4',
'profile_05dtmzwl',
'profile_8qxt95v9']
```

And also:

```
In [6]: from os import listdir
```

Importing shorthands:

```
In [7]: import numpy as np
```

Warning

```
from os import *
```

This is called the *star import* and please, **Do not use it**

- Makes the code harder to read and understand: where do symbols come from?
- Makes it impossible to guess the functionality by the context and the name (hint: *os.name* is the name of the OS), and to profit usefully from tab completion.
- Restricts the variable names you can use: *os.name* might override *name*, or vise-versa.
- Creates possible name clashes between modules.
- Makes the code impossible to statically check for undefined symbols.

Tip

Modules are thus a good way to organize code in a hierarchical way. Actually, all the scientific computing tools we are going to use are modules:

```
>>> import numpy as np # data arrays
>>> np.linspace(0, 10, 6)
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> import scipy as sp  # scientific computing
```

2.5.3 Creating modules

Tip

If we want to write larger and better organized programs (compared to simple scripts), where some objects are defined, (variables, functions, classes) and that we want to reuse several times, we have to create our own *modules*.

Let us create a module `demo` contained in the file `demo.py`:

```
"A demo module."


def print_b():
    "Prints b."
    print("b")


def print_a():
    "Prints a."
    print("a")


c = 2
d = 2
```

Tip

In this file, we defined two functions `print_a` and `print_b`. Suppose we want to call the `print_a` function from the interpreter. We could execute the file as a script, but since we just want to have access to the function `print_a`, we are rather going to **import it as a module**. The syntax is as follows.

```
In [8]: import demo

In [9]: demo.print_a()
a

In [10]: demo.print_b()
b
```

Importing the module gives access to its objects, using the `module.object` syntax. Don't forget to put the module's name before the object's name, otherwise Python won't recognize the instruction.

Introspection

```
In [11]: demo?
Type:               module
Base Class: <type 'module'>
String Form:        <module 'demo' from 'demo.py'>
Namespace:  Interactive
File:               /home/varoquau/Projects/Python_talks/scipy_2009_tutorial/source/
↪demo.py
Docstring:
    A demo module.

In [12]: who
demo

In [13]: whos
Variable   Type      Data/Info
------------------------------
demo       module    <module 'demo' from 'demo.py'>

In [14]: dir(demo)
Out[14]:
```

(continues on next page)

(continued from previous page)

```
['__builtins__',
'__doc__',
'__file__',
'__name__',
'__package__',
'c',
'd',
'print_a',
'print_b']

In [15]: demo.<TAB>
demo.c         demo.print_a  demo.py
demo.d         demo.print_b  demo.pyc
```

Importing objects from modules into the main namespace

```
In [16]: from demo import print_a, print_b

In [17]: whos
Variable   Type        Data/Info
-------------------------------
demo       module      <module 'demo' from 'demo.py'>
print_a    function    <function print_a at 0xb7421534>
print_b    function    <function print_b at 0xb74214c4>

In [18]: print_a()
a
```

Warning

Module caching

Modules are cached: if you modify `demo.py` and re-import it in the old session, you will get the old one.

Solution:

```
In [10]: importlib.reload(demo)
```

2.5.4 '__main__' and module loading

Tip

Sometimes we want code to be executed when a module is run directly, but not when it is imported by another module. `if __name__ == '__main__'` allows us to check whether the module is being run directly.

File `demo2.py`:

```
def print_b():
    "Prints b."
    print("b")


def print_a():
```

(continues on next page)

(continued from previous page)

```
    "Prints a."
    print("a")


# print_b() runs on import
print_b()

if __name__ == "__main__":
    # print_a() is only executed when the module is run directly.
    print_a()
```

Importing it:

```
In [19]: import demo2
b

In [20]: import demo2
```

Running it:

```
In [21]: %run demo2
b
a
```

2.5.5 Scripts or modules? How to organize your code

Note

Rule of thumb

- Sets of instructions that are called several times should be written inside **functions** for better code reusability.
- Functions (or other bits of code) that are called from several scripts should be written inside a **module**, so that only the module is imported in the different scripts (do not copy-and-paste your functions in the different scripts!).

How modules are found and imported

When the `import mymodule` statement is executed, the module `mymodule` is searched in a given list of directories. This list includes a list of installation-dependent default path (e.g., `/usr/lib64/python3.11`) as well as the list of directories specified by the environment variable `PYTHONPATH`.

The list of directories searched by Python is given by the `sys.path` variable

```
In [22]: import sys

In [23]: sys.path
Out[23]:
['/home/runner/work/scientific-python-lectures/scientific-python-lectures',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python312.zip',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/lib-dynload',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages']
```

Modules must be located in the search path, therefore you can:

- write your own modules within directories already defined in the search path (e.g. `$HOME/.venv/lectures/lib64/python3.11/site-packages`). You may use symbolic links (on Linux) to keep the code somewhere else.
- modify the environment variable `PYTHONPATH` to include the directories containing the user-defined modules.

Tip

On Linux/Unix, add the following line to a file read by the shell at startup (e.g. /etc/profile, .profile)

```
export PYTHONPATH=$PYTHONPATH:/home/emma/user_defined_modules
```

On Windows, https://support.microsoft.com/kb/310519 explains how to handle environment variables.

- or modify the `sys.path` variable itself within a Python script.

Tip

```
import sys
new_path = '/home/emma/user_defined_modules'
if new_path not in sys.path:
    sys.path.append(new_path)
```

This method is not very robust, however, because it makes the code less portable (user-dependent path) and because you have to add the directory to your sys.path each time you want to import from a module in this directory.

See also

See https://docs.python.org/3/tutorial/modules.html for more information about modules.

2.5.6 Packages

A directory that contains many modules is called a *package*. A package is a module with submodules (which can have submodules themselves, etc.). A special file called `__init__.py` (which may be empty) tells Python that the directory is a Python package, from which modules can be imported.

```
$ ls
_build_utils/         fft/          _lib/        odr/          spatial/
cluster/              fftpack/      linalg/      optimize/     special/
conftest.py           __init__.py   linalg.pxd   optimize.pxd  special.pxd
constants/            integrate/    meson.build  setup.py      stats/
datasets/             interpolate/  misc/        signal/
_distributor_init.py  io/           ndimage/     sparse/
$ cd ndimage
$ ls
_filters.py  __init__.py        _measurements.py  morphology.py      src/
filters.py   _interpolation.py  measurements.py   _ni_docstrings.py  tests/
_fourier.py  interpolation.py   meson.build       _ni_support.py     utils/
fourier.py   LICENSE.txt        _morphology.py    setup.py
```

From Ipython:

```
In [24]: import scipy as sp

In [25]: sp.__file__
Out[25]: '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages/scipy/_
↪_init__.py'

In [26]: sp.version.version
Out[26]: '1.15.2'

In [27]: sp.ndimage.morphology.binary_dilation?
Signature:
sp.ndimage.morphology.binary_dilation(
    input,
    structure=None,
    iterations=1,
    mask=None,
    output=None,
    border_value=0,
    origin=0,
    brute_force=False,
)
Docstring:
Multidimensional binary dilation with the given structuring element.
...
```

2.5.7 Good practices

- Use **meaningful** object **names**
- **Indentation: no choice!**

> **Tip**
>
> Indenting is compulsory in Python! Every command block following a colon bears an additional indentation level with respect to the previous line with a colon. One must therefore indent after `def f():` or `while:`. At the end of such logical blocks, one decreases the indentation depth (and re-increases it if a new block is entered, etc.)
>
> Strict respect of indentation is the price to pay for getting rid of `{` or `;` characters that delineate logical blocks in other languages. Improper indentation leads to errors such as
>
> ```
> ------------------------------------------------------------
> IndentationError: unexpected indent (test.py, line 2)
> ```
>
> All this indentation business can be a bit confusing in the beginning. However, with the clear indentation, and in the absence of extra characters, the resulting code is very nice to read compared to other languages.

- **Indentation depth**: Inside your text editor, you may choose to indent with any positive number of spaces (1, 2, 3, 4, ...). However, it is considered good practice to **indent with 4 spaces**. You may configure your editor to map the `Tab` key to a 4-space indentation.
- **Style guidelines**

  Long lines: you should not write very long lines that span over more than (e.g.) 80 characters. Long lines can be broken with the `\` character

  ```
  >>> long_line = "Here is a very very long line \
  ... that we break in two parts."
  ```

Spaces

Write well-spaced code: put whitespaces after commas, around arithmetic operators, etc.:

```
>>> a = 1 # yes
>>> a=1 # too cramped
```

A certain number of rules for writing "beautiful" code (and more importantly using the same conventions as anybody else!) are given in the Style Guide for Python Code.

Quick read

If you want to do a first quick pass through the Scientific Python Lectures to learn the ecosystem, you can directly skip to the next chapter: *NumPy: creating and manipulating numerical data*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter later.

2.6 Input and Output

To be exhaustive, here are some information about input and output in Python. Since we will use the NumPy methods to read and write files, **you may skip this chapter at first reading**.

We write or read **strings** to/from files (other types must be converted to strings). To write in a file:

```
>>> f = open('workfile', 'w') # opens the workfile file
>>> type(f)
<class '_io.TextIOWrapper'>
>>> f.write('This is a test \nand another test')
>>> f.close()
```

To read from a file

```
In [1]: f = open('workfile', 'r')

In [2]: s = f.read()

In [3]: print(s)
This is a test
and another test

In [4]: f.close()
```

See also

For more details: https://docs.python.org/3/tutorial/inputoutput.html

2.6.1 Iterating over a file

```
In [5]: f = open('workfile', 'r')

In [6]: for line in f:
   ...:     print(line)
```

(continues on next page)

(continued from previous page)

```
   ...:
This is a test
and another test

In [7]: f.close()
```

File modes

- Read-only: `r`
- Write-only: `w`
  - Note: Create a new file or *overwrite* existing file.
- Append a file: `a`
- Read and Write: `r+`
- Binary mode: `b`
  - Note: Use for binary files, especially on Windows.

2.7 Standard Library

Note

Reference document for this section:

- The Python Standard Library documentation: https://docs.python.org/3/library/index.html
- Python Essential Reference, David Beazley, Addison-Wesley Professional

2.7.1 `os` module: operating system functionality

"A portable way of using operating system dependent functionality."

Directory and file manipulation

Current directory:

```
In [1]: import os

In [2]: os.getcwd()
Out[2]: '/tmp'
```

List a directory:

```
In [3]: os.listdir(os.curdir)
Out[3]:
['profile_fipa_q3s',
 'profile_zpw1zshu',
 'profile_yx2v9j1u',
 'profile_hvadjvo0',
 'profile_vjnmbn9t',
 'profile_3e7b7mnh',
 'profile_u_dkva9k',
 'profile_1_ho6wsw',
```

(continues on next page)

(continued from previous page)

```
'profile_17z9xbiu',
'profile_nhsyjqfm',
'profile_uyk_j767',
'profile_g6ucuiy9',
'profile_4anwydli',
'profile_wej8tpc7',
'dotnet-diagnostic-831-1101-socket',
'profile_x_2rv34s',
'profile_nnfonejt',
'clr-debug-pipe-831-1101-out',
'profile_h8ahy7fq',
'profile_hebci4u3',
'profile_662qdwyj',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-chrony.service-aXXAhc',
'profile_odcbc8pq',
'profile_rkxkqy6r',
'clr-debug-pipe-1824-12120-out',
'profile_un100jp7',
'profile_805ffpez',
'profile_ov3bujrd',
'profile_glp4r62l',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-polkit.service-kkpwfH',
'profile_5vwjni03',
'profile_buspmxtb',
'.ICE-unix',
'clr-debug-pipe-1824-12120-in',
'clr-debug-pipe-831-1101-in',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-ModemManager.service-Empwlt',
'snap-private-tmp',
'profile_sfdld8m4',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-systemd-logind.service-cYg7Es',
'profile_yc76tsby',
'profile_dcwdz0l1',
'profile_yho0ln7h',
'profile_99cs1dgd',
'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-systemd-resolved.service-nnkqv9',
'profile_a3cfjro5',
'profile_xxx9__92',
'profile_1e7iroyb',
'profile_dj6lnbhy',
'clr-debug-pipe-1808-11921-in',
'profile_0p2hwy1o',
'profile_md53_vd8',
'profile_u0zph5is',
'profile_lci7bgu5',
'dotnet-diagnostic-1824-12120-socket',
'profile_fo__gd4r',
'profile_c547dl1i',
'profile_oxmjht_b',
'profile_8j1j5lyh',
'profile_e07ps17i',
'profile_bklgzk93',
'profile_ypn7g29s',
'.XIM-unix',
'profile_ebl3yo6f',
'.X11-unix',
```

(continues on next page)

(continued from previous page)

```
 'profile_ajv63h_v',
 'profile_hh8uhz1r',
 'hsperfdata_root',
 'profile_mn8s6x17',
 'dotnet-diagnostic-1808-11921-socket',
 'profile_7h7xd5mi',
 'profile_3ckq9d03',
 'profile_x8bqc_g8',
 'profile_c57e_94a',
 'profile_b49of1_b',
 'clr-debug-pipe-1808-11921-out',
 'profile_iohwvmzf',
 'profile_gz7tsu91',
 'profile_6f04cas7',
 'profile_daahkszg',
 'profile_ogv3klaq',
 'profile_y0hi0gcb',
 'profile_0j7pkhjb',
 'profile_yyac79ne',
 'systemd-private-5a1e73d7020f41aebb5ccae2a9ef7f53-haveged.service-KLck1c',
 '.font-unix',
 'profile_lu41b8dd',
 'profile_1t3sdp4r',
 'profile_hbu89y3v',
 'profile_wmjz7lc4',
 'profile_05dtmzwl',
 'profile_8qxt95v9']
```

Make a directory:

```
In [4]: os.mkdir('junkdir')

In [5]: 'junkdir' in os.listdir(os.curdir)
Out[5]: True
```

Rename the directory:

```
In [6]: os.rename('junkdir', 'foodir')

In [7]: 'junkdir' in os.listdir(os.curdir)
Out[7]: False

In [8]: 'foodir' in os.listdir(os.curdir)
Out[8]: True

In [9]: os.rmdir('foodir')

In [10]: 'foodir' in os.listdir(os.curdir)
Out[10]: False
```

Delete a file:

```
In [11]: fp = open('junk.txt', 'w')

In [12]: fp.close()

In [13]: 'junk.txt' in os.listdir(os.curdir)
```

(continues on next page)

(continued from previous page)

```
Out[13]: True

In [14]: os.remove('junk.txt')

In [15]: 'junk.txt' in os.listdir(os.curdir)
Out[15]: False
```

os.path: path manipulations

os.path provides common operations on pathnames.

```
In [16]: fp = open('junk.txt', 'w')

In [17]: fp.close()

In [18]: a = os.path.abspath('junk.txt')

In [19]: a
Out[19]: '/tmp/junk.txt'

In [20]: os.path.split(a)
Out[20]: ('/tmp', 'junk.txt')

In [21]: os.path.dirname(a)
Out[21]: '/tmp'

In [22]: os.path.basename(a)
Out[22]: 'junk.txt'

In [23]: os.path.splitext(os.path.basename(a))
Out[23]: ('junk', '.txt')

In [24]: os.path.exists('junk.txt')
Out[24]: True

In [25]: os.path.isfile('junk.txt')
Out[25]: True

In [26]: os.path.isdir('junk.txt')
Out[26]: False

In [27]: os.path.expanduser('~/local')
Out[27]: '/home/runner/local'

In [28]: os.path.join(os.path.expanduser('~'), 'local', 'bin')
Out[28]: '/home/runner/local/bin'
```

Running an external command

```
In [29]: os.system('ls')
Out[29]: 0
```

> **Note**
>
> Alternative to os.system

A noteworthy alternative to `os.system` is the sh module. Which provides much more convenient ways to obtain the output, error stream and exit code of the external command.

```
In [30]: import sh
In [31]: com = sh.ls()

In [31]: print(com)
basic_types.rst   exceptions.rst   oop.rst              standard_library.rst
control_flow.rst  first_steps.rst  python_language.rst
demo2.py          functions.rst    python-logo.png
demo.py           io.rst           reusing_code.rst

In [32]: type(com)
Out[32]: str
```

Walking a directory

`os.path.walk` generates a list of filenames in a directory tree.

```
In [33]: for dirpath, dirnames, filenames in os.walk(os.curdir):
   ....:     for fp in filenames:
   ....:         print(os.path.abspath(fp))
   ....:
   ....:
/tmp/dotnet-diagnostic-831-1101-socket
/tmp/clr-debug-pipe-831-1101-out
/tmp/clr-debug-pipe-1824-12120-out
/tmp/clr-debug-pipe-1824-12120-in
/tmp/clr-debug-pipe-831-1101-in
/tmp/junk.txt
/tmp/clr-debug-pipe-1808-11921-in
/tmp/dotnet-diagnostic-1824-12120-socket
/tmp/dotnet-diagnostic-1808-11921-socket
/tmp/clr-debug-pipe-1808-11921-out
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
```

(continues on next page)

(continued from previous page)

```
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/dhist
/tmp/bookmarks
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
/tmp/README
```

Environment variables:

```
In [34]: os.environ.keys()
Out[34]: KeysView(environ({'SHELL': '/bin/bash', 'COLORTERM': 'truecolor', ...}))

In [35]: os.environ['SHELL']
Out[35]: '/bin/bash'
```

2.7.2 `shutil`: high-level file operations

The `shutil` provides useful file operations:

- `shutil.rmtree`: Recursively delete a directory tree.
- `shutil.move`: Recursively move a file or directory to another location.
- `shutil.copy`: Copy files or directories.

2.7.3 `glob`: Pattern matching on files

The `glob` module provides convenient file pattern matching.

Find all files ending in `.txt`:

```
In [36]: import glob

In [37]: glob.glob('*.txt')
Out[37]: ['junk.txt']
```

2.7.4 `sys` module: system-specific information

System-specific information related to the Python interpreter.

- Which version of python are you running and where is it installed:

```
In [38]: import sys

In [39]: sys.platform
Out[39]: 'linux'

In [40]: sys.version
Out[40]: '3.12.11 (main, Jun  4 2025, 04:14:08) [GCC 13.3.0]'

In [41]: sys.prefix
Out[41]: '/opt/hostedtoolcache/Python/3.12.11/x64'
```

- List of command line arguments passed to a Python script:

```
In [42]: sys.argv
Out[42]:
['/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages/sphinx/__main__
↪.py',
 '-b',
 'latex',
 '-d',
 'build/doctrees',
 '.',
 'build/latex']
```

`sys.path` is a list of strings that specifies the search path for modules. Initialized from PYTHONPATH:

```
In [43]: sys.path
Out[43]:
['/home/runner/work/scientific-python-lectures/scientific-python-lectures',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python312.zip',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/lib-dynload',
 '/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages']
```

2.7.5 `pickle`: easy persistence

Useful to store arbitrary objects to a file. Not safe or fast!

```
In [44]: import pickle

In [45]: l = [1, None, 'Stan']

In [46]: with open('test.pkl', 'wb') as file:
   ....:     pickle.dump(l, file)
   ....:

In [47]: with open('test.pkl', 'rb') as file:
   ....:     out = pickle.load(file)
   ....:

In [48]: out
Out[48]: [1, None, 'Stan']
```

Exercise

Write a program to search your `PYTHONPATH` for the module `site.py`.

path_site

2.8 Exception handling in Python

It is likely that you have raised Exceptions if you have typed all the previous commands of the tutorial. For example, you may have raised an exception if you entered a command with a typo.

Exceptions are raised by different kinds of errors arising when executing Python code. In your own code, you may also catch errors, or define custom error types. You may want to look at the descriptions of the the built-in Exceptions when looking for the right exception type.

2.8.1 Exceptions

Exceptions are raised by errors in Python:

```
In [1]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[1], line 1
----> 1 1/0

ZeroDivisionError: division by zero
```

(continues on next page)

(continued from previous page)

```
In [2]: 1 + 'e'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 1 + 'e'

TypeError: unsupported operand type(s) for +: 'int' and 'str'

In [3]: d = {1:1, 2:2}

In [4]: d[3]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[4], line 1
----> 1 d[3]

KeyError: 3

In [5]: l = [1, 2, 3]

In [6]: l[4]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 l[4]

IndexError: list index out of range

In [7]: l.foobar
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[7], line 1
----> 1 l.foobar

AttributeError: 'list' object has no attribute 'foobar'
```

As you can see, there are **different types** of exceptions for different errors.

2.8.2 Catching exceptions

try/except

```
In [8]: while True:
   ...:     try:
   ...:         x = int(input('Please enter a number: '))
   ...:         break
   ...:     except ValueError:
   ...:         print('That was no valid number.  Try again...')
   ...:
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1

In [9]: x
Out[9]: 1
```

try/finally

```
In [10]: try:
   ....:     x = int(input('Please enter a number: '))
   ....: finally:
   ....:     print('Thank you for your input')
   ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[10], line 2
      1 try:
----> 2     x = int(input('Please enter a number: '))
      3 finally:
      4     print('Thank you for your input')
ValueError: invalid literal for int() with base 10: 'a'
```

Important for resource management (e.g. closing a file)

Easier to ask for forgiveness than for permission

```
In [11]: def print_sorted(collection):
   ....:     try:
   ....:         collection.sort()
   ....:     except AttributeError:
   ....:         pass # The pass statement does nothing
   ....:     print(collection)
   ....:

In [12]: print_sorted([1, 3, 2])
[1, 2, 3]

In [13]: print_sorted(set((1, 3, 2)))
{1, 2, 3}

In [14]: print_sorted('132')
132
```

2.8.3 Raising exceptions

- Capturing and reraising an exception:

```
In [15]: def filter_name(name):
   ....:     try:
   ....:         name = name.encode('ascii')
   ....:     except UnicodeError as e:
   ....:         if name == 'Gaël':
   ....:             print('OK, Gaël')
   ....:         else:
   ....:             raise e
   ....:     return name
   ....:

In [16]: filter_name('Gaël')
OK, Gaël
Out[16]: 'Gaël'
```

(continues on next page)

(continued from previous page)

```
In [17]: filter_name('Stéfan')
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
Cell In[17], line 1
----> 1 filter_name('Stéfan')

Cell In[15], line 8, in filter_name(name)
      6         print('OK, Gaël')
      7     else:
----> 8         raise e
      9 return name

Cell In[15], line 3, in filter_name(name)
      1 def filter_name(name):
      2     try:
----> 3         name = name.encode('ascii')
      4     except UnicodeError as e:
      5         if name == 'Gaël':

UnicodeEncodeError: 'ascii' codec can't encode character '\xe9' in position 2:␣
↪ordinal not in range(128)
```

- Exceptions to pass messages between parts of the code:

```
In [18]: def achilles_arrow(x):
   ....:     if abs(x - 1) < 1e-3:
   ....:         raise StopIteration
   ....:     x = 1 - (1-x)/2.
   ....:     return x
   ....:

In [19]: x = 0

In [20]: while True:
   ....:     try:
   ....:         x = achilles_arrow(x)
   ....:     except StopIteration:
   ....:         break
   ....:
   ....:

In [21]: x
Out[21]: 0.9990234375
```

Use exceptions to notify certain conditions are met (e.g. StopIteration) or not (e.g. custom error raising)

2.9 Object-oriented programming (OOP)

Python supports object-oriented programming (OOP). The goals of OOP are:

- to organize the code, and
- to reuse code in similar contexts.

Here is a small example: we create a Student *class*, which is an object gathering several custom functions

(*methods*) and variables (*attributes*), we will be able to use:

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def set_age(self, age):
...         self.age = age
...     def set_major(self, major):
...         self.major = major
...
>>> anna = Student('anna')
>>> anna.set_age(21)
>>> anna.set_major('physics')
```

In the previous example, the Student class has `__init__`, `set_age` and `set_major` methods. Its attributes are `name`, `age` and `major`. We can call these methods and attributes with the following notation: `classinstance.method` or `classinstance.attribute`. The `__init__` constructor is a special method we call with: `MyClass(init parameters if any)`.

Now, suppose we want to create a new class MasterStudent with the same methods and attributes as the previous one, but with an additional `internship` attribute. We won't copy the previous class, but **inherit** from it:

```
>>> class MasterStudent(Student):
...     internship = 'mandatory, from March to June'
...
>>> james = MasterStudent('james')
>>> james.internship
'mandatory, from March to June'
>>> james.set_age(23)
>>> james.age
23
```

The MasterStudent class inherited from the Student attributes and methods.

Thanks to classes and object-oriented programming, we can organize code with different classes corresponding to different objects we encounter (an Experiment class, an Image class, a Flow class, etc.), with their own methods and attributes. Then we can use inheritance to consider variations around a base class and **reuse** code. Ex : from a Flow base class, we can create derived StokesFlow, TurbulentFlow, PotentialFlow, etc.

CHAPTER 3

NumPy: creating and manipulating numerical data

Authors: *Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux, and Pauli Virtanen*

This chapter gives an overview of NumPy, the core tool for performant numerical computing with Python.

3.1 The NumPy array object

Section contents

- *What are NumPy and NumPy arrays?*
- *Creating arrays*
- *Basic data types*
- *Basic visualization*
- *Indexing and slicing*
- *Copies and views*
- *Fancy indexing*

3.1.1 What are NumPy and NumPy arrays?

NumPy arrays

Python objects

- high-level number objects: integers, floating point
- containers: lists (costless insertion and append), dictionaries (fast lookup)

NumPy provides

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)
- Also known as *array oriented computing*

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

Tip

For example, An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Why it is useful: Memory-efficient container that provides fast numerical operations.

```
In [1]: L = range(1000)

In [2]: %timeit [i**2 for i in L]
51.1 us +- 955 ns per loop (mean +- std. dev. of 7 runs, 10,000 loops each)

In [3]: a = np.arange(1000)

In [4]: %timeit a**2
944 ns +- 8.18 ns per loop (mean +- std. dev. of 7 runs, 1,000,000 loops each)
```

NumPy Reference documentation

- On the web: https://numpy.org/doc/
- Interactive help:

```
In [5]: np.array?
Docstring:
```

(continues on next page)

(continued from previous page)

```
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
      like=None)

Create an array.

Parameters
----------
object : array_like
    An array, any object exposing the array interface, an object whose
    ``__array__`` method returns an array, or any (nested) sequence.
    If object is a scalar, a 0-dimensional array containing object is
    returned.
dtype : data-type, optional
    The desired data-type for the array. If not given, NumPy will try to use
    a default ``dtype`` that can represent the values (by applying promotion
    rules when necessary.)
copy : bool, optional
    If ``True`` (default), then the array data is copied. If ``None``,
    a copy will only be made if ``__array__`` returns a copy, if obj is
    a nested sequence, or if a copy is needed to satisfy any of the other
    requirements (``dtype``, ``order``, etc.). Note that any copy of
    the data is shallow, i.e., for arrays with object dtype, the new
    array will point to the same objects. See Examples for `ndarray.copy`.
    For ``False`` it raises a ``ValueError`` if a copy cannot be avoided.
    Default: ``True``.
order : {'K', 'A', 'C', 'F'}, optional
    Specify the memory layout of the array. If object is not an array, the
    newly created array will be in C order (row major) unless 'F' is
    specified, in which case it will be in Fortran order (column major).
    If object is an array the following holds.

    ===== ========= ===================================================
    order  no copy                     copy=True
    ===== ========= ===================================================
    'K'   unchanged F & C order preserved, otherwise most similar order
    'A'   unchanged F order if input is F and not C, otherwise C order
    'C'   C order   C order
    'F'   F order   F order
    ===== ========= ===================================================

    When ``copy=None`` and a copy is made for other reasons, the result is
    the same as if ``copy=True``, with some exceptions for 'A', see the
    Notes section. The default order is 'K'.
subok : bool, optional
    If True, then sub-classes will be passed-through, otherwise
    the returned array will be forced to be a base-class array (default).
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting
    array should have.  Ones will be prepended to the shape as
    needed to meet this requirement.
like : array_like, optional
    Reference object to allow the creation of arrays which are not
    NumPy arrays. If an array-like passed in as ``like`` supports
    the ``__array_function__`` protocol, the result will be defined
    by it. In this case, it ensures the creation of an array object
    compatible with that passed in via this argument.
```

(continues on next page)

(continued from previous page)

```
    .. versionadded:: 1.20.0

Returns
-------
out : ndarray
    An array object satisfying the specified requirements.

See Also
--------
empty_like : Return an empty array with shape and type of input.
ones_like : Return an array of ones with shape and type of input.
zeros_like : Return an array of zeros with shape and type of input.
full_like : Return a new array with shape of input filled with value.
empty : Return a new uninitialized array.
ones : Return a new array setting values to one.
zeros : Return a new array setting values to zero.
full : Return a new array of given shape filled with value.
copy: Return an array copy of the given object.


Notes
-----
When order is 'A' and ``object`` is an array in neither 'C' nor 'F' order,
and a copy is forced by a change in dtype, then the order of the result is
not necessarily 'C' as expected. This is likely a bug.

Examples
--------
>>> import numpy as np
>>> np.array([1, 2, 3])
array([1, 2, 3])

Upcasting:

>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])

More than one dimension:

>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])

Minimum dimensions 2:

>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])

Type provided:

>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])

Data-type consisting of more than one element:
```

(continues on next page)

(continued from previous page)

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3], dtype=int32)

Creating an array from sub-classes:

>>> np.array(np.asmatrix('1 2; 3 4'))
array([[1, 2],
       [3, 4]])

>>> np.array(np.asmatrix('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
Type:      builtin_function_or_method
```

Tip

```
>>> help(np.array)
Help on built-in function array in module numpy:

array(...)
    array(object, dtype=None, ...
```

- Looking for something:

```
In [6]: np.con*?
np.concat
np.concatenate
np.conj
np.conjugate
np.convolve
```

Import conventions

The recommended convention to import NumPy is:

```
>>> import numpy as np
```

3.1.2 Creating arrays

Manual construction of arrays

- **1-D**:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

- **2-D, 3-D, . . .**:

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)     # returns the size of the first dimension
2

>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],

       [[3],
        [4]]])
>>> c.shape
(2, 2, 1)
```

Exercise: Simple arrays

- Create a simple two dimensional array. First, redo the examples from above. And then create your own: how about odd numbers counting backwards on the first row, and even numbers on the second?
- Use the functions `len()`, `numpy.shape()` on these arrays. How do they relate to each other? And to the `ndim` attribute of the arrays?

Functions for creating arrays

Tip

In practice, we rarely enter items one by one...

- Evenly spaced:

```
>>> a = np.arange(10) # 0 .. n-1  (!)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exclusive), step
>>> b
array([1, 3, 5, 7])
```

- or by number of points:

```
>>> c = np.linspace(0, 1, 6)   # start, end, num-points
>>> c
array([0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([0. ,  0.2,  0.4,  0.6,  0.8])
```

- Common arrays:

```
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[1.,  1.,  1.],
       [1.,  1.,  1.],
       [1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[0.,  0.],
       [0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

- `np.random`: random numbers (Mersenne Twister PRNG):

```
>>> rng = np.random.default_rng(27446968)
>>> a = rng.random(4)       # uniform in [0, 1]
>>> a
array([0.64613018, 0.48984931, 0.50851229, 0.22563948])

>>> b = rng.standard_normal(4)      # Gaussian
>>> b
array([-0.38250769, -0.61536465,  0.98131732,  0.59353096])
```

Exercise: Creating arrays using functions

- Experiment with `arange`, `linspace`, `ones`, `zeros`, `eye` and `diag`.
- Create different kinds of arrays with random numbers.
- Try setting the seed before creating an array with random values.
- Look at the function `np.empty`. What does it do? When might this be useful?

3.1.3 Basic data types

You may have noticed that, in some instances, array elements are displayed with a trailing dot (e.g. `2.` vs `2`). This is due to a difference in the data-type used:

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

Tip

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```

The **default** data type is floating point:

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

There are also other types:

Complex

```
>>> d = np.array([1+2j, 3+4j, 5+6*1j])
>>> d.dtype
dtype('complex128')
```

Bool

```
>>> e = np.array([True, False, False, True])
>>> e.dtype
dtype('bool')
```

Strings

```
>>> f = np.array(['Bonjour', 'Hello', 'Hallo'])
>>> f.dtype     # <--- strings containing max. 7 letters
dtype('<U7')
```

Much more

- int32
- int64
- uint32
- uint64

3.1.4 Basic visualization

Now that we have our first data arrays, we are going to visualize them.

Start by launching IPython:

```
$ ipython # or ipython3 depending on your install
```

Or the notebook:

```
$ jupyter notebook
```

Once IPython has started, enable interactive plots:

```
>>> %matplotlib
```

Or, from the notebook, enable plots in the notebook:

```
>>> %matplotlib inline
```

The `inline` is important for the notebook, so that plots are displayed in the notebook and not in a new window.

Matplotlib is a 2D plotting package. We can import its functions as below:

```
>>> import matplotlib.pyplot as plt  # the tidy way
```

And then use (note that you have to use `show` explicitly if you have not enabled interactive plots with `%matplotlib`):

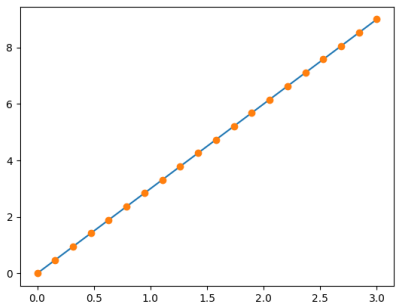
```
>>> plt.plot(x, y)       # line plot
>>> plt.show()           # <-- shows the plot (not needed with interactive plots)
```

Or, if you have enabled interactive plots with `%matplotlib`:

```
>>> plt.plot(x, y)       # line plot
```

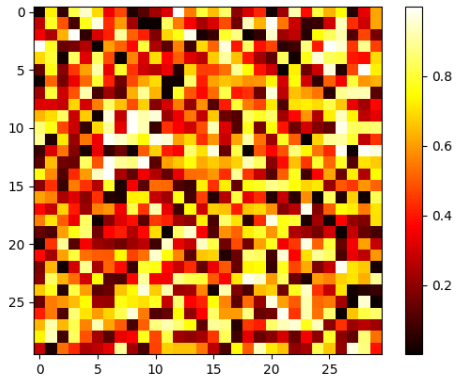
- **1D plotting**:

```
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at ...>]
```



- **2D arrays** (such as images):

```
>>> rng = np.random.default_rng(27446968)
>>> image = rng.random((30, 30))
>>> plt.imshow(image, cmap=plt.cm.hot)
<matplotlib.image.AxesImage object at ...>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar object at ...>
```



See also

More in the: *matplotlib chapter*

Exercise: Simple visualizations

- Plot some simple arrays: a cosine as a function of time and a 2D matrix.
- Try using the `gray` colormap on the 2D matrix.

3.1.5 Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(np.int64(0), np.int64(2), np.int64(9))
```

Warning

Indices begin at 0, like other Python sequences (and C/C++). In contrast, in Fortran or Matlab, indices begin at 1.

The usual python idiom for reversing a sequence is supported:

```
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

For multidimensional arrays, indices are tuples of integers:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
```

(continues on next page)

(continued from previous page)

```
np.int64(1)
>>> a[2, 1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

Note

- In 2D, the first dimension corresponds to **rows**, the second to **columns**.
- for multidimensional `a`, `a[0]` is interpreted by taking all elements in the unspecified dimensions.

Slicing: Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

Note that the last index is not included! :

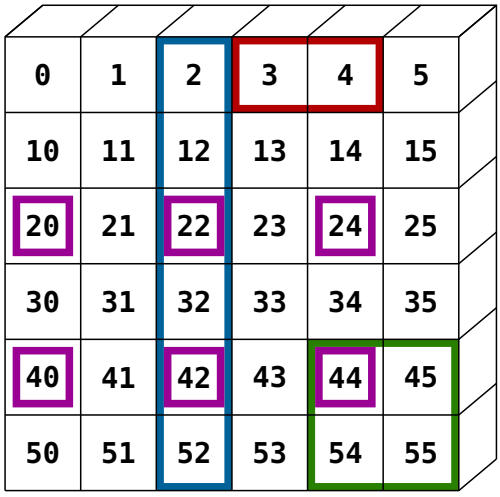
```
>>> a[:4]
array([0, 1, 2, 3])
```

All three slice components are not required: by default, *start* is 0, *end* is the last and *step* is 1:

```
>>> a[1:3]
array([1, 2])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```

A small illustrated summary of NumPy indexing and slicing...

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



You can also combine assignment and slicing:

```
>>> a = np.arange(10)
>>> a[5:] = 10
>>> a
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
>>> b = np.arange(5)
>>> a[5:] = b[::-1]
>>> a
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

Exercise: Indexing and slicing

- Try the different flavours of slicing, using `start`, `end` and `step`: starting from a linspace, try to obtain odd numbers counting backwards, and even numbers counting forwards.
- Reproduce the slices in the diagram above. You may use the following expression to create the array:

```
>>> np.arange(6) + np.arange(0, 51, 10)[:, np.newaxis]
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

Exercise: Array creation

Create the following arrays (with correct data types):

```
[[1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 2],
 [1, 6, 1, 1]]

[[0., 0., 0., 0., 0.],
 [2., 0., 0., 0., 0.],
 [0., 3., 0., 0., 0.],
 [0., 0., 4., 0., 0.],
 [0., 0., 0., 5., 0.],
 [0., 0., 0., 0., 6.]]
```

Par on course: 3 statements for each

Hint: Individual array elements can be accessed similarly to a list, e.g. `a[1]` or `a[1, 2]`.

Hint: Examine the docstring for `diag`.

Exercise: Tiling for array creation

Skim through the documentation for `np.tile`, and use this function to construct the array:

```
[[4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1],
 [4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1]]
```

3.1.6 Copies and views

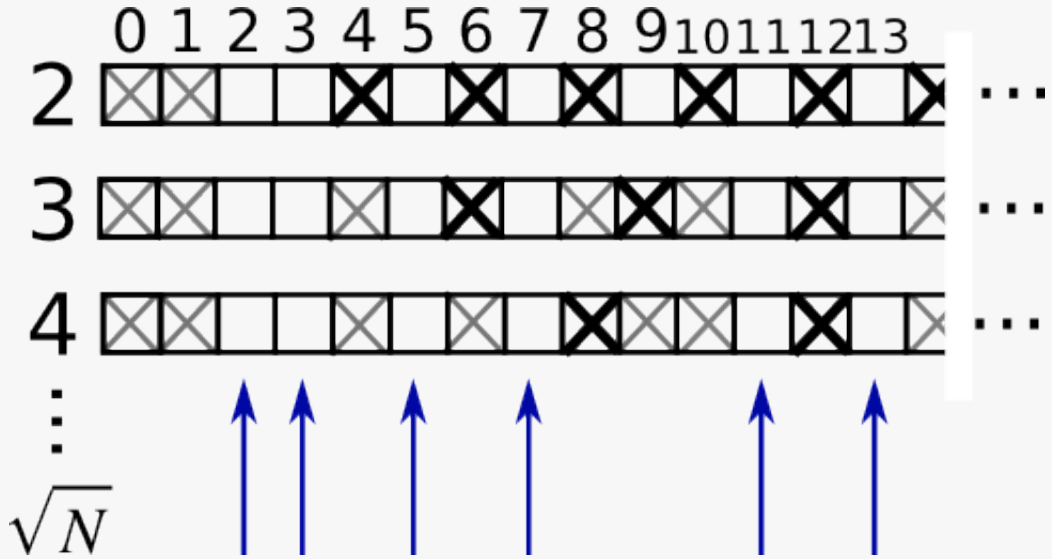
A slicing operation creates a **view** on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use `np.may_share_memory()` to check if two arrays share the same memory block. Note however, that this uses heuristics and may give you false positives.

When modifying the view, the original array is modified as well:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b
array([0, 2, 4, 6, 8])
>>> np.may_share_memory(a, b)
True
>>> b[0] = 12
>>> b
array([12,  2,  4,  6,  8])
>>> a   # (!)
array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9])

>>> a = np.arange(10)
>>> c = a[::2].copy()  # force a copy
>>> c[0] = 12
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> np.may_share_memory(a, c)
False
```

This behavior can be surprising at first sight... but it allows to save both memory and time.

Worked example: Prime number sieve



Compute prime numbers in 0–99, with a sieve

- Construct a shape (100,) boolean array `is_prime`, filled with True in the beginning:

```
>>> is_prime = np.ones((100,), dtype=bool)
```

- Cross out 0 and 1 which are not primes:

```
>>> is_prime[:2] = 0
```

- For each integer `j` starting from 2, cross out its higher multiples:

```
>>> N_max = int(np.sqrt(len(is_prime) - 1))
>>> for j in range(2, N_max + 1):
...     is_prime[2*j::j] = False
```

- Skim through `help(np.nonzero)`, and print the prime numbers
- Follow-up:
  - Move the above code into a script file named `prime_sieve.py`
  - Run it to check it works
  - Use the optimization suggested in the sieve of Eratosthenes:
  1. Skip `j` which are already known to not be primes
  2. The first number to cross out is j^2

3.1.7 Fancy indexing

Tip

NumPy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**). This method is called *fancy indexing*. It creates **copies not views**.

Using boolean masks

```
>>> rng = np.random.default_rng(27446968)
>>> a = rng.integers(0, 21, 15)
>>> a
array([ 3, 13, 12, 10, 10, 10, 18,  4,  8,  5,  6, 11, 12, 17,  3])
>>> (a % 3 == 0)
array([ True, False,  True, False, False, False,  True, False, False,
       False,  True, False,  True, False,  True])
>>> mask = (a % 3 == 0)
>>> extract_from_a = a[mask] # or,  a[a%3==0]
>>> extract_from_a           # extract a sub-array with the mask
array([ 3, 12, 18,  6, 12,  3])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

```
>>> a[a % 3 == 0] = -1
>>> a
array([-1, 13, -1, 10, 10, 10, -1,  4,  8,  5, -1, 11, -1, 17, -1])
```

Indexing with an array of integers

```
>>> a = np.arange(0, 100, 10)
>>> a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

```
>>> a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
array([20, 30, 20, 40, 20])
```

New values can be assigned with this kind of indexing:

```
>>> a[[9, 7]] = -100
>>> a
array([   0,   10,   20,   30,   40,   50,   60, -100,   80, -100])
```

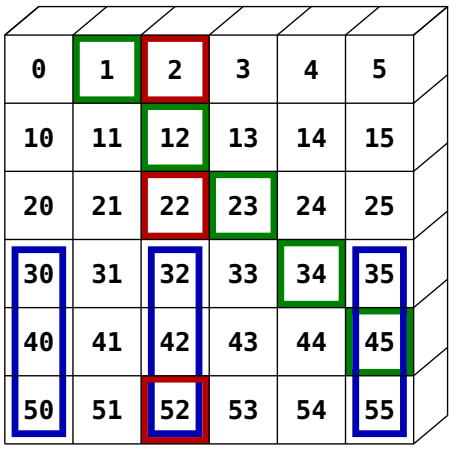
Tip

When a new array is created by indexing with an array of integers, the new array has the same shape as the array of integers:

```
>>> a = np.arange(10)
>>> idx = np.array([[3, 4], [9, 7]])
>>> idx.shape
(2, 2)
>>> a[idx]
array([[3, 4],
       [9, 7]])
```

The image below illustrates various fancy indexing applications

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



Exercise: Fancy indexing

- Again, reproduce the fancy indexing shown in the diagram above.
- Use fancy indexing on the left and array creation on the right to assign values into an array, for instance by setting parts of the array in the diagram above to zero.

3.2 Numerical operations on arrays

Section contents

- *Elementwise operations*
- *Basic reductions*
- *Broadcasting*
- *Array shape manipulation*
- *Sorting data*
- *Summary*

3.2.1 Elementwise operations

Basic operations

With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
```

All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([2.,  4.,  6.,  8.])

>>> j = np.arange(5)
>>> 2**(j + 1) - j
array([ 2,  3,  6, 13, 28])
```

These operations are of course much faster than if you did them in pure python:

```
>>> a = np.arange(10000)
>>> %timeit a + 1
10000 loops, best of 3: 24.3 us per loop
>>> l = range(10000)
>>> %timeit [i+1 for i in l]
1000 loops, best of 3: 861 us per loop
```

Warning

Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c                   # NOT matrix multiplication!
array([[1.,  1.,  1.],
       [1.,  1.,  1.],
       [1.,  1.,  1.]])
```

Note

Matrix multiplication:

```
>>> c @ c
array([[3.,  3.,  3.],
       [3.,  3.,  3.],
       [3.,  3.,  3.]])
```

Exercise: Elementwise operations

- Try simple arithmetic elementwise operations: add even elements with odd elements
- Time them against their pure python counterparts using `%timeit`.
- Generate:
  - `[2**0, 2**1, 2**2, 2**3, 2**4]`
  - `a_j = 2^(3*j) - j`

Other operations

Comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True])
>>> a > b
array([False, False,  True, False])
```

Tip

Array-wise comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> c = np.array([1, 2, 3, 4])
>>> np.array_equal(a, b)
False
>>> np.array_equal(a, c)
True
```

Logical operations:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> np.logical_or(a, b)
array([ True,  True,  True, False])
>>> np.logical_and(a, b)
array([ True, False, False, False])
```

Transcendental functions:

```
>>> a = np.arange(5)
>>> np.sin(a)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
>>> np.exp(a)
```

(continues on next page)

(continued from previous page)

```
array([ 1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
>>> np.log(np.exp(a))
array([0., 1., 2., 3., 4.])
```

Shape mismatches

```
>>> a = np.arange(4)
>>> a + np.array([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

Broadcasting? We'll return to that *later*.

Transposition:

```
>>> a = np.triu(np.ones((3, 3)), 1)   # see help(np.triu)
>>> a
array([[0.,  1.,  1.],
       [0.,  0.,  1.],
       [0.,  0.,  0.]])
>>> a.T
array([[0.,  0.,  0.],
       [1.,  0.,  0.],
       [1.,  1.,  0.]])
```

Note

The transposition is a view

The transpose returns a *view* of the original array:

```
>>> a = np.arange(9).reshape(3, 3)
>>> a.T[0, 2] = 999
>>> a.T
array([[  0,   3, 999],
       [  1,   4,   7],
       [  2,   5,   8]])
>>> a
array([[  0,   1,   2],
       [  3,   4,   5],
       [999,   7,   8]])
```

Note

Linear algebra

The sub-module `numpy.linalg` implements basic linear algebra, such as solving linear systems, singular value decomposition, etc. However, it is not guaranteed to be compiled using efficient routines, and thus we recommend the use of `scipy.linalg`, as detailed in section *Linear algebra operations: scipy.linalg*

Exercise other operations

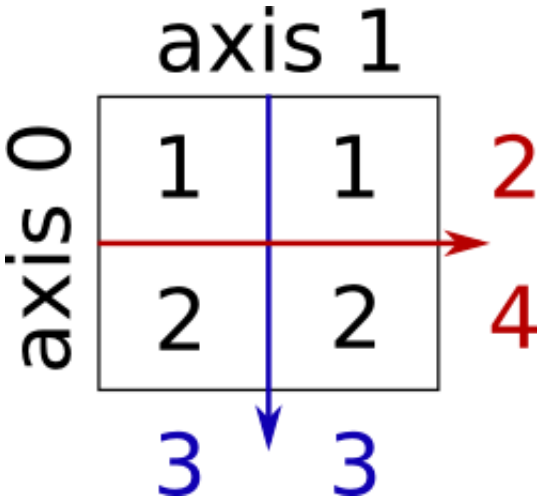
- Look at the help for `np.allclose`. When might this be useful?

- Look at the help for `np.triu` and `np.tril`.

3.2.2 Basic reductions

Computing sums

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
np.int64(10)
>>> x.sum()
np.int64(10)
```



Sum by rows and by columns:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)   # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(np.int64(3), np.int64(3))
>>> x.sum(axis=1)   # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(np.int64(2), np.int64(4))
```

Tip

Same idea in higher dimensions:

```
>>> rng = np.random.default_rng(27446968)
>>> x = rng.random((2, 2, 2))
>>> x.sum(axis=2)[0, 1]
np.float64(0.73415...)
>>> x[0, 1, :].sum()
np.float64(0.73415...)
```

Other reductions

— works the same way (and take `axis=`)

Extrema:

```
>>> x = np.array([1, 3, 2])
>>> x.min()
np.int64(1)
>>> x.max()
np.int64(3)

>>> x.argmin()  # index of minimum
np.int64(0)
>>> x.argmax()  # index of maximum
np.int64(1)
```

Logical operations:

```
>>> np.all([True, True, False])
np.False_
>>> np.any([True, True, False])
np.True_
```

Note

Can be used for array comparisons:

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
np.False_
>>> np.all(a == a)
np.True_

>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
np.True_
```

Statistics:

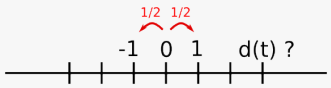
```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
np.float64(1.75)
>>> np.median(x)
np.float64(1.5)
>>> np.median(y, axis=-1) # last axis
array([2.,  5.])

>>> x.std()          # full population standard dev.
np.float64(0.82915619758884995)
```

... and many more (best to learn as you go).

Exercise: Reductions

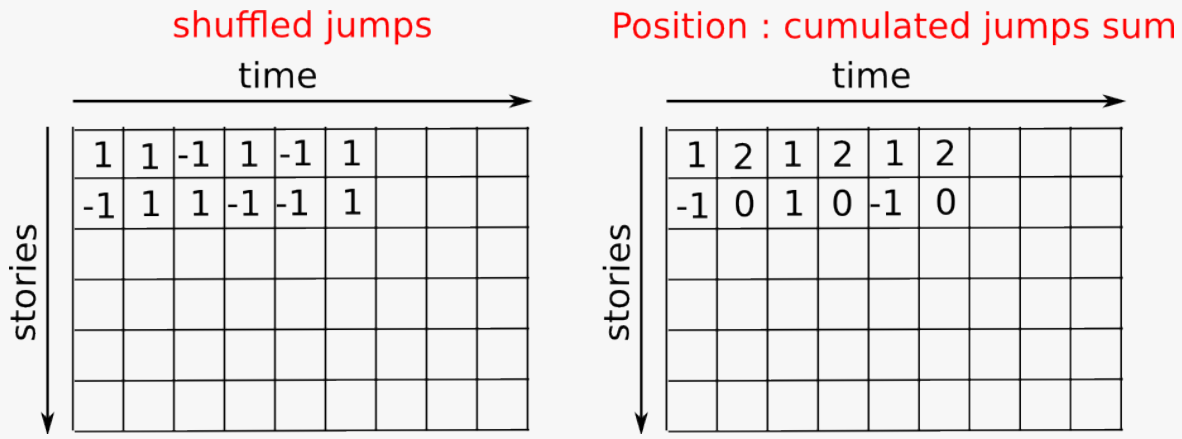
- Given there is a `sum`, what other function might you expect to see?
- What is the difference between `sum` and `cumsum`?

Worked Example: diffusion using a random walk algorithm



Tip

Let us consider a simple 1D random walk process: at each time step a walker jumps right or left with equal probability.
We are interested in finding the typical distance from the origin of a random walker after `t` left or right jumps? We are going to simulate many "walkers" to find this law, and we are going to do so using array computing tricks: we are going to create a 2D array with the "stories" (each walker has a story) in one direction, and the time in the other:




```
>>> n_stories = 1000 # number of walkers
>>> t_max = 200      # time during which we follow the walker
```

We randomly choose all the steps 1 or -1 of the walk:

```
>>> t = np.arange(t_max)
>>> rng = np.random.default_rng()
>>> steps = 2 * rng.integers(0, 1 + 1, (n_stories, t_max)) - 1 # +1 because the␣
↪high value is exclusive
>>> np.unique(steps) # Verification: all steps are 1 or -1
array([-1,  1])
```

We build the walks by summing steps along the time:

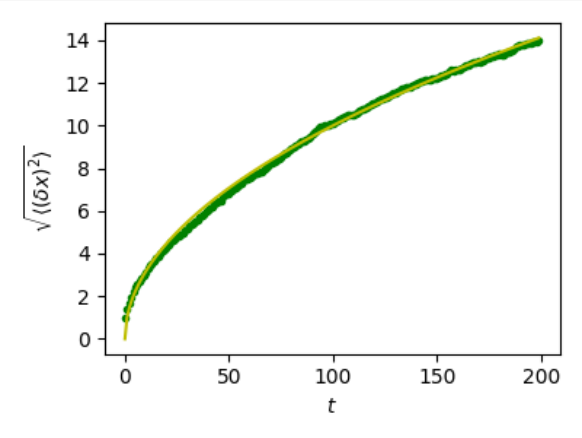
```
>>> positions = np.cumsum(steps, axis=1) # axis = 1: dimension of time
>>> sq_distance = positions**2
```

We get the mean in the axis of the stories:

```
>>> mean_sq_distance = np.mean(sq_distance, axis=0)
```

Plot the results:

```
>>> plt.figure(figsize=(4, 3))
<Figure size ... with 0 Axes>
>>> plt.plot(t, np.sqrt(mean_sq_distance), 'g.', t, np.sqrt(t), 'y-')
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
>>> plt.xlabel(r"$t$")
Text(...'$t$')
>>> plt.ylabel(r"$\sqrt{\langle (\delta x)^2 \rangle}$")
Text(...'$\\sqrt{\\langle (\\delta x)^2 \\rangle}$')
>>> plt.tight_layout() # provide sufficient space for labels
```



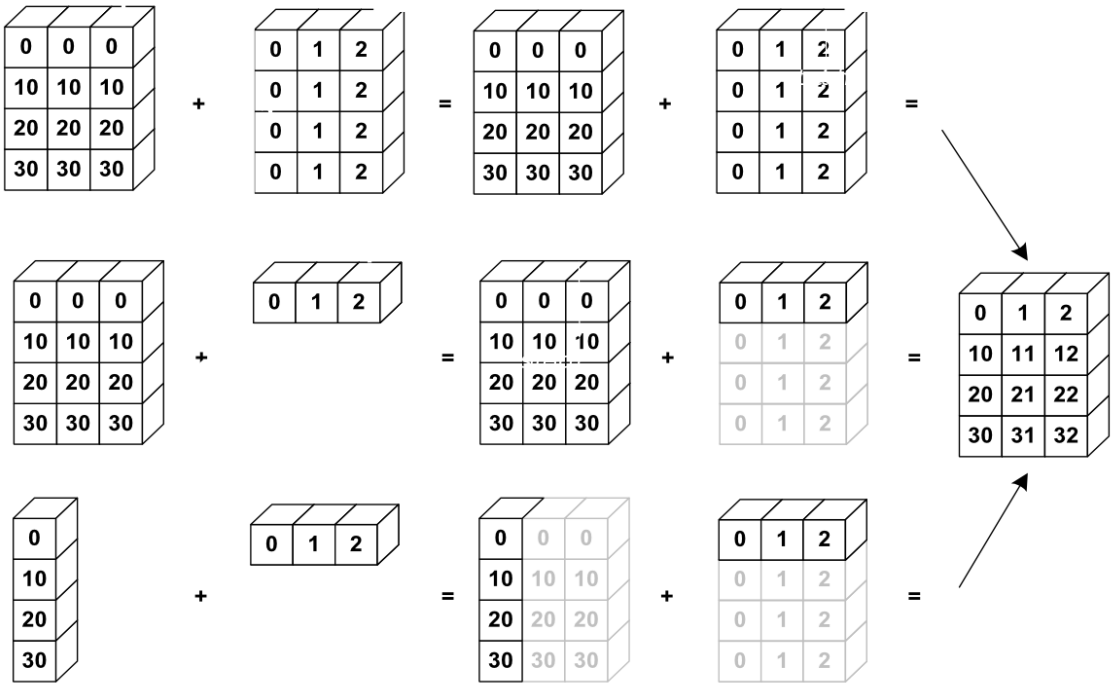
We find a well-known result in physics: the RMS distance grows as the square root of the time!

3.2.3 Broadcasting

- Basic operations on `numpy` arrays (addition, etc.) are elementwise
- This works on arrays of the same size.

  Nevertheless, It's also possible to do operations on arrays of different sizes if *NumPy* can transform these arrays so that they all have the same size: this conversion is called **broadcasting**.

The image below gives an example of broadcasting:



Let's verify:

```
>>> a = np.tile(np.arange(0, 40, 10), (3, 1)).T
>>> a
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
>>> b = np.array([0, 1, 2])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

We have already used broadcasting without knowing it!:

```
>>> a = np.ones((4, 5))
>>> a[0] = 2  # we assign an array of dimension 0 to an array of dimension 1
>>> a
array([[2.,  2.,  2.,  2.,  2.],
       [1.,  1.,  1.,  1.,  1.],
       [1.,  1.,  1.,  1.,  1.],
       [1.,  1.,  1.,  1.,  1.]])
```

A useful trick:

```
>>> a = np.arange(0, 40, 10)
>>> a.shape
(4,)
```

(continues on next page)

(continued from previous page)

```
>>> a = a[:, np.newaxis]  # adds a new axis -> 2D array
>>> a.shape
(4, 1)
>>> a
array([[ 0],
       [10],
       [20],
       [30]])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

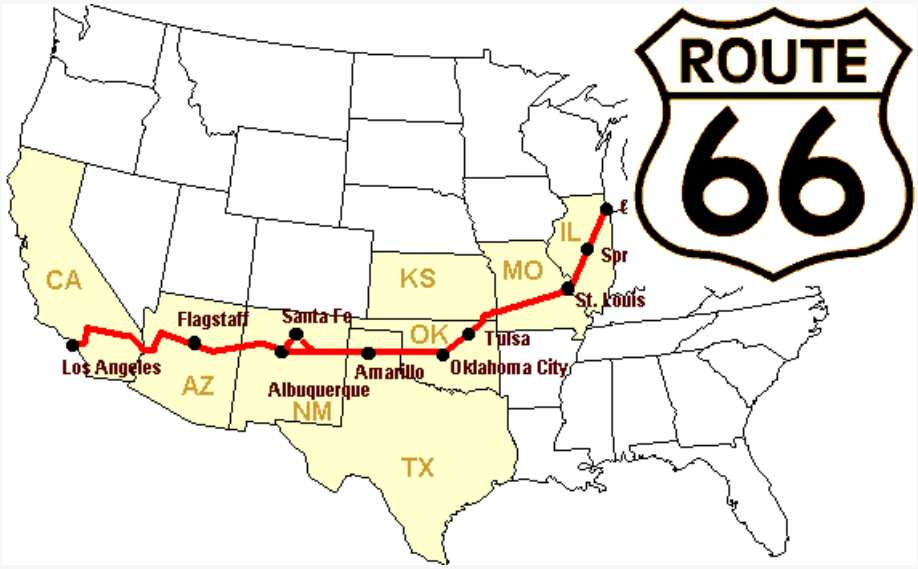
Tip

Broadcasting seems a bit magical, but it is actually quite natural to use it when we want to solve a problem whose output data is an array with more dimensions than input data.

Worked Example: Broadcasting

Let's construct an array of distances (in miles) between cities of Route 66: Chicago, Springfield, Saint-Louis, Tulsa, Oklahoma City, Amarillo, Santa Fe, Albuquerque, Flagstaff and Los Angeles.

```
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544,
...        1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
array([[   0,  198,  303,  736,  871, 1175, 1475, 1544, 1913, 2448],
       [ 198,    0,  105,  538,  673,  977, 1277, 1346, 1715, 2250],
       [ 303,  105,    0,  433,  568,  872, 1172, 1241, 1610, 2145],
       [ 736,  538,  433,    0,  135,  439,  739,  808, 1177, 1712],
       [ 871,  673,  568,  135,    0,  304,  604,  673, 1042, 1577],
       [1175,  977,  872,  439,  304,    0,  300,  369,  738, 1273],
       [1475, 1277, 1172,  739,  604,  300,    0,   69,  438,  973],
       [1544, 1346, 1241,  808,  673,  369,   69,    0,  369,  904],
       [1913, 1715, 1610, 1177, 1042,  738,  438,  369,    0,  535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973,  904,  535,    0]])
```

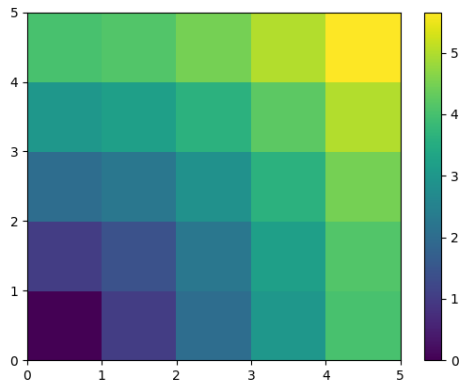


A lot of grid-based or network-based problems can also use broadcasting. For instance, if we want to compute the distance from the origin of points on a 5x5 grid, we can do

```
>>> x, y = np.arange(5), np.arange(5)[:, np.newaxis]
>>> distance = np.sqrt(x ** 2 + y ** 2)
>>> distance
array([[0.        , 1.        , 2.        , 3.        , 4.        ],
       [1.        , 1.41421356, 2.23606798, 3.16227766, 4.12310563],
       [2.        , 2.23606798, 2.82842712, 3.60555128, 4.47213595],
       [3.        , 3.16227766, 3.60555128, 4.24264069, 5.        ],
       [4.        , 4.12310563, 4.47213595, 5.        , 5.65685425]])
```

Or in color:

```
>>> plt.pcolor(distance)
<matplotlib.collections.PolyQuadMesh object at ...>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar object at ...>
```



Remark : the `numpy.ogrid()` function allows to directly create vectors x and y of the previous example, with two "significant dimensions":

```
>>> x, y = np.ogrid[0:5, 0:5]
>>> x, y
```

(continues on next page)

(continued from previous page)

```
(array([[0],
       [1],
       [2],
       [3],
       [4]]), array([[0, 1, 2, 3, 4]]))
>>> x.shape, y.shape
((5, 1), (1, 5))
>>> distance = np.sqrt(x ** 2 + y ** 2)
```

Tip

So, `np.ogrid` is very useful as soon as we have to handle computations on a grid. On the other hand, `np.mgrid` directly provides matrices full of indices for cases where we can't (or don't want to) benefit from broadcasting:

```
>>> x, y = np.mgrid[0:4, 0:4]
>>> x
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])
>>> y
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])
```

See also

Broadcasting: discussion of broadcasting in the *Advanced NumPy* chapter.

3.2.4 Array shape manipulation

Flattening

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

Higher dimensions: last dimensions ravel out "first".

Reshaping

The inverse operation to flattening:

```
>>> a.shape
(2, 3)
>>> b = a.ravel()
```

(continues on next page)

(continued from previous page)

```
>>> b = b.reshape((2, 3))
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
```

Or,

```
>>> a.reshape((2, -1))     # unspecified (-1) value is inferred
array([[1, 2, 3],
       [4, 5, 6]])
```

Warning

`ndarray.reshape` **may** return a view (cf `help(np.reshape)`), or copy

Tip

```
>>> b[0, 0] = 99
>>> a
array([[99,  2,  3],
       [ 4,  5,  6]])
```

Beware: reshape may also return a copy!:

```
>>> a = np.zeros((3, 2))
>>> b = a.T.reshape(3*2)
>>> b[0] = 9
>>> a
array([[0.,  0.],
       [0.,  0.],
       [0.,  0.]])
```

To understand this you need to learn more about the memory layout of a NumPy array.

Adding a dimension

Indexing with the `np.newaxis` object allows us to add an axis to an array (you have seen this already above in the broadcasting section):

```
>>> z = np.array([1, 2, 3])
>>> z
array([1, 2, 3])

>>> z[:, np.newaxis]
array([[1],
       [2],
       [3]])

>>> z[np.newaxis, :]
array([[1, 2, 3]])
```

Dimension shuffling

```
>>> a = np.arange(4*3*2).reshape(4, 3, 2)
>>> a.shape
(4, 3, 2)
>>> a[0, 2, 1]
np.int64(5)
>>> b = a.transpose(1, 2, 0)
>>> b.shape
(3, 2, 4)
>>> b[2, 1, 0]
np.int64(5)
```

Also creates a view:

```
>>> b[2, 1, 0] = -1
>>> a[0, 2, 1]
np.int64(-1)
```

Resizing

Size of an array can be changed with `ndarray.resize`:

```
>>> a = np.arange(4)
>>> a.resize((8,))
>>> a
array([0, 1, 2, 3, 0, 0, 0, 0])
```

However, it must not be referred to somewhere else:

```
>>> b = a
>>> a.resize((4,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

Exercise: Shape manipulations

- Look at the docstring for `reshape`, especially the notes section which has some more information about copies and views.
- Use `flatten` as an alternative to `ravel`. What is the difference? (Hint: check which one returns a view and which a copy)
- Experiment with `transpose` for dimension shuffling.

3.2.5 Sorting data

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

Note

Sorts each row separately!

In-place sort:

```
>>> a.sort(axis=1)
>>> a
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

Finding minima and maxima:

```
>>> a = np.array([4, 3, 1, 2])
>>> j_max = np.argmax(a)
>>> j_min = np.argmin(a)
>>> j_max, j_min
(np.int64(0), np.int64(2))
```

Exercise: Sorting

- Try both in-place and out-of-place sorting.
- Try creating arrays with different dtypes and sorting them.
- Use `all` or `array_equal` to check the results.
- Look at `np.random.shuffle` for a way to create sortable input quicker.
- Combine `ravel`, `sort` and `reshape`.
- Look at the `axis` keyword for `sort` and rewrite the previous exercise.

3.2.6 Summary

What do you need to know to get started?

- Know how to create arrays : `array`, `arange`, `ones`, `zeros`.
- Know the shape of the array with `array.shape`, then use slicing to obtain different views of the array: `array[::2]`, etc. Adjust the shape of the array using `reshape` or flatten it with `ravel`.
- Obtain a subset of the elements of an array and/or modify their values with masks

  ```
  >>> a[a < 0] = 0
  ```

- Know miscellaneous operations on arrays, such as finding the mean or max (`array.max()`, `array.mean()`). No need to retain everything, but have the reflex to search in the documentation (online docs, `help()`)!!
- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more NumPy functions to handle various array operations.

Quick read

If you want to do a first quick pass through the Scientific Python Lectures to learn the ecosystem, you can directly skip to the next chapter: *Matplotlib: plotting*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter, as well as to do some more *exercises*.

3.3 More elaborate arrays

Section contents

- *More data types*
- *Structured data types*
- *`maskedarray`: dealing with (propagation of) missing data*

3.3.1 More data types

Casting

"Bigger" type wins in mixed-type operations:

```
>>> np.array([1, 2, 3]) + 1.5
array([2.5,  3.5,  4.5])
```

Assignment never changes the type!

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a[0] = 1.9     # <-- float is truncated to integer
>>> a
array([1, 2, 3])
```

Forced casts:

```
>>> a = np.array([1.7, 1.2, 1.6])
>>> b = a.astype(int)  # <-- truncates to integer
>>> b
array([1, 1, 1])
```

Rounding:

```
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
>>> b = np.around(a)
>>> b                    # still floating-point
array([1.,  2.,  2.,  2.,  4.,  4.])
>>> c = np.around(a).astype(int)
>>> c
array([1, 2, 2, 2, 4, 4])
```

Different data type sizes

Integers (signed):

`int8`	8 bits
`int16`	16 bits
`int32`	32 bits (same as `int` on 32-bit platform)
`int64`	64 bits (same as `int` on 64-bit platform)

```
>>> np.array([1], dtype=int).dtype
dtype('int64')
>>> np.iinfo(np.int32).max, 2**31 - 1
(2147483647, 2147483647)
```

Unsigned integers:

`uint8`	8 bits
`uint16`	16 bits
`uint32`	32 bits
`uint64`	64 bits

```
>>> np.iinfo(np.uint32).max, 2**32 - 1
(4294967295, 4294967295)
```

Floating-point numbers:

`float16`	16 bits
`float32`	32 bits
`float64`	64 bits (same as `float`)
`float96`	96 bits, platform-dependent (same as `np.longdouble`)
`float128`	128 bits, platform-dependent (same as `np.longdouble`)

```
>>> np.finfo(np.float32).eps
np.float32(1.1920929e-07)
>>> np.finfo(np.float64).eps
np.float64(2.220446049250313e-16)

>>> np.float32(1e-8) + np.float32(1) == 1
np.True_
>>> np.float64(1e-8) + np.float64(1) == 1
np.False_
```

Complex floating-point numbers:

`complex64`	two 32-bit floats
`complex128`	two 64-bit floats
`complex192`	two 96-bit floats, platform-dependent
`complex256`	two 128-bit floats, platform-dependent

Smaller data types

If you don't know you need special data types, then you probably don't.

Comparison on using `float32` instead of `float64`:

- Half the size in memory and on disk
- Half the memory bandwidth required (may be a bit faster in some operations)

```
In [1]: a = np.zeros((int(1e6),), dtype=np.float64)

In [2]: b = np.zeros((int(1e6),), dtype=np.float32)

In [3]: %timeit a*a
275 us +- 4.48 us per loop (mean +- std. dev. of 7 runs, 1,000 loops each)

In [4]: %timeit b*b
139 us +- 2.02 us per loop (mean +- std. dev. of 7 runs, 10,000 loops each)
```

- But: bigger rounding errors — sometimes in surprising places (i.e., don't use them unless you really need them)

3.3.2 Structured data types

`sensor_code`	(4-character string)
`position`	(float)
`value`	(float)

```
>>> samples = np.zeros((6,), dtype=[('sensor_code', 'S4'),
...                                 ('position', float), ('value', float)])
>>> samples.ndim
1
>>> samples.shape
(6,)
>>> samples.dtype.names
('sensor_code', 'position', 'value')
>>> samples[:] = [('ALFA',   1, 0.37), ('BETA', 1, 0.11), ('TAU', 1,   0.13),
...               ('ALFA', 1.5, 0.37), ('ALFA', 3, 0.11), ('TAU', 1.2, 0.13)]
>>> samples
array([(b'ALFA', 1. , 0.37), (b'BETA', 1. , 0.11), (b'TAU', 1. , 0.13),
       (b'ALFA', 1.5, 0.37), (b'ALFA', 3. , 0.11), (b'TAU', 1.2, 0.13)],
      dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')])
```

Field access works by indexing with field names:

```
>>> samples['sensor_code']
array([b'ALFA', b'BETA', b'TAU', b'ALFA', b'ALFA', b'TAU'], dtype='|S4')
>>> samples['value']
array([0.37,  0.11,  0.13,  0.37,  0.11,  0.13])
>>> samples[0]
np.void((b'ALFA', 1.0, 0.37), dtype=[('sensor_code', 'S4'), ('position', '<f8'), (
↪'value', '<f8')])

>>> samples[0]['sensor_code'] = 'TAU'
>>> samples[0]
np.void((b'TAU', 1.0, 0.37), dtype=[('sensor_code', 'S4'), ('position', '<f8'), (
↪'value', '<f8')])
```

Multiple fields at once:

```
>>> samples[['position', 'value']]
array([(1. ,  0.37), (1. ,  0.11), (1. ,  0.13), (1.5,  0.37),
```

(continues on next page)

(continued from previous page)

```
       (3. ,  0.11), (1.2,  0.13)],
      dtype={'names': ['position', 'value'], 'formats': ['<f8', '<f8'], 'offsets': [4,
↪ 12], 'itemsize': 20})
```

Fancy indexing works, as usual:

```
>>> samples[samples['sensor_code'] == b'ALFA']
array([(b'ALFA', 1.5, 0.37), (b'ALFA', 3. , 0.11)],
      dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')])
```

Note

There are a bunch of other syntaxes for constructing structured arrays, see here and here.

3.3.3 `maskedarray`: dealing with (propagation of) missing data

- For floats one could use NaN's, but masks work for all types:

```
>>> x = np.ma.array([1, 2, 3, 4], mask=[0, 1, 0, 1])
>>> x
masked_array(data=[1, --, 3, --],
             mask=[False,  True, False,  True],
       fill_value=999999)


>>> y = np.ma.array([1, 2, 3, 4], mask=[0, 1, 1, 1])
>>> x + y
masked_array(data=[2, --, --, --],
             mask=[False,  True,  True,  True],
       fill_value=999999)
```

- Masking versions of common functions:

```
>>> np.ma.sqrt([1, -1, 2, -2])
masked_array(data=[1.0, --, 1.41421356237... --],
             mask=[False,  True, False,  True],
       fill_value=1e+20)
```

Note

There are other useful *array siblings*

While it is off topic in a chapter on NumPy, let's take a moment to recall good coding practice, which really do pay off in the long run:

Good practices

- Explicit variable names (no need of a comment to explain what is in the variable)
- Style: spaces after commas, around =, etc.

  A certain number of rules for writing "beautiful" code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).

- Except some rare cases, variable names and comments in English.

3.4 Advanced operations

Section contents

- *Polynomials*
- *Loading data files*

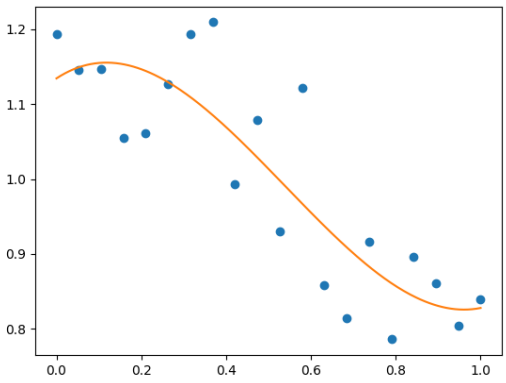
3.4.1 Polynomials

NumPy also contains polynomials in different bases:

For example, $3x^2 + 2x - 1$:

```
>>> p = np.poly1d([3, 2, -1])
>>> p(0)
np.int64(-1)
>>> p.roots
array([-1.        ,  0.33333333])
>>> p.order
2
```

```
>>> x = np.linspace(0, 1, 20)
>>> rng = np.random.default_rng()
>>> y = np.cos(x) + 0.3*rng.random(20)
>>> p = np.poly1d(np.polyfit(x, y, 3))

>>> t = np.linspace(0, 1, 200) # use a larger number of points for smoother plotting
>>> plt.plot(x, y, 'o', t, p(t), '-')
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
```



See https://numpy.org/doc/stable/reference/routines.polynomials.poly1d.html for more.

More polynomials (with more bases)

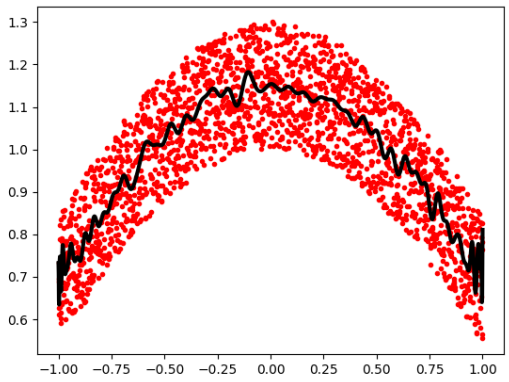
NumPy also has a more sophisticated polynomial interface, which supports e.g. the Chebyshev basis.

$3x^2 + 2x - 1$:

```
>>> p = np.polynomial.Polynomial([-1, 2, 3]) # coefs in different order!
>>> p(0)
np.float64(-1.0)
>>> p.roots()
array([-1.        ,  0.33333333])
>>> p.degree()  # In general polynomials do not always expose 'order'
2
```

Example using polynomials in Chebyshev basis, for polynomials in range `[-1, 1]`:

```
>>> x = np.linspace(-1, 1, 2000)
>>> rng = np.random.default_rng()
>>> y = np.cos(x) + 0.3*rng.random(2000)
>>> p = np.polynomial.Chebyshev.fit(x, y, 90)

>>> plt.plot(x, y, 'r.')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(x, p(x), 'k-', lw=3)
[<matplotlib.lines.Line2D object at ...>]
```



The Chebyshev polynomials have some advantages in interpolation.

3.4.2 Loading data files

Text files

Example: `populations.txt`:

```
# year  hare    lynx    carrot
1900    30e3    4e3     48300
1901    47.2e3  6.1e3   48200
1902    70.2e3  9.8e3   41500
1903    77.4e3  35.2e3  38200
```

```
>>> data = np.loadtxt('data/populations.txt')
>>> data
array([[  1900.,  30000.,   4000.,  48300.],
       [  1901.,  47200.,   6100.,  48200.],
```

(continues on next page)

(continued from previous page)

```
       [  1902.,  70200.,   9800.,  41500.],
...
```

```
>>> np.savetxt('pop2.txt', data)
>>> data2 = np.loadtxt('pop2.txt')
```

Note

If you have a complicated text file, what you can try are:

- `np.genfromtxt`
- Using Python's I/O functions and e.g. regexps for parsing (Python is quite well suited for this)

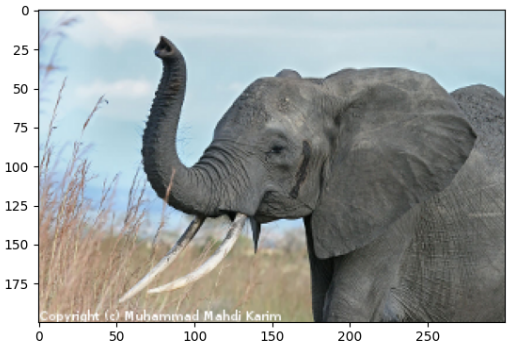
Reminder: Navigating the filesystem with IPython

```
In [1]: pwd       # show current directory
Out[1]: '/tmp'
```

Images

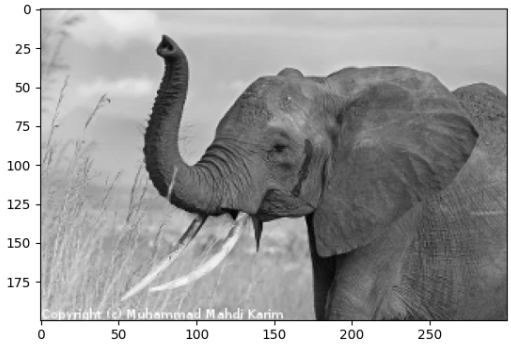
Using Matplotlib:

```
>>> img = plt.imread('data/elephant.png')
>>> img.shape, img.dtype
((200, 300, 3), dtype('float32'))
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at ...>
>>> plt.savefig('plot.png')

>>> plt.imsave('red_elephant.png', img[:,:,0], cmap=plt.cm.gray)
```



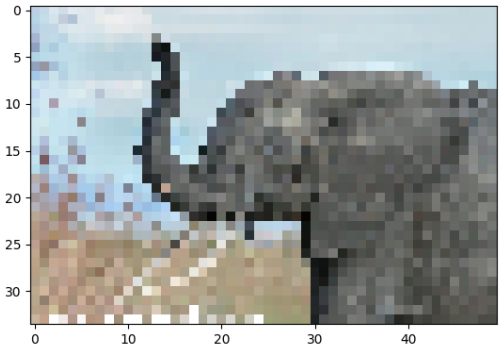
This saved only one channel (of RGB):

```
>>> plt.imshow(plt.imread('red_elephant.png'))
<matplotlib.image.AxesImage object at ...>
```



Other libraries:

```
>>> import imageio.v3 as iio
>>> iio.imwrite('tiny_elephant.png', (img[::6,::6] * 255).astype(np.uint8))
>>> plt.imshow(plt.imread('tiny_elephant.png'), interpolation='nearest')
<matplotlib.image.AxesImage object at ...>
```



NumPy's own format

NumPy has its own binary format, not portable but with efficient I/O:

```
>>> data = np.ones((3, 3))
>>> np.save('pop.npy', data)
>>> data3 = np.load('pop.npy')
```

Well-known (& more obscure) file formats

- HDF5: h5py, PyTables
- NetCDF: `scipy.io.netcdf_file`, netcdf4-python, ...
- Matlab: `scipy.io.loadmat`, `scipy.io.savemat`
- MatrixMarket: `scipy.io.mmread`, `scipy.io.mmwrite`
- IDL: `scipy.io.readsav`

... if somebody uses it, there's probably also a Python library for it.

Exercise: Text data files

Write a Python script that loads data from `populations.txt`:: and drop the last column and the first 5 rows. Save the smaller dataset to `pop2.txt`.

NumPy internals

If you are interested in the NumPy internals, there is a good discussion in *Advanced NumPy*.

3.5 Some exercises

3.5.1 Array manipulations

1. Form the 2-D array (without typing it in explicitly):

```
[[1,  6, 11],
 [2,  7, 12],
 [3,  8, 13],
 [4,  9, 14],
 [5, 10, 15]]
```

   and generate a new array containing its 2nd and 4th rows.

2. Divide each column of the array:

```
>>> import numpy as np
>>> a = np.arange(25).reshape(5, 5)
```

   elementwise with the array `b = np.array([1., 5, 10, 15, 20])`. (Hint: `np.newaxis`).

3. Harder one: Generate a 10 x 3 array of random numbers (in range [0,1]). For each row, pick the number closest to 0.5.

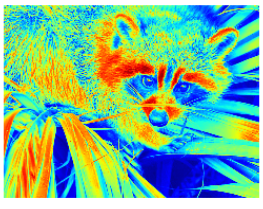
   - Use `abs` and `argmin` to find the column `j` closest for each row.
   - Use fancy indexing to extract the numbers. (Hint: `a[i,j]` – the array `i` must contain the row numbers corresponding to stuff in `j`.)

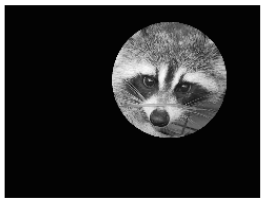
3.5.2 Picture manipulation: Framing a Face

Let's do some manipulations on NumPy arrays by starting with an image of a raccoon. `scipy` provides a 2D array of this image with the `scipy.datasets.face` function:

```
>>> import scipy as sp
>>> face = sp.datasets.face(gray=True)  # 2D grayscale image
```

Here are a few images we will be able to obtain with our manipulations: use different colormaps, crop the image, change some parts of the image.





- Let's use the imshow function of matplotlib to display the image.

```
>>> import matplotlib.pyplot as plt
>>> face = sp.datasets.face(gray=True)
>>> plt.imshow(face)
<matplotlib.image.AxesImage object at 0x...>
```

- **The face is displayed in false colors. A colormap must be** specified for it to be displayed in grey.

```
>>> plt.imshow(face, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>
```

- **Create an array of the image with a narrower centering** [for example,] remove 100 pixels from all the borders of the image. To check the result, display this new array with `imshow`.

```
>>> crop_face = face[100:-100, 100:-100]
```

- **We will now frame the face with a black locket. For this, we** need to create a mask corresponding to the pixels we want to be black. The center of the face is around (660, 330), so we defined the mask by this condition `(y-300)**2 + (x-660)**2`

```
>>> sy, sx = face.shape
>>> y, x = np.ogrid[0:sy, 0:sx] # x and y indices of pixels
>>> y.shape, x.shape
((768, 1), (1, 1024))
>>> centerx, centery = (660, 300) # center of the image
>>> mask = ((y - centery)**2 + (x - centerx)**2) > 230**2 # circle
```

then we assign the value 0 to the pixels of the image corresponding to the mask. The syntax is extremely simple and intuitive:

```
>>> face[mask] = 0
>>> plt.imshow(face)
<matplotlib.image.AxesImage object at 0x...>
```

- **Follow-up: copy all instructions of this exercise in a script called** `face_locket.py` then execute this script in IPython with `%run face_locket.py`.

  Change the circle to an ellipsoid.

3.5.3 Data statistics

The data in `populations.txt` describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables

>>> import matplotlib.pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
<Axes: >
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at ...>, ...]
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at ...>
```

Computes and print, based on the data in `populations.txt`...

1. The mean and std of the populations of each species for the years in the period.
2. Which year each species had the largest population.
3. Which species has the largest population for each year. (Hint: `argsort` & fancy indexing of `np.array(['H', 'L', 'C'])`)
4. Which years any of the populations is above 50000. (Hint: comparisons and `np.any`)
5. The top 2 years for each species when they had the lowest populations. (Hint: `argsort`, fancy indexing)
6. Compare (plot) the change in hare population (see `help(np.gradient)`) and the number of lynxes. Check correlation (see `help(np.corrcoef)`).

... all without for-loops.

Solution: `Python source file`

3.5.4 Crude integral approximations

Write a function `f(a, b, c)` that returns $a^b - c$. Form a 24x12x6 array containing its values in parameter ranges `[0,1] x [0,1] x [0,1]`.

Approximate the 3-d integral

$$\int_0^1 \int_0^1 \int_0^1 (a^b - c) da\, db\, dc$$

over this volume with the mean. The exact result is: $\ln 2 - \frac{1}{2} \approx 0.1931\ldots$ — what is your relative error?

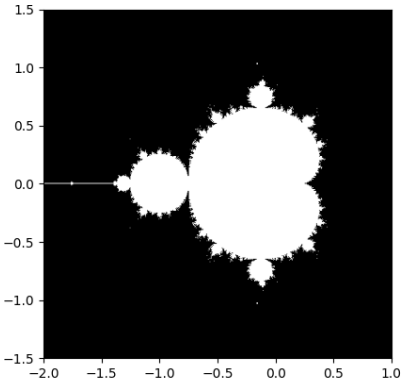
(Hints: use elementwise operations and broadcasting. You can make `np.ogrid` give a number of points in given range with `np.ogrid[0:1:20j]`.)

Reminder Python functions:

```
def f(a, b, c):
    return some_result
```

Solution: `Python source file`

3.5.5 Mandelbrot set



Write a script that computes the Mandelbrot fractal. The Mandelbrot iteration:

```
N_max = 50
some_threshold = 50

c = x + 1j*y

z = 0
for j in range(N_max):
    z = z**2 + c
```

Point (x, y) belongs to the Mandelbrot set if $|z| <$ `some_threshold`.

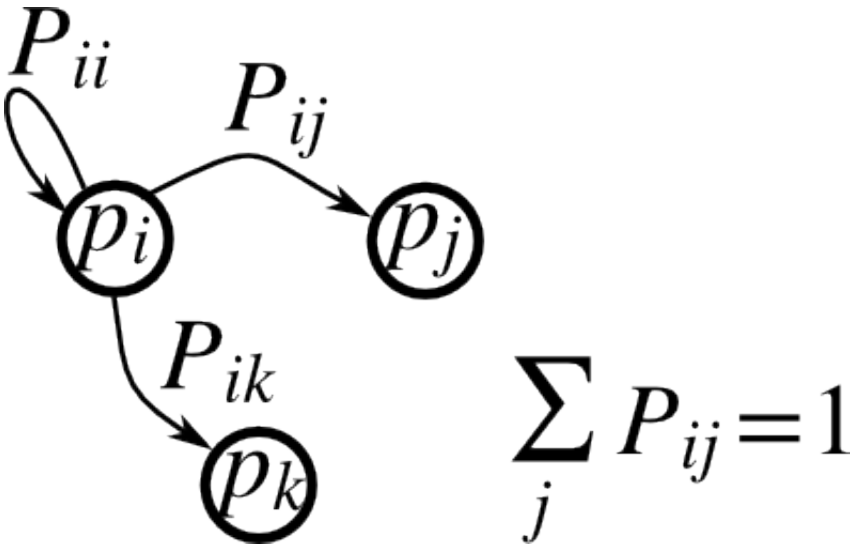
Do this computation by:

1. Construct a grid of c = x + 1j*y values in range [-2, 1] x [-1.5, 1.5]
2. Do the iteration
3. Form the 2-d boolean mask indicating which points are in the set
4. Save the result to an image with:

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(mask.T, extent=[-2, 1, -1.5, 1.5])
<matplotlib.image.AxesImage object at ...>
>>> plt.gray()
>>> plt.savefig('mandelbrot.png')
```

Solution: `Python source file`

3.5.6 Markov chain



Markov chain transition matrix `P`, and probability distribution on the states `p`:

1. `0 <= P[i,j] <= 1`: probability to go from state `i` to state `j`
2. Transition rule: $p_{new} = P^T p_{old}$
3. `all(sum(P, axis=1) == 1)`, `p.sum() == 1`: normalization

Write a script that works with 5 states, and:

- Constructs a random matrix, and normalizes each row so that it is a transition matrix.
- Starts from a random (normalized) probability distribution `p` and takes 50 steps => `p_50`
- Computes the stationary distribution: the eigenvector of `P.T` with eigenvalue 1 (numerically: closest to 1) => `p_stationary`

Remember to normalize the eigenvector — I didn't...

- Checks if `p_50` and `p_stationary` are equal to tolerance 1e-5

Toolbox: `np.random`, `@`, `np.linalg.eig`, reductions, `abs()`, `argmin`, comparisons, `all`, `np.linalg.norm`, etc.

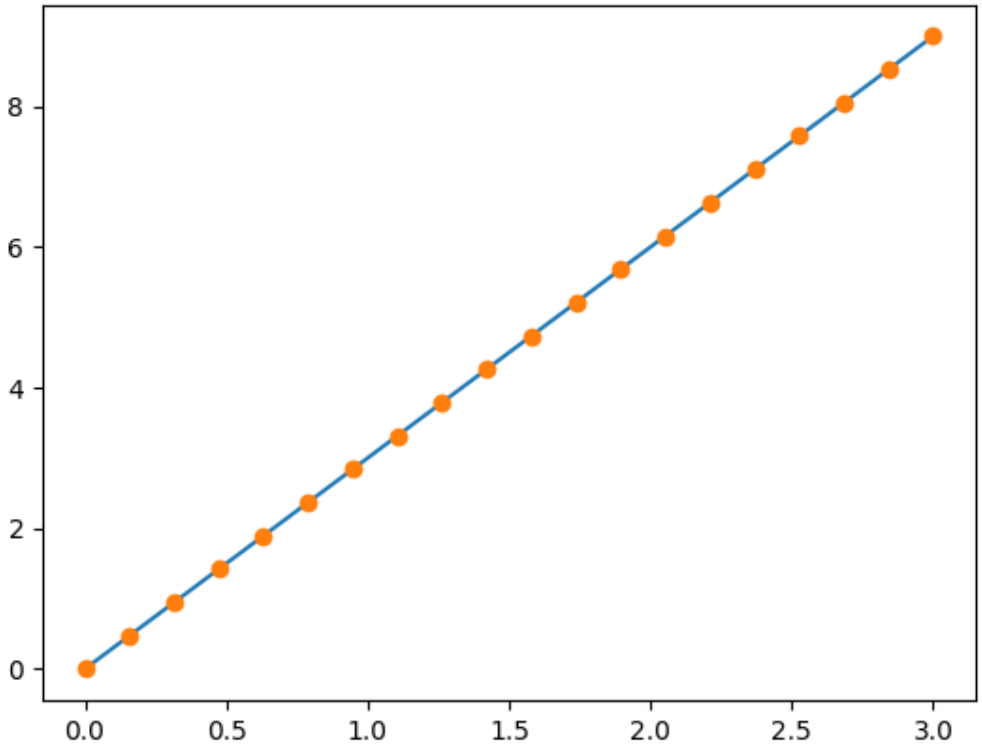
Solution: `Python source file`

3.6 Full code examples

3.6.1 Full code examples for the numpy chapter

1D plotting

Plot a basic 1D figure

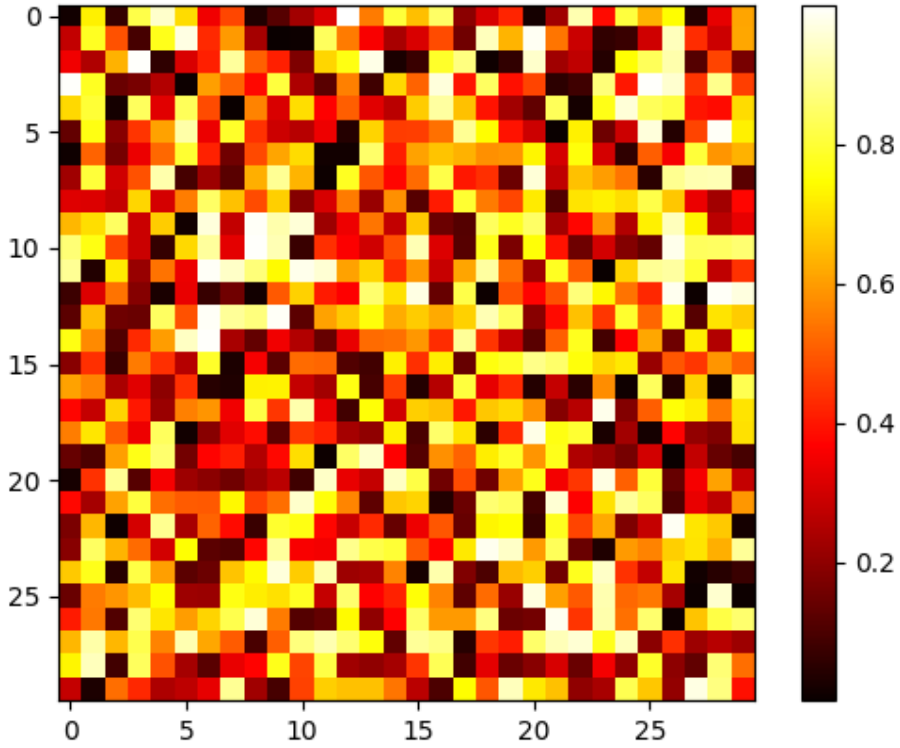



```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 3, 20)
y = np.linspace(0, 9, 20)
plt.plot(x, y)
plt.plot(x, y, "o")
plt.show()
```

Total running time of the script: (0 minutes 0.047 seconds)

2D plotting

Plot a basic 2D figure

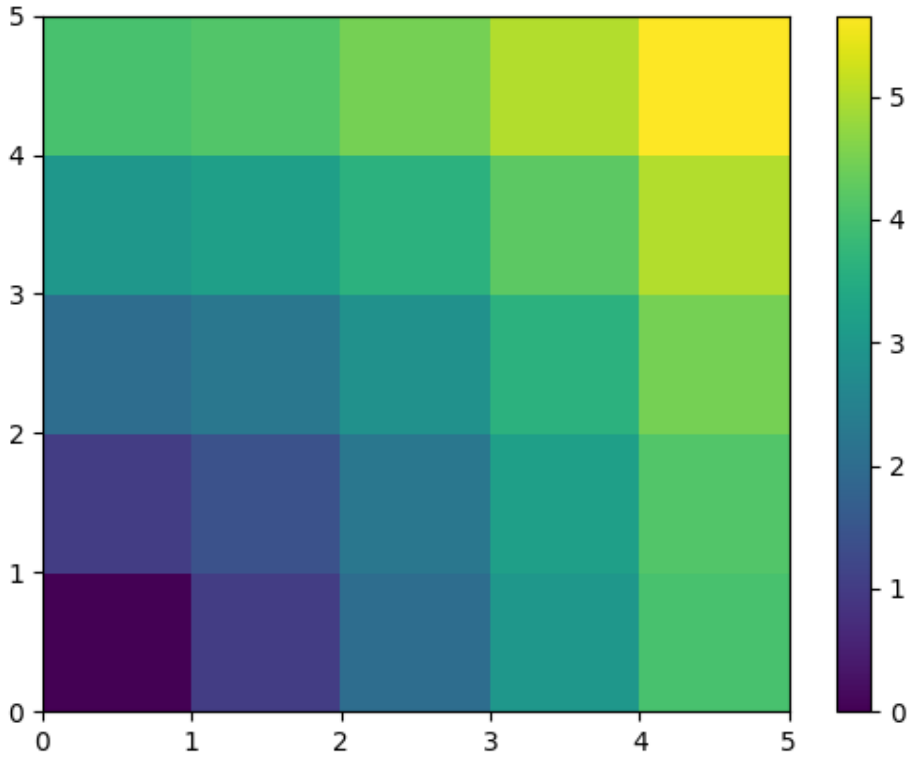



```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng()
image = rng.random((30, 30))
plt.imshow(image, cmap="hot")
plt.colorbar()
plt.show()
```

Total running time of the script: (0 minutes 0.069 seconds)

Distances exercise

Plot distances in a grid

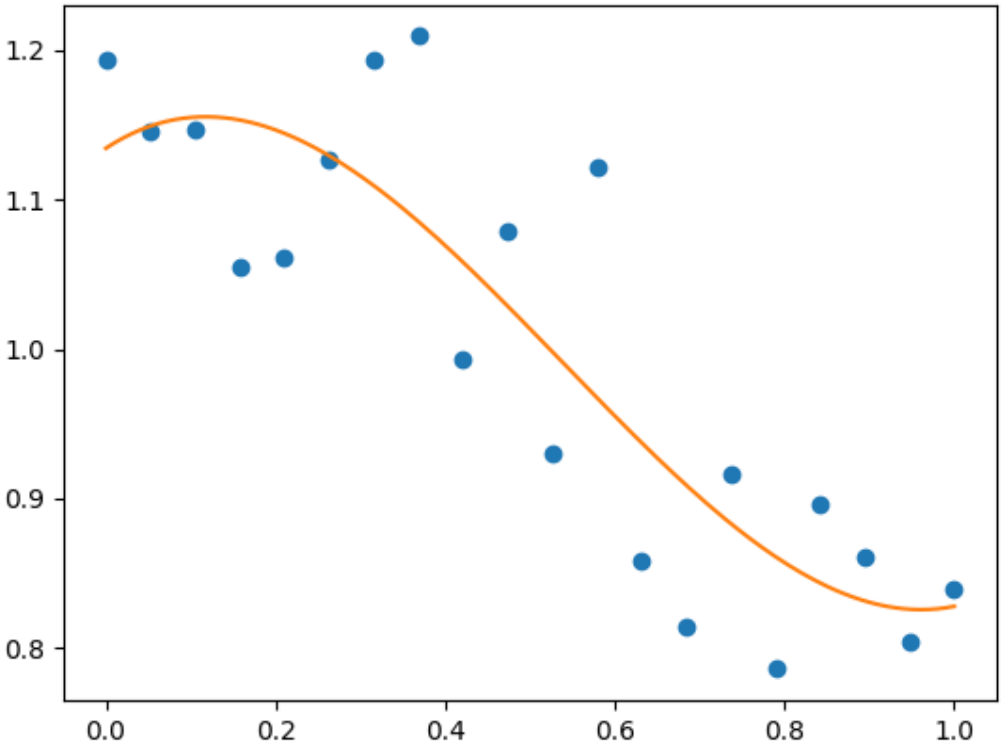



```
import numpy as np
import matplotlib.pyplot as plt

x, y = np.arange(5), np.arange(5)[:, np.newaxis]
distance = np.sqrt(x**2 + y**2)
plt.pcolor(distance)
plt.colorbar()
plt.show()
```

Total running time of the script: (0 minutes 0.063 seconds)

Fitting to polynomial

Plot noisy data and their polynomial fit

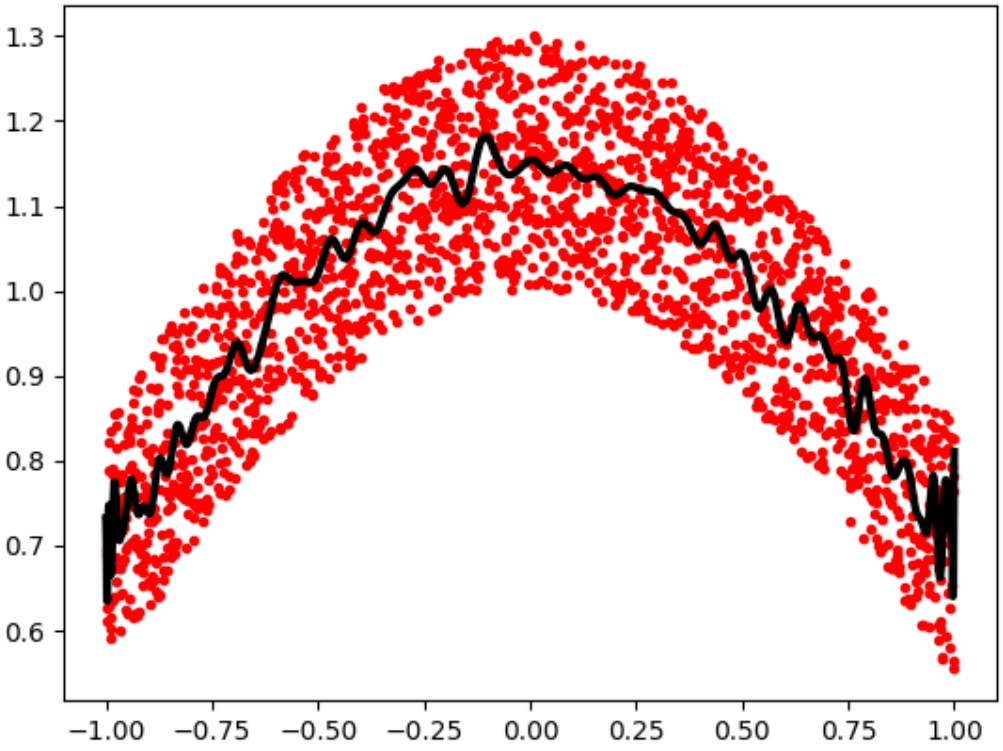



```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

x = np.linspace(0, 1, 20)
y = np.cos(x) + 0.3 * rng.random(20)
p = np.poly1d(np.polyfit(x, y, 3))

t = np.linspace(0, 1, 200)
plt.plot(x, y, "o", t, p(t), "-")
plt.show()
```

Total running time of the script: (0 minutes 0.046 seconds)

Fitting in Chebyshev basis

Plot noisy data and their polynomial fit in a Chebyshev basis




```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

x = np.linspace(-1, 1, 2000)
y = np.cos(x) + 0.3 * rng.random(2000)
p = np.polynomial.Chebyshev.fit(x, y, 90)

plt.plot(x, y, "r.")
plt.plot(x, p(x), "k-", lw=3)
plt.show()
```

Total running time of the script: (0 minutes 0.088 seconds)

Population exercise

Plot populations of hares, lynxes, and carrots

```
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt("../../../data/populations.txt")
year, hares, lynxes, carrots = data.T

plt.axes((0.2, 0.1, 0.5, 0.8))
plt.plot(year, hares, year, lynxes, year, carrots)
plt.legend(("Hare", "Lynx", "Carrot"), loc=(1.05, 0.5))
plt.show()
```

Total running time of the script: (0 minutes 0.059 seconds)

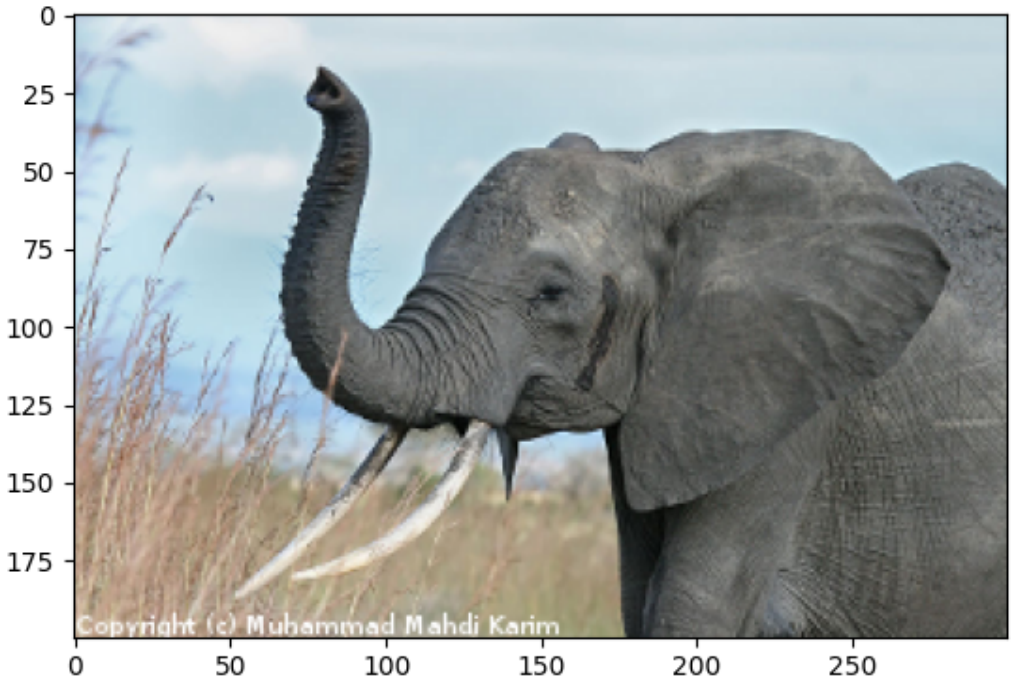
Reading and writing an elephant

Read and write images

```
import numpy as np
import matplotlib.pyplot as plt
```

original figure

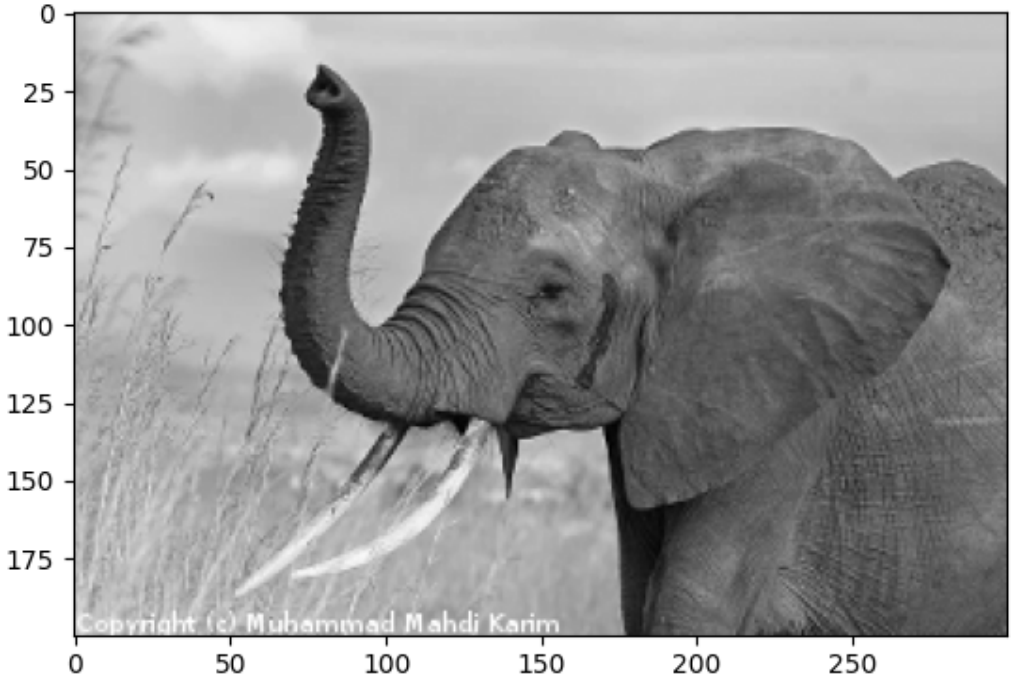
```
plt.figure()
img = plt.imread("../../../data/elephant.png")
plt.imshow(img)
```




```
<matplotlib.image.AxesImage object at 0x7f3b37add460>
```

red channel displayed in grey

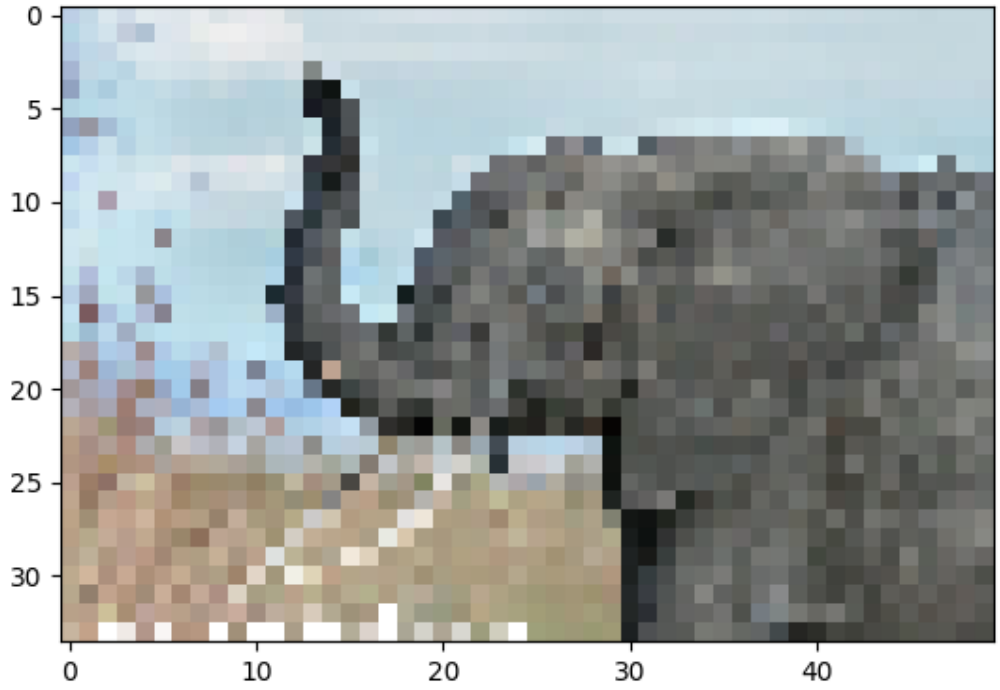
```
plt.figure()
img_red = img[:, :, 0]
plt.imshow(img_red, cmap="gray")
```




```
<matplotlib.image.AxesImage object at 0x7f3b371659a0>
```

lower resolution

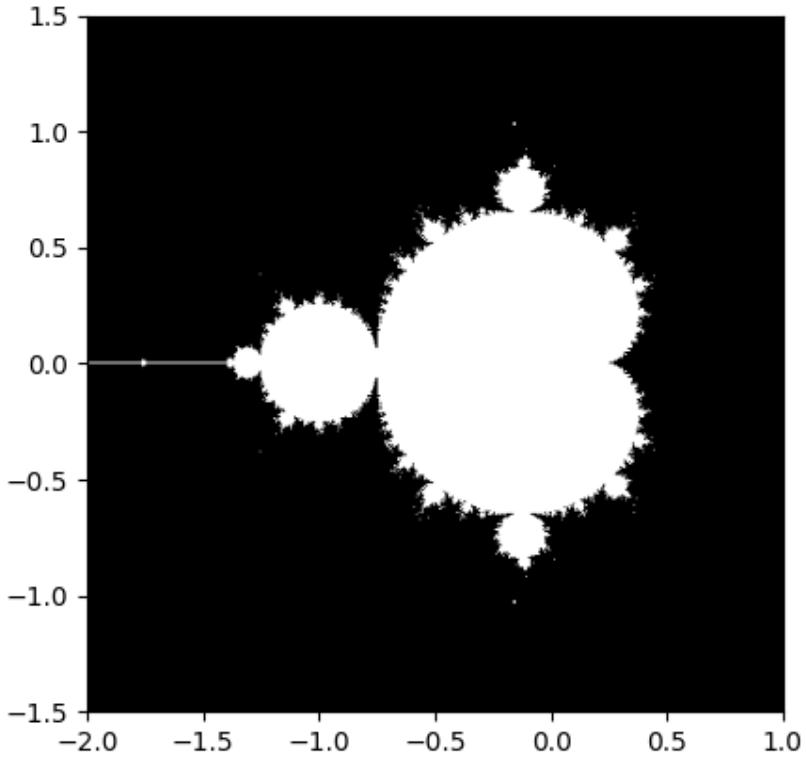
```
plt.figure()
img_tiny = img[::6, ::6]
plt.imshow(img_tiny, interpolation="nearest")

plt.show()
```



Total running time of the script: (0 minutes 0.258 seconds)

Mandelbrot set

Compute the Mandelbrot fractal and plot it




```
import numpy as np
import matplotlib.pyplot as plt
from numpy import newaxis
import warnings


def compute_mandelbrot(N_max, some_threshold, nx, ny):
    # A grid of c-values
    x = np.linspace(-2, 1, nx)
    y = np.linspace(-1.5, 1.5, ny)

    c = x[:, newaxis] + 1j * y[newaxis, :]

    # Mandelbrot iteration

    z = c

    # The code below overflows in many regions of the x-y grid, suppress
    # warnings temporarily
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        for j in range(N_max):
            z = z**2 + c
        mandelbrot_set = abs(z) < some_threshold

    return mandelbrot_set
```

(continues on next page)

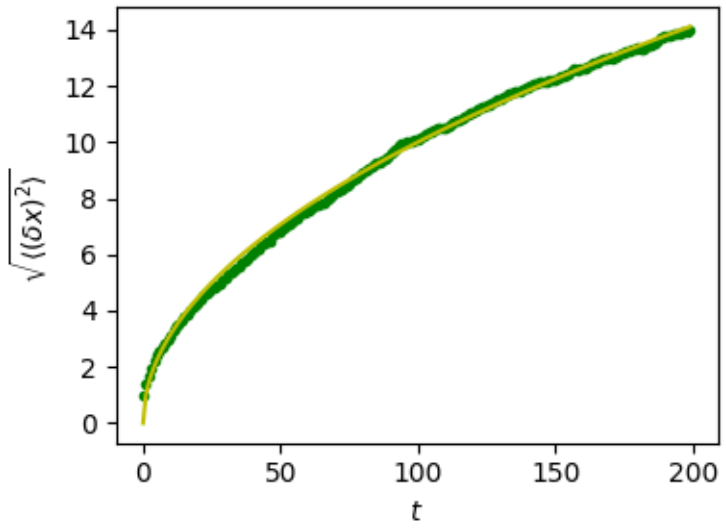
(continued from previous page)

```
mandelbrot_set = compute_mandelbrot(50, 50.0, 601, 401)

plt.imshow(mandelbrot_set.T, extent=(-2, 1, -1.5, 1.5))
plt.gray()
plt.show()
```

Total running time of the script: (0 minutes 0.080 seconds)

Random walk exercise

Plot distance as a function of time for a random walk together with the theoretical result




```
import numpy as np
import matplotlib.pyplot as plt

# We create 1000 realizations with 200 steps each
n_stories = 1000
t_max = 200

t = np.arange(t_max)
# Steps can be -1 or 1 (note that randint excludes the upper limit)
rng = np.random.default_rng()
steps = 2 * rng.integers(0, 1 + 1, (n_stories, t_max)) - 1

# The time evolution of the position is obtained by successively
# summing up individual steps. This is done for each of the
# realizations, i.e. along axis 1.
positions = np.cumsum(steps, axis=1)

# Determine the time evolution of the mean square distance.
sq_distance = positions**2
mean_sq_distance = np.mean(sq_distance, axis=0)

# Plot the distance d from the origin as a function of time and
# compare with the theoretically expected result where d(t)
# grows as a square root of time t.
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(4, 3))
plt.plot(t, np.sqrt(mean_sq_distance), "g.", t, np.sqrt(t), "y-")
plt.xlabel(r"$t$")
plt.ylabel(r"$\sqrt{\langle (\delta x)^2 \rangle}$")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.073 seconds)

CHAPTER 4

Matplotlib: plotting

Thanks

Many thanks to **Bill Wing** and **Christoph Deil** for review and corrections.

Authors: *Nicolas Rougier, Mike Müller, Gaël Varoquaux*

Chapter contents

- *Introduction*
- *Simple plot*
- *Figures, Subplots, Axes and Ticks*
- *Other Types of Plots: examples and exercises*
- *Beyond this tutorial*
- *Quick references*
- *Full code examples*

4.1 Introduction

Tip

Matplotlib is probably the most used Python package for 2D-graphics. It provides both a quick way

to visualize data from Python and publication-quality figures in many formats. We are going to explore matplotlib in interactive mode covering most common cases.

4.1.1 IPython, Jupyter, and matplotlib modes

Tip

The Jupyter notebook and the IPython enhanced interactive Python, are tuned for the scientific-computing workflow in Python, in combination with Matplotlib:

For interactive matplotlib sessions, turn on the **matplotlib mode**

IPython console
When using the IPython console, use:

```
In [1]: %matplotlib
```

Jupyter notebook
In the notebook, insert, **at the beginning of the notebook** the following magic:

```
%matplotlib inline
```

4.1.2 pyplot

Tip

pyplot provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab™. Therefore, the majority of plotting commands in pyplot have Matlab™ analogs with similar arguments. Important commands are explained with interactive examples.

```
import matplotlib.pyplot as plt
```

4.2 Simple plot

Tip

In this section, we want to draw the cosine and sine functions on the same plot. Starting from the default settings, we'll enrich the figure step by step to make it nicer.

First step is to get the data for the sine and cosine functions:

```
import numpy as np

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)
```

`X` is now a numpy array with 256 values ranging from $-\pi$ to $+\pi$ (included). `C` is the cosine (256 values) and `S` is the sine (256 values).

To run the example, you can type them in an IPython interactive session:

```
$ ipython --matplotlib
```

This brings us to the IPython prompt:

```
IPython 0.13 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.
```

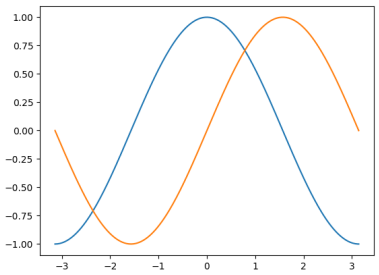
Tip

You can also download each of the examples and run it using regular python, but you will lose interactive data manipulation:

```
$ python plot_exercise_1.py
```

You can get source for each step by clicking on the corresponding figure.

4.2.1 Plotting with default settings



Hint

Documentation

- plot tutorial
- `plot()` command

Tip

Matplotlib comes with a set of default settings that allow customizing all kinds of properties. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on.

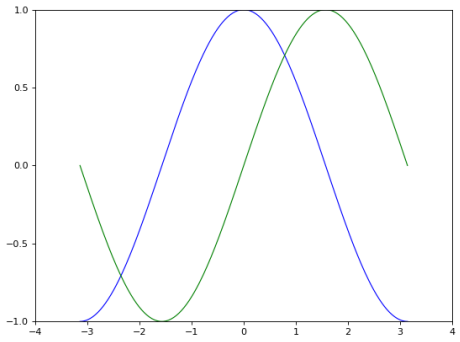
```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C)
plt.plot(X, S)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

4.2.2 Instantiating defaults



Hint

Documentation

- Customizing matplotlib

In the script below, we've instantiated (and commented) all the figure settings that influence the appearance of the plot.

Tip

The settings have been explicitly set to their default values, but now you can interactively play with the values to explore their affect (see *Line properties* and *Line styles* below).

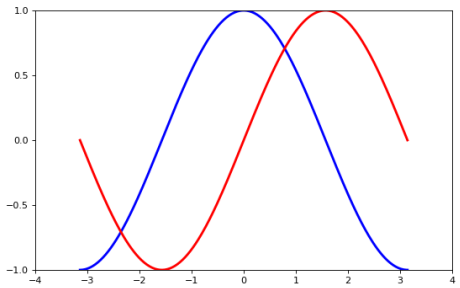
```
import numpy as np
import matplotlib.pyplot as plt

# Create a figure of size 8x6 inches, 80 dots per inch
plt.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
plt.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)

# Plot cosine with a blue continuous line of width 1 (pixels)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine with a green continuous line of width 1 (pixels)
plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
```

(continues on next page)

(continued from previous page)

```
plt.xlim(-4.0, 4.0)

# Set x ticks
plt.xticks(np.linspace(-4, 4, 9))

# Set y limits
plt.ylim(-1.0, 1.0)

# Set y ticks
plt.yticks(np.linspace(-1, 1, 5))

# Save figure using 72 dots per inch
# plt.savefig("exercise_2.png", dpi=72)

# Show result on screen
plt.show()
```

4.2.3 Changing colors and line widths



Hint

Documentation

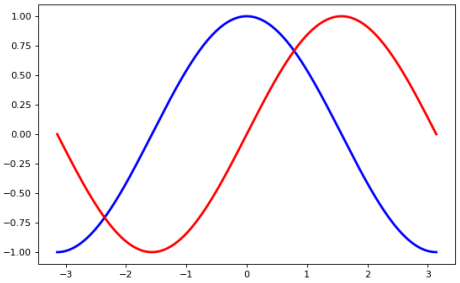
- Controlling line properties
- `Line2D` API

Tip

First step, we want to have the cosine in blue and the sine in red and a slightly thicker line for both of them. We'll also slightly alter the figure size to make it more horizontal.

```
...
plt.figure(figsize=(10, 6), dpi=80)
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
...
```

4.2.4 Setting limits



Hint

Documentation

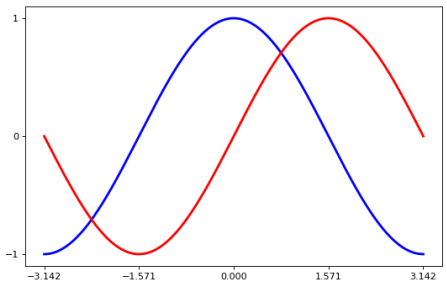
- `xlim()` command
- `ylim()` command

Tip

Current limits of the figure are a bit too tight and we want to make some space in order to clearly see all data points.

```
...
plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)
...
```

4.2.5 Setting ticks



Hint

Documentation

- `xticks()` command
- `yticks()` command
- Tick container

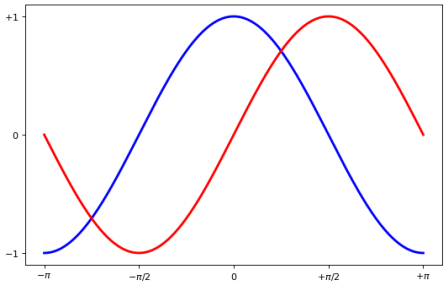
- Tick locating and formatting

Tip

Current ticks are not ideal because they do not show the interesting values ($\pm\pi$,:math:$pm\ pi/2$) for sine and cosine. We'll change them such that they show only these values.

```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
plt.yticks([-1, 0, +1])
...
```

4.2.6 Setting tick labels



Hint

Documentation

- Working with text
- `xticks()` command
- `yticks()` command
- `set_xticklabels()`
- `set_yticklabels()`

Tip

Ticks are now properly placed but their label is not very explicit. We could guess that 3.142 is π but it would be better to make it explicit. When we set tick values, we can also provide a corresponding label in the second argument list. Note that we'll use latex to allow for nice rendering of the label.

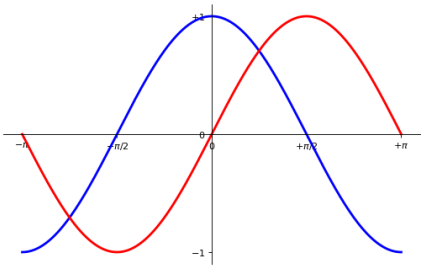
```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
           [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.yticks([-1, 0, +1],
```

(continues on next page)

(continued from previous page)

```
          [r'$-1$', r'$0$', r'$+1$'])
...
```

4.2.7 Moving spines



Hint

Documentation

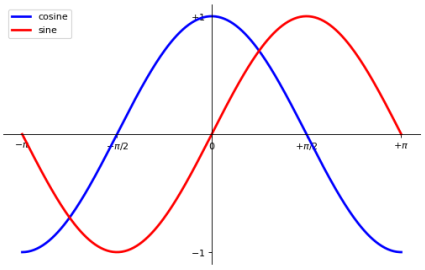
- `spines` API
- Axis container
- Transformations tutorial

Tip

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions and until now, they were on the border of the axis. We'll change that since we want to have them in the middle. Since there are four of them (top/bottom/left/right), we'll discard the top and right by setting their color to none and we'll move the bottom and left ones to coordinate 0 in data space coordinates.

```
...
ax = plt.gca()  # gca stands for 'get current axis'
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
...
```

4.2.8 Adding a legend



Hint

Documentation

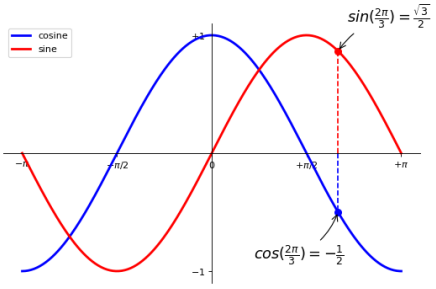
- Legend guide
- `legend()` command
- `legend` API

Tip

Let's add a legend in the upper left corner. This only requires adding the keyword argument label (that will be used in the legend box) to the plot commands.

```
...
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-", label="sine")

plt.legend(loc='upper left')
...
```

4.2.9 Annotate some points



Hint

Documentation

- Annotating axis

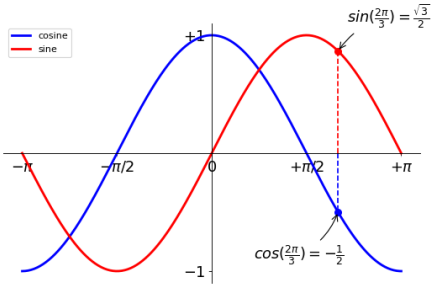
- `annotate()` command

Tip

Let's annotate some interesting points using the annotate command. We chose the $2\pi/3$ value and we want to annotate both the sine and the cosine. We'll first draw a marker on the curve as well as a straight dotted line. Then, we'll use the annotate command to display some text with an arrow.

```
...

t = 2 * np.pi / 3
plt.plot([t, t], [0, np.cos(t)], color='blue', linewidth=2.5, linestyle="--")
plt.scatter([t, ], [np.cos(t), ], 50, color='blue')

plt.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$',
             xy=(t, np.cos(t)), xycoords='data',
             xytext=(-90, -50), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plt.plot([t, t],[0, np.sin(t)], color='red', linewidth=2.5, linestyle="--")
plt.scatter([t, ],[np.sin(t), ], 50, color='red')

plt.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
             xy=(t, np.sin(t)), xycoords='data',
             xytext=(+10, +30), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
...
```

4.2.10 Devil is in the details



Hint

Documentation

- `artist` API
- `set_bbox()` method

Tip

The tick labels are now hardly visible because of the blue and red lines. We can make them bigger and we can also adjust their properties such that they'll be rendered on a semi-transparent white background. This will allow us to see both the data and the labels.

```
...
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
    label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65))
...
```

4.3 Figures, Subplots, Axes and Ticks

A **"figure"** in matplotlib means the whole window in the user interface. Within this figure there can be **"subplots"**.

Tip

So far we have used implicit figure and axes creation. This is handy for fast plots. We can have more control over the display using figure, subplot, and axes explicitly. While subplot positions the plots in a regular grid, axes allows free placement within the figure. Both can be useful depending on your intention. We've already worked with figures and subplots without explicitly calling them. When we call plot, matplotlib calls `gca()` to get the current axes and gca in turn calls `gcf()` to get the current figure. If there is none it calls `figure()` to make one, strictly speaking, to make a `subplot(111)`. Let's look at the details.

4.3.1 Figures

Tip

A figure is the windows in the GUI that has "Figure #" as title. Figures are numbered starting from 1 as opposed to the normal Python way starting from 0. This is clearly MATLAB-style. There are several parameters that determine what the figure looks like:

Argument	Default	Description
`num`	`1`	number of figure
`figsize`	`figure.figsize`	figure size in inches (width, height)
`dpi`	`figure.dpi`	resolution in dots per inch
`facecolor`	`figure.facecolor`	color of the drawing background
`edgecolor`	`figure.edgecolor`	color of edge around the drawing background
`frameon`	`True`	draw figure frame or not

Tip

The defaults can be specified in the resource file and will be used most of the time. Only the number of the figure is frequently changed.

As with other objects, you can set figure properties also setp or with the set_something methods.

When you work with the GUI you can close a figure by clicking on the x in the upper right corner. But you can close a figure programmatically by calling close. Depending on the argument it closes (1)

the current figure (no argument), (2) a specific figure (figure number or figure instance as argument), or (3) all figures (`"all"` as argument).

```
plt.close(1)     # Closes figure 1
```

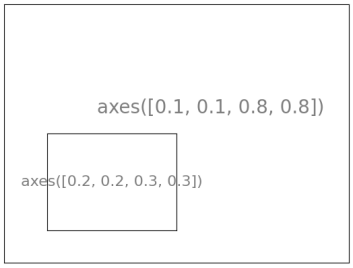
4.3.2 Subplots

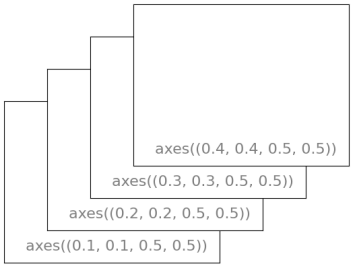
Tip

With subplot you can arrange plots in a regular grid. You need to specify the number of rows and columns and the number of the plot. Note that the gridspec command is a more powerful alternative.

4.3.3 Axes

Axes are very similar to subplots but allow placement of plots at any location in the figure. So if we want to put a smaller plot inside a bigger one we do so with axes.





4.3.4 Ticks

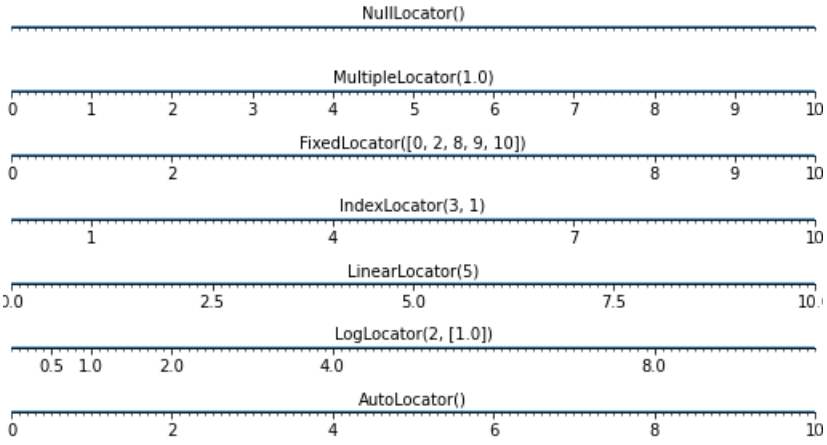
Well formatted ticks are an important part of publishing-ready figures. Matplotlib provides a totally configurable system for ticks. There are tick locators to specify where ticks should appear and tick formatters to give ticks the appearance you want. Major and minor ticks can be located and formatted independently from each other. Per default minor ticks are not shown, i.e. there is only an empty list for them because it is as `NullLocator` (see below).

Tick Locators

Tick locators control the positions of the ticks. They are set as follows:

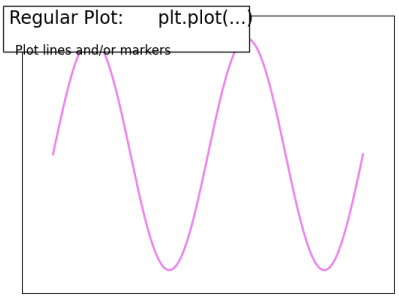
```
ax = plt.gca()
ax.xaxis.set_major_locator(eval(locator))
```

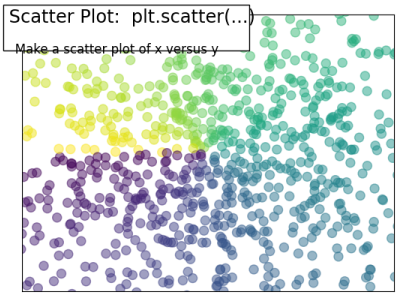
There are several locators for different kind of requirements:

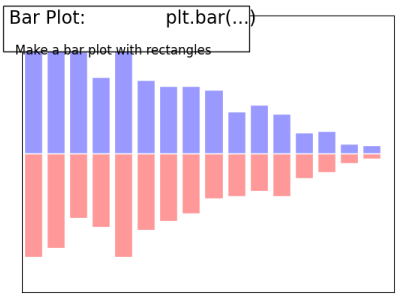


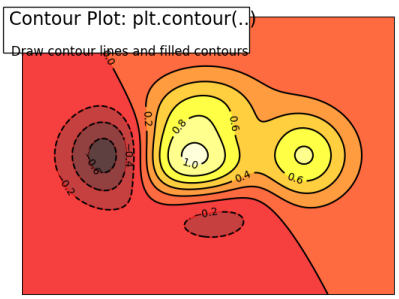
All of these locators derive from the base class `matplotlib.ticker.Locator`. You can make your own locator deriving from it. Handling dates as ticks can be especially tricky. Therefore, matplotlib provides special locators in matplotlib.dates.

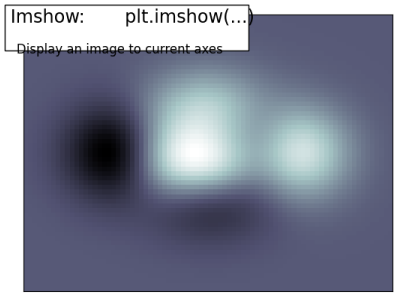
4.4 Other Types of Plots: examples and exercises

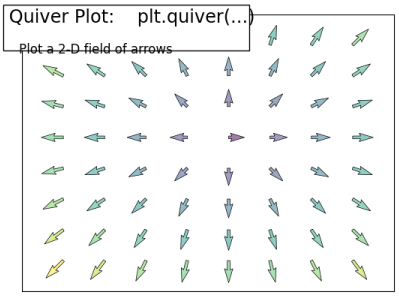


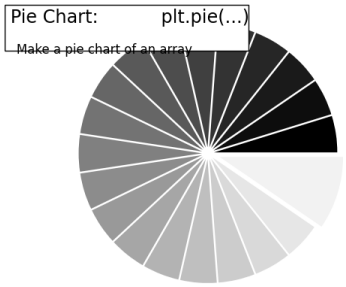


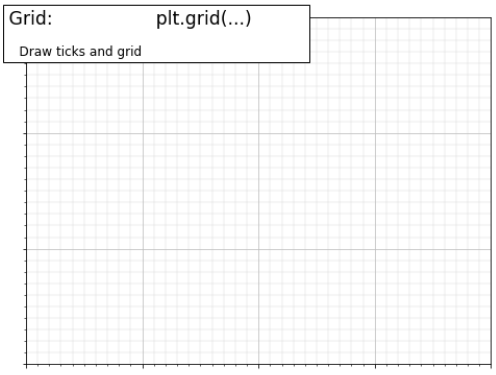


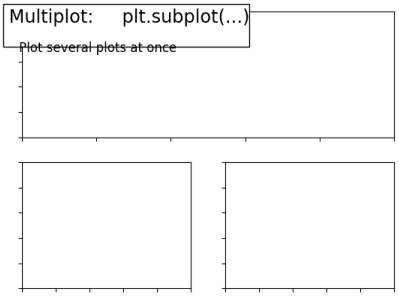


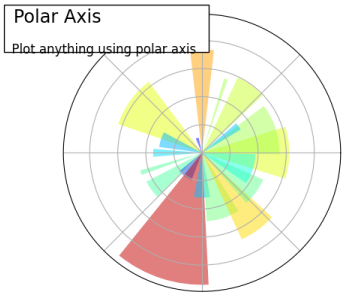

Imshow: plt.imshow(...)
Display an image to current axes

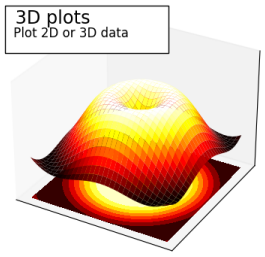

Quiver Plot: plt.quiver(...)
Plot a 2-D field of arrows

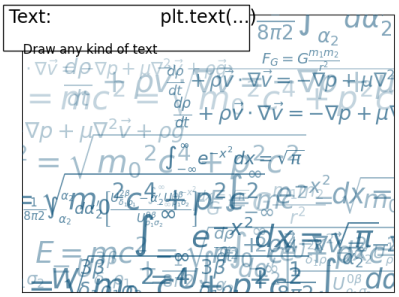

Pie Chart: plt.pie(...)
Make a pie chart of an array


Grid: plt.grid(...)
Draw ticks and grid

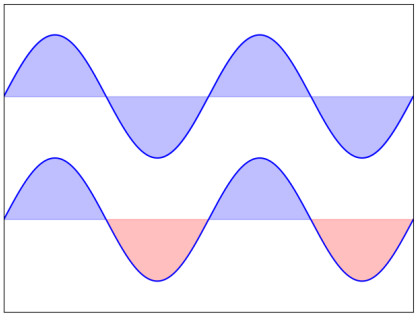

Multiplot: plt.subplot(...)
Plot several plots at once


Polar Axis
Plot anything using polar axis


3D plots
Plot 2D or 3D data


Text: plt.text(...)
Draw any kind of text

4.4.1 Regular Plots



Starting from the code below, try to reproduce the graphic taking care of filled areas:

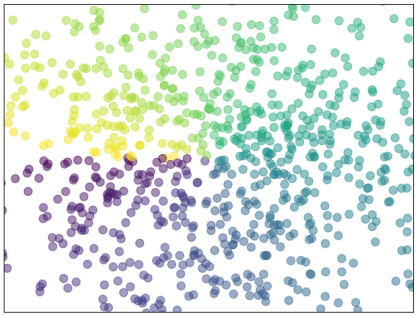
Hint

You need to use the `fill_between()` command.

```
n = 256
X = np.linspace(-np.pi, np.pi, n)
Y = np.sin(2 * X)

plt.plot(X, Y + 1, color='blue', alpha=1.00)
plt.plot(X, Y - 1, color='blue', alpha=1.00)
```

Click on the figure for solution.

4.4.2 Scatter Plots



Starting from the code below, try to reproduce the graphic taking care of marker size, color and transparency.

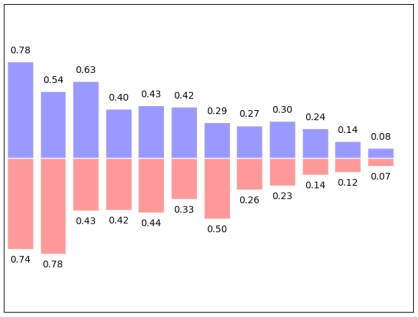
Hint

Color is given by angle of (X,Y).

```
n = 1024
rng = np.random.default_rng()
X = rng.normal(0,1,n)
Y = rng.normal(0,1,n)

plt.scatter(X,Y)
```

Click on figure for solution.

4.4.3 Bar Plots



Starting from the code below, try to reproduce the graphic by adding labels for red bars.

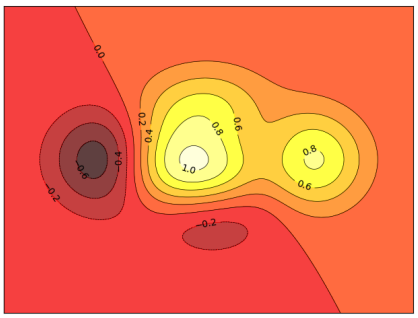
Hint

You need to take care of text alignment.

```
n = 12
X = np.arange(n)
rng = np.random.default_rng()
Y1 = (1 - X / float(n)) * rng.uniform(0.5, 1.0, n)
Y2 = (1 - X / float(n)) * rng.uniform(0.5, 1.0, n)

plt.bar(X, +Y1, facecolor='#9999ff', edgecolor='white')
plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x, y in zip(X, Y1):
    plt.text(x + 0.4, y + 0.05, '%.2f' % y, ha='center', va='bottom')

plt.ylim(-1.25, +1.25)
```

Click on figure for solution.

4.4.4 Contour Plots



Starting from the code below, try to reproduce the graphic taking care of the colormap (see *Colormaps* below).

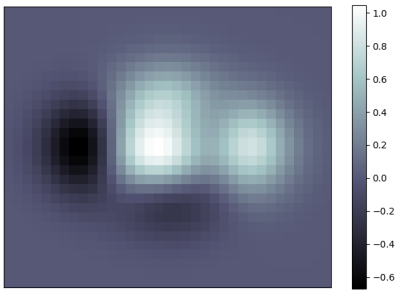
Hint

You need to use the `clabel()` command.

```
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 -y ** 2)

n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)

plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='jet')
C = plt.contour(X, Y, f(X, Y), 8, colors='black', linewidth=.5)
```

Click on figure for solution.

4.4.5 Imshow



Starting from the code below, try to reproduce the graphic taking care of colormap, image interpolation and origin.

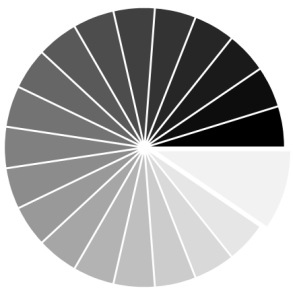
Hint

You need to take care of the `origin` of the image in the imshow command and use a `colorbar()`

```
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)

n = 10
x = np.linspace(-3, 3, 4 * n)
y = np.linspace(-3, 3, 3 * n)
X, Y = np.meshgrid(x, y)
plt.imshow(f(X, Y))
```

Click on the figure for the solution.

4.4.6 Pie Charts



Starting from the code below, try to reproduce the graphic taking care of colors and slices size.

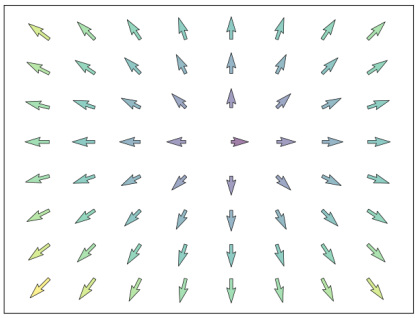
Hint

You need to modify Z.

```
rng = np.random.default_rng()
Z = rng.uniform(0, 1, 20)
plt.pie(Z)
```

Click on the figure for the solution.

4.4.7 Quiver Plots



Starting from the code below, try to reproduce the graphic taking care of colors and orientations.

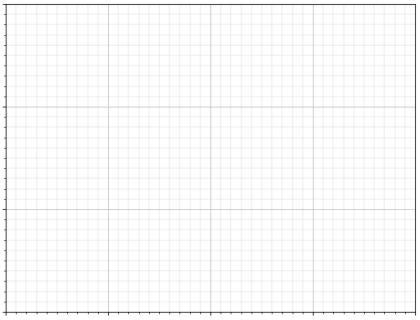
Hint

You need to draw arrows twice.

```
n = 8
X, Y = np.mgrid[0:n, 0:n]
plt.quiver(X, Y)
```

Click on figure for solution.

4.4.8 Grids



Starting from the code below, try to reproduce the graphic taking care of line styles.

```
axes = plt.gca()
axes.set_xlim(0, 4)
axes.set_ylim(0, 3)
```

(continues on next page)

(continued from previous page)

```
axes.set_xticklabels([])
axes.set_yticklabels([])
```

Click on figure for solution.

4.4.9 Multi Plots

Starting from the code below, try to reproduce the graphic.

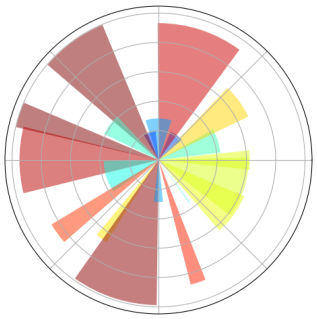
Hint

You can use several subplots with different partition.

```
plt.subplot(2, 2, 1)
plt.subplot(2, 2, 3)
plt.subplot(2, 2, 4)
```

Click on figure for solution.

4.4.10 Polar Axis



Hint

You only need to modify the `axes` line

Starting from the code below, try to reproduce the graphic.

```
plt.axes([0, 0, 1, 1])

N = 20
theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
rng = np.random.default_rng()
```

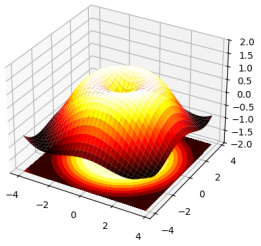
(continues on next page)

(continued from previous page)

```
radii = 10 * rng.random(N)
width = np.pi / 4 * rng.random(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)
```

Click on figure for solution.

4.4.11 3D Plots



Starting from the code below, try to reproduce the graphic.

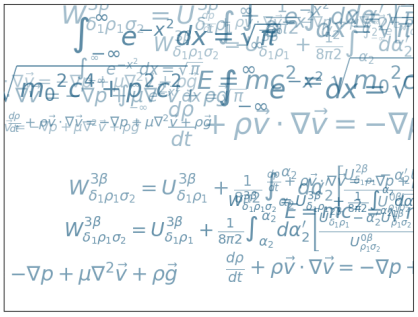
Hint

You need to use `contourf()`

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
Y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

Click on figure for solution.

4.4.12 Text



Try to do the same from scratch !

Hint

Have a look at the matplotlib logo.

Click on figure for solution.

Quick read

If you want to do a first quick pass through the Scientific Python Lectures to learn the ecosystem, you can directly skip to the next chapter: *SciPy : high-level scientific computing*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter later.

4.5 Beyond this tutorial

Matplotlib benefits from extensive documentation as well as a large community of users and developers. Here are some links of interest:

4.5.1 Tutorials

- Pyplot tutorial
- Introduction
- Controlling line properties
- Working with multiple figures and axes
- Working with text
- Image tutorial
- Startup commands
- Importing image data into NumPy arrays
- Plotting NumPy arrays as images
- Text tutorial
- Text introduction
- Basic text commands
- Text properties and layout
- Writing mathematical expressions
- Text rendering With LaTeX
- Annotating text
- Artist tutorial
- Introduction
- Customizing your objects
- Object containers
- Figure container
- Axes container
- Axis containers
- Tick containers
- Path tutorial
- Introduction
- Bézier example
- Compound paths
- Transforms tutorial
- Introduction
- Data coordinates
- Axes coordinates
- Blended transformations
- Using offset transforms to create a shadow effect
- The transformation pipeline

4.5.2 Matplotlib documentation

- User guide
- FAQ
- Installation
- Usage
- How-To
- Troubleshooting
- Environment Variables
- Screenshots

4.5.3 Code documentation

The code is well documented and you can quickly access a specific command from within a python session:

```
>>> import matplotlib.pyplot as plt
>>> help(plt.plot)
Help on function plot in module matplotlib.pyplot:

plot(*args: ...) -> 'list[Line2D]'
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
...
```

4.5.4 Galleries

The matplotlib gallery is also incredibly useful when you search how to render a given graphic. Each example comes with its source.

4.5.5 Mailing lists

Finally, there is a user mailing list where you can ask for help and a developers mailing list that is more technical.

4.6 Quick references

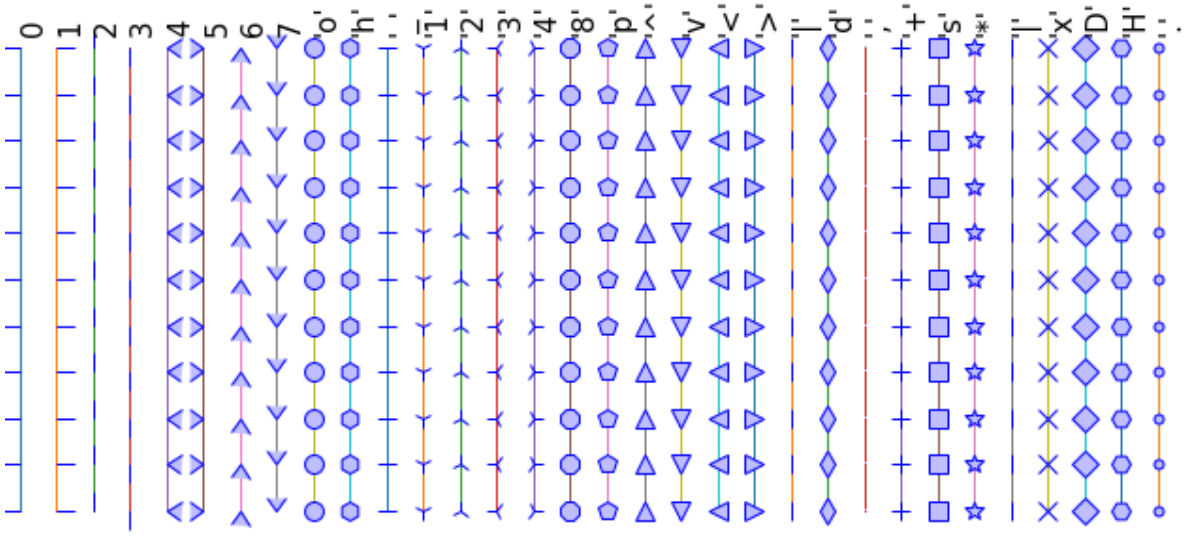
Here is a set of tables that show main properties and styles.

4.6.1 Line properties

Property	Description	Appearance
alpha (or a)	alpha transparency on 0-1 scale	
antialiased	True or False - use antialised rendering	Aliased Anti-aliased
color (or c)	matplotlib color arg	
linestyle (or ls)	see *Line properties*	
linewidth (or lw)	float, the line width in points	
solid_capstyle	Cap style for solid lines	
solid_joinstyle	Join style for solid lines	
dash_capstyle	Cap style for dashes	
dash_joinstyle	Join style for dashes	
marker	see *Markers*	
markeredgewidth (mew)	line width around the marker symbol	
markeredgecolor (mec)	edge color if a marker is used	
markerfacecolor (mfc)	face color if a marker is used	
markersize (ms)	size of the marker in points	

4.6.2 Line styles

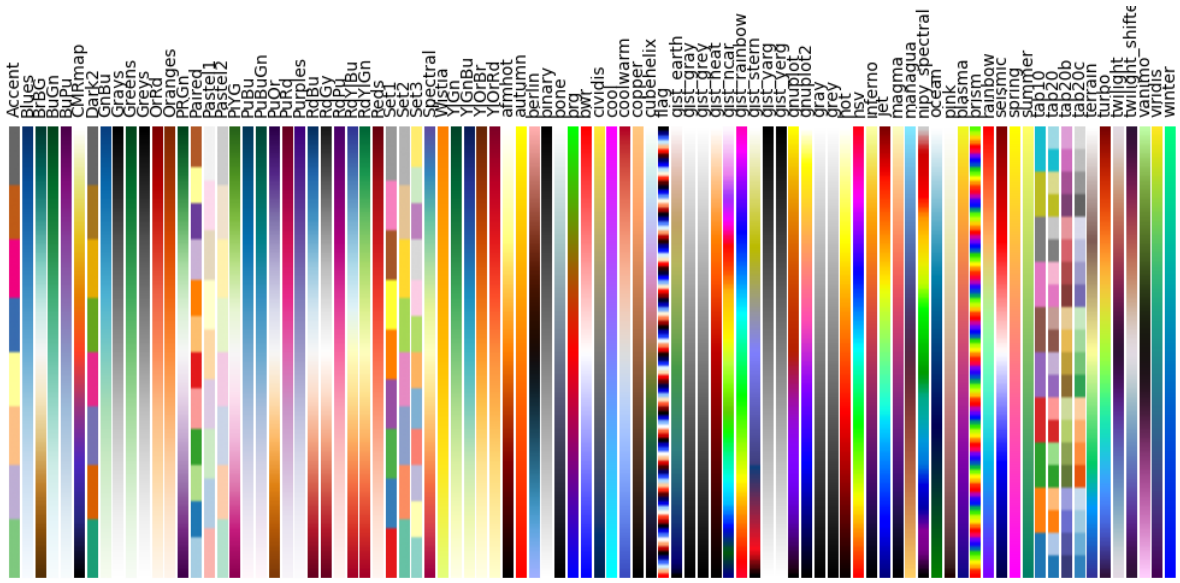
4.6.3 Markers



4.6.4 Colormaps

All colormaps can be reversed by appending `_r`. For instance, `gray_r` is the reverse of `gray`.

If you want to know more about colormaps, check the documentation on Colormaps in matplotlib.



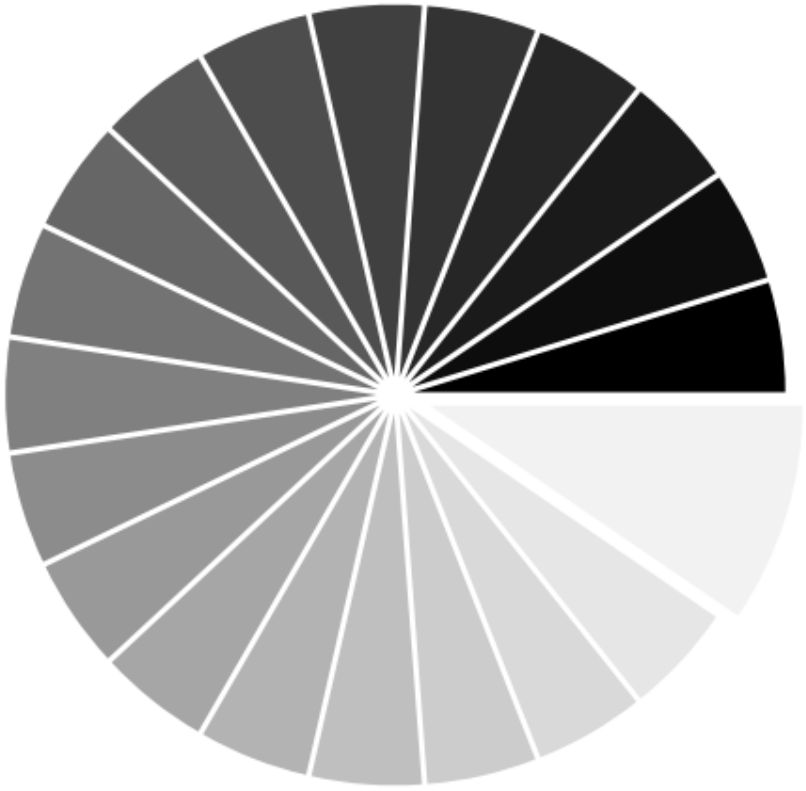
4.7 Full code examples

4.7.1 Code samples for Matplotlib

The examples here are only examples relevant to the points raised in this chapter. The matplotlib documentation comes with a much more exhaustive gallery.

Pie chart

A simple pie chart example with matplotlib.

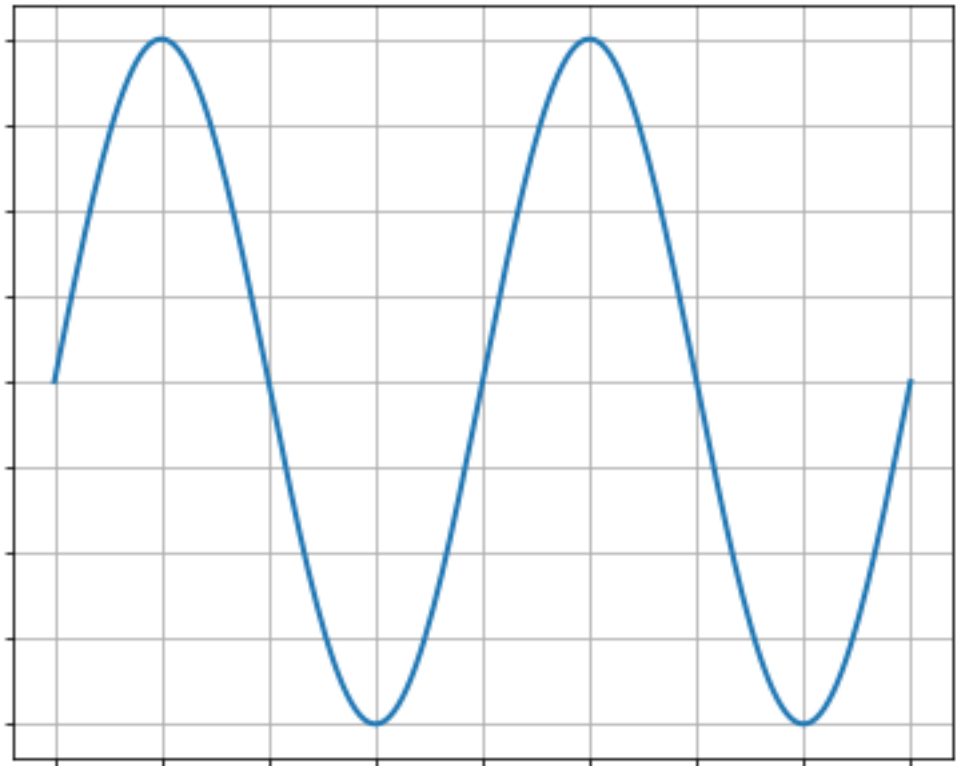


```
import numpy as np
import matplotlib.pyplot as plt

n = 20
Z = np.ones(n)
Z[-1] *= 2

plt.axes((0.025, 0.025, 0.95, 0.95))

plt.pie(Z, explode=Z * 0.05, colors=[f"{i / float(n):f}" for i in range(n)])
plt.axis("equal")
plt.xticks([])
plt.yticks()

plt.show()
```

Total running time of the script: (0 minutes 0.050 seconds)

A simple, good-looking plot

Demoing some simple features of matplotlib

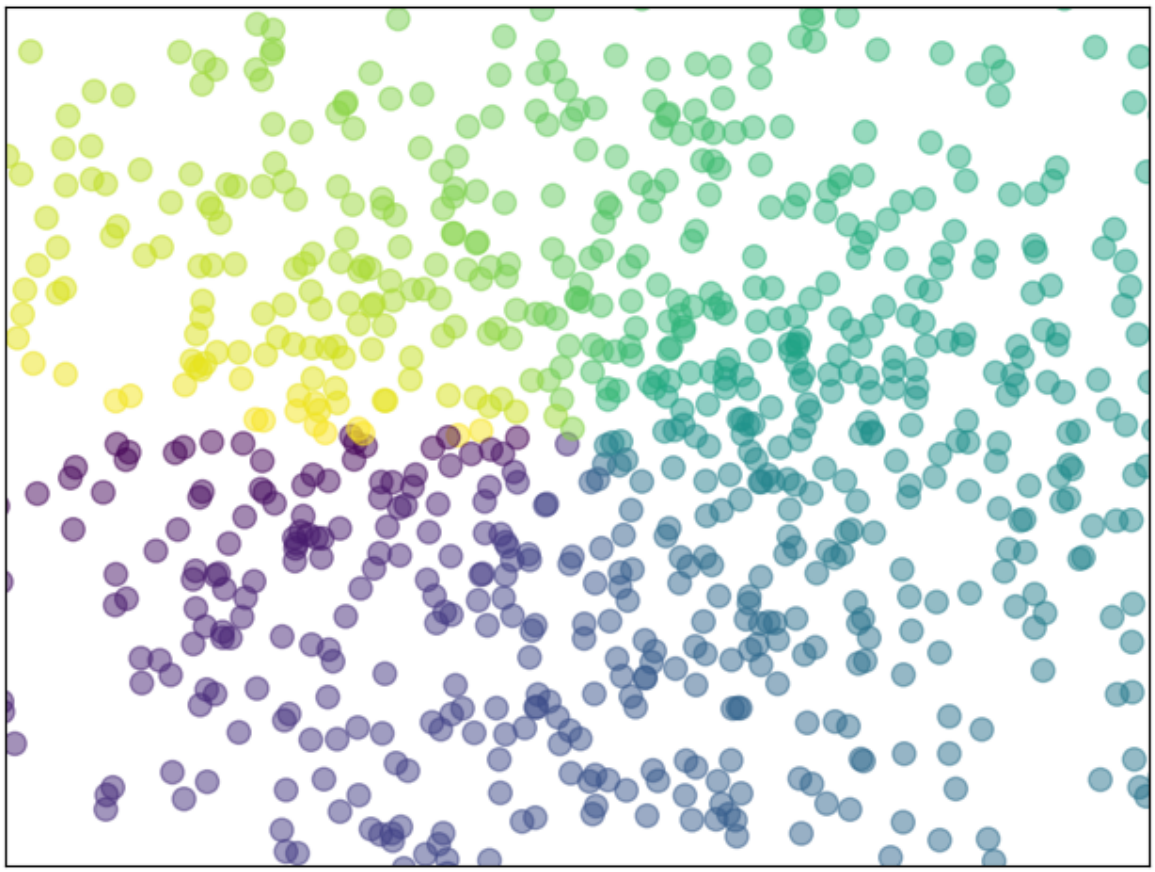


```
import numpy as np
import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5, 4), dpi=72)
axes = fig.add_axes((0.01, 0.01, 0.98, 0.98))
X = np.linspace(0, 2, 200)
Y = np.sin(2 * np.pi * X)
plt.plot(X, Y, lw=2)
plt.ylim(-1.1, 1.1)
plt.grid()

plt.show()
```

Total running time of the script: (0 minutes 0.051 seconds)

Plotting a scatter of points

A simple example showing how to plot a scatter of points with matplotlib.

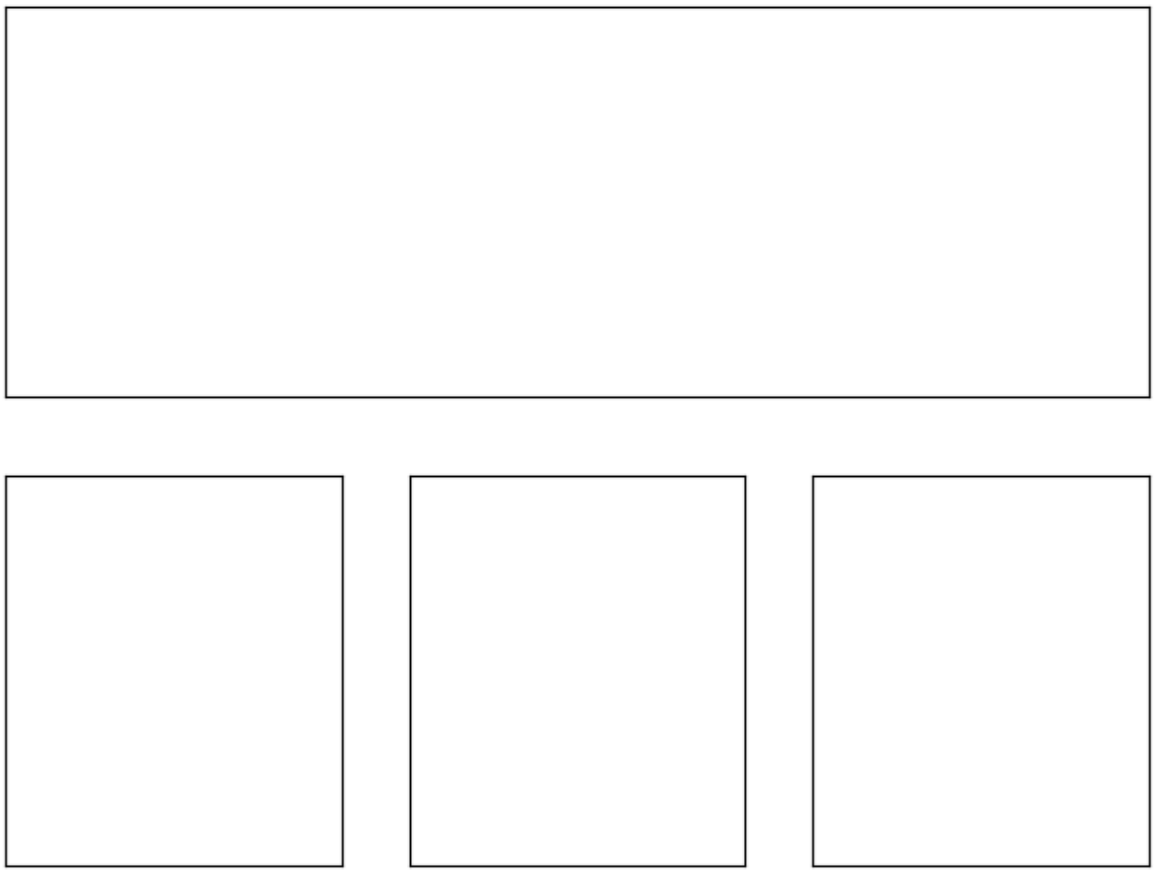


```
import numpy as np
import matplotlib.pyplot as plt

n = 1024
rng = np.random.default_rng()
X = rng.normal(0, 1, n)
Y = rng.normal(0, 1, n)
T = np.arctan2(Y, X)

plt.axes((0.025, 0.025, 0.95, 0.95))
plt.scatter(X, Y, s=75, c=T, alpha=0.5)

plt.xlim(-1.5, 1.5)
plt.xticks([])
plt.ylim(-1.5, 1.5)
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.058 seconds)

Subplots

Show multiple subplots in matplotlib.

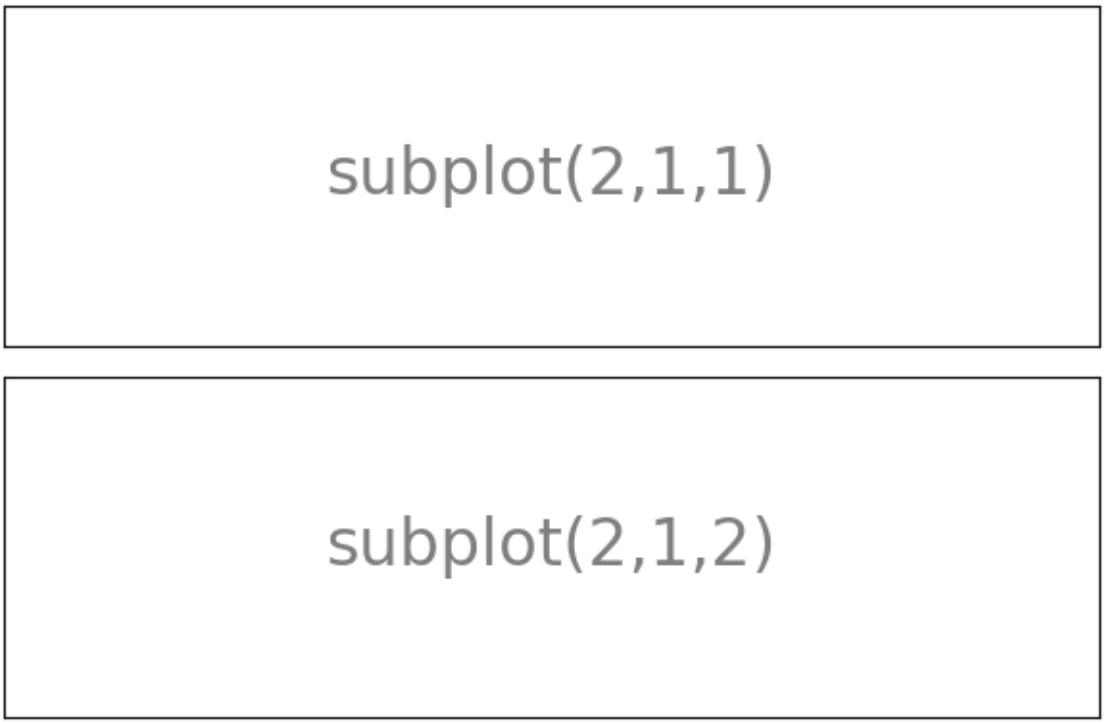


```
import matplotlib.pyplot as plt

fig = plt.figure()
fig.subplots_adjust(bottom=0.025, left=0.025, top=0.975, right=0.975)

plt.subplot(2, 1, 1)
plt.xticks([]), plt.yticks([])

plt.subplot(2, 3, 4)
plt.xticks([])
plt.yticks([])

plt.subplot(2, 3, 5)
plt.xticks([])
plt.yticks([])

plt.subplot(2, 3, 6)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.038 seconds)

Horizontal arrangement of subplots

An example showing horizontal arrangement of subplots with matplotlib.

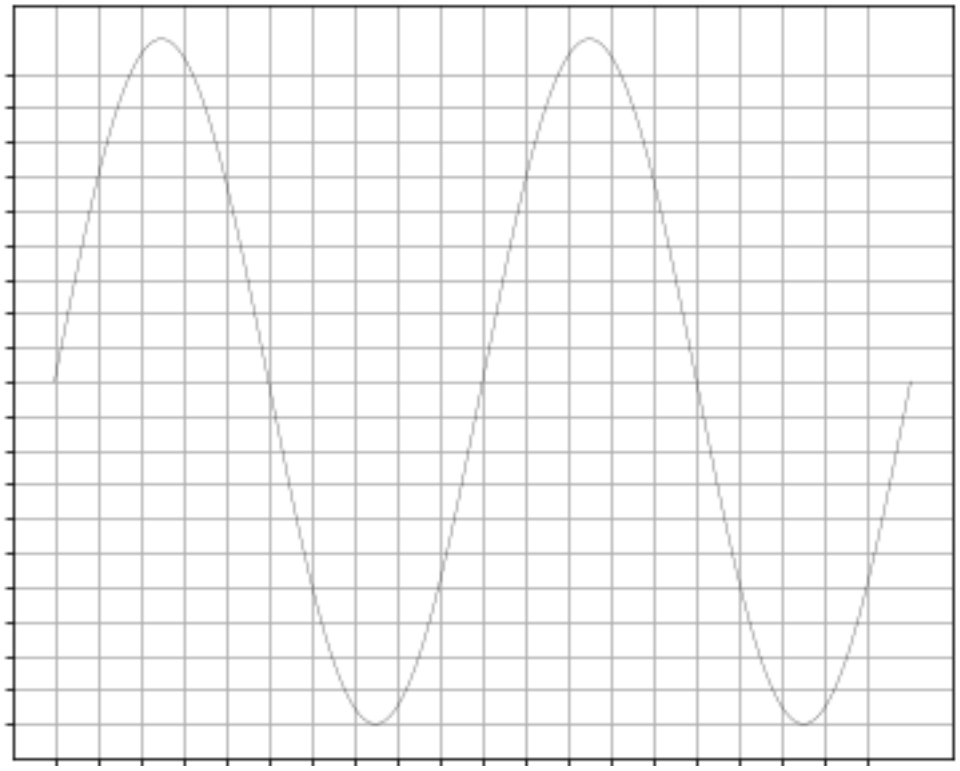



```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.subplot(2, 1, 1)
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "subplot(2,1,1)", ha="center", va="center", size=24, alpha=0.5)

plt.subplot(2, 1, 2)
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "subplot(2,1,2)", ha="center", va="center", size=24, alpha=0.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.041 seconds)

A simple plotting example

A plotting example with a few simple tweaks

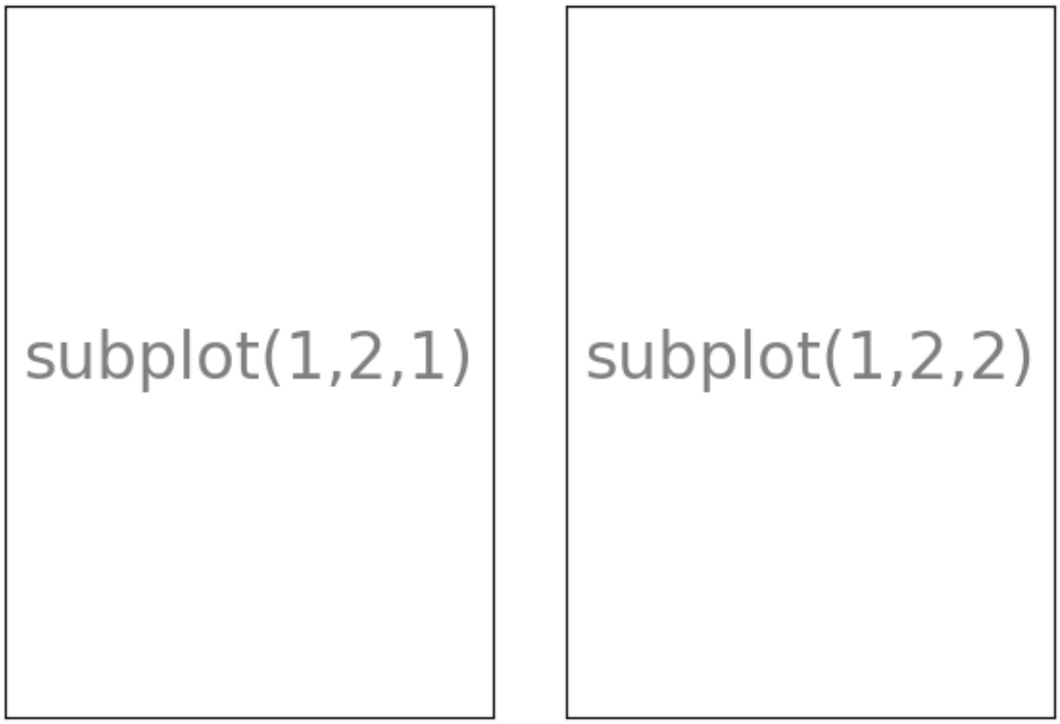


```
import numpy as np
import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5, 4), dpi=72)
axes = fig.add_axes((0.01, 0.01, 0.98, 0.98))
x = np.linspace(0, 2, 200)
y = np.sin(2 * np.pi * x)
plt.plot(x, y, lw=0.25, c="k")
plt.xticks(np.arange(0.0, 2.0, 0.1))
plt.yticks(np.arange(-1.0, 1.0, 0.1))
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.084 seconds)

Subplot plot arrangement vertical

An example showing vertical arrangement of subplots with matplotlib.

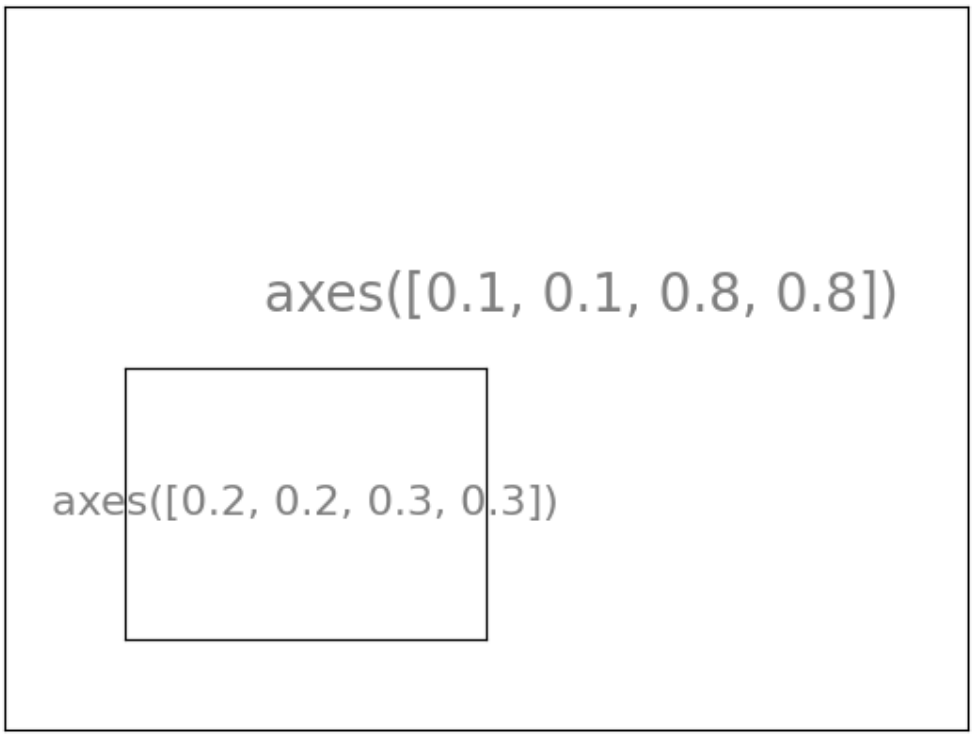



```
import matplotlib.pyplot as plt


plt.figure(figsize=(6, 4))
plt.subplot(1, 2, 1)
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "subplot(1,2,1)", ha="center", va="center", size=24, alpha=0.5)

plt.subplot(1, 2, 2)
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "subplot(1,2,2)", ha="center", va="center", size=24, alpha=0.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.036 seconds)

Simple axes example

This example shows a couple of simple usage of axes.

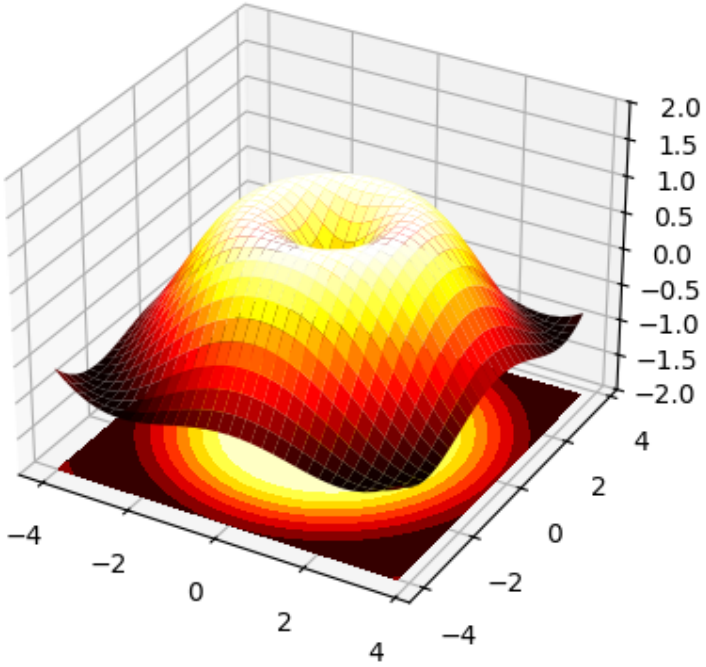



```
import matplotlib.pyplot as plt

plt.axes((0.1, 0.1, 0.8, 0.8))
plt.xticks([])
plt.yticks([])
plt.text(
    0.6, 0.6, "axes([0.1, 0.1, 0.8, 0.8])", ha="center", va="center", size=20,␣
↪alpha=0.5
)

plt.axes((0.2, 0.2, 0.3, 0.3))
plt.xticks([])
plt.yticks([])
plt.text(
    0.5, 0.5, "axes([0.2, 0.2, 0.3, 0.3])", ha="center", va="center", size=16,␣
↪alpha=0.5
)

plt.show()
```

Total running time of the script: (0 minutes 0.041 seconds)

3D plotting

A simple example of 3D plotting.

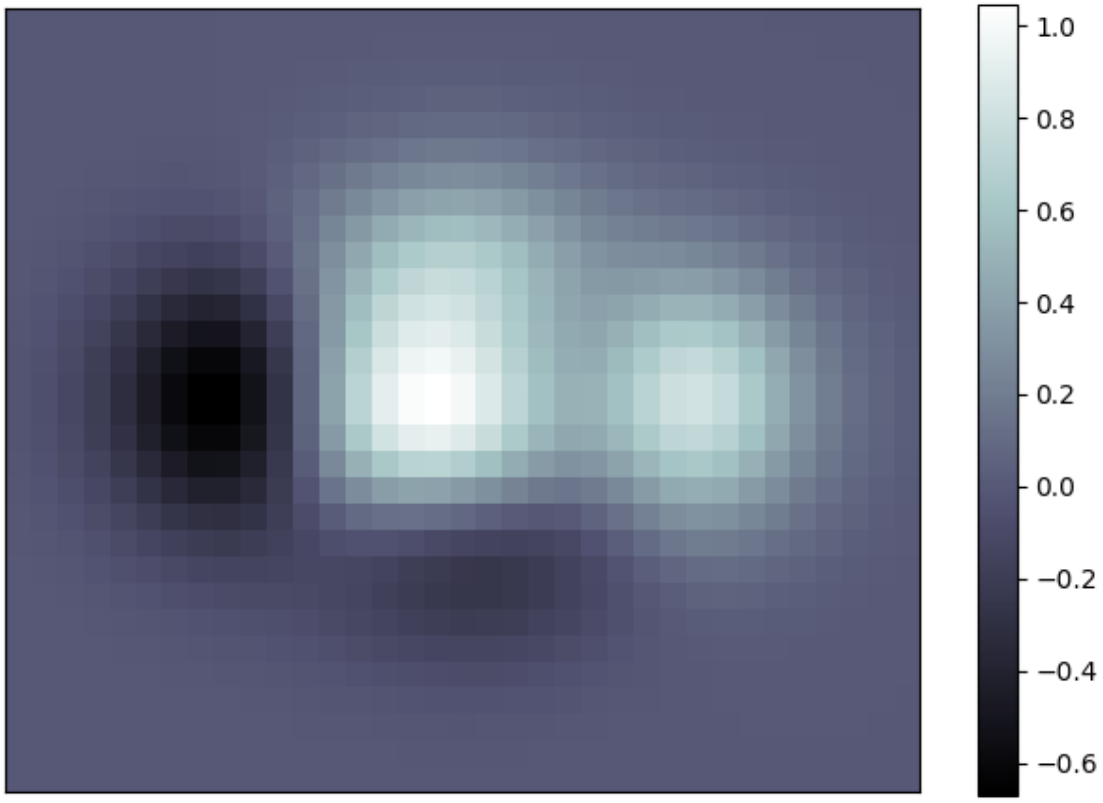



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

ax: Axes3D = plt.figure().add_subplot(projection="3d")
x = np.arange(-4, 4, 0.25)
y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap="hot")
ax.contourf(X, Y, Z, zdir="z", offset=-2, cmap="hot")
ax.set_zlim(-2, 2)

plt.show()
```

Total running time of the script: (0 minutes 0.141 seconds)

Imshow elaborate

An example demoing imshow and styling the figure.

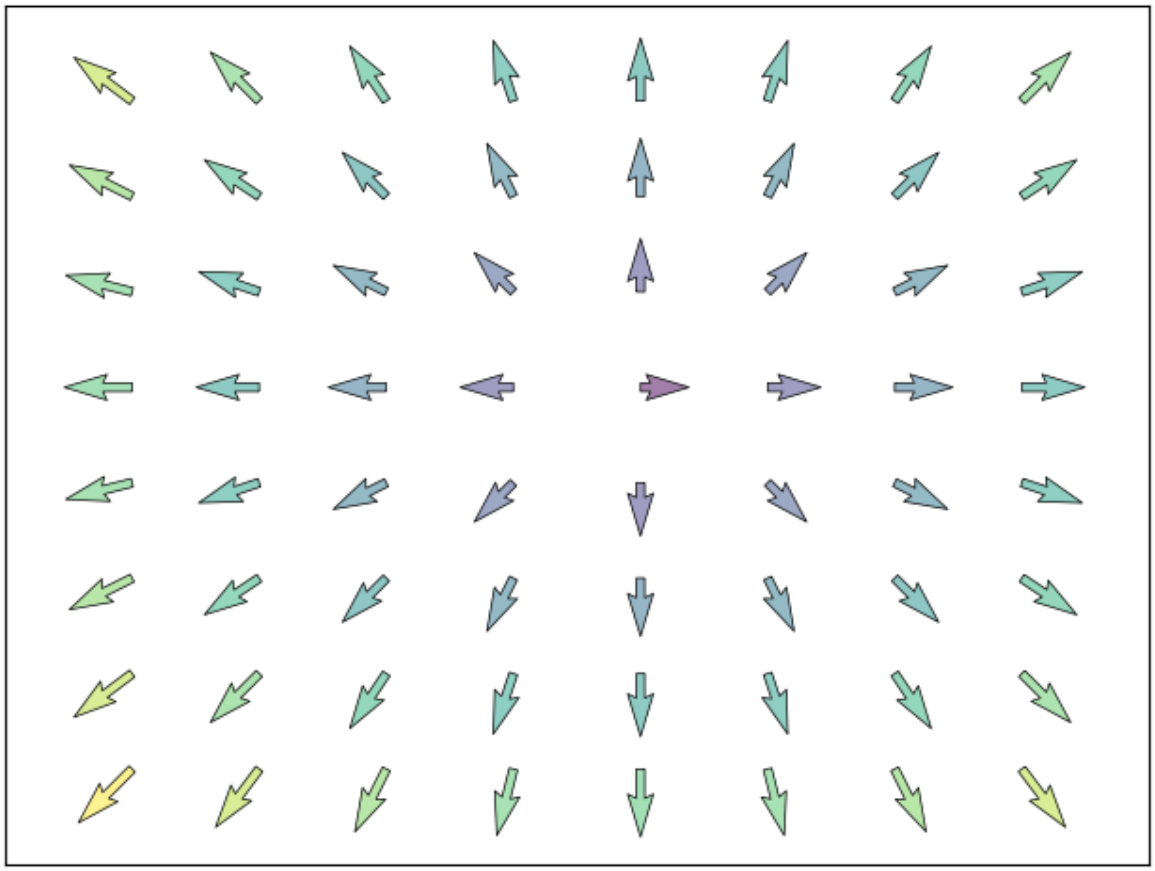



```
import numpy as np
import matplotlib.pyplot as plt


def f(x, y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-(x**2) - y**2)


n = 10
x = np.linspace(-3, 3, int(3.5 * n))
y = np.linspace(-3, 3, int(3.0 * n))
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

plt.axes((0.025, 0.025, 0.95, 0.95))
plt.imshow(Z, interpolation="nearest", cmap="bone", origin="lower")
plt.colorbar(shrink=0.92)

plt.xticks([])
plt.yticks([])
plt.show()
```

Total running time of the script: (0 minutes 0.076 seconds)

Plotting a vector field: quiver

A simple example showing how to plot a vector field (quiver) with matplotlib.

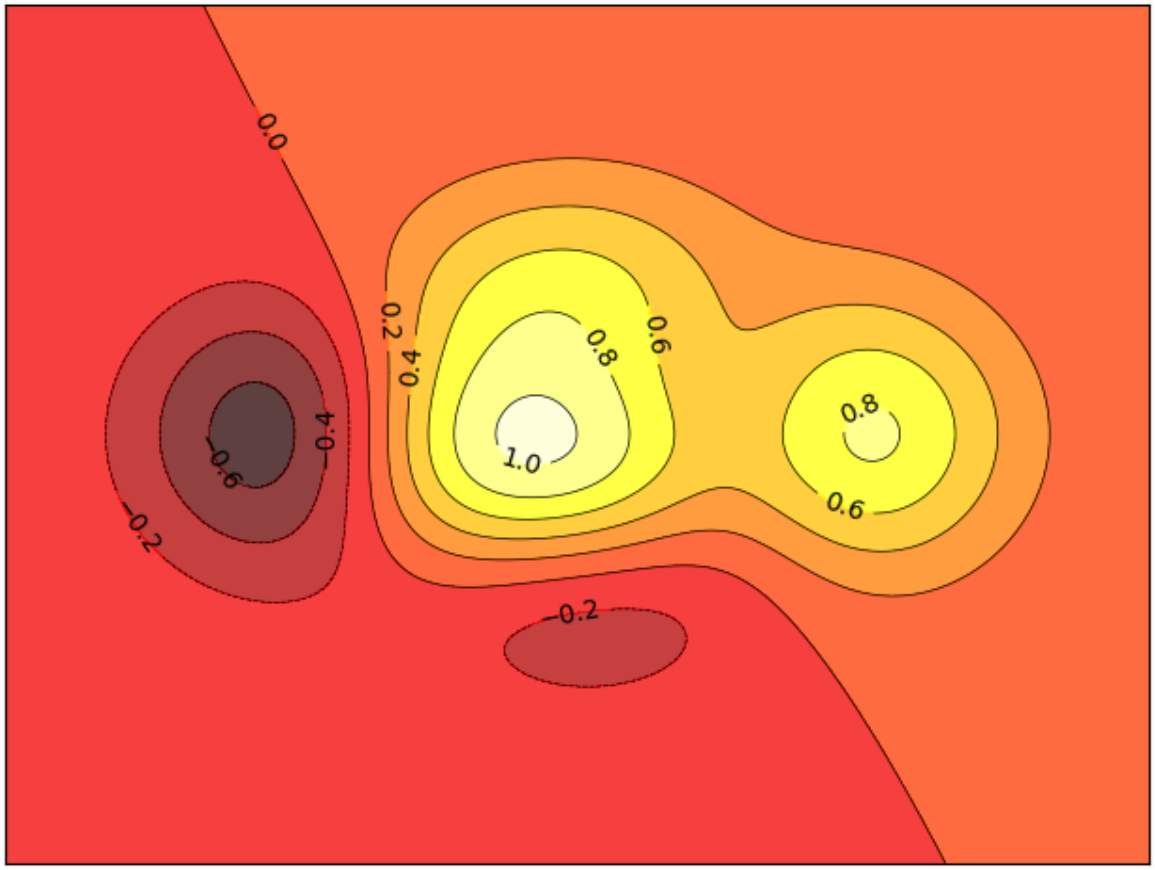


```
import numpy as np
import matplotlib.pyplot as plt

n = 8
X, Y = np.mgrid[0:n, 0:n]
T = np.arctan2(Y - n / 2.0, X - n / 2.0)
R = 10 + np.sqrt((Y - n / 2.0) ** 2 + (X - n / 2.0) ** 2)
U, V = R * np.cos(T), R * np.sin(T)

plt.axes((0.025, 0.025, 0.95, 0.95))
plt.quiver(X, Y, U, V, R, alpha=0.5)
plt.quiver(X, Y, U, V, edgecolor="k", facecolor="None", linewidth=0.5)

plt.xlim(-1, n)
plt.xticks([])
plt.ylim(-1, n)
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.029 seconds)

Displaying the contours of a function

An example showing how to display the contours of a function with matplotlib.

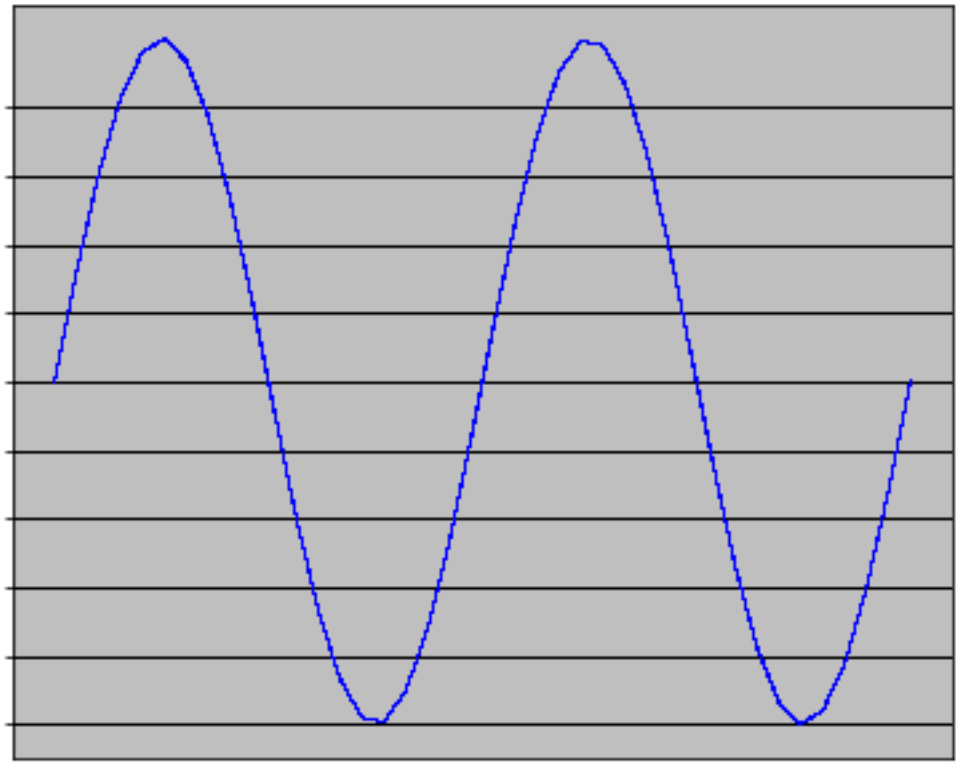



```
import numpy as np
import matplotlib.pyplot as plt


def f(x, y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-(x**2) - y**2)


n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)

plt.axes((0.025, 0.025, 0.95, 0.95))

plt.contourf(X, Y, f(X, Y), 8, alpha=0.75, cmap="hot")
C = plt.contour(X, Y, f(X, Y), 8, colors="black", linewidths=0.5)
plt.clabel(C, inline=1, fontsize=10)

plt.xticks([])
plt.yticks([])
plt.show()
```

Total running time of the script: (0 minutes 0.074 seconds)

A example of plotting not quite right

An “ugly” example of plotting.

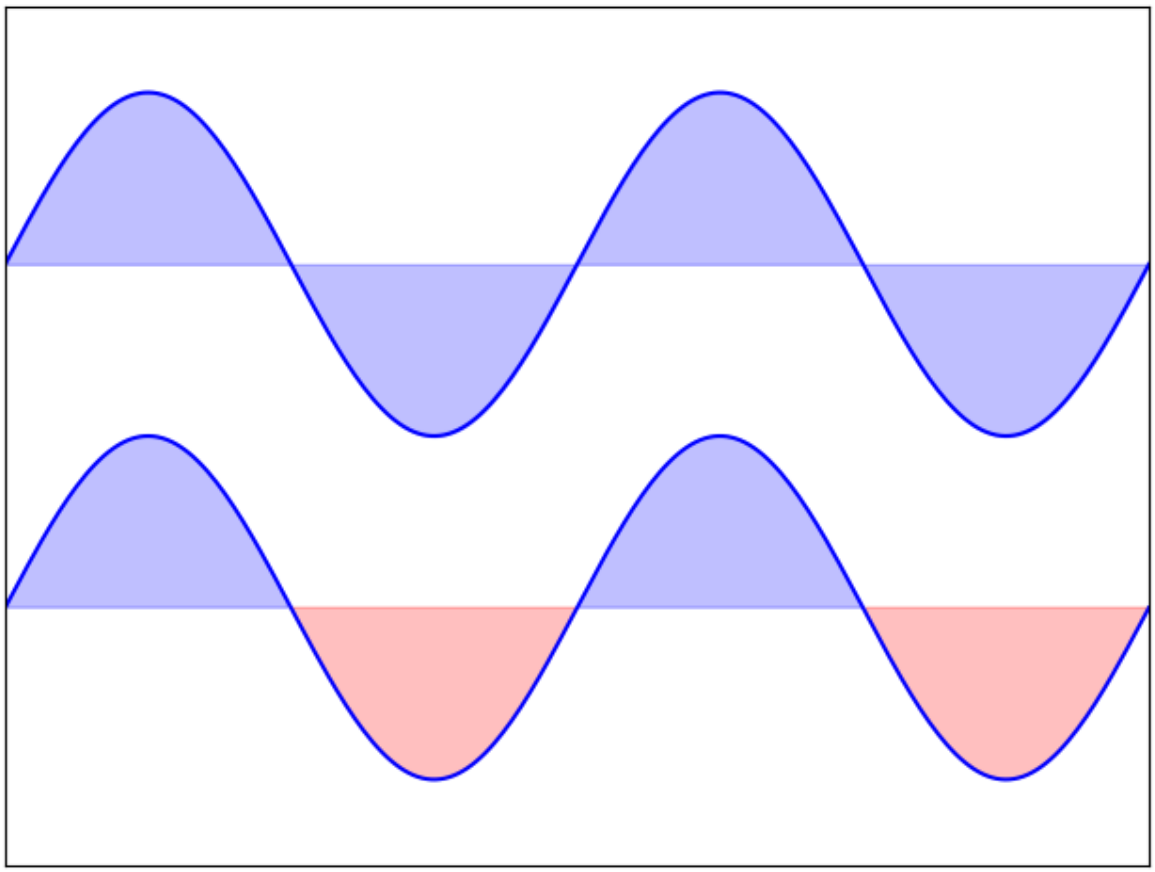


```
import numpy as np
import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt

matplotlib.rc("grid", color="black", linestyle="-", linewidth=1)

fig = plt.figure(figsize=(5, 4), dpi=72)
axes = fig.add_axes((0.01, 0.01, 0.98, 0.98), facecolor=".75")
X = np.linspace(0, 2, 40)
Y = np.sin(2 * np.pi * X)
plt.plot(X, Y, lw=0.05, c="b", antialiased=False)

plt.xticks([])
plt.yticks(np.arange(-1.0, 1.0, 0.2))
plt.grid()
ax = plt.gca()

plt.show()
```

Total running time of the script: (0 minutes 0.033 seconds)

Plot and filled plots

Simple example of plots and filling between them with matplotlib.

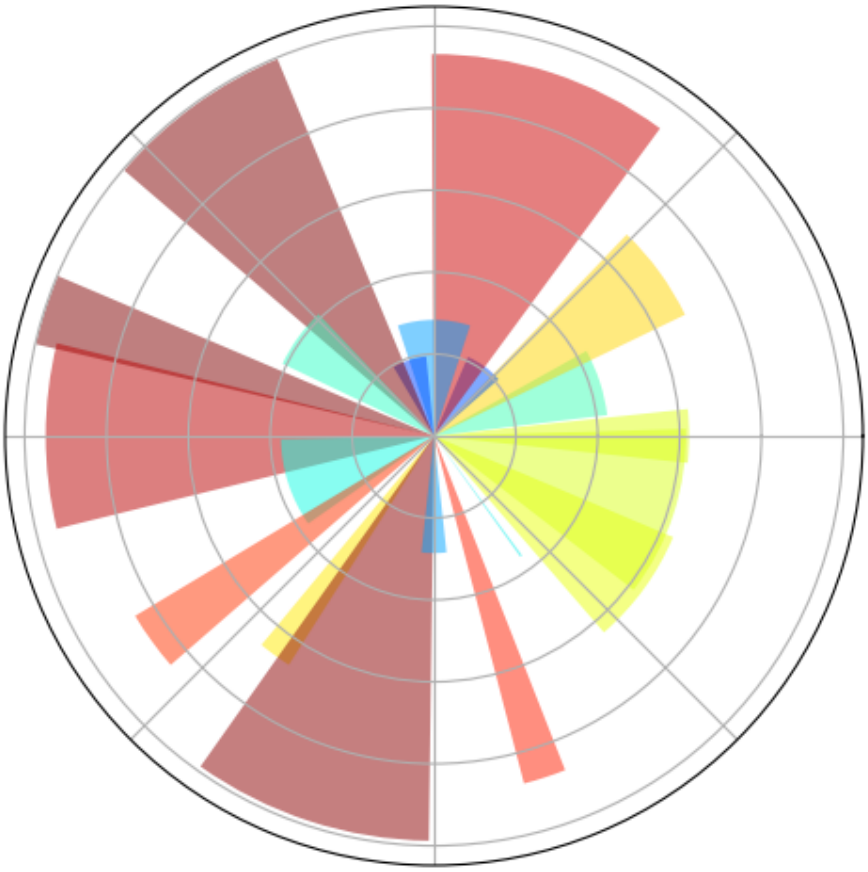


```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi, np.pi, n)
Y = np.sin(2 * X)

plt.axes((0.025, 0.025, 0.95, 0.95))

plt.plot(X, Y + 1, color="blue", alpha=1.00)
plt.fill_between(X, 1, Y + 1, color="blue", alpha=0.25)

plt.plot(X, Y - 1, color="blue", alpha=1.00)
plt.fill_between(X, -1, Y - 1, (Y - 1) > -1, color="blue", alpha=0.25)
plt.fill_between(X, -1, Y - 1, (Y - 1) < -1, color="red", alpha=0.25)

plt.xlim(-np.pi, np.pi)
plt.xticks([])
plt.ylim(-2.5, 2.5)
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.031 seconds)

Plotting in polar coordinates

A simple example showing how to plot in polar coordinates with matplotlib.

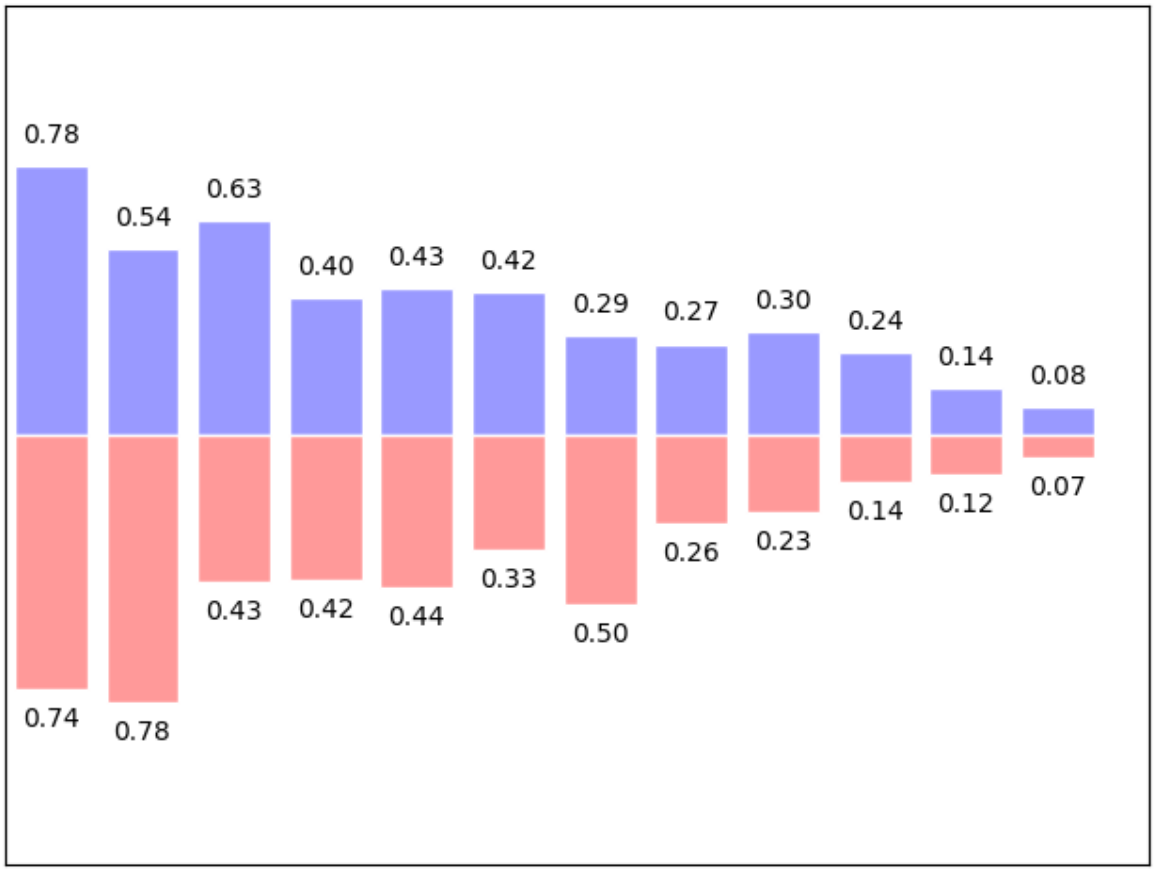


```
import numpy as np

import matplotlib
import matplotlib.pyplot as plt


jet = matplotlib.colormaps["jet"]

ax = plt.axes((0.025, 0.025, 0.95, 0.95), polar=True)

N = 20
theta = np.arange(0.0, 2 * np.pi, 2 * np.pi / N)
rng = np.random.default_rng()
radii = 10 * rng.random(N)
width = np.pi / 4 * rng.random(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars, strict=True):
    bar.set_facecolor(jet(r / 10.0))
    bar.set_alpha(0.5)

ax.set_xticklabels([])
ax.set_yticklabels([])
plt.show()
```

Total running time of the script: (0 minutes 0.082 seconds)

Bar plots

An example of bar plots with matplotlib.




```
import numpy as np
import matplotlib.pyplot as plt

n = 12
X = np.arange(n)
rng = np.random.default_rng()
Y1 = (1 - X / n) * rng.uniform(0.5, 1.0, n)
Y2 = (1 - X / n) * rng.uniform(0.5, 1.0, n)

plt.axes((0.025, 0.025, 0.95, 0.95))
plt.bar(X, +Y1, facecolor="#9999ff", edgecolor="white")
plt.bar(X, -Y2, facecolor="#ff9999", edgecolor="white")

for x, y in zip(X, Y1):
    plt.text(x, y + 0.05, f"{y:.2f}", ha="center", va="bottom")

for x, y in zip(X, Y2):
    plt.text(x, -y - 0.05, f"{y:.2f}", ha="center", va="top")

plt.xlim(-0.5, n)
plt.xticks([])
plt.ylim(-1.25, 1.25)
plt.yticks([])
```

(continues on next page)

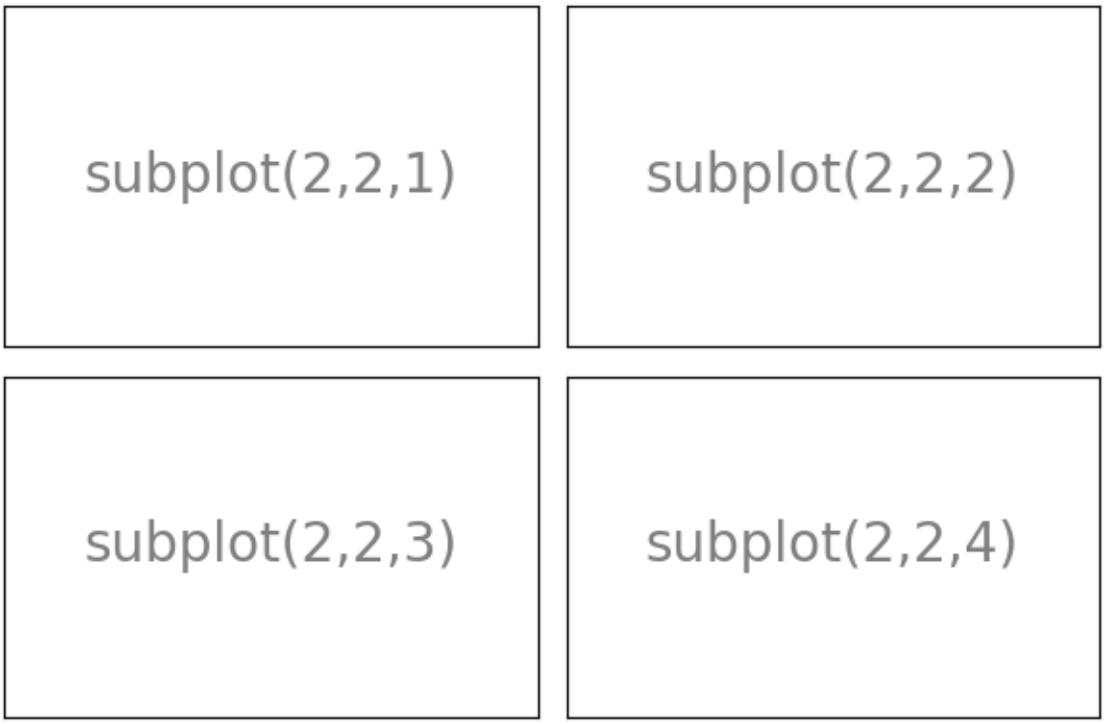
(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 0.056 seconds)

Subplot grid

An example showing the subplot grid in matplotlib.




```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.subplot(2, 2, 1)
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "subplot(2,2,1)", ha="center", va="center", size=20, alpha=0.5)

plt.subplot(2, 2, 2)
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "subplot(2,2,2)", ha="center", va="center", size=20, alpha=0.5)

plt.subplot(2, 2, 3)
plt.xticks([])
plt.yticks([])

plt.text(0.5, 0.5, "subplot(2,2,3)", ha="center", va="center", size=20, alpha=0.5)

plt.subplot(2, 2, 4)
plt.xticks([])
```

(continues on next page)

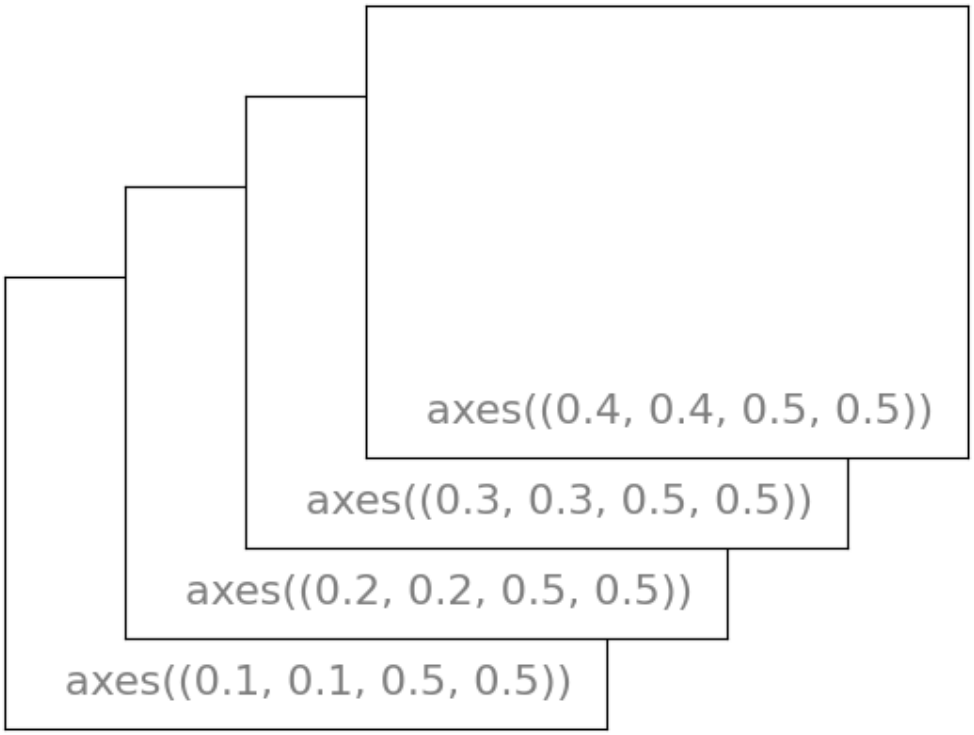
(continued from previous page)

```
plt.yticks([])
plt.text(0.5, 0.5, "subplot(2,2,4)", ha="center", va="center", size=20, alpha=0.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.058 seconds)

Axes

This example shows various axes command to position matplotlib axes.




```
import matplotlib.pyplot as plt

plt.axes((0.1, 0.1, 0.5, 0.5))
plt.xticks([])
plt.yticks([])
plt.text(
    0.1, 0.1, "axes((0.1, 0.1, 0.5, 0.5))", ha="left", va="center", size=16, alpha=0.5
)

plt.axes((0.2, 0.2, 0.5, 0.5))
plt.xticks([])
plt.yticks([])
plt.text(
    0.1, 0.1, "axes((0.2, 0.2, 0.5, 0.5))", ha="left", va="center", size=16, alpha=0.5
```

(continues on next page)

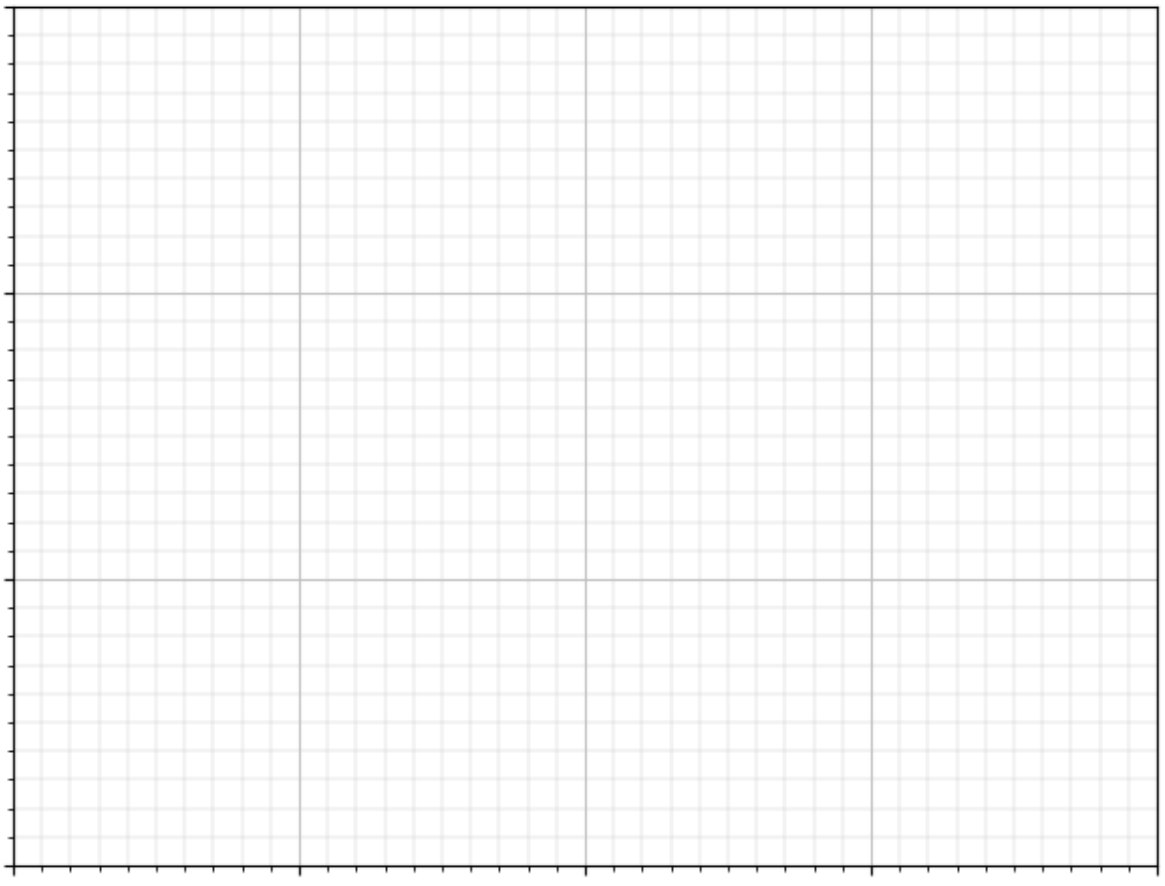
(continued from previous page)

```
)

plt.axes((0.3, 0.3, 0.5, 0.5))
plt.xticks([])
plt.yticks([])
plt.text(
    0.1, 0.1, "axes((0.3, 0.3, 0.5, 0.5))", ha="left", va="center", size=16, alpha=0.5
)

plt.axes((0.4, 0.4, 0.5, 0.5))
plt.xticks([])
plt.yticks([])
plt.text(
    0.1, 0.1, "axes((0.4, 0.4, 0.5, 0.5))", ha="left", va="center", size=16, alpha=0.5
)

plt.show()
```

Total running time of the script: (0 minutes 0.046 seconds)

Grid

Displaying a grid on the axes in matploblib.



```
import matplotlib.pyplot as plt
from matplotlib import ticker
```

(continues on next page)

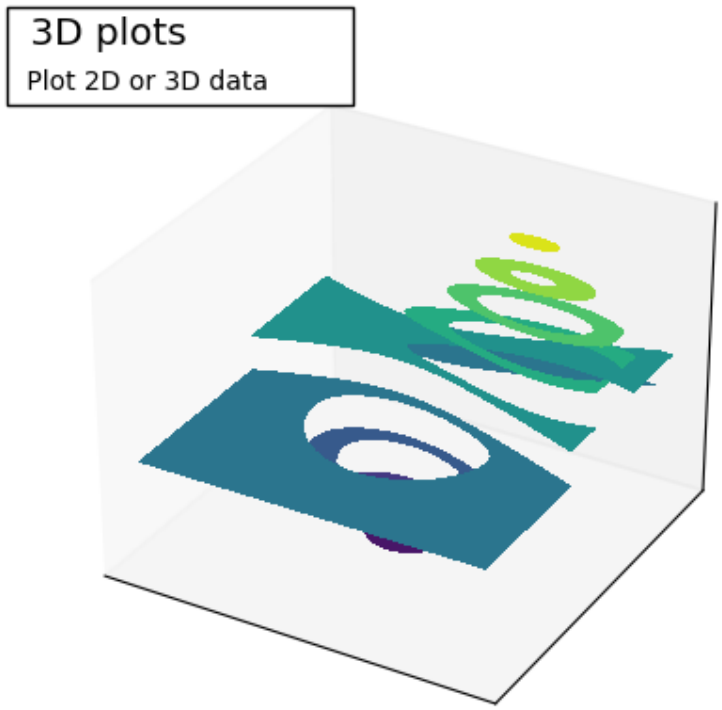
(continued from previous page)

```
ax = plt.axes((0.025, 0.025, 0.95, 0.95))

ax.set_xlim(0, 4)
ax.set_ylim(0, 3)
ax.xaxis.set_major_locator(ticker.MultipleLocator(1.0))
ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1.0))
ax.yaxis.set_minor_locator(ticker.MultipleLocator(0.1))
ax.grid(which="major", axis="x", linewidth=0.75, linestyle="-", color="0.75")
ax.grid(which="minor", axis="x", linewidth=0.25, linestyle="-", color="0.75")
ax.grid(which="major", axis="y", linewidth=0.75, linestyle="-", color="0.75")
ax.grid(which="minor", axis="y", linewidth=0.25, linestyle="-", color="0.75")
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
```

Total running time of the script: (0 minutes 0.097 seconds)

3D plotting

Demo 3D plotting with matplotlib and style the figure.




```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D, get_test_data
```

(continues on next page)

(continued from previous page)

```
ax: Axes3D = plt.figure().add_subplot(projection="3d")
X, Y, Z = get_test_data(0.05)
cset = ax.contourf(X, Y, Z)
ax.clabel(cset, fontsize=9, inline=1)

plt.xticks([])
plt.yticks([])
ax.set_zticks([])


ax.text2D(
    -0.05,
    1.05,
    " 3D plots              \n",
    horizontalalignment="left",
    verticalalignment="top",
    bbox={"facecolor": "white", "alpha": 1.0},
    family="DejaVu Sans",
    size="x-large",
    transform=plt.gca().transAxes,
)

ax.text2D(
    -0.05,
    0.975,
    " Plot 2D or 3D data",
    horizontalalignment="left",
    verticalalignment="top",
    family="DejaVu Sans",
    size="medium",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.049 seconds)

GridSpec

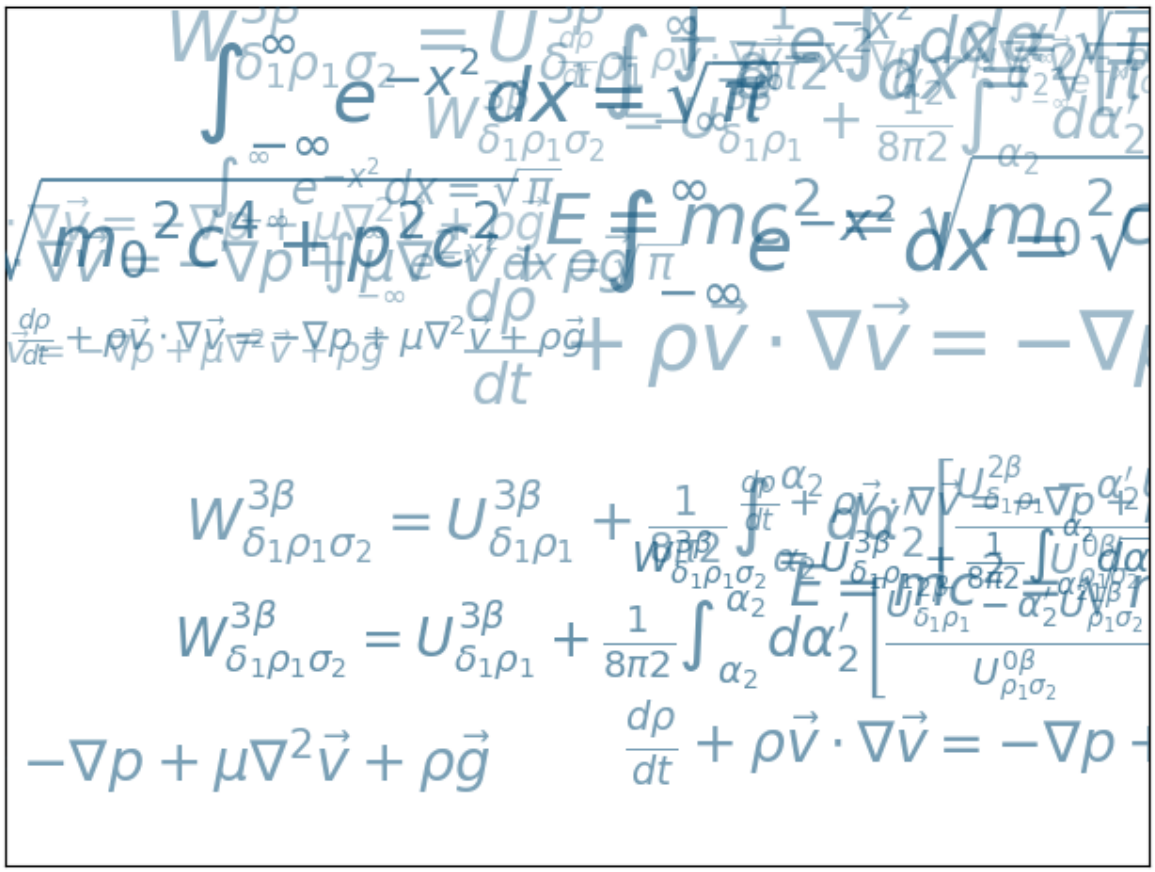
An example demoing gridspec

```
import matplotlib.pyplot as plt
from matplotlib import gridspec

plt.figure(figsize=(6, 4))
G = gridspec.GridSpec(3, 3)

axes_1 = plt.subplot(G[0, :])
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "Axes 1", ha="center", va="center", size=24, alpha=0.5)

axes_2 = plt.subplot(G[1, :-1])
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "Axes 2", ha="center", va="center", size=24, alpha=0.5)

axes_3 = plt.subplot(G[1:, -1])
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "Axes 3", ha="center", va="center", size=24, alpha=0.5)

axes_4 = plt.subplot(G[-1, 0])
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "Axes 4", ha="center", va="center", size=24, alpha=0.5)

axes_5 = plt.subplot(G[-1, -2])
plt.xticks([])
plt.yticks([])
plt.text(0.5, 0.5, "Axes 5", ha="center", va="center", size=24, alpha=0.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.068 seconds)

Demo text printing

A example showing off elaborate text printing with matplotlib.



```
import numpy as np
import matplotlib.pyplot as plt


eqs = []
eqs.append(
    r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}
↪{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\
↪delta_1 \rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \
↪sigma_2}}\right]$"
)
eqs.append(
    r"$\frac{d\rho}{d t} + \rho \vec{v}\cdot\nabla\vec{v} = -\nabla p + \mu\nabla^2 \
↪vec{v} + \rho \vec{g}$"
)
eqs.append(r"$\int_{-\infty}^\infty e^{-x^2}dx=\sqrt{\pi}$")
eqs.append(r"$E = mc^2 = \sqrt{{m_0}^2c^4 + p^2c^2}$")
eqs.append(r"$F_G = G\frac{m_1m_2}{r^2}$")

plt.axes((0.025, 0.025, 0.95, 0.95))
```

(continues on next page)

(continued from previous page)

```
rng = np.random.default_rng()

for i in range(24):
    index = rng.integers(0, len(eqs))
    eq = eqs[index]
    size = np.random.uniform(12, 32)
    x, y = np.random.uniform(0, 1, 2)
    alpha = np.random.uniform(0.25, 0.75)
    plt.text(
        x,
        y,
        eq,
        ha="center",
        va="center",
        color="#11557c",
        alpha=alpha,
        transform=plt.gca().transAxes,
        fontsize=size,
        clip_on=True,
    )
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.698 seconds)

4.7.2 Code for the chapter's exercises

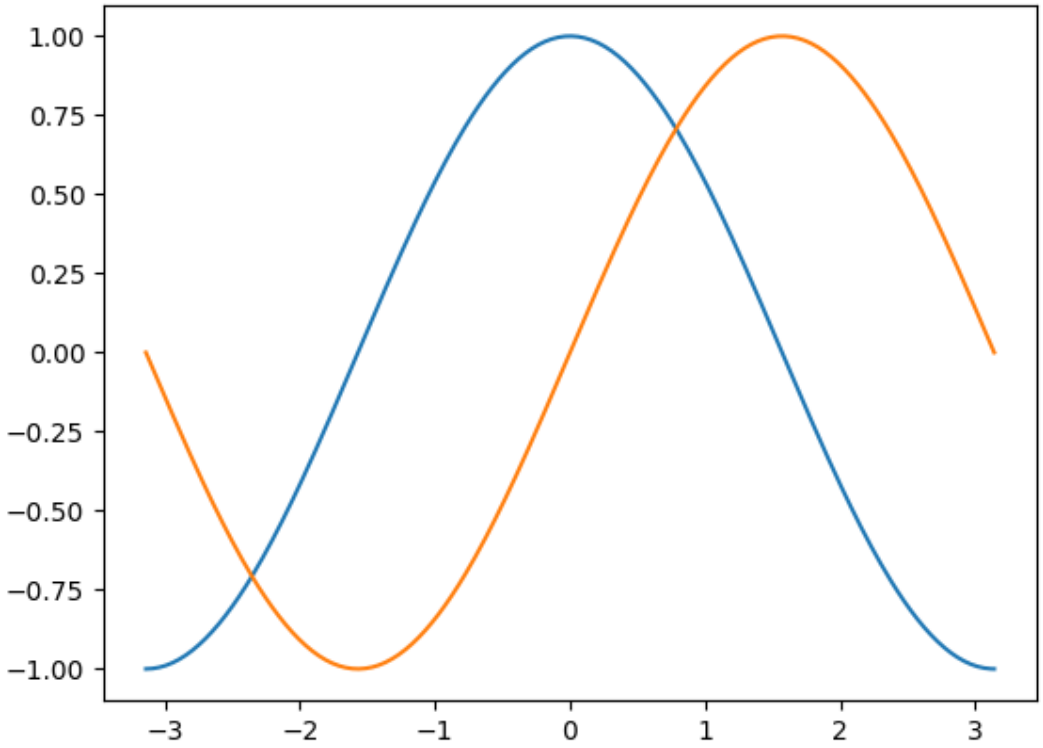
4.7.3 Example demoing choices for an option

4.7.4 Code generating the summary figures with a title

Code for the chapter's exercises

Exercise 1

Solution of the exercise 1 with matplotlib.

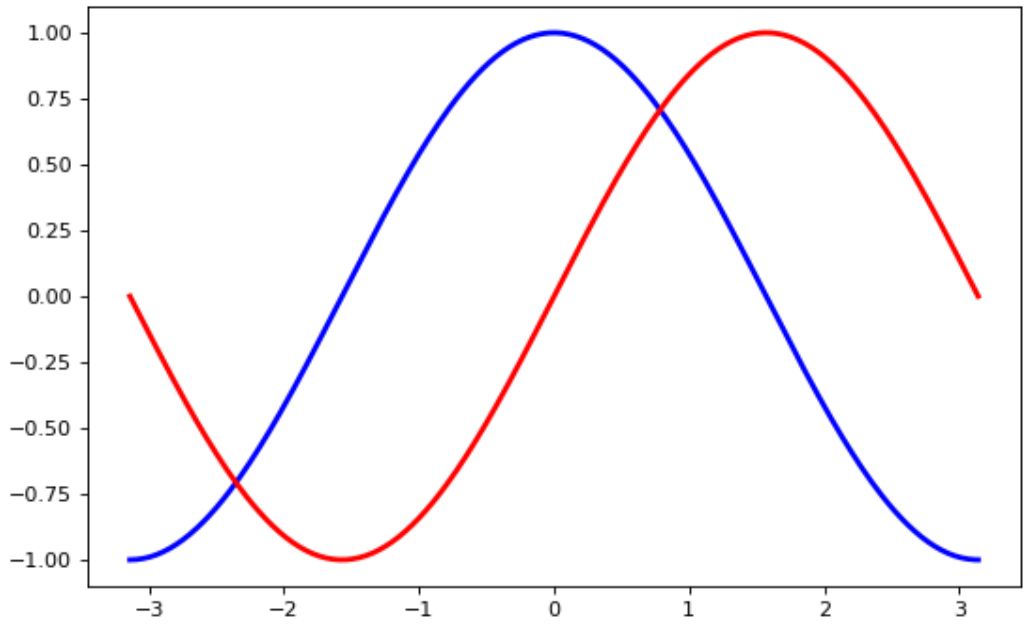



```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)
plt.plot(X, C)
plt.plot(X, S)

plt.show()
```

Total running time of the script: (0 minutes 0.055 seconds)

Exercise 4

Exercise 4 with matplotlib.

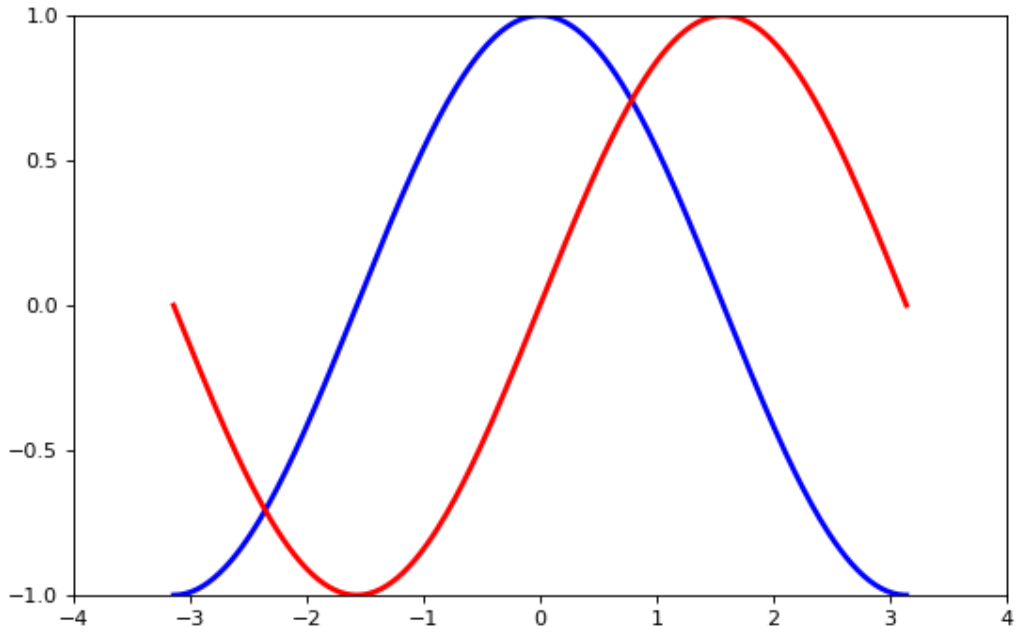



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256)
S = np.sin(X)
C = np.cos(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)

plt.show()
```

Total running time of the script: (0 minutes 0.052 seconds)

Exercise 3

Exercise 3 with matplotlib.

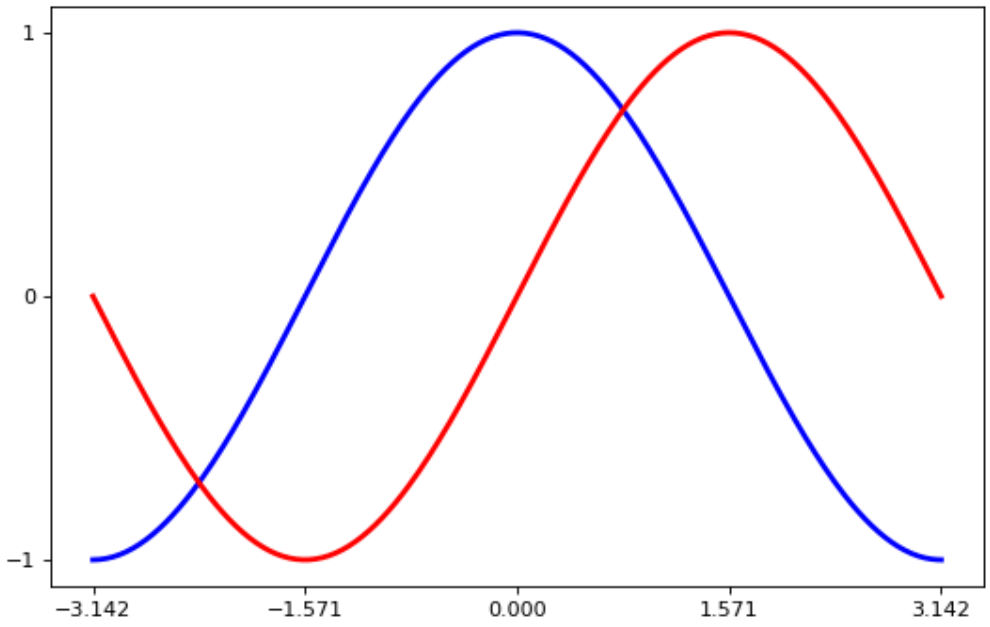



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(-4.0, 4.0)
plt.xticks(np.linspace(-4, 4, 9))

plt.ylim(-1.0, 1.0)
plt.yticks(np.linspace(-1, 1, 5))

plt.show()
```

Total running time of the script: (0 minutes 0.045 seconds)

Exercise 5

Exercise 5 with matplotlib.

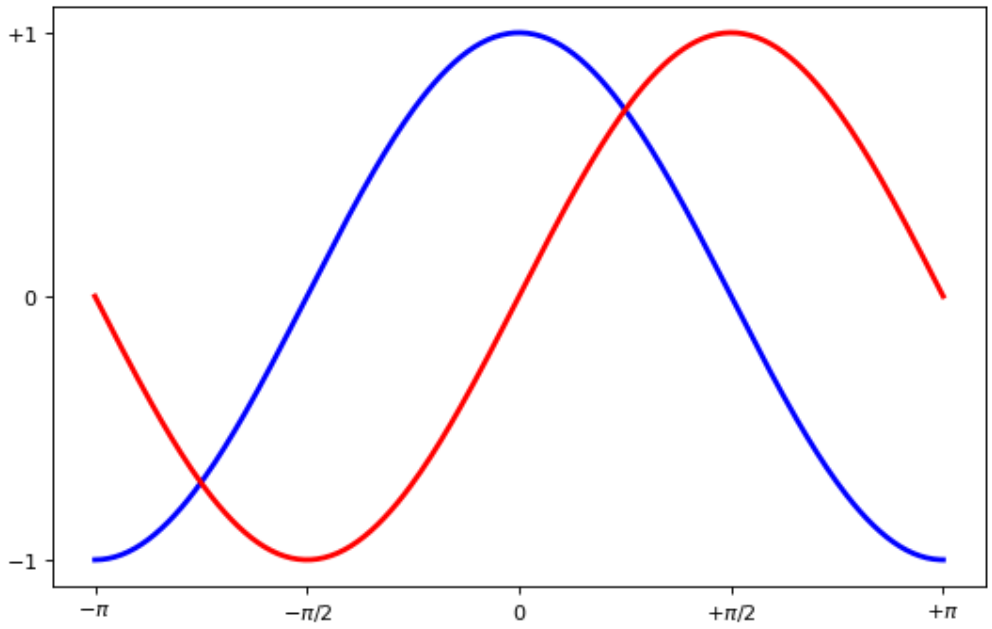



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256)
S = np.sin(X)
C = np.cos(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi])

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 0, +1])

plt.show()
```

Total running time of the script: (0 minutes 0.036 seconds)

Exercise 6

Exercise 6 with matplotlib.

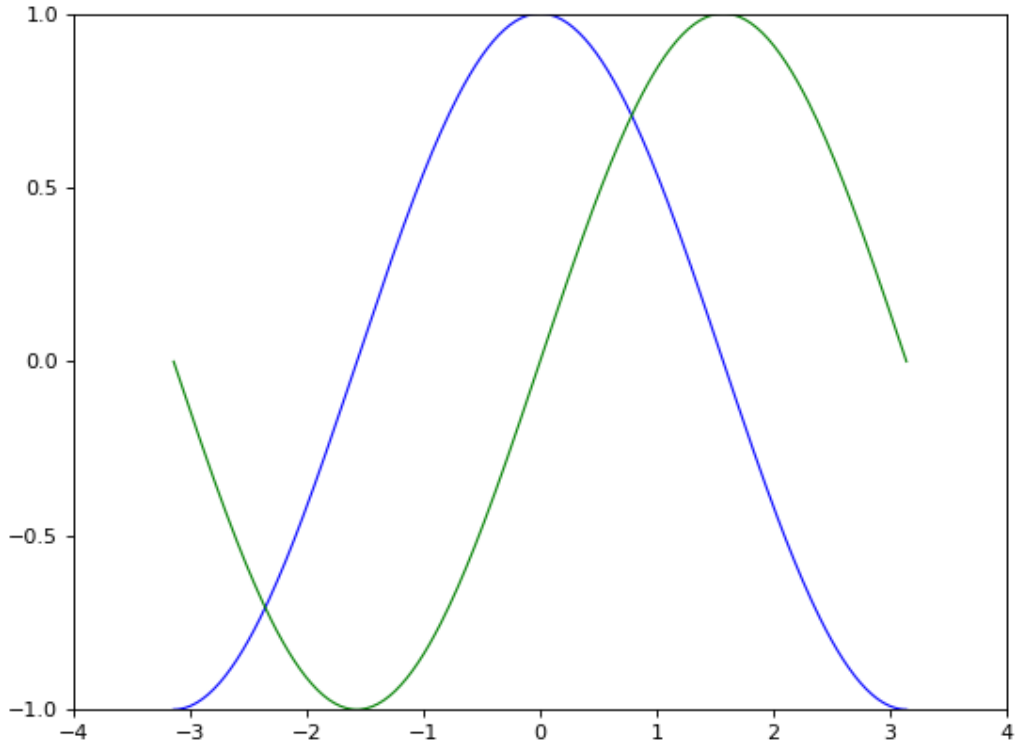



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks(
    [-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi],
    [r"$-\pi$", r"$-\pi/2$", r"$0$", r"$+\pi/2$", r"$+\pi$"],
)

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 0, +1], [r"$-1$", r"$0$", r"$+1$"])

plt.show()
```

Total running time of the script: (0 minutes 0.048 seconds)

Exercise 2

Exercise 2 with matplotlib.




```
import numpy as np
import matplotlib.pyplot as plt

# Create a new figure of size 8x6 points, using 100 dots per inch
plt.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)

# Plot cosine using blue color with a continuous line of width 1 (pixels)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine using green color with a continuous line of width 1 (pixels)
plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
plt.xlim(-4.0, 4.0)

# Set x ticks
plt.xticks(np.linspace(-4, 4, 9))

# Set y limits
```

(continues on next page)

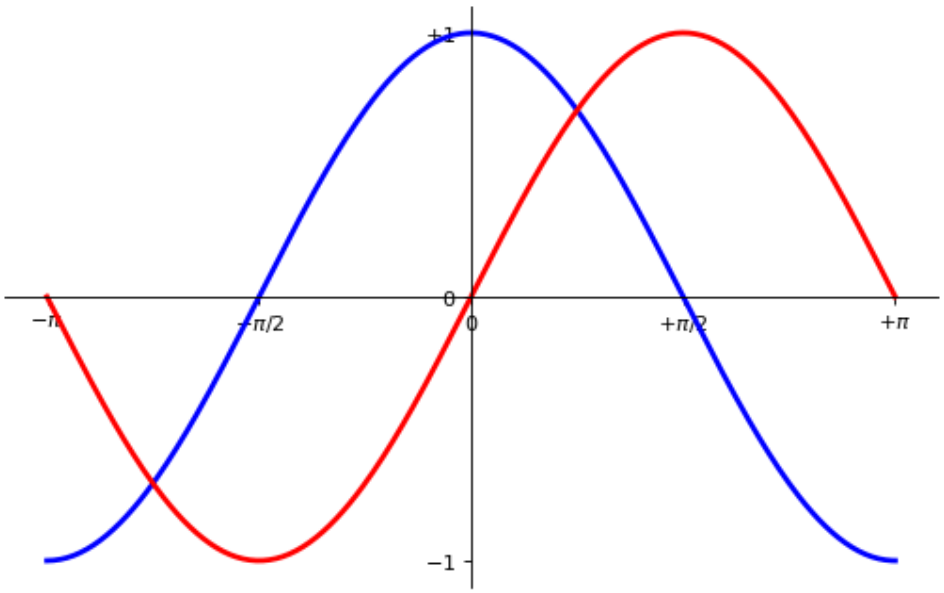
(continued from previous page)

```
plt.ylim(-1.0, 1.0)

# Set y ticks
plt.yticks(np.linspace(-1, 1, 5))

# Show result on screen
plt.show()
```

Total running time of the script: (0 minutes 0.048 seconds)

Exercise 7

Exercise 7 with matplotlib




```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

ax = plt.gca()
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
```

(continues on next page)

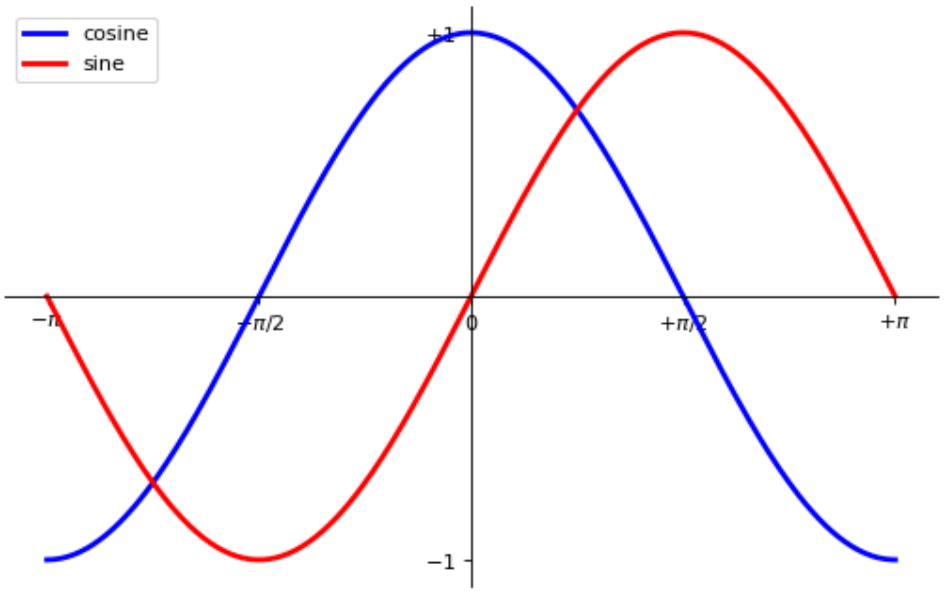
(continued from previous page)

```
ax.xaxis.set_ticks_position("bottom")
ax.spines["bottom"].set_position(("data", 0))
ax.yaxis.set_ticks_position("left")
ax.spines["left"].set_position(("data", 0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks(
    [-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi],
    [r"$-\pi$", r"$-\pi/2$", r"$0$", r"$+\pi/2$", r"$+\pi$"],
)

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 0, +1], [r"$-1$", r"$0$", r"$+1$"])

plt.show()
```

Total running time of the script: (0 minutes 0.057 seconds)

Exercise 8

Exercise 8 with matplotlib.




```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C = np.cos(X)
```

(continues on next page)

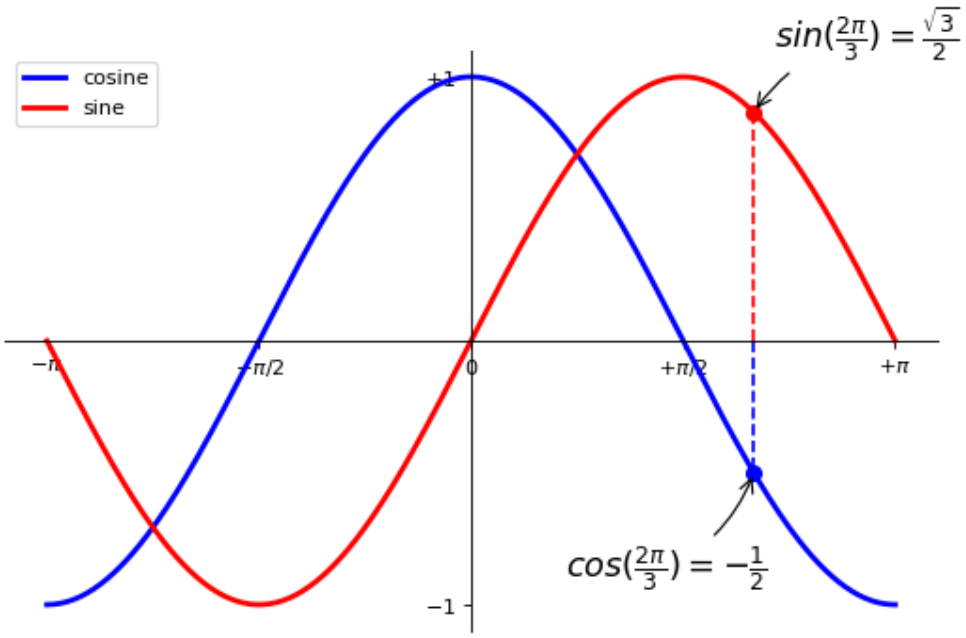
(continued from previous page)

```
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-", label="sine")

ax = plt.gca()
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
ax.xaxis.set_ticks_position("bottom")
ax.spines["bottom"].set_position(("data", 0))
ax.yaxis.set_ticks_position("left")
ax.spines["left"].set_position(("data", 0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks(
    [-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi],
    [r"$-\pi$", r"$-\pi/2$", r"$0$", r"$+\pi/2$", r"$+\pi$"],
)

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, +1], [r"$-1$", r"$+1$"])

plt.legend(loc="upper left")

plt.show()
```

Total running time of the script: (0 minutes 0.058 seconds)

Exercise 9

Exercise 9 with matplotlib.



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-", label="sine")

ax = plt.gca()
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
ax.xaxis.set_ticks_position("bottom")
ax.spines["bottom"].set_position(("data", 0))
ax.yaxis.set_ticks_position("left")
ax.spines["left"].set_position(("data", 0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks(
    [-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi],
    [r"$-\pi$", r"$-\pi/2$", r"$0$", r"$+\pi/2$", r"$+\pi$"],
)

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, +1], [r"$-1$", r"$+1$"])

t = 2 * np.pi / 3
plt.plot([t, t], [0, np.cos(t)], color="blue", linewidth=1.5, linestyle="--")
plt.scatter(
    [
        t,
    ],
    [
        np.cos(t),
    ],
    50,
    color="blue",
)
plt.annotate(
    r"$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$",
    xy=(t, np.sin(t)),
    xycoords="data",
    xytext=(+10, +30),
    textcoords="offset points",
    fontsize=16,
    arrowprops={"arrowstyle": "->", "connectionstyle": "arc3,rad=.2"},
)

plt.plot([t, t], [0, np.sin(t)], color="red", linewidth=1.5, linestyle="--")
plt.scatter(
    [
        t,
    ],
```

(continues on next page)

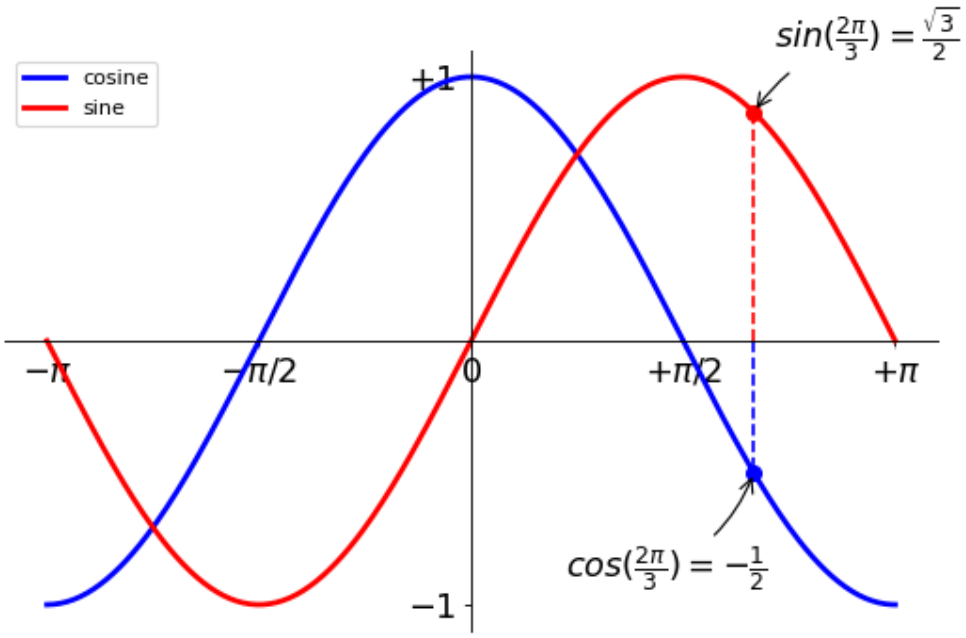
(continued from previous page)

```
    [
        np.sin(t),
    ],
    50,
    color="red",
)
plt.annotate(
    r"$cos(\frac{2\pi}{3})=-\frac{1}{2}$",
    xy=(t, np.cos(t)),
    xycoords="data",
    xytext=(-90, -50),
    textcoords="offset points",
    fontsize=16,
    arrowprops={"arrowstyle": "->", "connectionstyle": "arc3,rad=.2"},
)

plt.legend(loc="upper left")

plt.show()
```

Total running time of the script: (0 minutes 0.098 seconds)

Exercise

Exercises with matplotlib.




```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
```

(continues on next page)

(continued from previous page)

```
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-", label="sine")

ax = plt.gca()
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
ax.xaxis.set_ticks_position("bottom")
ax.spines["bottom"].set_position(("data", 0))
ax.yaxis.set_ticks_position("left")
ax.spines["left"].set_position(("data", 0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks(
    [-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi],
    [r"$-\pi$", r"$-\pi/2$", r"$0$", r"$+\pi/2$", r"$+\pi$"],
)

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 1], [r"$-1$", r"$+1$"])

plt.legend(loc="upper left")

t = 2 * np.pi / 3
plt.plot([t, t], [0, np.cos(t)], color="blue", linewidth=1.5, linestyle="--")
plt.scatter(
    [
        t,
    ],
    [
        np.cos(t),
    ],
    50,
    color="blue",
)
plt.annotate(
    r"$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$",
    xy=(t, np.sin(t)),
    xycoords="data",
    xytext=(10, 30),
    textcoords="offset points",
    fontsize=16,
    arrowprops={"arrowstyle": "->", "connectionstyle": "arc3,rad=.2"},
)

plt.plot([t, t], [0, np.sin(t)], color="red", linewidth=1.5, linestyle="--")
plt.scatter(
    [
        t,
    ],
    [
        np.sin(t),
```

(continues on next page)

(continued from previous page)

```
    ],
    50,
    color="red",
)
plt.annotate(
    r"$cos(\frac{2\pi}{3})=-\frac{1}{2}$",
    xy=(t, np.cos(t)),
    xycoords="data",
    xytext=(-90, -50),
    textcoords="offset points",
    fontsize=16,
    arrowprops={"arrowstyle": "->", "connectionstyle": "arc3,rad=.2"},
)

for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
    label.set_bbox({"facecolor": "white", "edgecolor": "None", "alpha": 0.65})

plt.show()
```

Total running time of the script: (0 minutes 0.095 seconds)

Example demoing choices for an option

The colors matplotlib line plots

An example demoing the various colors taken by matplotlib's plot.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0.1, 1, 0.8), frameon=False)

for i in range(1, 11):
    plt.plot([i, i], [0, 1], lw=1.5)

plt.xlim(0, 11)
plt.xticks([])
plt.yticks([])
plt.show()
```

Total running time of the script: (0 minutes 0.013 seconds)

Linewidth

Plot various linewidth with matplotlib.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0.1, 1, 0.8), frameon=False)

for i in range(1, 11):
    plt.plot([i, i], [0, 1], color="b", lw=i / 2.0)

plt.xlim(0, 11)
plt.ylim(0, 1)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.014 seconds)

Alpha: transparency

This example demonstrates using alpha for transparency.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0.1, 1, 0.8), frameon=False)

for i in range(1, 11):
    plt.axvline(i, linewidth=1, color="blue", alpha=0.25 + 0.75 * i / 10.0)

plt.xlim(0, 11)
plt.xticks([])
plt.yticks([])
plt.show()
```

Total running time of the script: (0 minutes 0.017 seconds)

Aliased versus anti-aliased

This example demonstrates aliased versus anti-aliased text.

```
import matplotlib.pyplot as plt

size = 128, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)

plt.axes((0, 0, 1, 1), frameon=False)

plt.rcParams["text.antialiased"] = False
plt.text(0.5, 0.5, "Aliased", ha="center", va="center")

plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.011 seconds)

Aliased versus anti-aliased

The example shows aliased versus anti-aliased text.

```
import matplotlib.pyplot as plt

size = 128, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

plt.rcParams["text.antialiased"] = True
plt.text(0.5, 0.5, "Anti-aliased", ha="center", va="center")

plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.010 seconds)

Marker size

Demo the marker size control in matplotlib.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

for i in range(1, 11):
    plt.plot(
        [
            i,
        ],
        [
            1,
        ],
        "s",
        markersize=i,
        markerfacecolor="w",
        markeredgewidth=0.5,
        markeredgecolor="k",
    )

plt.xlim(0, 11)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.014 seconds)

Marker edge width

Demo the marker edge widths of matplotlib's markers.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

for i in range(1, 11):
    plt.plot(
        [
```

(continues on next page)

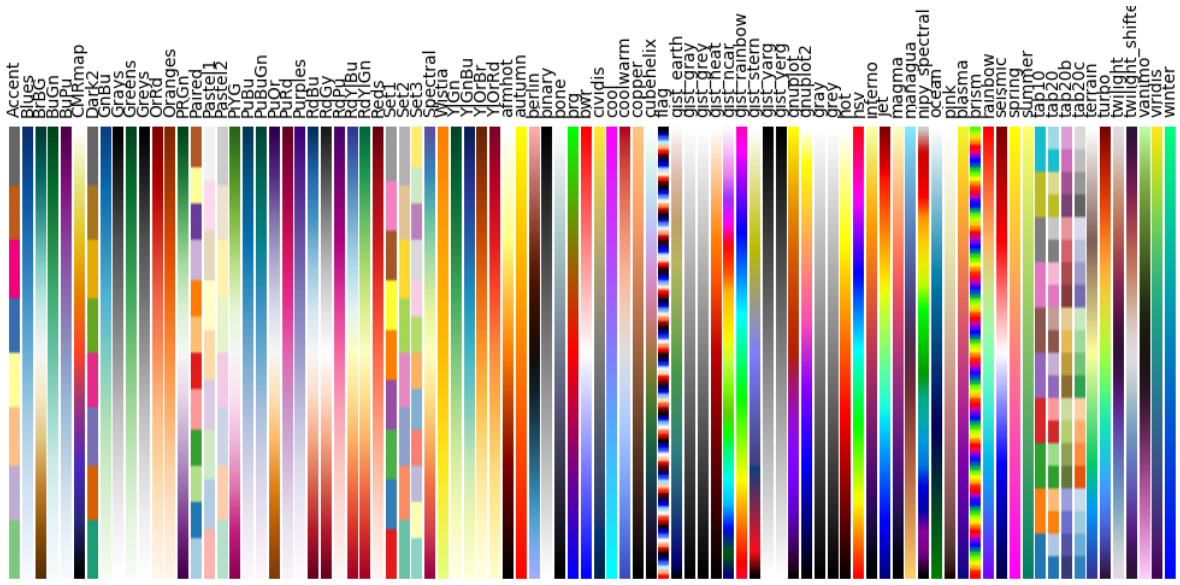
(continued from previous page)

```
            i,
        ],
        [
            1,
        ],
        "s",
        markersize=5,
        markeredgewidth=1 + i / 10.0,
        markeredgecolor="k",
        markerfacecolor="w",
    )
plt.xlim(0, 11)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.014 seconds)

Colormaps

An example plotting the matplotlib colormaps.




```
import numpy as np

import matplotlib
import matplotlib.pyplot as plt


plt.rc("text", usetex=False)
a = np.outer(np.arange(0, 1, 0.01), np.ones(10))

plt.figure(figsize=(10, 5))
plt.subplots_adjust(top=0.8, bottom=0.05, left=0.01, right=0.99)
maps = [m for m in matplotlib.colormaps if not m.endswith("_r")]
```

(continues on next page)

(continued from previous page)

```
maps.sort()
l = len(maps) + 1

for i, m in enumerate(maps):
    plt.subplot(1, l, i + 1)
    plt.axis("off")
    plt.imshow(a, aspect="auto", cmap=plt.get_cmap(m), origin="lower")
    plt.title(m, rotation=90, fontsize=10, va="bottom")

plt.show()
```

Total running time of the script: (0 minutes 1.812 seconds)

Solid joint style

An example showing the different solid joint styles in matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

plt.plot(np.arange(3), [0, 1, 0], color="blue", linewidth=8, solid_joinstyle="miter")
plt.plot(
    4 + np.arange(3), [0, 1, 0], color="blue", linewidth=8, solid_joinstyle="bevel"
)
plt.plot(
    8 + np.arange(3), [0, 1, 0], color="blue", linewidth=8, solid_joinstyle="round"
)

plt.xlim(0, 12)
plt.ylim(-1, 2)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.011 seconds)

Solid cap style

An example demoing the solide cap style in matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

plt.plot(np.arange(4), np.ones(4), color="blue", linewidth=8, solid_capstyle="butt")

plt.plot(
    5 + np.arange(4), np.ones(4), color="blue", linewidth=8, solid_capstyle="round"
)

plt.plot(
    10 + np.arange(4),
    np.ones(4),
    color="blue",
    linewidth=8,
    solid_capstyle="projecting",
)

plt.xlim(0, 14)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.011 seconds)

Marker edge color

Demo the marker edge color of matplotlib's markers.

```
import numpy as np
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

rng = np.random.default_rng()

for i in range(1, 11):
    r, g, b = np.random.uniform(0, 1, 3)
    plt.plot(
        [
            i,
```

(continues on next page)

(continued from previous page)

```
        ],
        [
            1,
        ],
        "s",
        markersize=5,
        markerfacecolor="w",
        markeredgewidth=1.5,
        markeredgecolor=(r, g, b, 1),
    )

plt.xlim(0, 11)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.015 seconds)

Marker face color

Demo the marker face color of matplotlib's markers.

```
import numpy as np
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

rng = np.random.default_rng()

for i in range(1, 11):
    r, g, b = np.random.uniform(0, 1, 3)
    plt.plot(
        [
            i,
        ],
        [
            1,
        ],
        "s",
        markersize=8,
        markerfacecolor=(r, g, b, 1),
        markeredgewidth=0.1,
        markeredgecolor=(0, 0, 0, 0.5),
    )
plt.xlim(0, 11)
plt.xticks([])
```

(continues on next page)

(continued from previous page)

```
plt.yticks([])
plt.show()
```

Total running time of the script: (0 minutes 0.014 seconds)

Dash capstyle

An example demoing the dash capstyle.

```
import numpy as np
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

plt.plot(
    np.arange(4),
    np.ones(4),
    color="blue",
    dashes=[15, 15],
    linewidth=8,
    dash_capstyle="butt",
)

plt.plot(
    5 + np.arange(4),
    np.ones(4),
    color="blue",
    dashes=[15, 15],
    linewidth=8,
    dash_capstyle="round",
)

plt.plot(
    10 + np.arange(4),
    np.ones(4),
    color="blue",
    dashes=[15, 15],
    linewidth=8,
    dash_capstyle="projecting",
)

plt.xlim(0, 14)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.011 seconds)

Dash join style

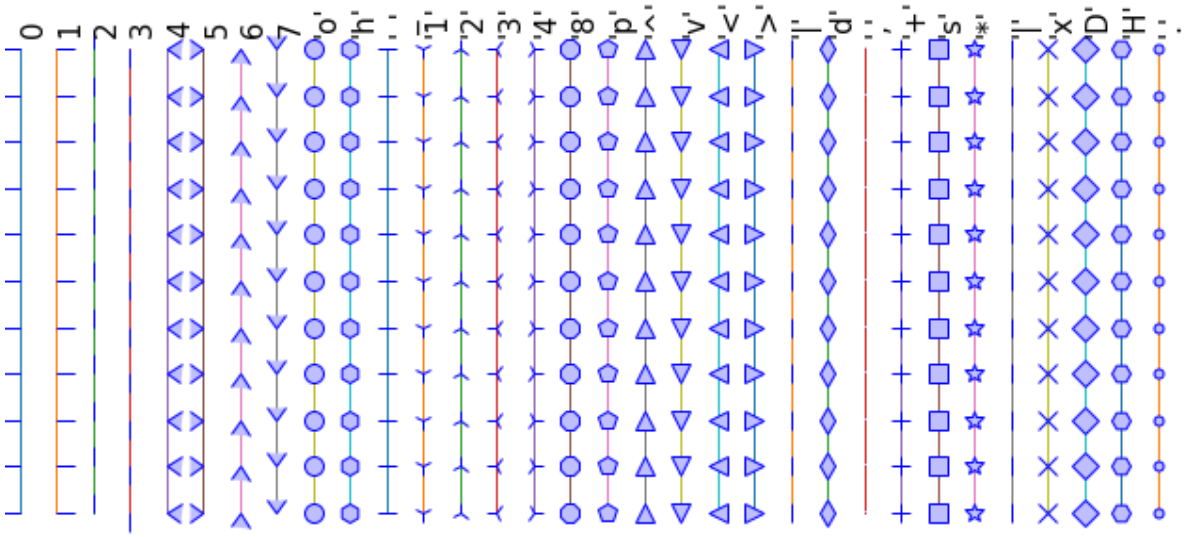
Example demoing the dash join style.

```
import numpy as np
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes((0, 0, 1, 1), frameon=False)

plt.plot(
    np.arange(3),
    [0, 1, 0],
    color="blue",
    dashes=[12, 5],
    linewidth=8,
    dash_joinstyle="miter",
)
plt.plot(
    4 + np.arange(3),
    [0, 1, 0],
    color="blue",
    dashes=[12, 5],
    linewidth=8,
    dash_joinstyle="bevel",
)
plt.plot(
    8 + np.arange(3),
    [0, 1, 0],
    color="blue",
    dashes=[12, 5],
    linewidth=8,
    dash_joinstyle="round",
)

plt.xlim(0, 12)
plt.ylim(-1, 2)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.012 seconds)

Markers

Show the different markers of matplotlib.



```
import numpy as np
import matplotlib.pyplot as plt


def marker(m, i):
    X = i * 0.5 * np.ones(11)
    Y = np.arange(11)

    plt.plot(X, Y, lw=1, marker=m, ms=10, mfc=(0.75, 0.75, 1, 1), mec=(0, 0, 1, 1))
    plt.text(0.5 * i, 10.25, repr(m), rotation=90, fontsize=15, va="bottom")


markers = [
    0,
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    "o",
    "h",
    "_",
    "1",
    "2",
    "3",
    "4",
    "8",
    "p",
    "^",
    "v",
    "<",
    ">",
    "|",
```

(continues on next page)

(continued from previous page)

```
    "d",
    ",",
    "+",
    "s",
    "*",
    "|",
    "x",
    "D",
    "H",
    ".",
]

n_markers = len(markers)

size = 20 * n_markers, 300
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
plt.axes((0, 0.01, 1, 0.9), frameon=False)

for i, m in enumerate(markers):
    marker(m, i)

plt.xlim(-0.2, 0.2 + 0.5 * n_markers)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.071 seconds)

Linestyles

Plot the different line styles.

```
import numpy as np
import matplotlib.pyplot as plt


def linestyle(ls, i):
    X = i * 0.5 * np.ones(11)
    Y = np.arange(11)
    plt.plot(
        X,
        Y,
        ls,
        color=(0.0, 0.0, 1, 1),
        lw=3,
        ms=8,
        mfc=(0.75, 0.75, 1, 1),
        mec=(0, 0, 1, 1),
    )
    plt.text(0.5 * i, 10.25, ls, rotation=90, fontsize=15, va="bottom")


linestyles = [
    "-",
    "--",
    ":",
    "-.",
    ".",
    ",",
    "o",
    "^",
    "v",
    "<",
    ">",
    "s",
    "+",
```

(continues on next page)

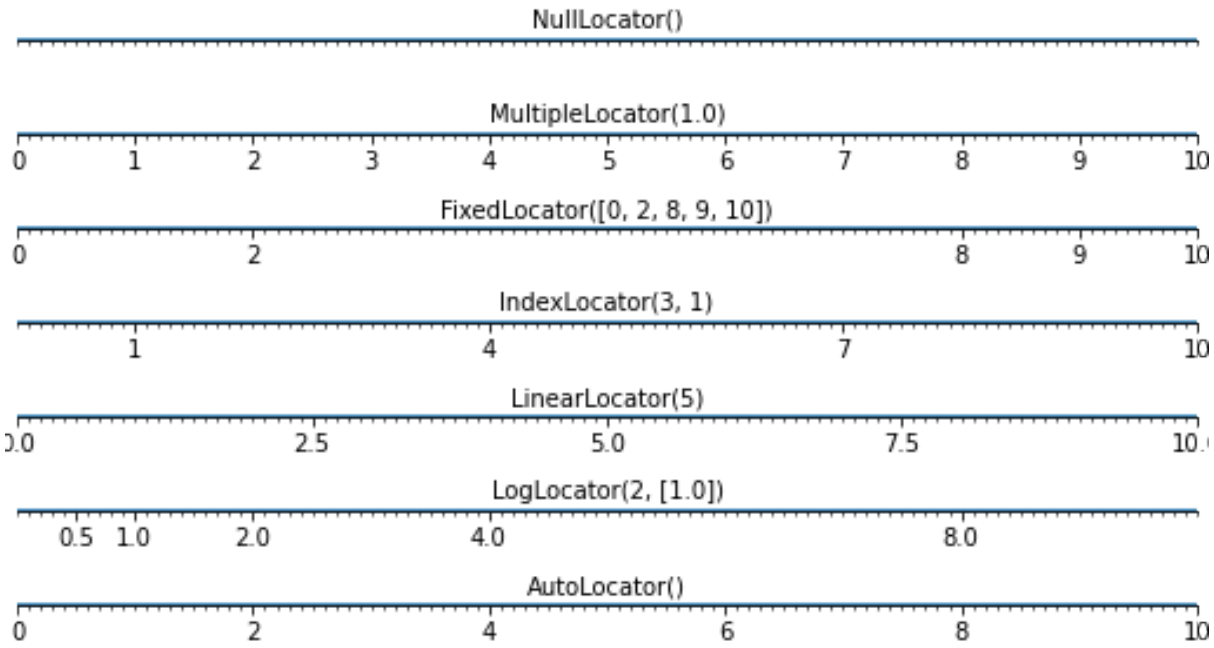
(continued from previous page)

```
    "x",
    "d",
    "1",
    "2",
    "3",
    "4",
    "h",
    "p",
    "|",
    "_",
    "D",
    "H",
]
n_lines = len(linestyles)

size = 20 * n_lines, 300
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
plt.axes((0, 0.01, 1, 0.9), frameon=False)

for i, ls in enumerate(linestyles):
    linestyle(ls, i)

plt.xlim(-0.2, 0.2 + 0.5 * n_lines)
plt.xticks([])
plt.yticks([])

plt.show()
```

Total running time of the script: (0 minutes 0.041 seconds)

Locators for tick on axis

An example demoing different locators to position ticks on axis for matplotlib.




```
import numpy as np

from matplotlib import ticker
import matplotlib.pyplot as plt


def tickline():
    plt.xlim(0, 10), plt.ylim(-1, 1), plt.yticks([])
    ax = plt.gca()
    ax.spines["right"].set_color("none")
    ax.spines["left"].set_color("none")
    ax.spines["top"].set_color("none")
    ax.xaxis.set_ticks_position("bottom")
    ax.spines["bottom"].set_position(("data", 0))
    ax.yaxis.set_ticks_position("none")
    ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.1))
    ax.plot(np.arange(11), np.zeros(11))
    return ax


locators = [
    "ticker.NullLocator()",
    "ticker.MultipleLocator(1.0)",
    "ticker.FixedLocator([0, 2, 8, 9, 10])",
    "ticker.IndexLocator(3, 1)",
    "ticker.LinearLocator(5)",
    "ticker.LogLocator(2, [1.0])",
    "ticker.AutoLocator()",
]

n_locators = len(locators)

size = 512, 40 * n_locators
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
```

(continues on next page)

(continued from previous page)

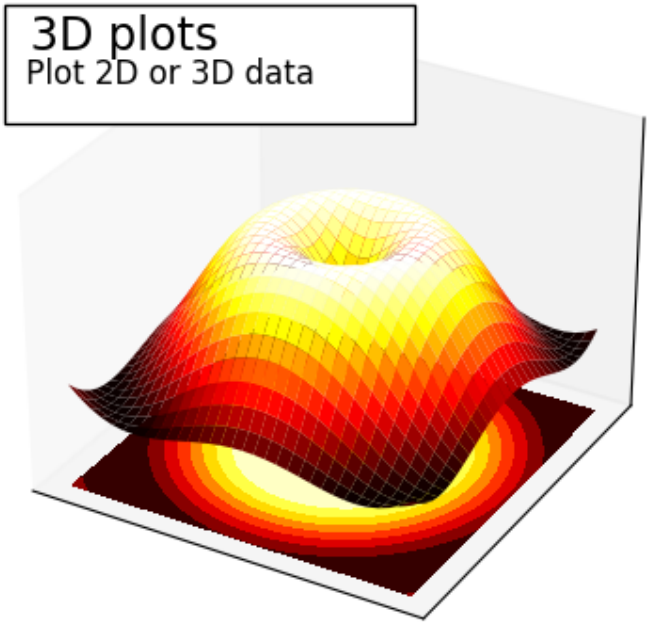
```
fig.patch.set_alpha(0)


for i, locator in enumerate(locators):
    plt.subplot(n_locators, 1, i + 1)
    ax = tickline()
    ax.xaxis.set_major_locator(eval(locator))
    plt.text(5, 0.3, locator[7:], ha="center")

plt.subplots_adjust(bottom=0.01, top=0.99, left=0.01, right=0.99)
plt.show()
```

Total running time of the script: (0 minutes 0.879 seconds)

Code generating the summary figures with a title

3D plotting vignette

Demo 3D plotting with matplotlib and decorate the figure.




```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.arange(-4, 4, 0.25)
```

(continues on next page)

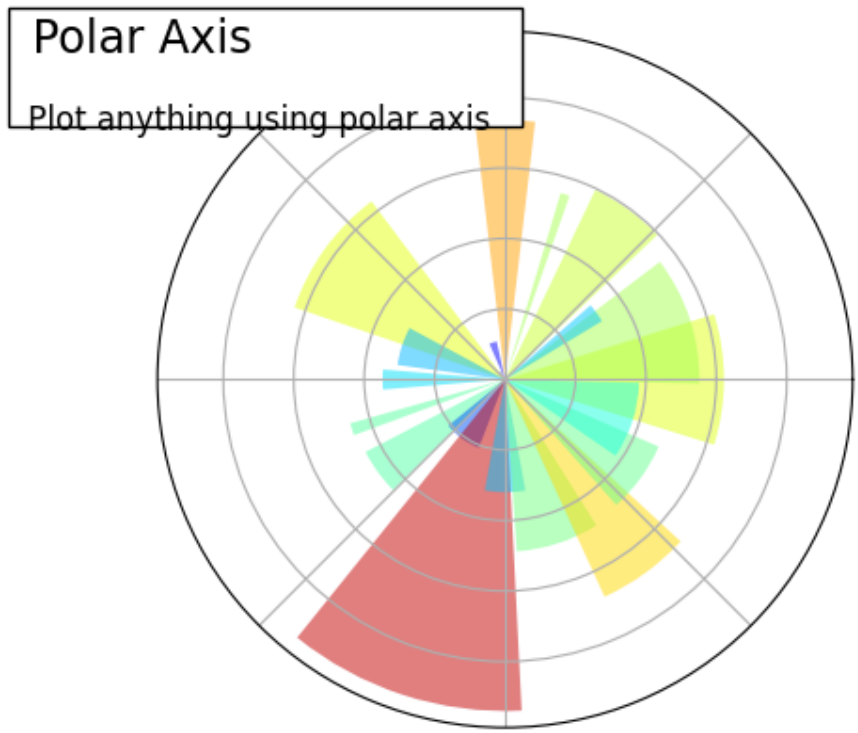
(continued from previous page)

```
y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

fig = plt.figure()
ax: Axes3D = fig.add_subplot(111, projection="3d")

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap="hot")
ax.contourf(X, Y, Z, zdir="z", offset=-2, cmap="hot")

ax.set_zlim(-2, 2)
plt.xticks([])
plt.yticks([])
ax.set_zticks([])

ax.text2D(
    0.05,
    0.93,
    " 3D plots              \n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    bbox={"facecolor": "white", "alpha": 1.0},
    transform=plt.gca().transAxes,
)

ax.text2D(
    0.05,
    0.87,
    " Plot 2D or 3D data",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.115 seconds)

Plotting in polar, decorated

An example showing how to plot in polar coordinate, and some decorations.




```
import numpy as np

import matplotlib
import matplotlib.pyplot as plt


plt.subplot(1, 1, 1, polar=True)

N = 20
theta = np.arange(0.0, 2 * np.pi, 2 * np.pi / N)
rng = np.random.default_rng()
radii = 10 * rng.random(N)
width = np.pi / 4 * rng.random(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)
jet = matplotlib.colormaps["jet"]

for r, bar in zip(radii, bars, strict=True):
    bar.set_facecolor(jet(r / 10.0))
    bar.set_alpha(0.5)
plt.gca().set_xticklabels([])
plt.gca().set_yticklabels([])


plt.text(
    -0.2,
    1.02,
    " Polar Axis                    \n",
    horizontalalignment="left",
```

(continues on next page)

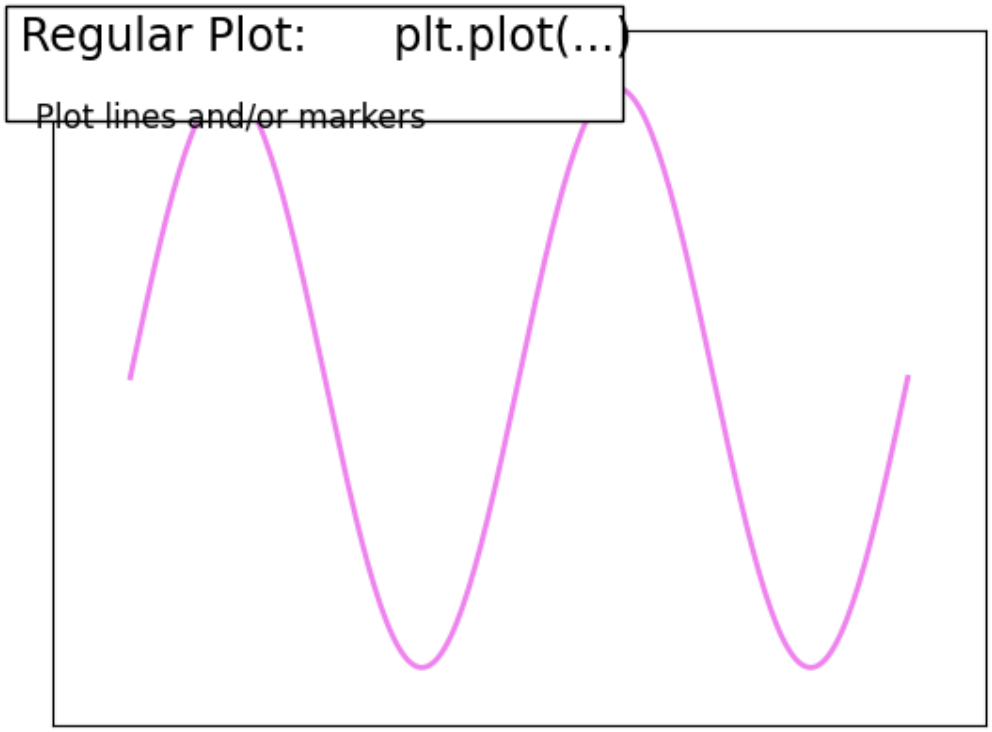
(continued from previous page)

```
    verticalalignment="top",
    size="xx-large",
    bbox={"facecolor": "white", "alpha": 1.0},
    transform=plt.gca().transAxes,
)

plt.text(
    -0.2,
    1.01,
    "\n\n Plot anything using polar axis ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.080 seconds)

Plot example vignette

An example of plots with matplotlib, and added annotations.




```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

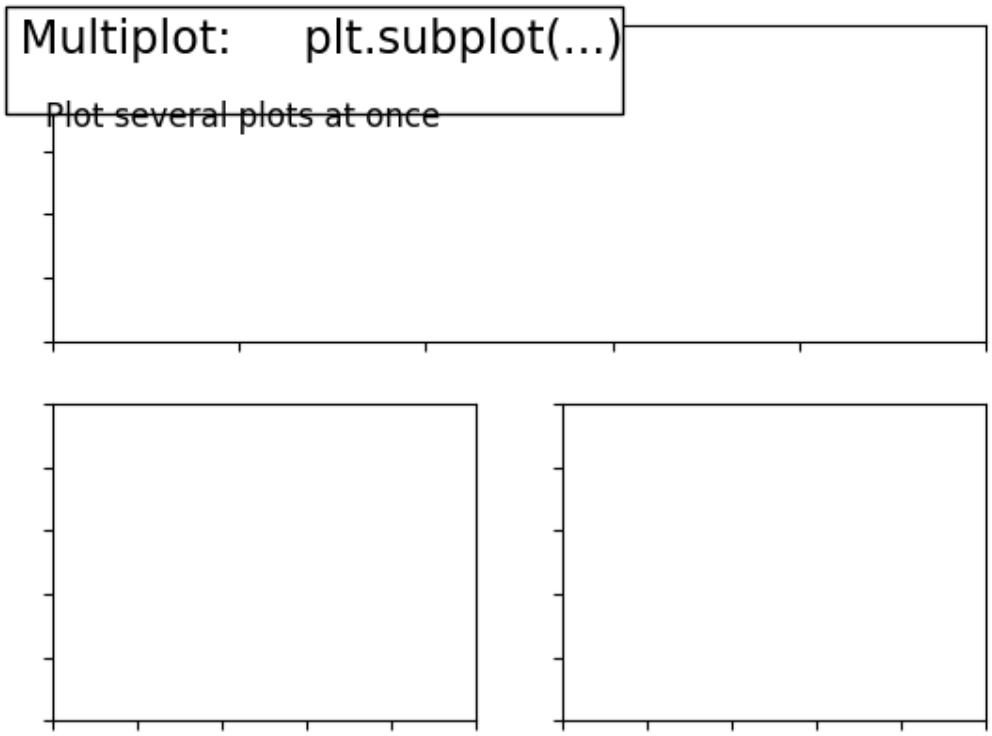
(continued from previous page)

```
n = 256
X = np.linspace(0, 2, n)
Y = np.sin(2 * np.pi * X)

plt.plot(X, Y, lw=2, color="violet")
plt.xlim(-0.2, 2.2)
plt.xticks([])
plt.ylim(-1.2, 1.2)
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Regular Plot:      plt.plot(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n   Plot lines and/or markers ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.028 seconds)

Multiple plots vignette

Demo multiple plots and style the figure.




```
import matplotlib.pyplot as plt

ax = plt.subplot(2, 1, 1)
ax.set_xticklabels([])
ax.set_yticklabels([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.72),
        width=0.66,
        height=0.34,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)
```

(continues on next page)

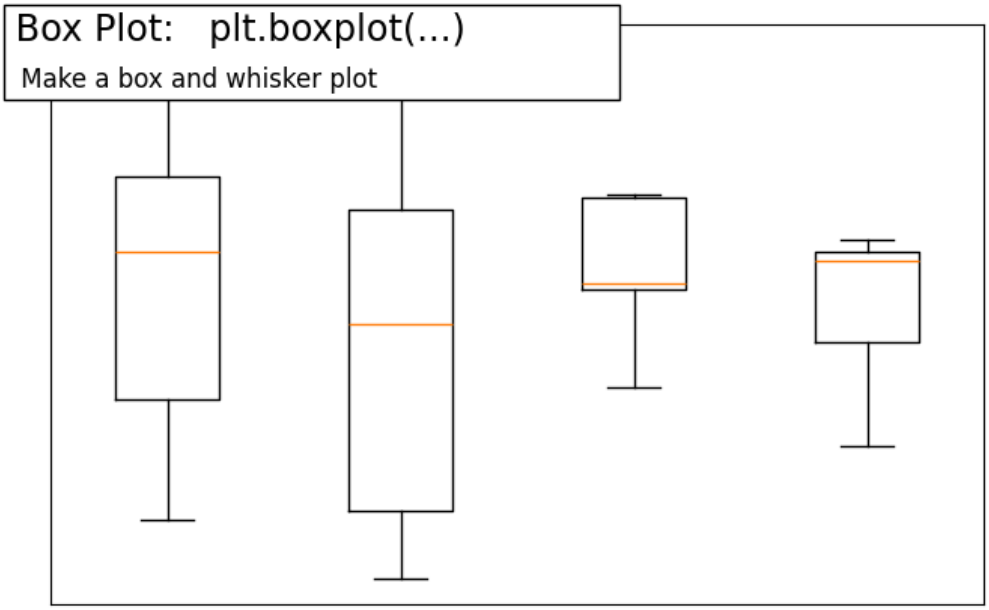
(continued from previous page)

```
plt.text(
    -0.05,
    1.02,
    " Multiplot:     plt.subplot(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=ax.transAxes,
)
plt.text(
    -0.05,
    1.01,
    "\n\n    Plot several plots at once ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=ax.transAxes,
)

ax = plt.subplot(2, 2, 3)
ax.set_xticklabels([])
ax.set_yticklabels([])

ax = plt.subplot(2, 2, 4)
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
```

Total running time of the script: (0 minutes 0.074 seconds)

Boxplot with matplotlib

An example of doing box plots with matplotlib




```
import numpy as np
import matplotlib.pyplot as plt


fig = plt.figure(figsize=(8, 5))
axes = plt.subplot(111)

n = 5
Z = np.zeros((n, 4))
X = np.linspace(0, 2, n)
rng = np.random.default_rng()
Y = rng.random((n, 4))
plt.boxplot(Y)

plt.xticks([])
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
```

(continues on next page)

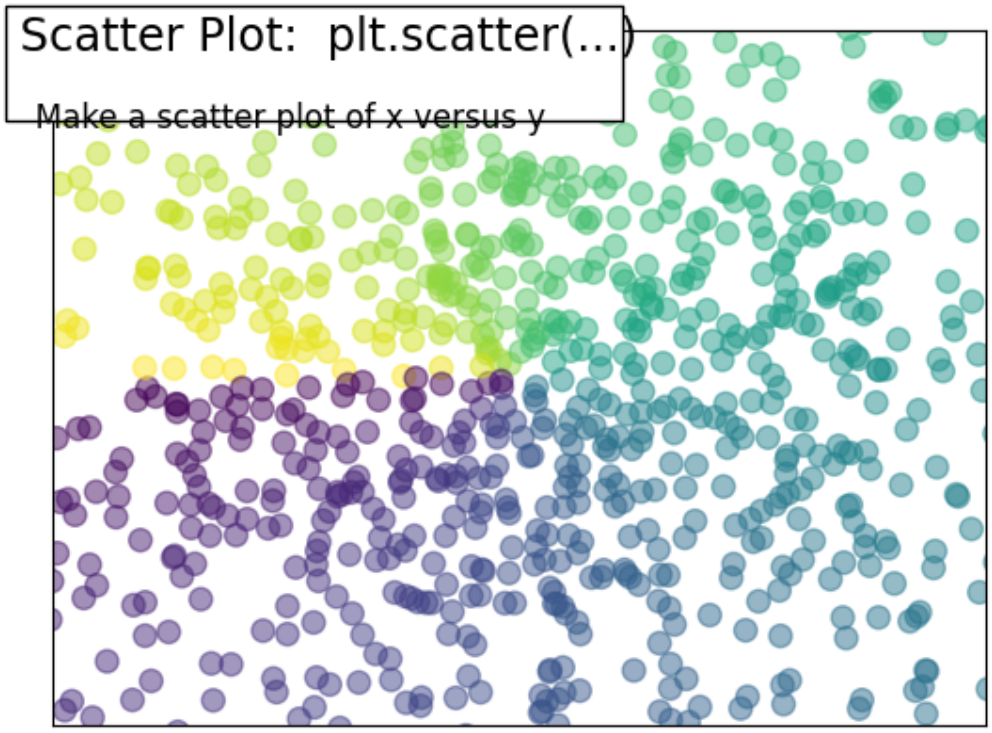
(continued from previous page)

```
    )
)

plt.text(
    -0.05,
    1.02,
    " Box Plot:   plt.boxplot(...)\n ",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=axes.transAxes,
)

plt.text(
    -0.04,
    0.98,
    "\n Make a box and whisker plot ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=axes.transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.040 seconds)

Plot scatter decorated

An example showing the scatter function, with decorations.




```
import numpy as np
import matplotlib.pyplot as plt

n = 1024
rng = np.random.default_rng()
X = rng.normal(0, 1, n)
Y = rng.normal(0, 1, n)

T = np.arctan2(Y, X)

plt.scatter(X, Y, s=75, c=T, alpha=0.5)
plt.xlim(-1.5, 1.5)
plt.xticks([])
plt.ylim(-1.5, 1.5)
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
```

(continues on next page)

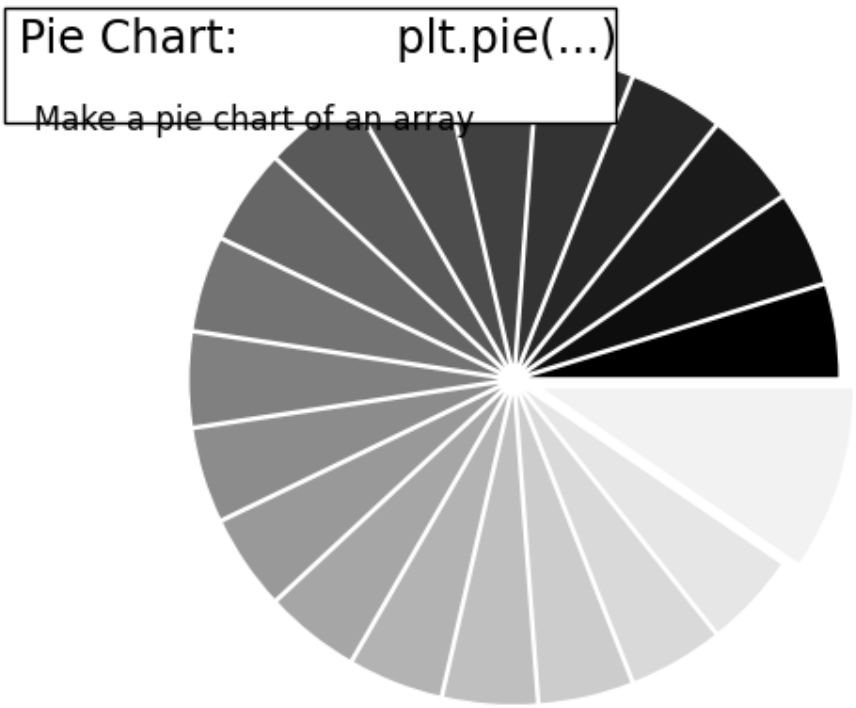
(continued from previous page)

```
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Scatter Plot:  plt.scatter(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n   Make a scatter plot of x versus y ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.060 seconds)

Pie chart vignette

Demo pie chart with matplotlib and style the figure.




```
import numpy as np
import matplotlib.pyplot as plt

n = 20
X = np.ones(n)
X[-1] *= 2
plt.pie(X, explode=X * 0.05, colors=[f"{i / float(n):f}" for i in range(n)])

fig = plt.gcf()
w, h = fig.get_figwidth(), fig.get_figheight()
r = h / float(w)

plt.xlim(-1.5, 1.5)
plt.ylim(-1.5 * r, 1.5 * r)
plt.xticks([])
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
```

(continues on next page)

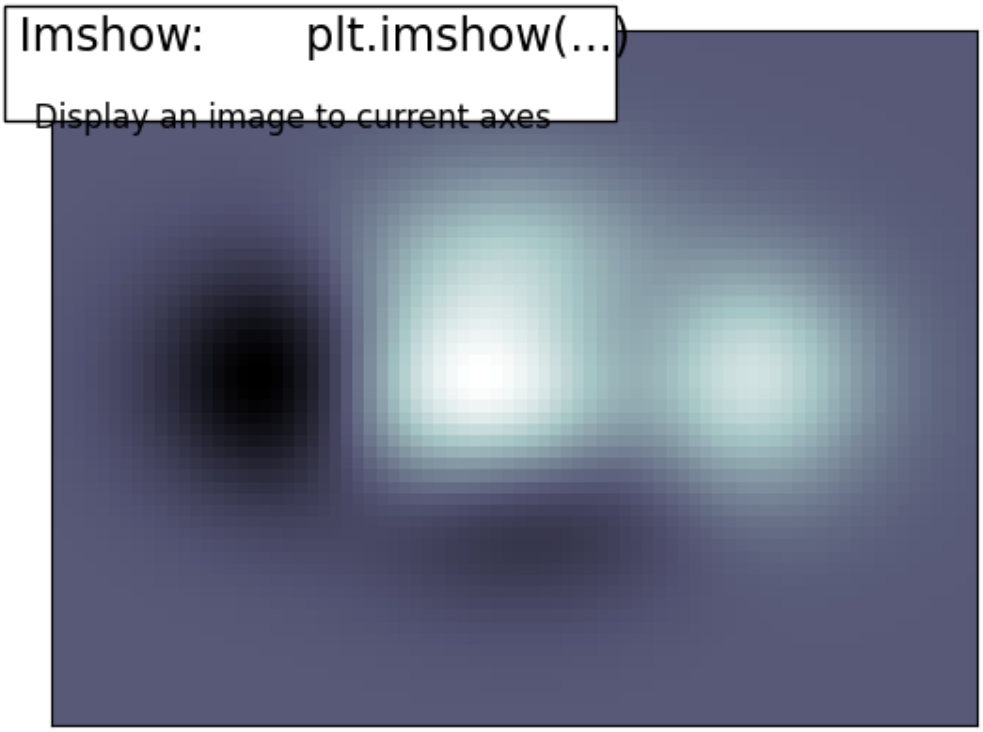
(continued from previous page)

```
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Pie Chart:           plt.pie(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n   Make a pie chart of an array ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

Imshow demo

Demoing imshow




```
import numpy as np
import matplotlib.pyplot as plt


def f(x, y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-(x**2) - y**2)


n = 10
x = np.linspace(-3, 3, 8 * n)
y = np.linspace(-3, 3, 6 * n)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.imshow(Z, interpolation="nearest", cmap="bone", origin="lower")
plt.xticks([])
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
```

(continues on next page)

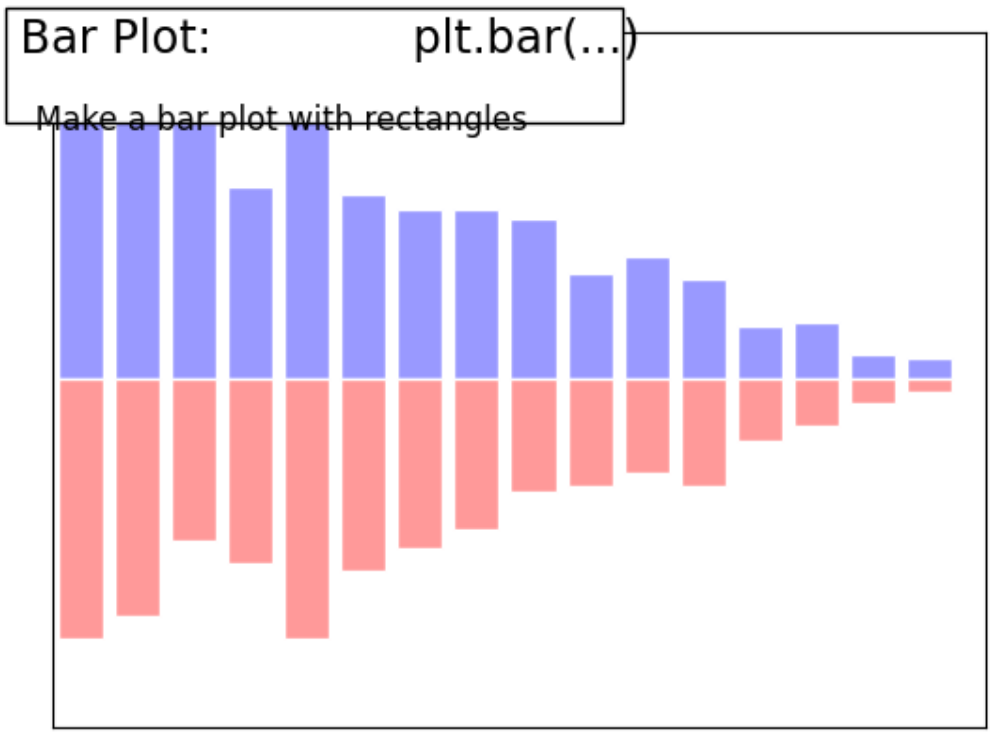
(continued from previous page)

```
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Imshow:       plt.imshow(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n   Display an image to current axes ",
    horizontalalignment="left",
    verticalalignment="top",
    family="DejaVu Sans",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.035 seconds)

Bar plot advanced

An more elaborate bar plot example




```
import numpy as np
import matplotlib.pyplot as plt

n = 16
X = np.arange(n)
Y1 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
Y2 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
plt.bar(X, Y1, facecolor="#9999ff", edgecolor="white")
plt.bar(X, -Y2, facecolor="#ff9999", edgecolor="white")
plt.xlim(-0.5, n)
plt.xticks([])
plt.ylim(-1, 1)
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
```

(continues on next page)

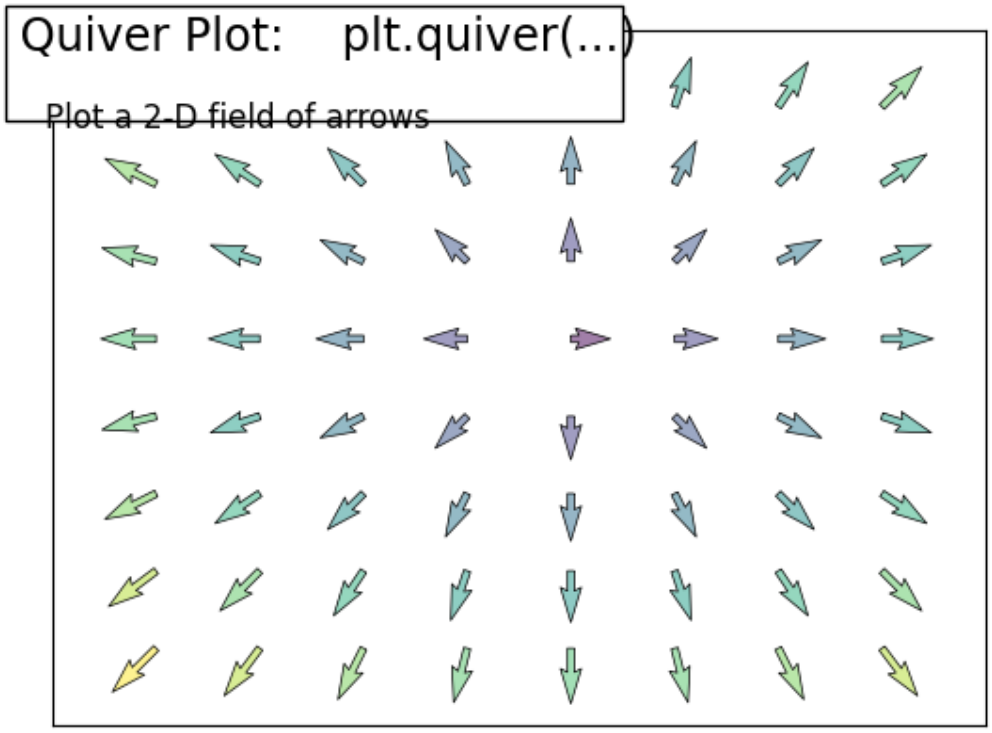
(continued from previous page)

```
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Bar Plot:              plt.bar(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n   Make a bar plot with rectangles ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.045 seconds)

Plotting quiver decorated

An example showing quiver with decorations.




```
import numpy as np
import matplotlib.pyplot as plt

n = 8
X, Y = np.mgrid[0:n, 0:n]
T = np.arctan2(Y - n / 2.0, X - n / 2.0)
R = 10 + np.sqrt((Y - n / 2.0) ** 2 + (X - n / 2.0) ** 2)
U, V = R * np.cos(T), R * np.sin(T)

plt.quiver(X, Y, U, V, R, alpha=0.5)
plt.quiver(X, Y, U, V, edgecolor="k", facecolor="None", linewidth=0.5)

plt.xlim(-1, n)
plt.xticks([])
plt.ylim(-1, n)
plt.yticks([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
```

(continues on next page)

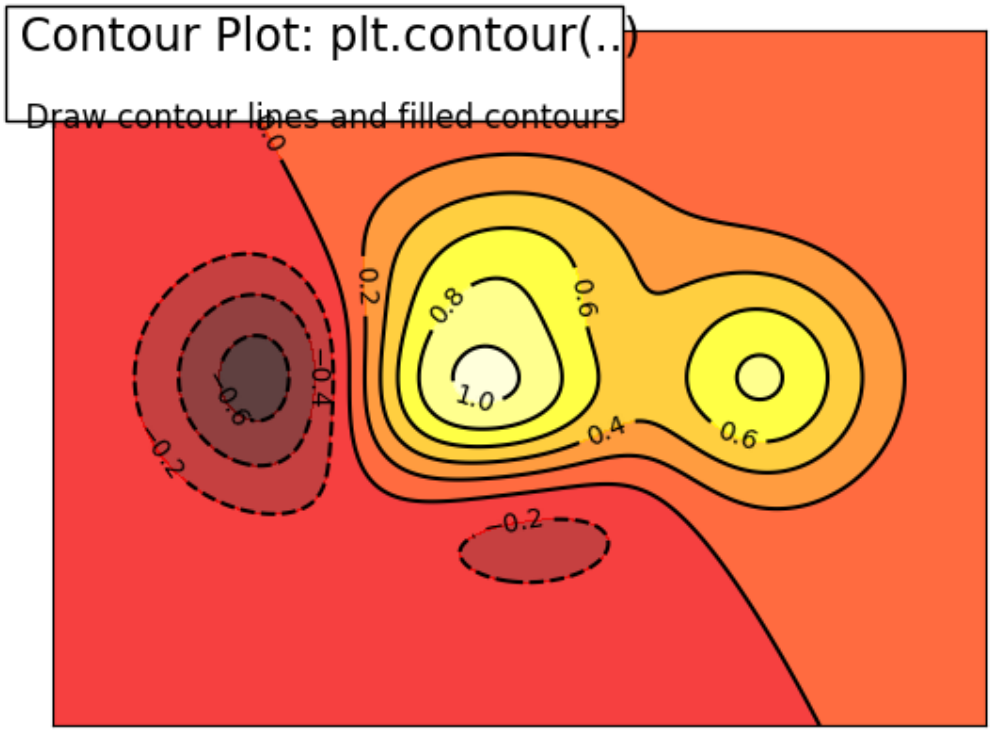
(continued from previous page)

```
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Quiver Plot:    plt.quiver(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n    Plot a 2-D field of arrows ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)


plt.show()
```

Total running time of the script: (0 minutes 0.034 seconds)

Display the contours of a function

An example demoing how to plot the contours of a function, with additional layout tweaks.




```
/home/runner/work/scientific-python-lectures/scientific-python-lectures/intro/
↪matplotlib/examples/pretty_plots/plot_contour_ext.py:24: UserWarning: The following␣
↪kwargs were not used by contour: 'linewidth'
  C = plt.contour(X, Y, f(X, Y), 8, colors="black", linewidth=0.5)
```

```
import numpy as np
import matplotlib.pyplot as plt


def f(x, y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-(x**2) - y**2)


n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)

plt.contourf(X, Y, f(X, Y), 8, alpha=0.75, cmap="hot")
C = plt.contour(X, Y, f(X, Y), 8, colors="black", linewidth=0.5)
plt.clabel(C, inline=1, fontsize=10)
plt.xticks([])
plt.yticks([])
```

(continues on next page)

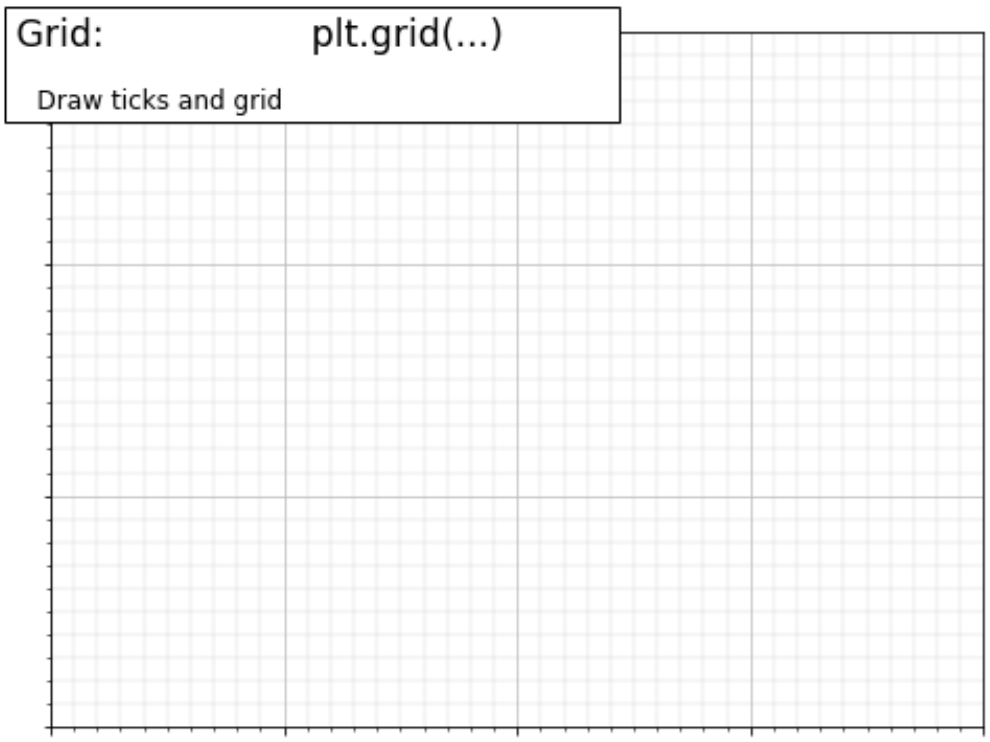
(continued from previous page)

```
# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Contour Plot: plt.contour(..)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n  Draw contour lines and filled contours ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.077 seconds)

Grid elaborate

An example displaying a grid on the axes and tweaking the layout.




```
Text(-0.05, 1.01, '\n\n    Draw ticks and grid ')
```

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator

fig = plt.figure(figsize=(8, 6), dpi=72, facecolor="white")
axes = plt.subplot(111)
axes.set_xlim(0, 4)
axes.set_ylim(0, 3)

axes.xaxis.set_major_locator(MultipleLocator(1.0))
axes.xaxis.set_minor_locator(MultipleLocator(0.1))
axes.yaxis.set_major_locator(MultipleLocator(1.0))
axes.yaxis.set_minor_locator(MultipleLocator(0.1))
axes.grid(which="major", axis="x", linewidth=0.75, linestyle="-", color="0.75")
axes.grid(which="minor", axis="x", linewidth=0.25, linestyle="-", color="0.75")
axes.grid(which="major", axis="y", linewidth=0.75, linestyle="-", color="0.75")
axes.grid(which="minor", axis="y", linewidth=0.25, linestyle="-", color="0.75")
axes.set_xticklabels([])
axes.set_yticklabels([])


# Add a title and a box around it
```

(continues on next page)

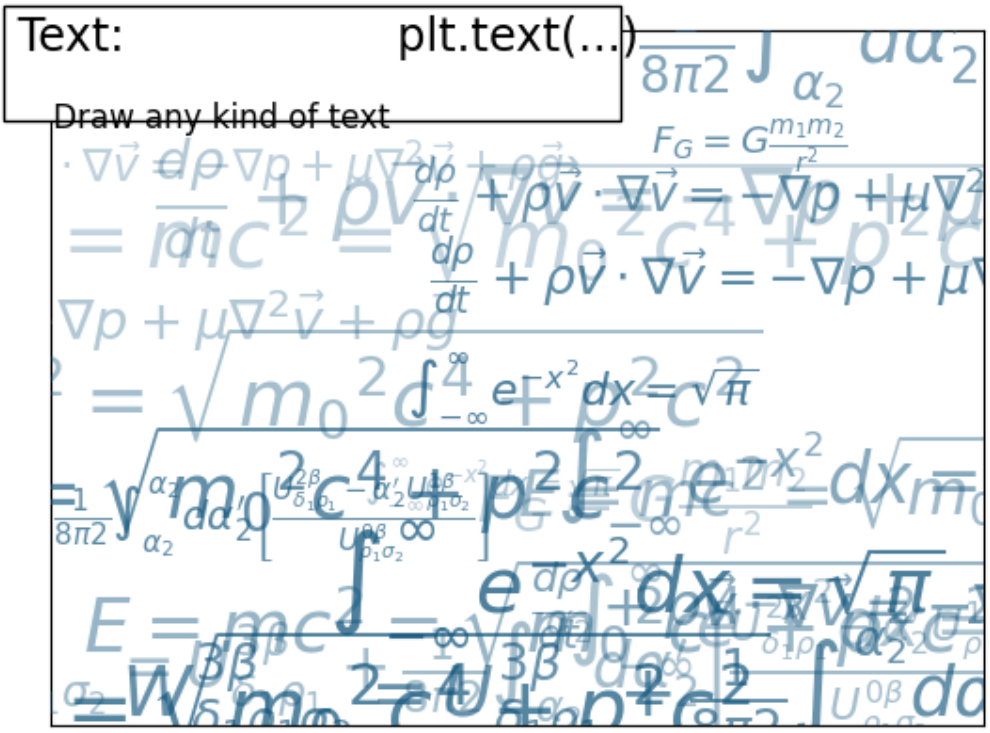
(continued from previous page)

```
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Grid:                   plt.grid(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=axes.transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n    Draw ticks and grid ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=axes.transAxes,
)
```

Total running time of the script: (0 minutes 0.099 seconds)

Text printing decorated

An example showing text printing and decorating the resulting figure.




```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
plt.xticks([])
plt.yticks([])

eqs = []
eqs.append(
    r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}
↪{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\
↪delta_1 \rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \
↪sigma_2}}\right]$"
)
eqs.append(
    r"$\frac{d\rho}{d t} + \rho \vec{v}\cdot\nabla\vec{v} = -\nabla p + \mu\nabla^2 \
↪vec{v} + \rho \vec{g}$"
)
eqs.append(r"$\int_{-\infty}^\infty e^{-x^2}dx=\sqrt{\pi}$")
eqs.append(r"$E = mc^2 = \sqrt{{m_0}^2c^4 + p^2c^2}$")
eqs.append(r"$F_G = G\frac{m_1m_2}{r^2}$")

rng = np.random.default_rng()

for i in range(24):
    index = rng.integers(0, len(eqs))
    eq = eqs[index]
    size = rng.uniform(12, 32)
```

(continues on next page)

(continued from previous page)

```
    x, y = rng.uniform(0, 1, 2)
    alpha = rng.uniform(0.25, 0.75)
    plt.text(
        x,
        y,
        eq,
        ha="center",
        va="center",
        color="#11557c",
        alpha=alpha,
        transform=plt.gca().transAxes,
        fontsize=size,
        clip_on=True,
    )


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch

ax = plt.gca()
ax.add_patch(
    FancyBboxPatch(
        (-0.05, 0.87),
        width=0.66,
        height=0.165,
        clip_on=False,
        boxstyle="square,pad=0",
        zorder=3,
        facecolor="white",
        alpha=1.0,
        transform=plt.gca().transAxes,
    )
)

plt.text(
    -0.05,
    1.02,
    " Text:                    plt.text(...)\n",
    horizontalalignment="left",
    verticalalignment="top",
    size="xx-large",
    transform=plt.gca().transAxes,
)

plt.text(
    -0.05,
    1.01,
    "\n\n     Draw any kind of text ",
    horizontalalignment="left",
    verticalalignment="top",
    size="large",
    transform=plt.gca().transAxes,
)

plt.show()
```

Total running time of the script: (0 minutes 0.658 seconds)

CHAPTER 5

SciPy : high-level scientific computing

Authors: *Gaël Varoquaux, Adrien Chauve, Andre Espaze, Emmanuelle Gouillart, Ralf Gommers*

Scipy

The `scipy` package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

Tip

`scipy` can be compared to other standard scientific-computing libraries, such as the GSL (GNU Scientific Library for C and C++), or Matlab's toolboxes. `scipy` is the core package for scientific routines in Python; it is meant to operate efficiently on `numpy` arrays, so that NumPy and SciPy work hand in hand.

Before implementing a routine, it is worth checking if the desired data processing is not already implemented in SciPy. As non-professional programmers, scientists often tend to **re-invent the wheel**, which leads to buggy, non-optimal, difficult-to-share and unmaintainable code. By contrast, `SciPy`'s routines are optimized and tested, and should therefore be used when possible.

Chapters contents

- *File input/output:* `scipy.io`
- *Special functions:* `scipy.special`

- *Linear algebra operations:* `scipy.linalg`
- *Interpolation:* `scipy.interpolate`
- *Optimization and fit:* `scipy.optimize`
- *Statistics and random numbers:* `scipy.stats`
- *Numerical integration:* `scipy.integrate`
- *Fast Fourier transforms:* `scipy.fft`
- *Signal processing:* `scipy.signal`
- *Image manipulation:* `scipy.ndimage`
- *Summary exercises on scientific computing*
- *Full code examples for the SciPy chapter*

Warning

This tutorial is far from an introduction to numerical computing. As enumerating the different submodules and functions in SciPy would be very boring, we concentrate instead on a few examples to give a general idea of how to use `scipy` for scientific computing.

`scipy` is composed of task-specific sub-modules:

`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fft`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

Tip

They all depend on `numpy`, but are mostly independent of each other. The standard way of importing NumPy and these SciPy modules is:

```
>>> import numpy as np
>>> import scipy as sp
```

5.1 File input/output: `scipy.io`

`scipy.io` contains functions for loading and saving data in several common formats including Matlab, IDL, Matrix Market, and Harwell-Boeing.

Matlab files: Loading and saving:

```
>>> import scipy as sp
>>> a = np.ones((3, 3))
>>> sp.io.savemat('file.mat', {'a': a})  # savemat expects a dictionary
>>> data = sp.io.loadmat('file.mat')
>>> data['a']
array([[1.,  1.,  1.],
       [1.,  1.,  1.],
       [1.,  1.,  1.]])
```

Warning

Python / Matlab mismatch: The Matlab file format does not support 1D arrays.

```
>>> a = np.ones(3)
>>> a
array([1.,  1.,  1.])
>>> a.shape
(3,)
>>> sp.io.savemat('file.mat', {'a': a})
>>> a2 = sp.io.loadmat('file.mat')['a']
>>> a2
array([[1.,  1.,  1.]])
>>> a2.shape
(1, 3)
```

Notice that the original array was a one-dimensional array, whereas the saved and reloaded array is a two-dimensional array with a single row.

For other formats, see the `scipy.io` documentation.

See also

- Load text files: `numpy.loadtxt()`/`numpy.savetxt()`
- Clever loading of text/csv files: `numpy.genfromtxt()`
- Fast and efficient, but NumPy-specific, binary format: `numpy.save()`/`numpy.load()`
- Basic input/output of images in Matplotlib: `matplotlib.pyplot.imread()`/`matplotlib.pyplot.imsave()`
- More advanced input/output of images: `imageio`

5.2 Special functions: `scipy.special`

"Special" functions are functions commonly used in science and mathematics that are not considered to be "elementary" functions. Examples include

- the gamma function, `scipy.special.gamma()`,
- the error function, `scipy.special.erf()`,

- Bessel functions, such as `scipy.special.jv()` (Bessel function of the first kind), and
- elliptic functions, such as `scipy.special.ellipj()` (Jacobi elliptic functions).

Other special functions are combinations of familiar elementary functions, but they offer better accuracy or robustness than their naive implementations would.

Most of these function are computed elementwise and follow standard NumPy broadcasting rules when the input arrays have different shapes. For example, `scipy.special.xlog1py()` is mathematically equivalent to $x \log(1 + y)$.

```
>>> import scipy as sp
>>> x = np.asarray([1, 2])
>>> y = np.asarray([[3], [4], [5]])
>>> res = sp.special.xlog1py(x, y)
>>> res.shape
(3, 2)
>>> ref = x * np.log(1 + y)
>>> np.allclose(res, ref)
True
```

However, `scipy.special.xlog1py()` is numerically favorable for small y, when explicit addition of `1` would lead to loss of precision due to floating point truncation error.

```
>>> x = 2.5
>>> y = 1e-18
>>> x * np.log(1 + y)
np.float64(0.0)
>>> sp.special.xlog1py(x, y)
np.float64(2.5e-18)
```

Many special functions also have "logarithmized" variants. For instance, the gamma function $\Gamma(\cdot)$ is related to the factorial function by $n! = \Gamma(n + 1)$, but it extends the domain from the positive integers to the complex plane.

```
>>> x = np.arange(10)
>>> np.allclose(sp.special.gamma(x + 1), sp.special.factorial(x))
True
>>> sp.special.gamma(5) < sp.special.gamma(5.5) < sp.special.gamma(6)
np.True_
```

The factorial function grows quickly, and so the gamma function overflows for moderate values of the argument. However, sometimes only the logarithm of the gamma function is needed. In such cases, we can compute the logarithm of the gamma function directly using `scipy.special.gammaln()`.

```
>>> x = [5, 50, 500]
>>> np.log(sp.special.gamma(x))
array([  3.17805383, 144.56574395,          inf])
>>> sp.special.gammaln(x)
array([   3.17805383,  144.56574395, 2605.11585036])
```

Such functions can often be used when the intermediate components of a calculation would overflow or underflow, but the final result would not. For example, suppose we wish to compute the ratio $\Gamma(500)/\Gamma(499)$.

```
>>> a = sp.special.gamma(500)
>>> b = sp.special.gamma(499)
>>> a, b
(np.float64(inf), np.float64(inf))
```

Both the numerator and denominator overflow, so performing a/b will not return the result we seek. However, the magnitude of the result should be moderate, so the use of logarithms comes to mind. Combining the identities $\log(a/b) = \log(a) - \log(b)$ and $\exp(\log(x)) = x$, we get:

```
>>> log_a = sp.special.gammaln(500)
>>> log_b = sp.special.gammaln(499)
>>> log_res = log_a - log_b
>>> res = np.exp(log_res)
>>> res
np.float64(499.0000000...)
```

Similarly, suppose we wish to compute the difference $\log(\Gamma(500) - \Gamma(499))$. For this, we use `scipy.special.logsumexp()`, which computes $\log(\exp(x)+\exp(y))$ using a numerical trick that avoids overflow.

```
>>> res = sp.special.logsumexp([log_a, log_b],
...                            b=[1, -1])  # weights the terms of the sum
>>> res
np.float64(2605.113844343...)
```

For more information about these and many other special functions, see the documentation of `scipy.special`.

5.3 Linear algebra operations: `scipy.linalg`

`scipy.linalg` provides a Python interface to efficient, compiled implementations of standard linear algebra operations: the BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage) libraries.

For example, the `scipy.linalg.det()` function computes the determinant of a square matrix:

```
>>> import scipy as sp
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> sp.linalg.det(arr)
np.float64(-2.0)
```

Mathematically, the solution of a linear system $Ax = b$ is $x = A^{-1}b$, but explicit inversion of a matrix is numerically unstable and should be avoided. Instead, use `scipy.linalg.solve()`:

```
>>> A = np.array([[1, 2],
...               [2, 3]])
>>> b = np.array([14, 23])
>>> x = sp.linalg.solve(A, b)
>>> x
array([4., 5.])
>>> np.allclose(A @ x, b)
True
```

Linear systems with special structure can often be solved more efficiently than more general systems. For example, systems with triangular matrices can be solved using `scipy.linalg.solve_triangular()`:

```
>>> A_upper = np.triu(A)
>>> A_upper
array([[1, 2],
       [0, 3]])
>>> np.allclose(sp.linalg.solve_triangular(A_upper, b, lower=False),
...             sp.linalg.solve(A_upper, b))
True
```

`scipy.linalg` also features matrix factorizations/decompositions such as the singular value decomposition.

```
>>> A = np.array([[1, 2],
...               [2, 3]])
>>> U, s, Vh = sp.linalg.svd(A)
>>> s  # singular values
array([4.23606798, 0.23606798])
```

The original matrix can be recovered by matrix multiplication of the factors:

```
>>> S = np.diag(s)  # convert to diagonal matrix before matrix multiplication
>>> A2 = U @ S @ Vh
>>> np.allclose(A2, A)
True
>>> A3 = (U * s) @ Vh  # more efficient: use array math broadcasting rules!
>>> np.allclose(A3, A)
True
```

Many other decompositions (e.g. LU, Cholesky, QR), solvers for structured linear systems (e.g. triangular, circulant), eigenvalue problem algorithms, matrix functions (e.g. matrix exponential), and routines for special matrix creation (e.g. block diagonal, toeplitz) are available in `scipy.linalg`.

5.4 Interpolation: `scipy.interpolate`

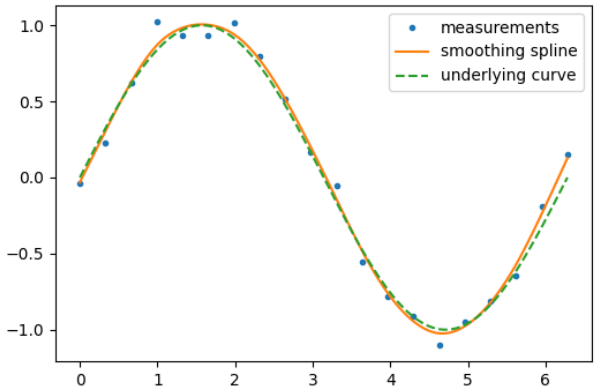
`scipy.interpolate` is used for fitting a function – an "interpolant" – to experimental or computed data. Once fit, the interpolant can be used to approximate the underlying function at intermediate points; it can also be used to compute the integral, derivative, or inverse of the function.

Some kinds of interpolants, known as "smoothing splines", are designed to generate smooth curves from noisy data. For example, suppose we have the following data:

```
>>> rng = np.random.default_rng(27446968)
>>> measured_time = np.linspace(0, 2*np.pi, 20)
>>> function = np.sin(measured_time)
>>> noise = rng.normal(loc=0, scale=0.1, size=20)
>>> measurements = function + noise
```

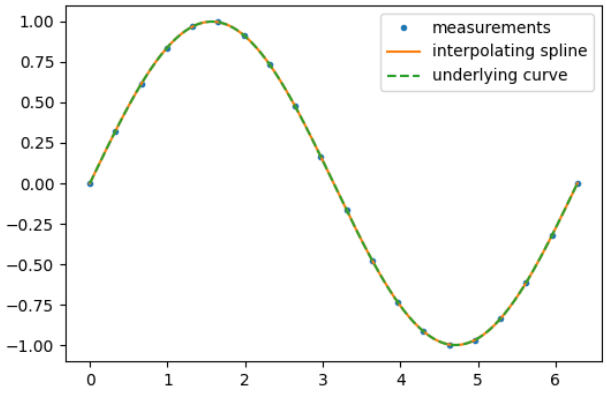
`scipy.interpolate.make_smoothing_spline()` can be used to form a curve similar to the underlying sine function.

```
>>> smoothing_spline = sp.interpolate.make_smoothing_spline(measured_time,␣
↪measurements)
>>> interpolation_time = np.linspace(0, 2*np.pi, 200)
>>> smooth_results = smoothing_spline(interpolation_time)
```



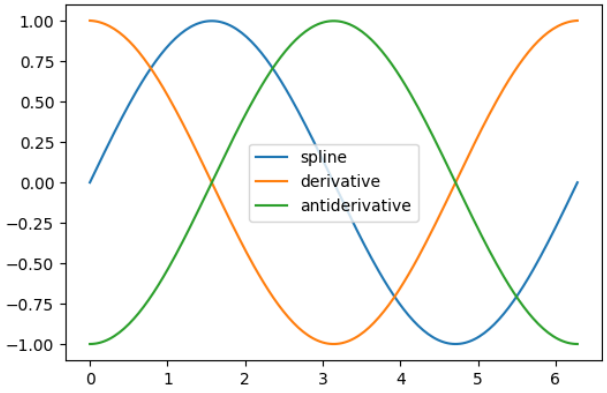
On the other hand, if the data are not noisy, it may be desirable to pass exactly through each point.

```
>>> interp_spline = sp.interpolate.make_interp_spline(measured_time, function)
>>> interp_results = interp_spline(interpolation_time)
```



The `derivative` and `antiderivative` methods of the result object can be used for differentiation and integration. For the latter, the constant of integration is assumed to be zero, but we can "wrap" the antiderivative to include a nonzero constant of integration.

```
>>> d_interp_spline = interp_spline.derivative()
>>> d_interp_results = d_interp_spline(interpolation_time)
>>> i_interp_spline = lambda t: interp_spline.antiderivative()(t) - 1
>>> i_interp_results = i_interp_spline(interpolation_time)
```



For functions that are monotonic on an interval (e.g. sin from $\pi/2$ to $3\pi/2$), we can reverse the arguments of `make_interp_spline` to interpolate the inverse function. Because the first argument is expected to be monotonically *increasing*, we also reverse the order of elements in the arrays with `numpy.flip()`.

```
>>> i = (measured_time > np.pi/2) & (measured_time < 3*np.pi/2)
>>> inverse_spline = sp.interpolate.make_interp_spline(np.flip(function[i]),
...                                                    np.flip(measured_time[i]))
>>> inverse_spline(0)
array(3.14159265)
```

See the summary exercise on *Maximum wind speed prediction at the Sprogø station* for a more advanced spline interpolation example, and read the SciPy interpolation tutorial and the `scipy.interpolate` documentation for much more information.

5.5 Optimization and fit: `scipy.optimize`

`scipy.optimize` provides algorithms for root finding, curve fitting, and more general optimization.

5.5.1 Root Finding

`scipy.optimize.root_scalar()` attempts to find a root of a specified scalar-valued function (i.e., an argument at which the function value is zero). Like many `scipy.optimize` functions, the function needs an initial guess of the solution, which the algorithm will refine until it converges or recognizes failure. We also provide the derivative to improve the rate of convergence.

```
>>> def f(x):
...     return (x-1)*(x-2)
>>> def df(x):
...     return 2*x - 3
>>> x0 = 0  # guess
>>> res = sp.optimize.root_scalar(f, x0=x0, fprime=df)
>>> res
      converged: True
           flag: converged
 function_calls: 12
     iterations: 6
           root: 1.0
         method: newton
```

Warning

None of the functions in `scipy.optimize` that accept a guess are guaranteed to converge for all possible guesses! (For example, try `x0=1.5` in the example above, where the derivative of the function is exactly zero.) If this occurs, try a different guess, adjust the options (like providing a `bracket` as shown below), or consider whether SciPy offers a more appropriate method for the problem.

Note that only one the root at `1.0` is found. By inspection, we can tell that there is a second root at `2.0`. We can direct the function toward a particular root by changing the guess or by passing a bracket that contains only the root we seek.

```
>>> res = sp.optimize.root_scalar(f, bracket=(1.5, 10))
>>> res.root
2.0
```

For multivariate problems, use `scipy.optimize.root()`.

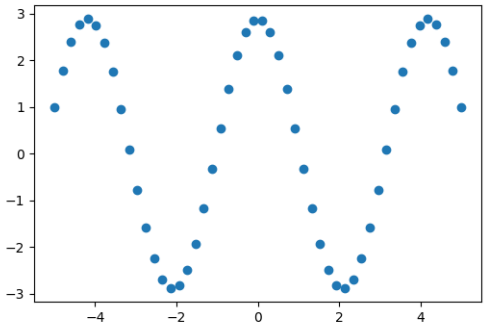
```
>>> def f(x):
...     # intersection of unit circle and line from origin
...     return [x[0]**2 + x[1]**2 - 1,
...             x[1] - x[0]]
>>> res = sp.optimize.root(f, x0=[0, 0])
>>> np.allclose(f(res.x), 0, atol=1e-10)
True
>>> np.allclose(res.x, np.sqrt(2)/2)
True
```

Over-constrained problems can be solved in the least-squares sense using `scipy.optimize.root()` with `method='lm'` (Levenberg-Marquardt).

```
>>> def f(x):
...     # intersection of unit circle, line from origin, and parabola
...     return [x[0]**2 + x[1]**2 - 1,
...             x[1] - x[0],
...             x[1] - x[0]**2]
>>> res = sp.optimize.root(f, x0=[1, 1], method='lm')
>>> res.success
True
>>> res.x
array([0.76096066, 0.66017736])
```

See the documentation of `scipy.optimize.root_scalar()` and `scipy.optimize.root()` for a variety of other solution algorithms and options.

5.5.2 Curve fitting

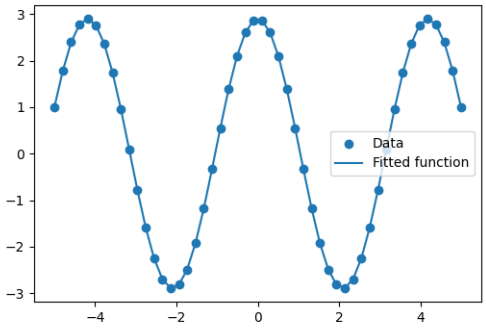


Suppose we have data that is sinusoidal but noisy:

```
>>> x = np.linspace(-5, 5, num=50)  # 50 values between -5 and 5
>>> noise = 0.01 * np.cos(100 * x)
>>> a, b = 2.9, 1.5
>>> y = a * np.cos(b * x) + noise
```

We can approximate the underlying amplitude, frequency, and phase from the data by least squares curve fitting. To begin, we write a function that accepts the independent variable as the first argument and all parameters to fit as separate arguments:

```
>>> def f(x, a, b, c):
...     return a * np.sin(b * x + c)
```



We then use `scipy.optimize.curve_fit()` to find a and b:

```
>>> params, _ = sp.optimize.curve_fit(f, x, y, p0=[2, 1, 3])
>>> params
array([2.900026  , 1.50012043, 1.57079633])
>>> ref = [a, b, np.pi/2]  # what we'd expect
>>> np.allclose(params, ref, rtol=1e-3)
True
```

Exercise: Curve fitting of temperature data

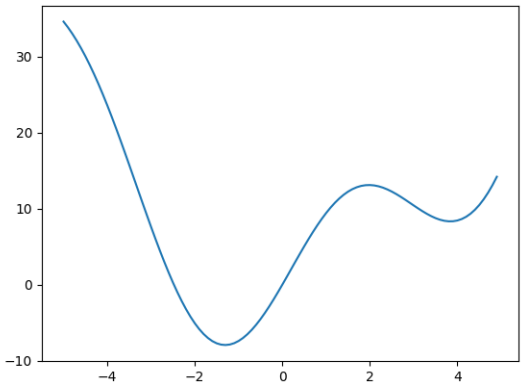
The temperature extremes in Alaska for each month, starting in January, are given by (in degrees Celsius):

```
max:  17,  19,  21,  28,  33,  38, 37,  37,  31,  23,  19,  18
min: -62, -59, -56, -46, -32, -18, -9, -13, -25, -46, -52, -58
```

1. Plot these temperature extremes.
2. Define a function that can describe min and max temperatures. Hint: this function has to have a period of 1 year. Hint: include a time offset.
3. Fit this function to the data with `scipy.optimize.curve_fit()`.
4. Plot the result. Is the fit reasonable? If not, why?
5. Is the time offset for min and max temperatures the same within the fit accuracy?

solution

5.5.3 Optimization



Suppose we wish to minimize the scalar-valued function of a single variable $f(x) = x^2 + 10\sin(x)$:

```
>>> def f(x):
...     return x**2 + 10*np.sin(x)
>>> x = np.arange(-5, 5, 0.1)
>>> plt.plot(x, f(x))
[<matplotlib.lines.Line2D object at ...>]
>>> plt.show()
```

We can see that the function has a local minimizer near $x = 3.8$ and a global minimizer near $x = -1.3$, but the precise values cannot be determined from the plot.

The most appropriate function for this purpose is `scipy.optimize.minimize_scalar()`. Since we know the approximate locations of the minima, we will provide bounds that restrict the search to the vicinity of the global minimum.

```
>>> res = sp.optimize.minimize_scalar(f, bounds=(-2, -1))
>>> res
 message: Solution found.
 success: True
  status: 0
     fun: -7.9458233756...
       x: -1.306440997...
     nit: 8
    nfev: 8
>>> res.fun == f(res.x)
np.True_
```

If we did not already know the approximate location of the global minimum, we could use one of SciPy's global minimizers, such as `scipy.optimize.differential_evolution()`. We are required to pass `bounds`, but they do not need to be tight.

```
>>> bounds=[(-5, 5)]  # list of lower, upper bound for each variable
>>> res = sp.optimize.differential_evolution(f, bounds=bounds)
>>> res
 message: Optimization terminated successfully.
 success: True
     fun: -7.9458233756...
       x: [-1.306e+00]
     nit: 6
    nfev: 111
     jac: [ 9.948e-06]
```

For multivariate optimization, a good choice for many problems is `scipy.optimize.minimize()`. Suppose we wish to find the minimum of a quadratic function of two variables, $f(x_0, x_1) = (x_0-1)^2+(x_1-2)^2$.

```
>>> def f(x):
...     return (x[0] - 1)**2 + (x[1] - 2)**2
```

Like `scipy.optimize.root()`, `scipy.optimize.minimize()` requires a guess `x0`. (Note that this is the initial value of *both* variables rather than the value of the variable we happened to label x_0.)

```
>>> res = sp.optimize.minimize(f, x0=[0, 0])
>>> res
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 1.70578...e-16
        x: [ 1.000e+00  2.000e+00]
```

(continues on next page)

(continued from previous page)

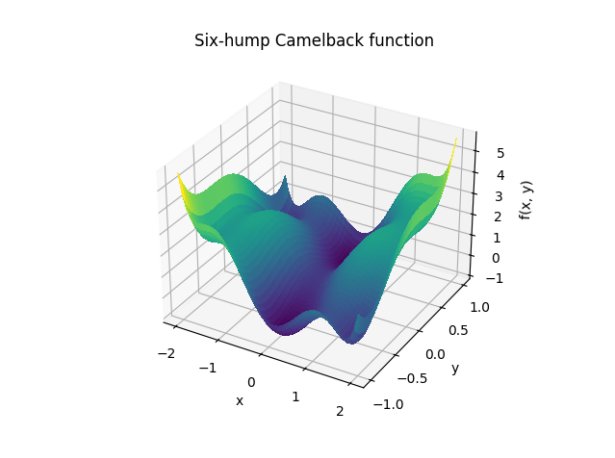
```
     nit: 2
     jac: [ 3.219e-09 -8.462e-09]
hess_inv: [[ 9.000e-01 -2.000e-01]
           [-2.000e-01  6.000e-01]]
    nfev: 9
    njev: 3
```

Maximization?

Is `scipy.optimize.minimize()` restricted to the solution of minimization problems? Nope! To solve a maximization problem, simply minimize the *negative* of the original objective function.

This barely scratches the surface of SciPy's optimization features, which include mixed integer linear programming, constrained nonlinear programming, and the solution of assignment problems. For much more information, see the documentation of `scipy.optimize` and the advanced chapter *Mathematical optimization: finding minima of functions*.

Exercise: 2-D minimization



The six-hump camelback function

$$f(x, y) = (4 - 2.1x^2 + \frac{x^4}{3})x^2 + xy + (4y^2 - 4)y^2$$

has multiple local minima. Find a global minimum (there is more than one, each with the same value of the objective function) and at least one other local minimum.

Hints:

- Variables can be restricted to $-2 < x < 2$ and $-1 < y < 1$.
- `numpy.meshgrid()` and `matplotlib.pyplot.imshow()` can help with visualization.
- Try minimizing with `scipy.optimize.minimize()` with an initial guess of $(x, y) = (0, 0)$. Does it find the global minimum, or converge to a local minimum? What about other initial guesses?
- Try minimizing with `scipy.optimize.differential_evolution()`.

solution

See the summary exercise on *Non linear least squares curve fitting: application to point extraction in topographical lidar data* for another, more advanced example.

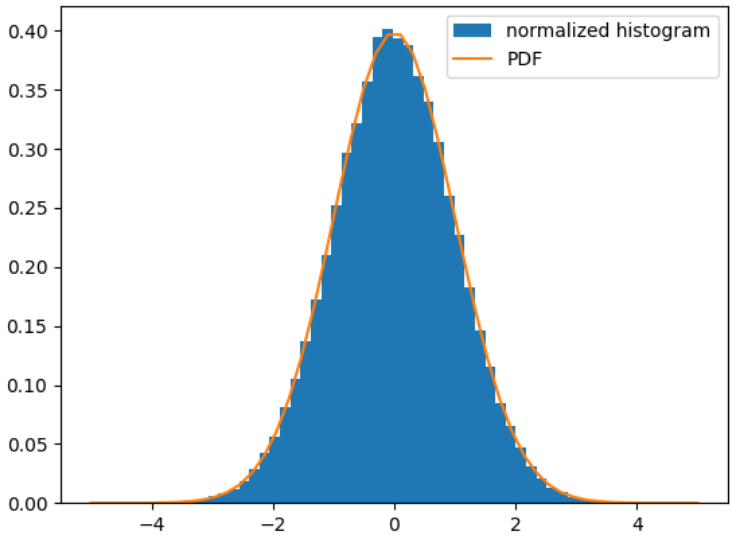
5.6 Statistics and random numbers: `scipy.stats`

`scipy.stats` contains fundamental tools for statistics in Python.

5.6.1 Statistical Distributions

Consider a random variable distributed according to the standard normal. We draw a sample consisting of 100000 observations from the random variable. The normalized histogram of the sample is an estimator of the random variable's probability density function (PDF):

```
>>> dist = sp.stats.norm(loc=0, scale=1)  # standard normal distribution
>>> sample = dist.rvs(size=100000)  # "random variate sample"
>>> plt.hist(sample, bins=50, density=True, label='normalized histogram')
>>> x = np.linspace(-5, 5)
>>> plt.plot(x, dist.pdf(x), label='PDF')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.legend()
<matplotlib.legend.Legend object at ...>
```



Distribution objects and frozen distributions

Each of the 100+ `scipy.stats` distribution families is represented by an *object* with a *__call__* method. Here, we call the `scipy.stats.norm` object to specify its location and scale, and it returns a *frozen* distribution: a particular element of a distribution family with all parameters fixed. The frozen distribution object has methods to compute essential functions of the particular distribution.

Suppose we knew that the sample had been drawn from a distribution belonging to the family of normal distributions, but we did not know the particular distribution's location (mean) and scale (standard deviation). We perform maximum likelihood estimation of the unknown parameters using the distribution family's `fit` method:

```
>>> loc, scale = sp.stats.norm.fit(sample)
>>> loc
np.float64(0.0015767005...)
>>> scale
np.float64(0.9973396878...)
```

Since we know the true parameters of the distribution from which the sample was drawn, we are not surprised that these estimates are similar.

Exercise: Probability distributions

Generate 1000 random variates from a gamma distribution with a shape parameter of 1. *Hint: the shape parameter is passed as the first argument when freezing the distribution.* Plot the histogram of the sample, and overlay the distribution's PDF. Estimate the shape parameter from the sample using the `fit` method.

Extra: the distributions have many useful methods. Explore them using tab completion. Plot the cumulative density function of the distribution, and compute the variance.

5.6.2 Sample Statistics and Hypothesis Tests

The sample mean is an estimator of the mean of the distribution from which the sample was drawn:

```
>>> np.mean(sample)
np.float64(0.001576700508...)
```

NumPy includes some of the most fundamental sample statistics (e.g. `numpy.mean()`, `numpy.var()`, `numpy.percentile()`); `scipy.stats` includes many more. For instance, the geometric mean is a common measure of central tendency for data that tends to be distributed over many orders of magnitude.

```
>>> sp.stats.gmean(2**sample)
np.float64(1.0010934829...)
```

SciPy also includes a variety of hypothesis tests that produce a sample statistic and a p-value. For instance, suppose we wish to test the null hypothesis that `sample` was drawn from a normal distribution:

```
>>> res = sp.stats.normaltest(sample)
>>> res.statistic
np.float64(5.20841759...)
>>> res.pvalue
np.float64(0.07396163283...)
```

Here, `statistic` is a sample statistic that tends to be high for samples that are drawn from non-normal distributions. `pvalue` is the probability of observing such a high value of the statistic for a sample that *has* been drawn from a normal distribution. If the p-value is unusually small, this may be taken as evidence that `sample` was *not* drawn from the normal distribution. Our statistic and p-value are moderate, so the test is inconclusive.

There are many other features of `scipy.stats`, including circular statistics, quasi-Monte Carlo methods, and resampling methods. For much more information, see the documentation of `scipy.stats` and the advanced chapter *statistics*.

5.7 Numerical integration: `scipy.integrate`

5.7.1 Quadrature

Suppose we wish to compute the definite integral $\int_0^{\pi/2} \sin(t)dt$ numerically. `scipy.integrate.quad()` chooses one of several adaptive techniques depending on the parameters, and is therefore the recommended first choice for integration of function of a single variable:

```
>>> integral, error_estimate = sp.integrate.quad(np.sin, 0, np.pi/2)
>>> np.allclose(integral, 1)  # numerical result ~ analytical result
True
```

(continues on next page)

(continued from previous page)

```
>>> abs(integral - 1) < error_estimate  #  actual error < estimated error
True
```

Other functions for *numerical quadrature*, including integration of multivariate functions and approximating integrals from samples, are available in `scipy.integrate`.

5.7.2 Initial Value Problems

`scipy.integrate` also features routines for integrating Ordinary Differential Equations (ODE). For example, `scipy.integrate.solve_ivp()` integrates ODEs of the form:

$$\frac{dy}{dt} = f(t, y(t))$$

from an initial time t_0 and initial state $y(t = t_0) = t_0$ to a final time t_f or until an event occurs (e.g. a specified state is reached).

As an introduction, consider the initial value problem given by $\frac{dy}{dt} = -2y$ and the initial condition $y(t = 0) = 1$ on the interval $t = 0 \ldots 4$. We begin by defining a callable that computes $f(t, y(t))$ given the current time and state.

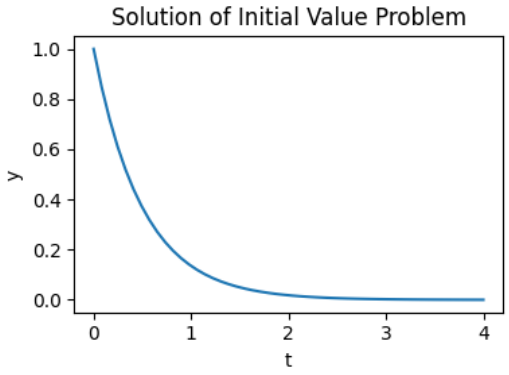
```
>>> def f(t, y):
...     return -2 * y
```

Then, to compute `y` as a function of time:

```
>>> t_span = (0, 4)  # time interval
>>> t_eval = np.linspace(*t_span)  # times at which to evaluate `y`
>>> y0 = [1,]  # initial state
>>> res = sp.integrate.solve_ivp(f, t_span=t_span, y0=y0, t_eval=t_eval)
```

and plot the result:

```
>>> plt.plot(res.t, res.y[0])
[<matplotlib.lines.Line2D object at ...>]
>>> plt.xlabel('t')
Text(0.5, ..., 't')
>>> plt.ylabel('y')
Text(..., 0.5, 'y')
>>> plt.title('Solution of Initial Value Problem')
Text(0.5, 1.0, 'Solution of Initial Value Problem')
```



Let us integrate a more complex ODE: a damped spring-mass oscillator. The position of a mass attached to a spring obeys the 2nd order ODE $\ddot{y} + 2\zeta\omega_0\dot{y} + \omega_0^2 y = 0$ with natural frequency $\omega_0 = \sqrt{k/m}$, damping ratio $\zeta = c/(2m\omega_0)$, spring constant k, mass m, and damping coefficient c.

Before using `scipy.integrate.solve_ivp()`, the 2nd order ODE needs to be transformed into a system of first-order ODEs. Note that

$$\frac{dy}{dt} = \dot{y}\frac{d\dot{y}}{dt} = \ddot{y} = -(2\zeta\omega_0\dot{y} + \omega_0^2 y)$$

If we define $z = [z_0, z_1]$ where $z_0 = y$ and $z_1 = \dot{y}$, then the first order equation:

$$\frac{dz}{dt} = \begin{bmatrix} \frac{dz_0}{dt} \\ \frac{dz_1}{dt} \end{bmatrix} = \begin{bmatrix} z_1 \\ -(2\zeta\omega_0 z_1 + \omega_0^2 z_0) \end{bmatrix}$$

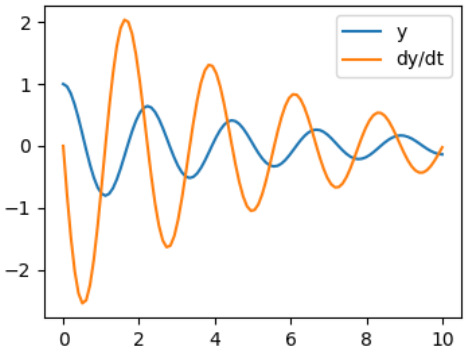
is equivalent to the original second order equation.

We set:

```
>>> m = 0.5  # kg
>>> k = 4  # N/m
>>> c = 0.4  # N s/m
>>> zeta = c / (2 * m * np.sqrt(k/m))
>>> omega = np.sqrt(k / m)
```

and define the function that computes $\dot{z} = f(t, z(t))$:

```
>>> def f(t, z, zeta, omega):
...     return (z[1], -2.0 * zeta * omega * z[1] - omega**2 * z[0])
```



Integration of the system follows:

```
>>> t_span = (0, 10)
>>> t_eval = np.linspace(*t_span, 100)
>>> z0 = [1, 0]
>>> res = sp.integrate.solve_ivp(f, t_span, z0, t_eval=t_eval,
...                              args=(zeta, omega), method='LSODA')
```

Tip

With the option *method='LSODA'*, `scipy.integrate.solve_ivp()` uses the LSODA (Livermore Solver for Ordinary Differential equations with Automatic method switching for stiff and non-stiff problems). See the ODEPACK Fortran library for more details.

See also

Partial Differential Equations

There is no Partial Differential Equations (PDE) solver in SciPy. Some Python packages for solving PDE's are available, such as fipy or SfePy.

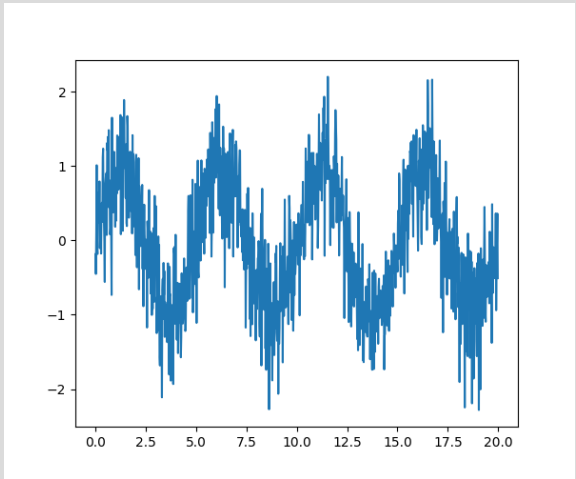
5.8 Fast Fourier transforms: `scipy.fft`

The `scipy.fft` module computes fast Fourier transforms (FFTs) and offers utilities to handle them. Some important functions are:

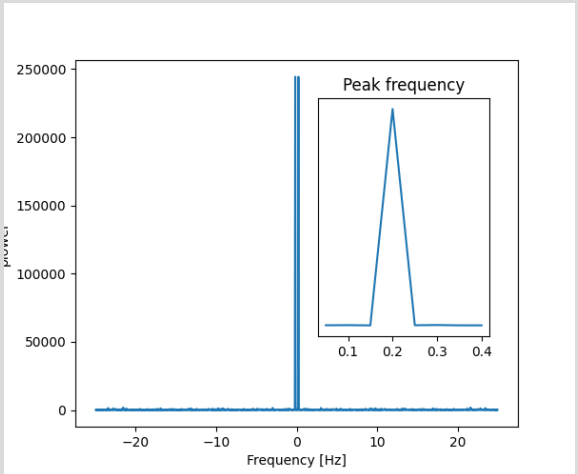
- `scipy.fft.fft()` to compute the FFT
- `scipy.fft.fftfreq()` to generate the sampling frequencies
- `scipy.fft.ifft()` to compute the inverse FFT, from frequency space to signal space

As an illustration, a (noisy) input signal (`sig`), and its FFT:

```
>>> sig_fft = sp.fft.fft(sig)
>>> freqs = sp.fft.fftfreq(sig.size, d=time_step)
```



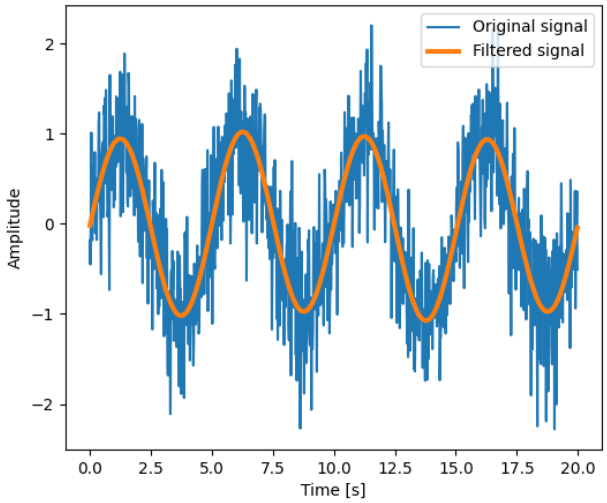
Signal



FFT

As the signal comes from a real-valued function, the Fourier transform is symmetric.

The peak signal frequency can be found with `freqs[power.argmax()]`



Setting the Fourier component above this frequency to zero and inverting the FFT with `scipy.fft.ifft()`, gives a filtered signal.

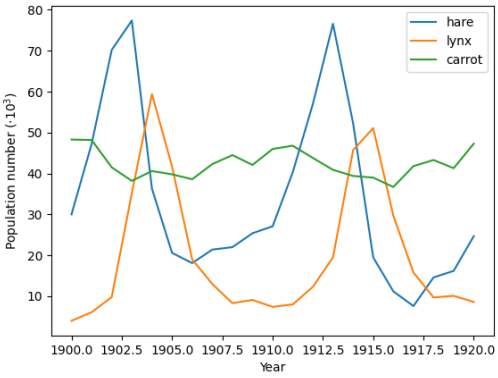
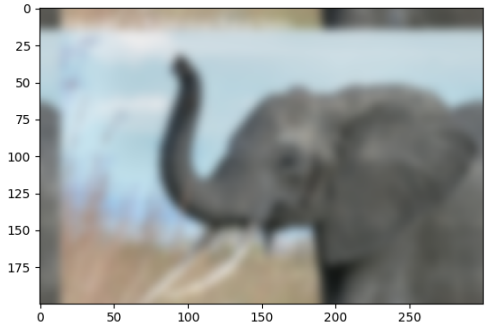
Note

The code of this example can be found *here*

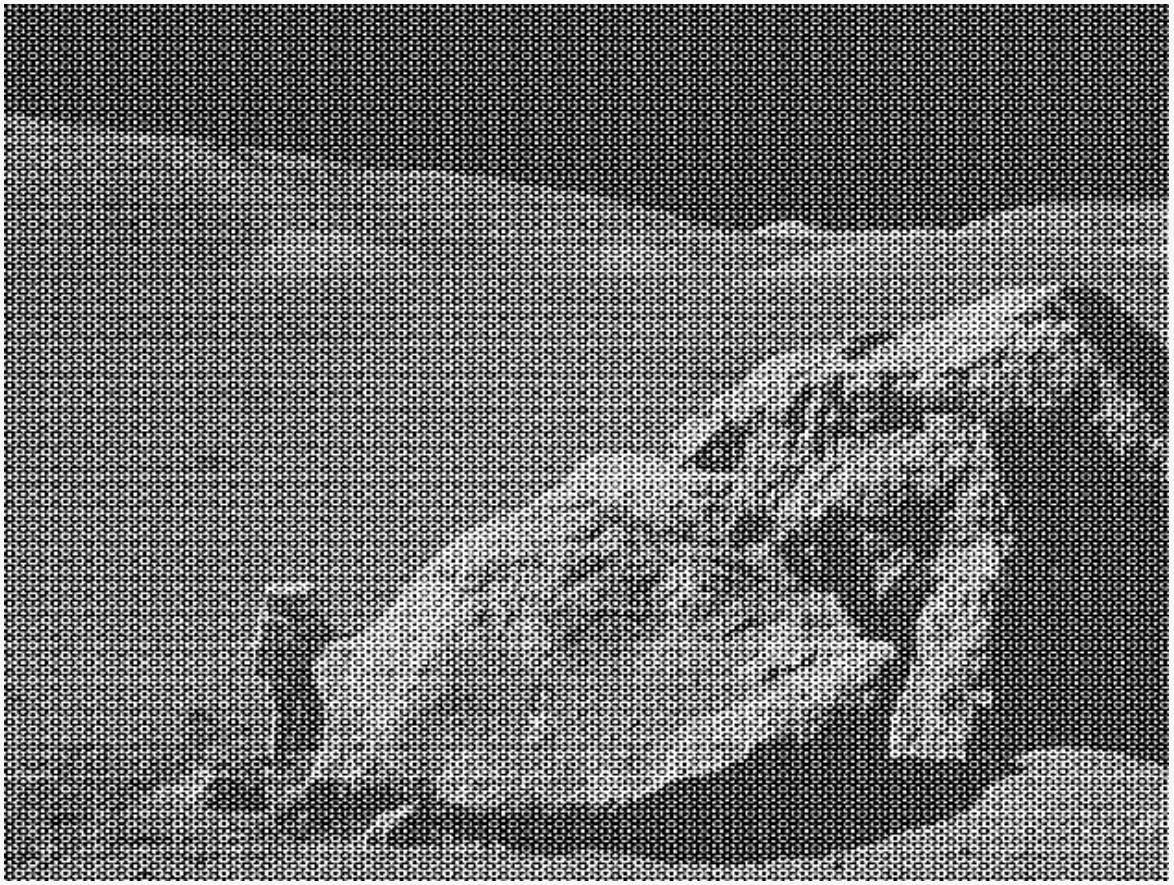
numpy.fft

NumPy also has an implementation of FFT (`numpy.fft`). However, the SciPy one should be preferred, as it uses more efficient underlying implementations.

Fully worked examples:

Crude periodicity finding (*link*)	Gaussian image blur (*link*)
	

Exercise: Denoise moon landing image



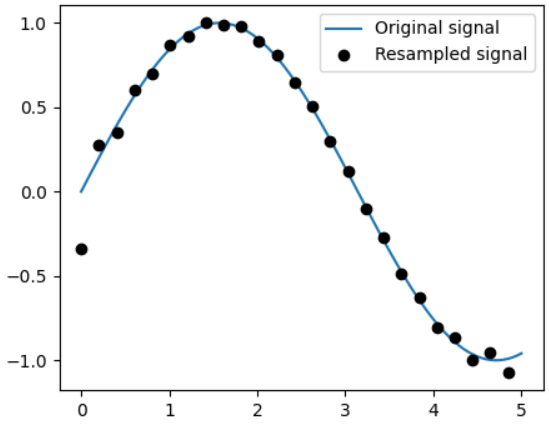
1. Examine the provided image `moonlanding.png`, which is heavily contaminated with periodic noise. In this exercise, we aim to clean up the noise using the Fast Fourier Transform.
2. Load the image using `matplotlib.pyplot.imread()`.
3. Find and use the 2-D FFT function in `scipy.fft`, and plot the spectrum (Fourier transform of) the image. Do you have any trouble visualising the spectrum? If so, why?
4. The spectrum consists of high and low frequency components. The noise is contained in the high-frequency part of the spectrum, so set some of those components to zero (use array slicing).
5. Apply the inverse Fourier transform to see the resulting image.

Solution

5.9 Signal processing: `scipy.signal`

Tip

`scipy.signal` is for typical signal processing: 1D, regularly-sampled signals.



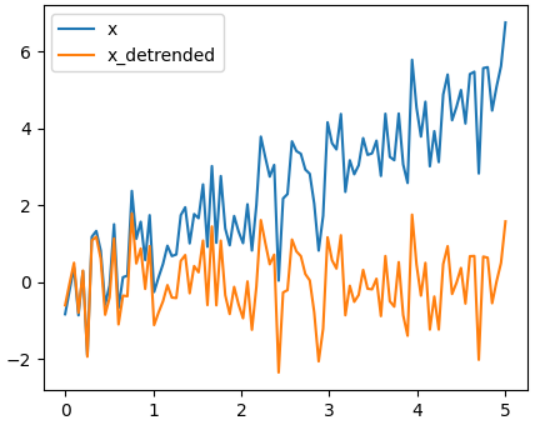
Resampling `scipy.signal.resample()`: resample a signal to n points using FFT.

```
>>> t = np.linspace(0, 5, 100)
>>> x = np.sin(t)

>>> x_resampled = sp.signal.resample(x, 25)

>>> plt.plot(t, x)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t[::4], x_resampled, 'ko')
[<matplotlib.lines.Line2D object at ...>]
```

Tip

Notice how on the side of the window the resampling is less accurate and has a rippling effect.

This resampling is different from the *interpolation* provided by `scipy.interpolate` as it only applies to regularly sampled data.



Detrending `scipy.signal.detrend()`: remove linear trend from signal:

```
>>> t = np.linspace(0, 5, 100)
>>> rng = np.random.default_rng()
>>> x = t + rng.normal(size=100)
```

(continues on next page)

(continued from previous page)

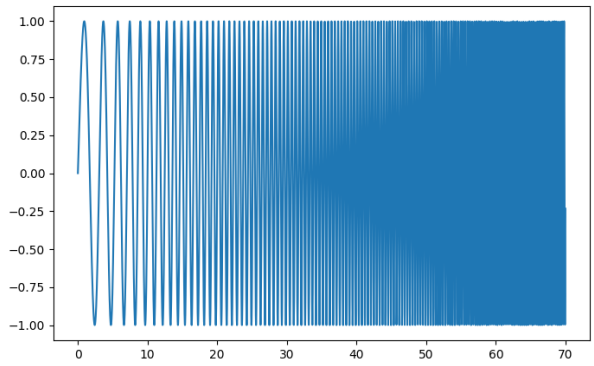
```
>>> x_detrended = sp.signal.detrend(x)

>>> plt.plot(t, x)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t, x_detrended)
[<matplotlib.lines.Line2D object at ...>]
```

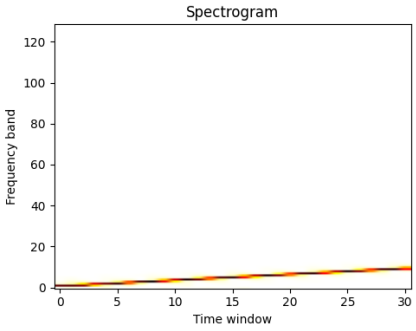
Filtering: For non-linear filtering, `scipy.signal` has filtering (median filter `scipy.signal.medfilt()`, Wiener `scipy.signal.wiener()`), but we will discuss this in the image section.

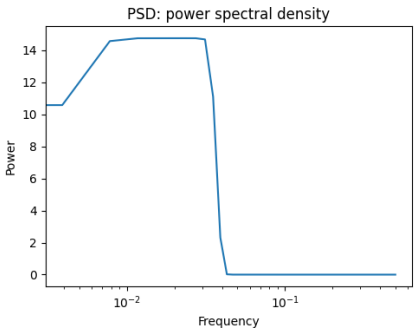
Tip

`scipy.signal` also has a full-blown set of tools for the design of linear filter (finite and infinite response filters), but this is out of the scope of this tutorial.

Spectral analysis: `scipy.signal.spectrogram()` compute a spectrogram –frequency spectrums over consecutive time windows–, while `scipy.signal.welch()` comptes a power spectrum density (PSD).







5.10 Image manipulation: `scipy.ndimage`

`scipy.ndimage` provides manipulation of n-dimensional arrays as images.

5.10.1 Geometrical transformations on images

Changing orientation, resolution, ..

```
>>> import scipy as sp
```

(continues on next page)

(continued from previous page)

```
>>> # Load an image
>>> face = sp.datasets.face(gray=True)

>>> # Shift, rotate and zoom it
>>> shifted_face = sp.ndimage.shift(face, (50, 50))
>>> shifted_face2 = sp.ndimage.shift(face, (50, 50), mode='nearest')
>>> rotated_face = sp.ndimage.rotate(face, 30)
>>> cropped_face = face[50:-50, 50:-50]
>>> zoomed_face = sp.ndimage.zoom(face, 2)
>>> zoomed_face.shape
(1536, 2048)
```

```
>>> plt.subplot(151)
<Axes: >

>>> plt.imshow(shifted_face, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>

>>> plt.axis('off')
(np.float64(-0.5), np.float64(1023.5), np.float64(767.5), np.float64(-0.5))

>>> # etc.
```

5.10.2 Image filtering

Generate a noisy face:

```
>>> import scipy as sp
>>> face = sp.datasets.face(gray=True)
>>> face = face[:512, -512:]  # crop out square on right
>>> import numpy as np
>>> noisy_face = np.copy(face).astype(float)
>>> rng = np.random.default_rng()
>>> noisy_face += face.std() * 0.5 * rng.standard_normal(face.shape)
```

Apply a variety of filters on it:

```
>>> blurred_face = sp.ndimage.gaussian_filter(noisy_face, sigma=3)
>>> median_face = sp.ndimage.median_filter(noisy_face, size=5)
>>> wiener_face = sp.signal.wiener(noisy_face, (5, 5))
```

Other filters in `scipy.ndimage.filters` and `scipy.signal` can be applied to images.

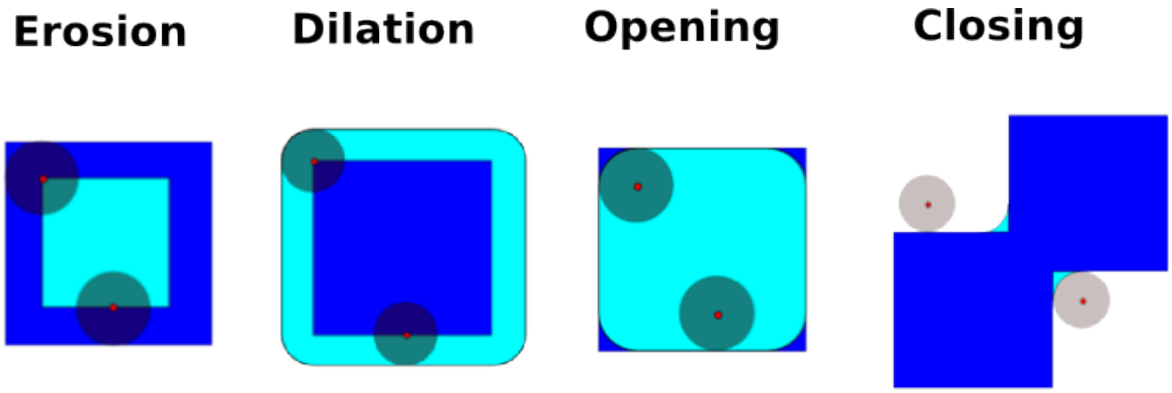
Exercise

Compare histograms for the different filtered images.

5.10.3 Mathematical morphology

Tip

Mathematical morphology stems from set theory. It characterizes and transforms geometrical structures. Binary (black and white) images, in particular, can be transformed using this theory: the sets to be transformed are the sets of neighboring non-zero-valued pixels. The theory was also extended to gray-valued images.



Mathematical-morphology operations use a *structuring element* in order to modify geometrical structures.

Let us first generate a structuring element:

```
>>> el = sp.ndimage.generate_binary_structure(2, 1)
>>> el
array([[False, True, False],
       [...True, True, True],
       [False, True, False]])
>>> el.astype(int)
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```

- **Erosion** `scipy.ndimage.binary_erosion()`

```
>>> a = np.zeros((7, 7), dtype=int)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> sp.ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> # Erosion removes objects smaller than the structure
>>> sp.ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

- **Dilation** `scipy.ndimage.binary_dilation()`

```
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.]])
>>> sp.ndimage.binary_dilation(a).astype(a.dtype)
array([[0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  1.,  1.,  1.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.]])
```

- **Opening** `scipy.ndimage.binary_opening()`

```
>>> a = np.zeros((5, 5), dtype=int)
>>> a[1:4, 1:4] = 1
>>> a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
```

(continues on next page)

(continued from previous page)

```
>>> sp.ndimage.binary_opening(a, structure=np.ones((3, 3))).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> sp.ndimage.binary_opening(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

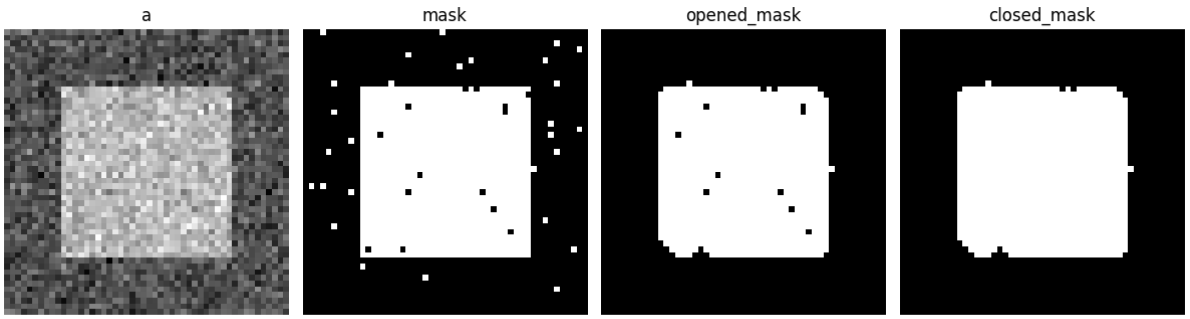
- **Closing:** `scipy.ndimage.binary_closing()`

Exercise

Check that opening amounts to eroding, then dilating.

An opening operation removes small structures, while a closing operation fills small holes. Such operations can therefore be used to "clean" an image.

```
>>> a = np.zeros((50, 50))
>>> a[10:-10, 10:-10] = 1
>>> rng = np.random.default_rng()
>>> a += 0.25 * rng.standard_normal(a.shape)
>>> mask = a>=0.5
>>> opened_mask = sp.ndimage.binary_opening(mask)
>>> closed_mask = sp.ndimage.binary_closing(opened_mask)
```



Exercise

Check that the area of the reconstructed square is smaller than the area of the initial square. (The opposite would occur if the closing step was performed *before* the opening).

For *gray-valued* images, eroding (resp. dilating) amounts to replacing a pixel by the minimal (resp. maximal) value among pixels covered by the structuring element centered on the pixel of interest.

```
>>> a = np.zeros((7, 7), dtype=int)
>>> a[1:6, 1:6] = 3
>>> a[4, 4] = 2; a[2, 3] = 1
>>> a
```

(continues on next page)

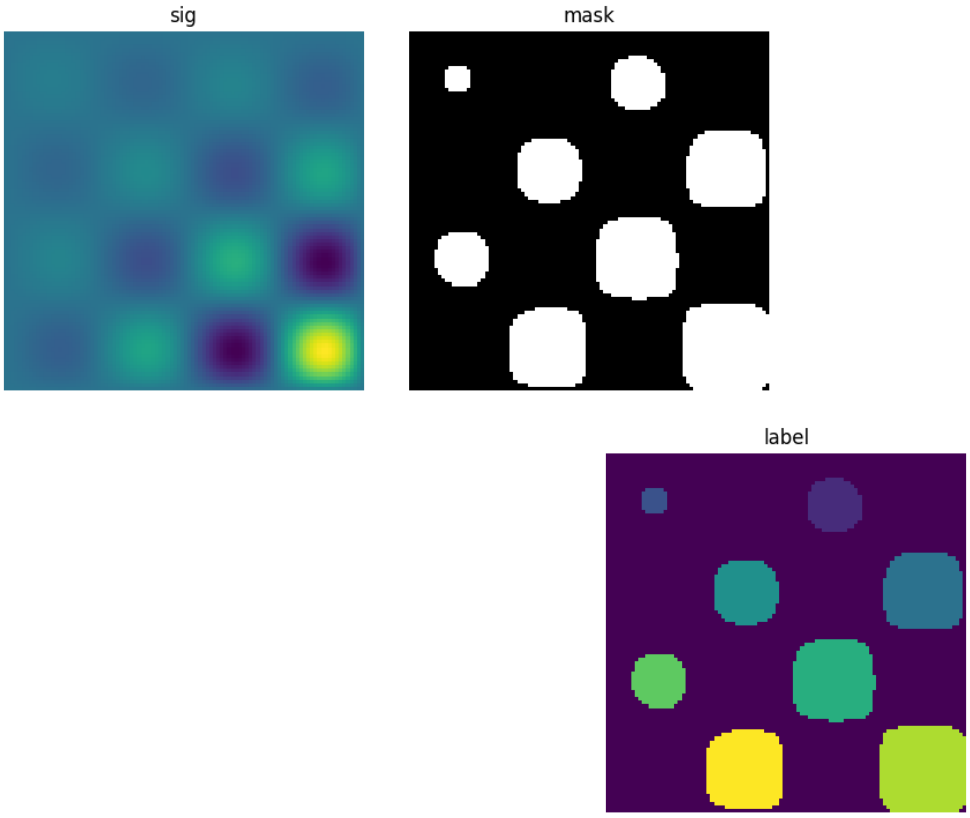
(continued from previous page)

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 1, 3, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 3, 2, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> sp.ndimage.grey_erosion(a, size=(3, 3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 3, 2, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

5.10.4 Connected components and measurements on images

Let us first generate a nice synthetic binary image.

```
>>> x, y = np.indices((100, 100))
>>> sig = np.sin(2*np.pi*x/50.) * np.sin(2*np.pi*y/50.) * (1+x*y/50.**2)**2
>>> mask = sig > 1
```

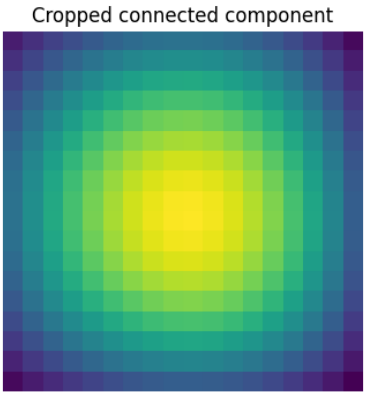


`scipy.ndimage.label()` assigns a different label to each connected component:

```
>>> labels, nb = sp.ndimage.label(mask)
>>> nb
8
```

Now compute measurements on each connected component:

```
>>> areas = sp.ndimage.sum(mask, labels, range(1, labels.max()+1))
>>> areas   # The number of pixels in each connected component
array([190.,   45.,  424.,  278.,  459.,  190.,  549.,  424.])
>>> maxima = sp.ndimage.maximum(sig, labels, range(1, labels.max()+1))
>>> maxima  # The maximum signal in each connected component
array([ 1.80238238,  1.13527605,  5.51954079,  2.49611818, 6.71673619,
        1.80238238, 16.76547217,  5.51954079])
```



Extract the 4th connected component, and crop the array around it:

```
>>> sp.ndimage.find_objects(labels)[3]
(slice(30, 48, None), slice(30, 48, None))
>>> sl = sp.ndimage.find_objects(labels)[3]
>>> import matplotlib.pyplot as plt
>>> plt.imshow(sig[sl])
<matplotlib.image.AxesImage object at ...>
```

See the summary exercise on *Image processing application: counting bubbles and unmolten grains* for a more advanced example.

5.11 Summary exercises on scientific computing

The summary exercises use mainly NumPy, SciPy and Matplotlib. They provide some real-life examples of scientific computing with Python. Now that the basics of working with NumPy and SciPy have been introduced, the interested user is invited to try these exercises.

5.11.1 Maximum wind speed prediction at the Sprogø station

The exercise goal is to predict the maximum wind speed occurring every 50 years even if no measure exists for such a period. The available data are only measured over 21 years at the Sprogø meteorological station located in Denmark. First, the statistical steps will be given and then illustrated with functions from the scipy.interpolate module. At the end the interested readers are invited to compute results from raw data and in a slightly different approach.

Statistical approach

The annual maxima are supposed to fit a normal probability density function. However such function is not going to be estimated because it gives a probability from a wind speed maxima. Finding the maximum wind speed occurring every 50 years requires the opposite approach, the result needs to be found from a defined probability. That is the quantile function role and the exercise goal will be to find it. In the current model, it is supposed that the maximum wind speed occurring every 50 years is defined as the upper 2% quantile.

By definition, the quantile function is the inverse of the cumulative distribution function. The latter describes the probability distribution of an annual maxima. In the exercise, the cumulative probability

`p_i` for a given year `i` is defined as `p_i = i/(N+1)` with `N = 21`, the number of measured years. Thus it will be possible to calculate the cumulative probability of every measured wind speed maxima. From those experimental points, the scipy.interpolate module will be very useful for fitting the quantile function. Finally the 50 years maxima is going to be evaluated from the cumulative probability of the 2% quantile.

Computing the cumulative probabilities

The annual wind speeds maxima have already been computed and saved in the NumPy format in the file `examples/max-speeds.npy`, thus they will be loaded by using NumPy:

```
>>> import numpy as np
>>> max_speeds = np.load('intro/scipy/summary-exercises/examples/max-speeds.npy')
>>> years_nb = max_speeds.shape[0]
```

Following the cumulative probability definition `p_i` from the previous section, the corresponding values will be:

```
>>> cprob = (np.arange(years_nb, dtype=np.float32) + 1)/(years_nb + 1)
```

and they are assumed to fit the given wind speeds:

```
>>> sorted_max_speeds = np.sort(max_speeds)
```

Prediction with UnivariateSpline

In this section the quantile function will be estimated by using the `UnivariateSpline` class which can represent a spline from points. The default behavior is to build a spline of degree 3 and points can have different weights according to their reliability. Variants are `InterpolatedUnivariateSpline` and `LSQUnivariateSpline` on which errors checking is going to change. In case a 2D spline is wanted, the `BivariateSpline` class family is provided. All those classes for 1D and 2D splines use the FITPACK Fortran subroutines, that's why a lower library access is available through the `splrep` and `splev` functions for respectively representing and evaluating a spline. Moreover interpolation functions without the use of FITPACK parameters are also provided for simpler use.

For the Sprogø maxima wind speeds, the `UnivariateSpline` will be used because a spline of degree 3 seems to correctly fit the data:

```
>>> import scipy as sp
>>> quantile_func = sp.interpolate.UnivariateSpline(cprob, sorted_max_speeds)
```

The quantile function is now going to be evaluated from the full range of probabilities:

```
>>> nprob = np.linspace(0, 1, 100)
>>> fitted_max_speeds = quantile_func(nprob)
```

In the current model, the maximum wind speed occurring every 50 years is defined as the upper 2% quantile. As a result, the cumulative probability value will be:

```
>>> fifty_prob = 1. - 0.02
```

So the storm wind speed occurring every 50 years can be guessed by:

```
>>> fifty_wind = quantile_func(fifty_prob)
>>> fifty_wind
array(32.97989825...)
```

The results are now gathered on a Matplotlib figure:

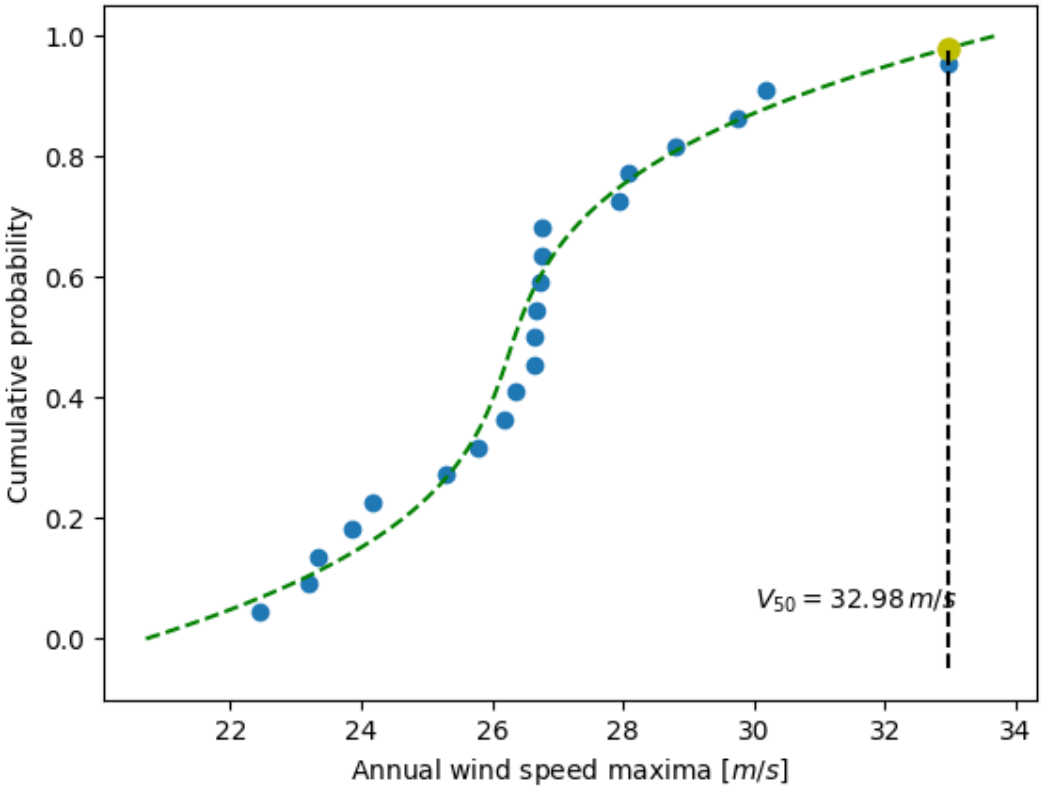


Fig. 1: Solution: `Python source file`

Exercise with the Gumbell distribution

The interested readers are now invited to make an exercise by using the wind speeds measured over 21 years. The measurement period is around 90 minutes (the original period was around 10 minutes but the file size has been reduced for making the exercise setup easier). The data are stored in NumPy format inside the file `examples/sprog-windspeeds.npy`. Do not look at the source code for the plots until you have completed the exercise.

- The first step will be to find the annual maxima by using NumPy and plot them as a matplotlib bar figure.

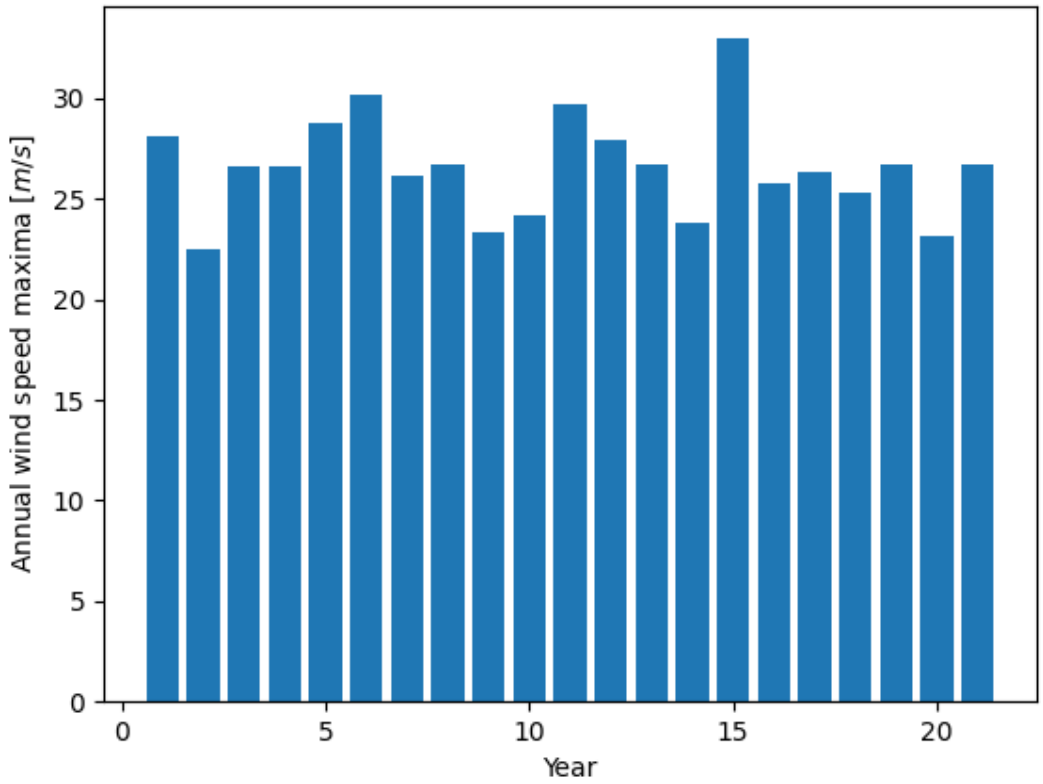


Fig. 2: Solution: `Python source file`

- The second step will be to use the Gumbell distribution on cumulative probabilities `p_i` defined as `-log( -log(p_i) )` for fitting a linear quantile function (remember that you can define the degree of the `UnivariateSpline`). Plotting the annual maxima versus the Gumbell distribution should give you the following figure.
- The last step will be to find 34.23 m/s for the maximum wind speed occurring every 50 years.

5.11.2 Non linear least squares curve fitting: application to point extraction in topographical lidar data

The goal of this exercise is to fit a model to some data. The data used in this tutorial are lidar data and are described in details in the following introductory paragraph. If you're impatient and want to practice now, please skip it and go directly to *Loading and visualization*.

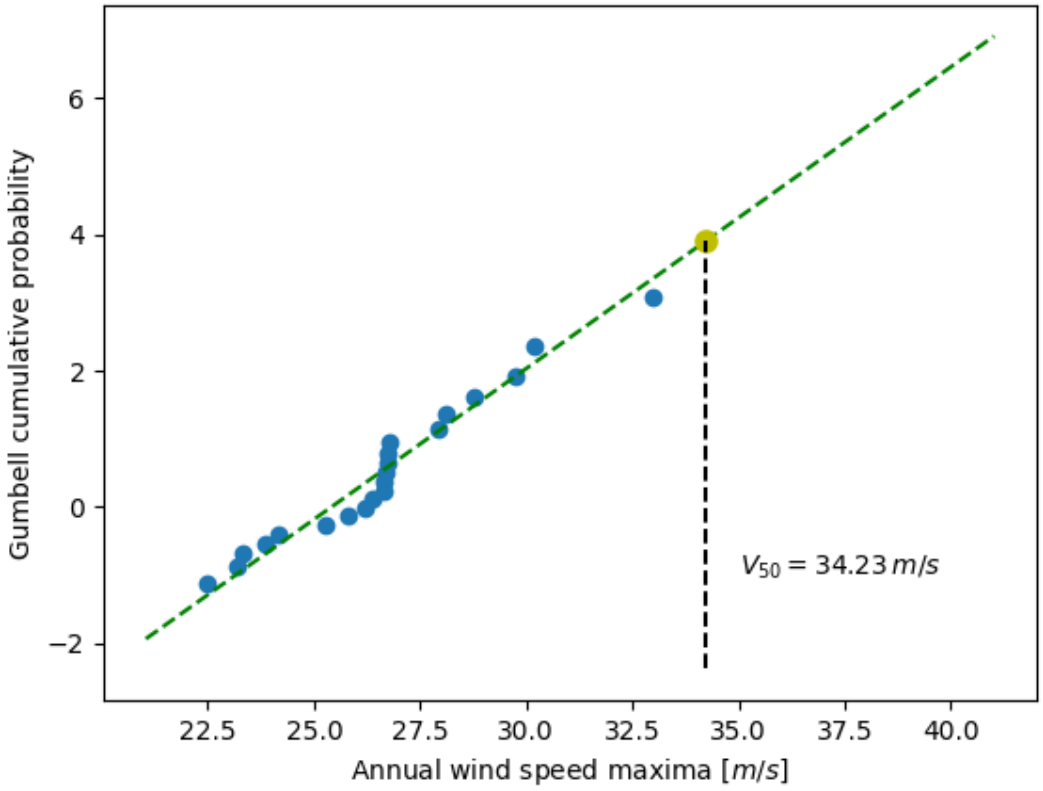


Fig. 3: Solution: `Python source file`

Introduction

Lidars systems are optical rangefinders that analyze property of scattered light to measure distances. Most of them emit a short light impulsion towards a target and record the reflected signal. This signal is then processed to extract the distance between the lidar system and the target.

Topographical lidar systems are such systems embedded in airborne platforms. They measure distances between the platform and the Earth, so as to deliver information on the Earth's topography (see[1] for more details).

In this tutorial, the goal is to analyze the waveform recorded by the lidar system[2]. Such a signal contains peaks whose center and amplitude permit to compute the position and some characteristics of the hit target. When the footprint of the laser beam is around 1m on the Earth surface, the beam can hit multiple targets during the two-way propagation (for example the ground and the top of a tree or building). The sum of the contributions of each target hit by the laser beam then produces a complex signal with multiple peaks, each one containing information about one target.

One state of the art method to extract information from these data is to decompose them in a sum of Gaussian functions where each function represents the contribution of a target hit by the laser beam.

Therefore, we use the `scipy.optimize` module to fit a waveform to one or a sum of Gaussian functions.

Loading and visualization

Load the first waveform using:

```
>>> import numpy as np
>>> waveform_1 = np.load('intro/scipy/summary-exercises/examples/waveform_1.npy')
```

and visualize it:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(len(waveform_1))
>>> plt.plot(t, waveform_1)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.show()
```

As shown below, this waveform is a 80-bin-length signal with a single peak with an amplitude of approximately 30 in the 15 nanosecond bin. Additionally, the base level of noise is approximately 3. These values can be used in the initial solution.

Fitting a waveform with a simple Gaussian model

The signal is very simple and can be modeled as a single Gaussian function and an offset corresponding to the background noise. To fit the signal with the function, we must:

- define the model
- propose an initial solution
- call `scipy.optimize.leastsq`

Model

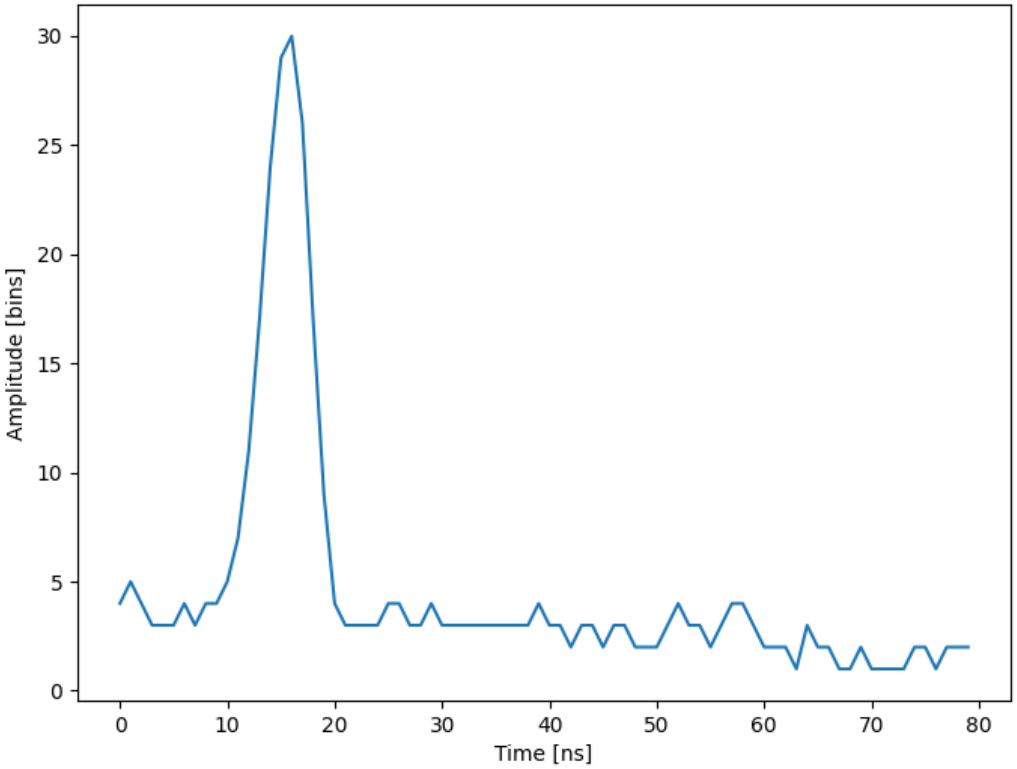
A Gaussian function defined by

$$B + A \exp\left\{-\left(\frac{t-\mu}{\sigma}\right)^2\right\}$$

can be defined in python by:

[1] Mallet, C. and Bretar, F. Full-Waveform Topographic Lidar: State-of-the-Art. *ISPRS Journal of Photogrammetry and Remote Sensing* 64(1), pp.1-16, January 2009 http://dx.doi.org/10.1016/j.isprsjprs.2008.09.007

[2] The data used for this tutorial are part of the demonstration data available for the FullAnalyze software and were kindly provided by the GIS DRAIX.




```
>>> def model(t, coeffs):
...     return coeffs[0] + coeffs[1] * np.exp( - ((t-coeffs[2])/coeffs[3])**2 )
```

where

- `coeffs[0]` is B (noise)
- `coeffs[1]` is A (amplitude)
- `coeffs[2]` is μ (center)
- `coeffs[3]` is σ (width)

Initial solution

One possible initial solution that we determine by inspection is:

```
>>> x0 = np.array([3, 30, 15, 1], dtype=float)
```

Fit

`scipy.optimize.leastsq` minimizes the sum of squares of the function given as an argument. Basically, the function to minimize is the residuals (the difference between the data and the model):

```
>>> def residuals(coeffs, y, t):
...     return y - model(t, coeffs)
```

So let's get our solution by calling `scipy.optimize.leastsq()` with the following arguments:

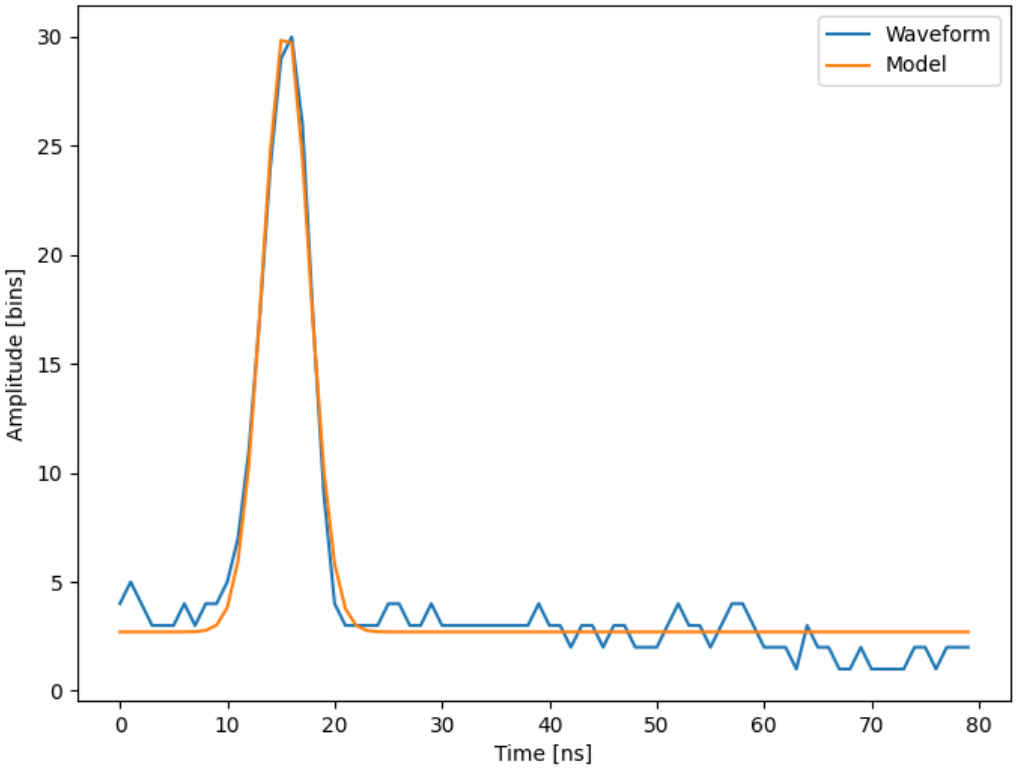
- the function to minimize

- an initial solution
- the additional arguments to pass to the function

```
>>> import scipy as sp
>>> t = np.arange(len(waveform_1))
>>> x, flag = sp.optimize.leastsq(residuals, x0, args=(waveform_1, t))
>>> x
array([ 2.70363, 27.82020, 15.47924,  3.05636])
```

And visualize the solution:

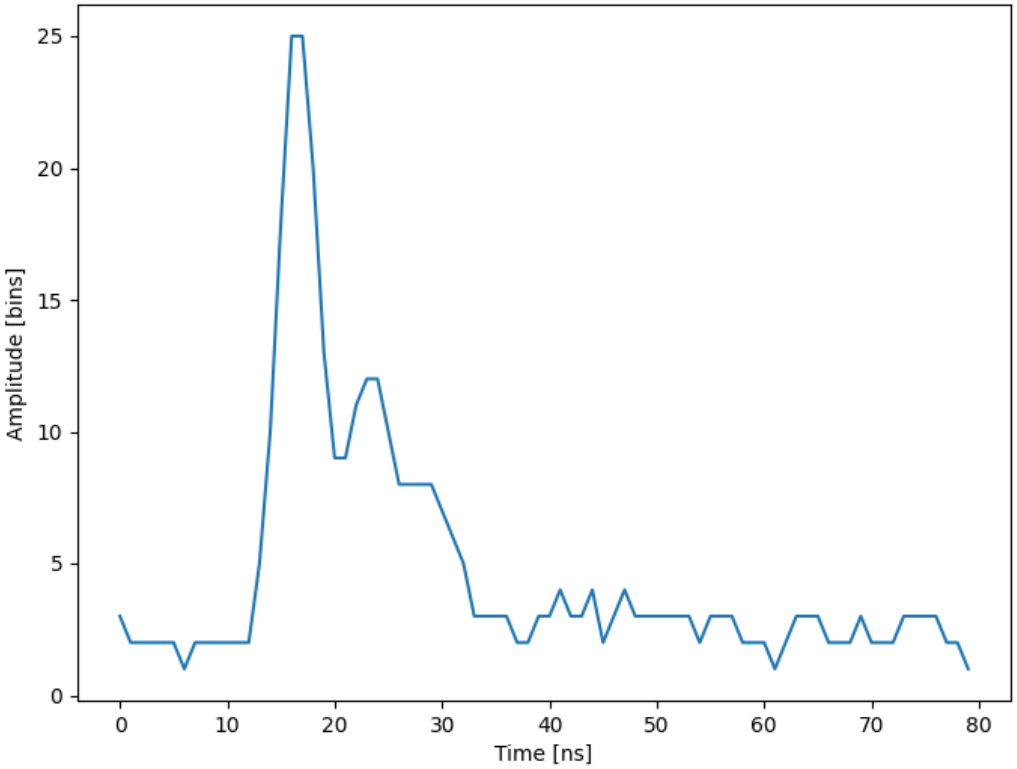
```
fig, ax = plt.subplots(figsize=(8, 6))
plt.plot(t, waveform_1, t, model(t, x))
plt.xlabel("Time [ns]")
plt.ylabel("Amplitude [bins]")
plt.legend(["Waveform", "Model"])
plt.show()
```



Remark: from scipy v0.8 and above, you should rather use `scipy.optimize.curve_fit()` which takes the model and the data as arguments, so you don't need to define the residuals any more.

Going further

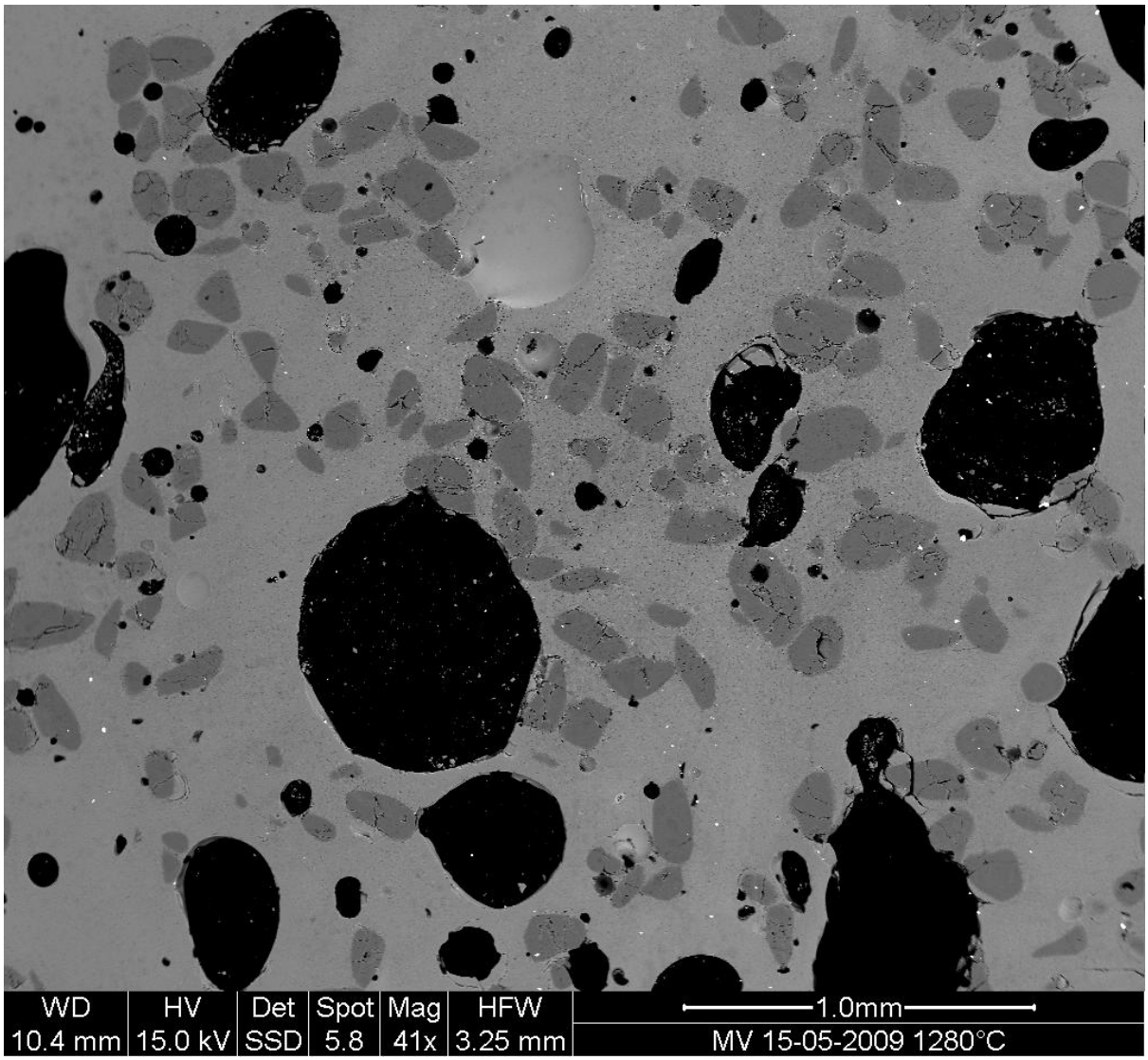
- Try with a more complex waveform (for instance `waveform_2.npy`) that contains three significant peaks. You must adapt the model which is now a sum of Gaussian functions instead of only one Gaussian peak.
- In some cases, writing an explicit function to compute the Jacobian is faster than letting `leastsq` estimate it numerically. Create a function to compute the Jacobian of the residuals and use it as



an input for `leastsq`.

- When we want to detect very small peaks in the signal, or when the initial guess is too far from a good solution, the result given by the algorithm is often not satisfying. Adding constraints to the parameters of the model enables to overcome such limitations. An example of *a priori* knowledge we can add is the sign of our variables (which are all positive).
- See the solution.
- Further exercise: compare the result of `scipy.optimize.leastsq()` and what you can get with `scipy.optimize.fmin_slsqp()` when adding boundary constraints.

5.11.3 Image processing application: counting bubbles and unmolten grains



Statement of the problem

1. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the docstring of `imshow` to display the image with the "right" orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).

   This Scanning Element Microscopy image shows a glass sample (light gray matrix) with some bubbles (on black) and unmolten sand grains (dark gray). We wish to determine the fraction of the sample covered by these three phases, and to estimate the typical size of sand grains and bubbles, their sizes, etc.

2. Crop the image to remove the lower panel with measure information.

3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.

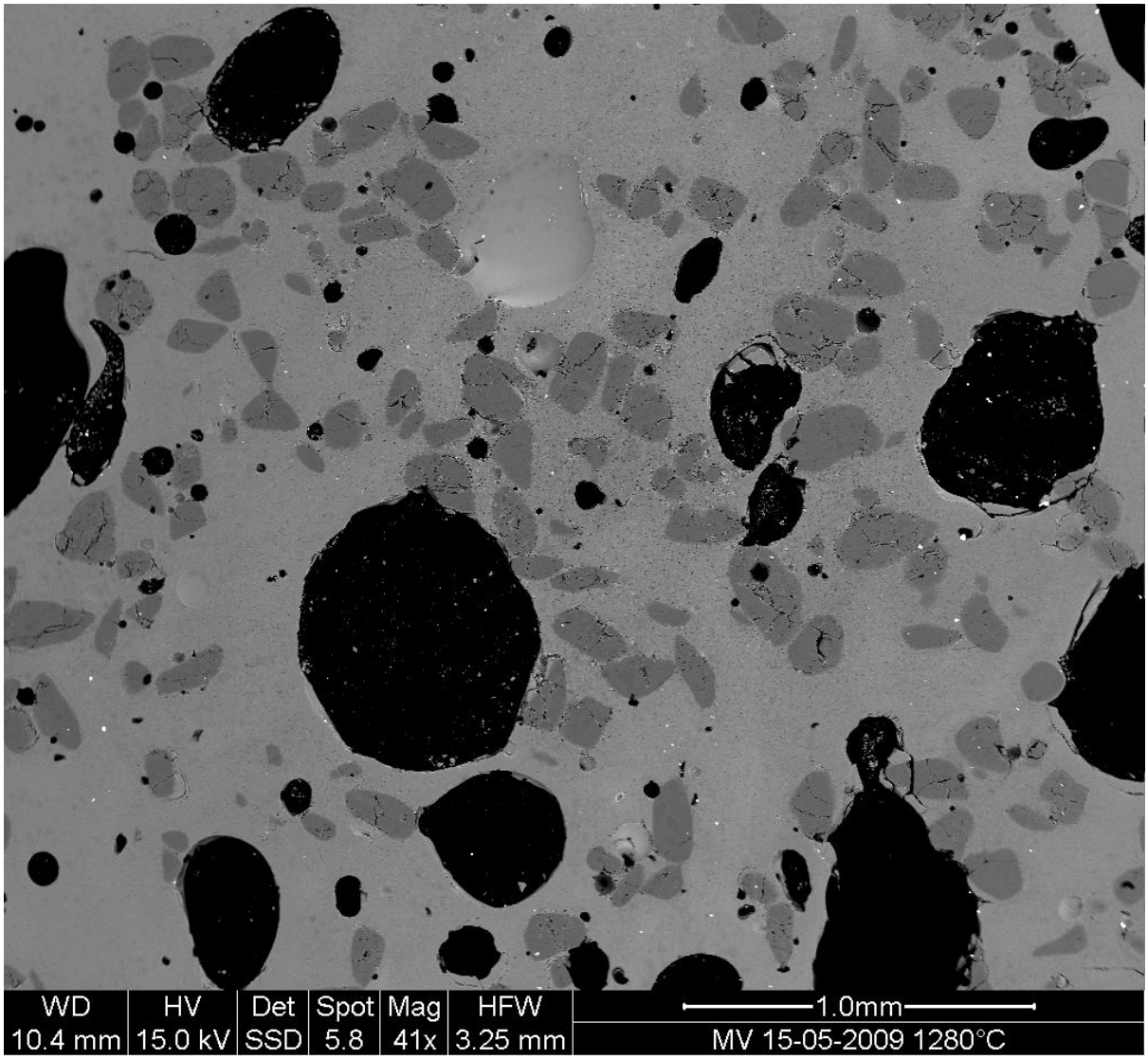
4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option (homework): write a function that determines automatically the thresholds from the minima of the histogram.

5. Display an image in which the three phases are colored with three different colors.

6. Use mathematical morphology to clean the different phases.

7. Attribute labels to all bubbles and sand grains, and remove from the sand mask grains that are smaller than 10 pixels. To do so, use `ndimage.sum` or `np.bincount` to compute the grain sizes.
8. Compute the mean size of bubbles.

5.11.4 Example of solution for the image processing exercise: unmolten grains in glass



1. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the docstring of `imshow` to display the image with the "right" orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).

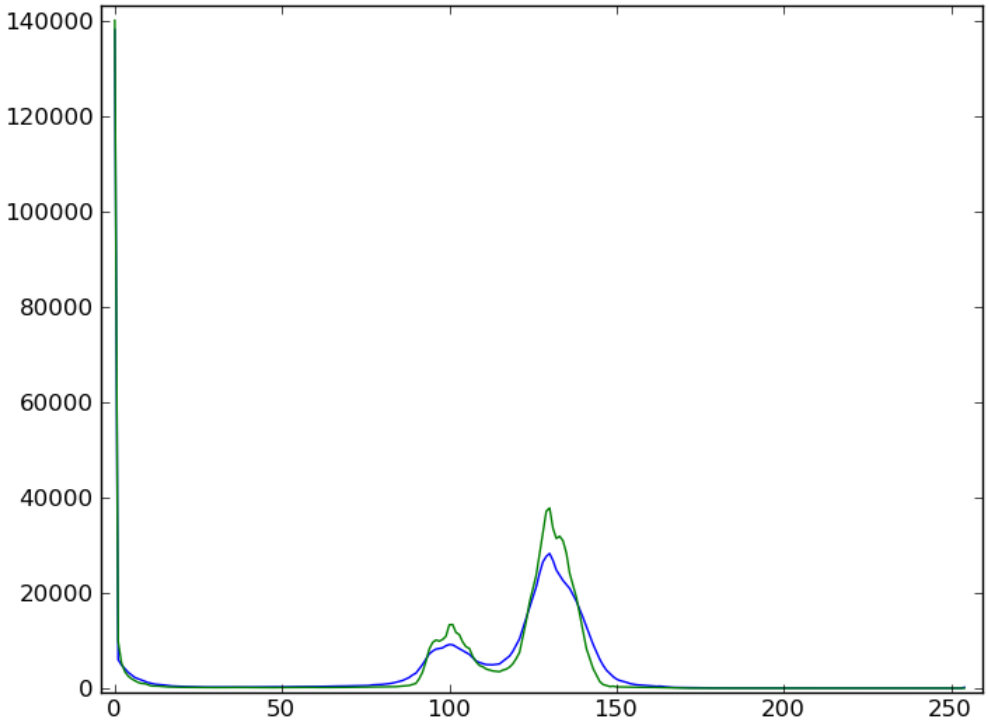
```
>>> dat = plt.imread('data/MV_HFV_012.jpg')
```

2. Crop the image to remove the lower panel with measure information.

```
>>> dat = dat[:-60]
```

3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.

```
>>> filtdat = sp.ndimage.median_filter(dat, size=(7,7))
>>> hi_dat = np.histogram(dat, bins=np.arange(256))
>>> hi_filtdat = np.histogram(filtdat, bins=np.arange(256))
```

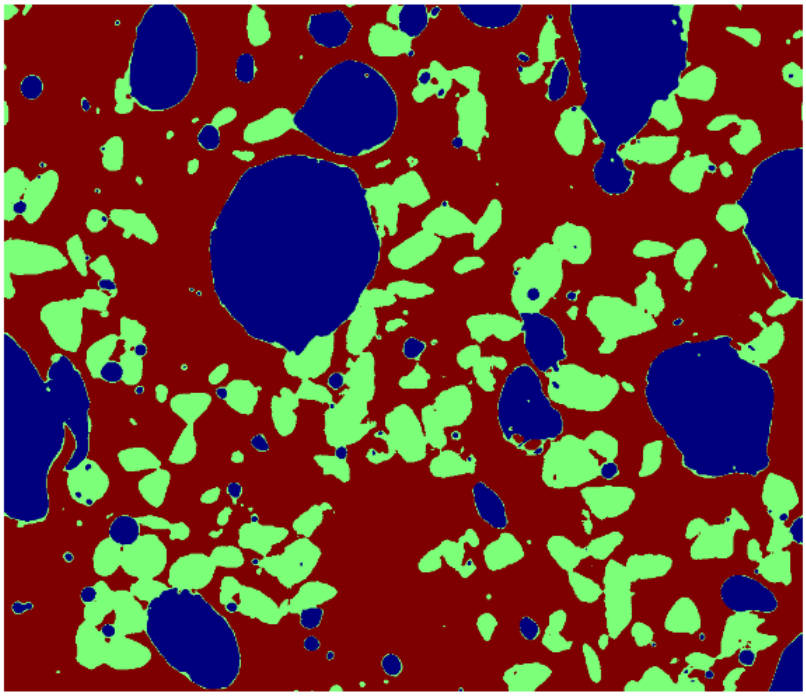


4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option (homework): write a function that determines automatically the thresholds from the minima of the histogram.

```
>>> void = filtdat <= 50
>>> sand = np.logical_and(filtdat > 50, filtdat <= 114)
>>> glass = filtdat > 114
```

5. Display an image in which the three phases are colored with three different colors.

```
>>> phases = void.astype(int) + 2*glass.astype(int) + 3*sand.astype(int)
```

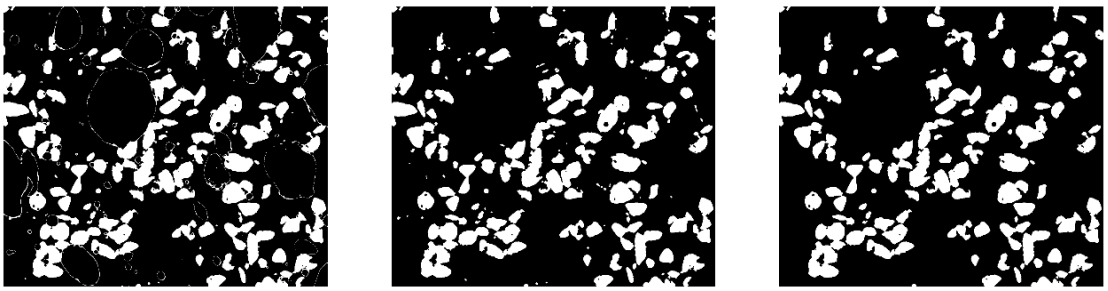


6. Use mathematical morphology to clean the different phases.

```
>>> sand_op = sp.ndimage.binary_opening(sand, iterations=2)
```

7. Attribute labels to all bubbles and sand grains, and remove from the sand mask grains that are smaller than 10 pixels. To do so, use `sp.ndimage.sum` or `np.bincount` to compute the grain sizes.

```
>>> sand_labels, sand_nb = sp.ndimage.label(sand_op)
>>> sand_areas = np.array(sp.ndimage.sum(sand_op, sand_labels, np.arange(sand_
↪labels.max()+1)))
>>> mask = sand_areas > 100
>>> remove_small_sand = mask[sand_labels.ravel()].reshape(sand_labels.shape)
```



8. Compute the mean size of bubbles.

```
>>> bubbles_labels, bubbles_nb = sp.ndimage.label(void)
>>> bubbles_areas = np.bincount(bubbles_labels.ravel())[1:]
>>> mean_bubble_size = bubbles_areas.mean()
>>> median_bubble_size = np.median(bubbles_areas)
```

(continues on next page)

(continued from previous page)

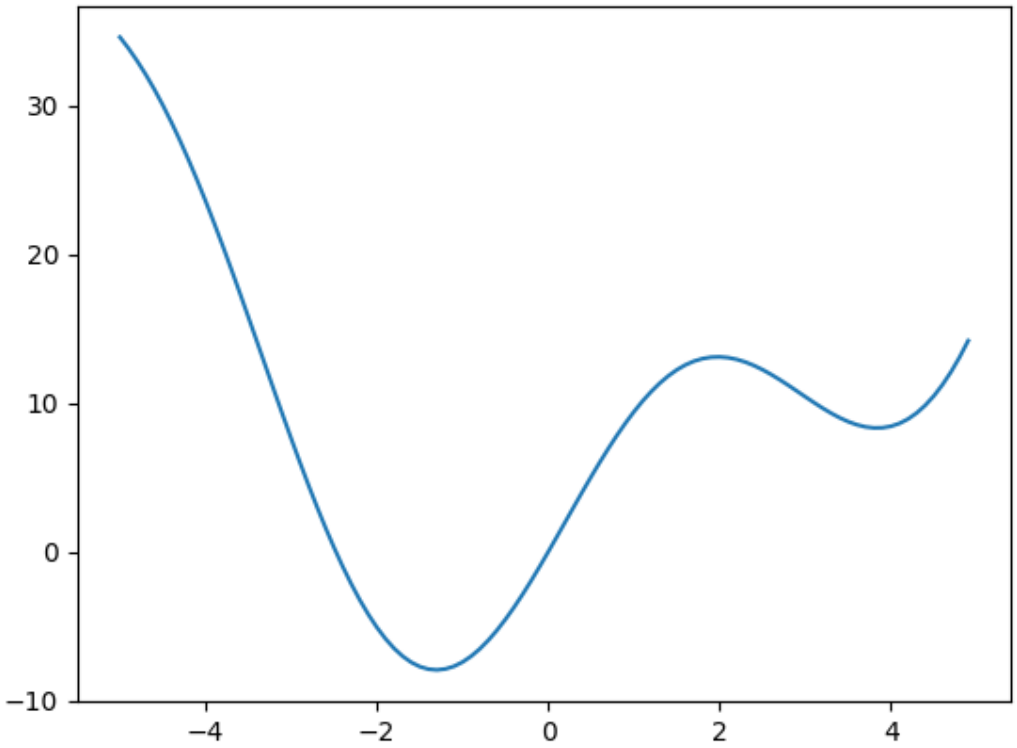
```
>>> mean_bubble_size, median_bubble_size
(np.float64(1699.875), np.float64(65.0))
```

5.12 Full code examples for the SciPy chapter

5.12.1 Finding the minimum of a smooth function

Demos various methods to find the minimum of a function.

```
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    return x**2 + 10 * np.sin(x)


x = np.arange(-5, 5, 0.1)
plt.plot(x, f(x))
```




```
[<matplotlib.lines.Line2D object at 0x7f3b36bc6900>]
```

Now find the minimum with a few methods

```
import scipy as sp

# The default (Nelder Mead)
print(sp.optimize.minimize(f, x0=0))
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -7.945823375615215
       x: [-1.306e+00]
     nit: 5
     jac: [-1.192e-06]
hess_inv: [[ 8.589e-02]]
    nfev: 12
    njev: 6
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.047 seconds)

5.12.2 Resample a signal with scipy.signal.resample

`scipy.signal.resample()` uses FFT to resample a 1D signal.

Generate a signal with 100 data point

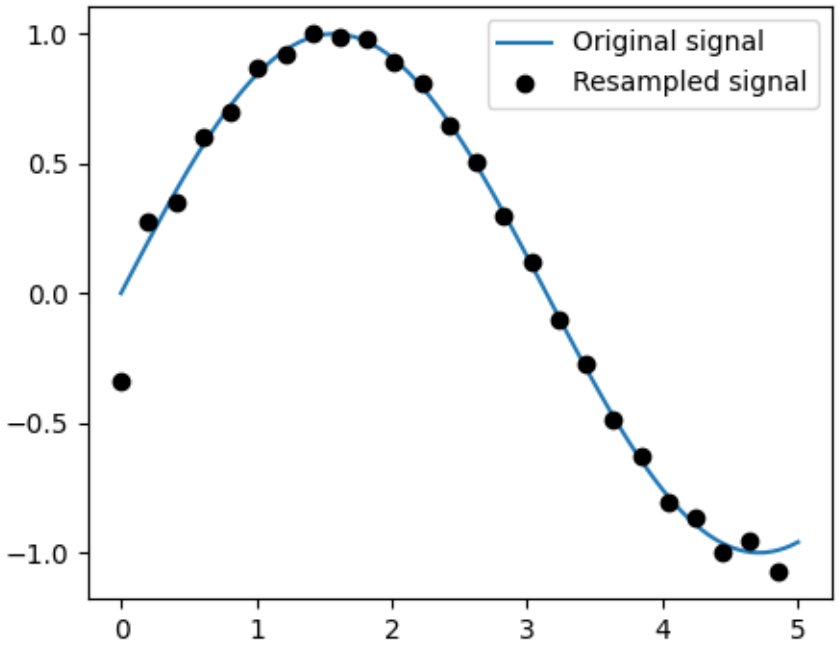
```
import numpy as np

t = np.linspace(0, 5, 100)
x = np.sin(t)
```

Downsample it by a factor of 4

```
import scipy as sp

x_resampled = sp.signal.resample(x, 25)
```

Plot

```
import matplotlib.pyplot as plt

plt.figure(figsize=(5, 4))
plt.plot(t, x, label="Original signal")
plt.plot(t[::4], x_resampled, "ko", label="Resampled signal")

plt.legend(loc="best")
plt.show()
```



Total running time of the script: (0 minutes 0.297 seconds)

5.12.3 Detrending a signal

`scipy.signal.detrend()` removes a linear trend.

Generate a random signal with a trend

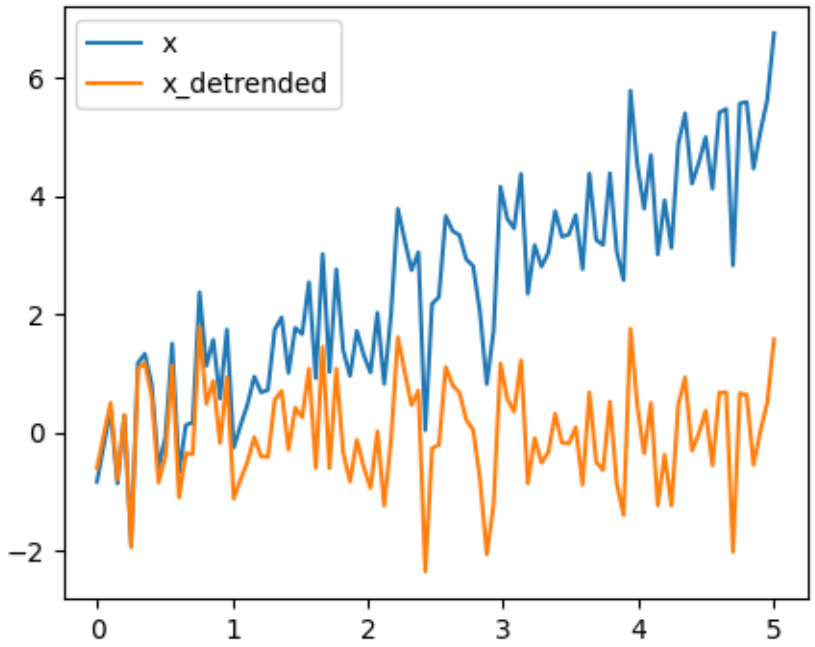
```
import numpy as np

t = np.linspace(0, 5, 100)
rng = np.random.default_rng()
x = t + rng.normal(size=100)
```

Detrend

```
import scipy as sp

x_detrended = sp.signal.detrend(x)
```

Plot

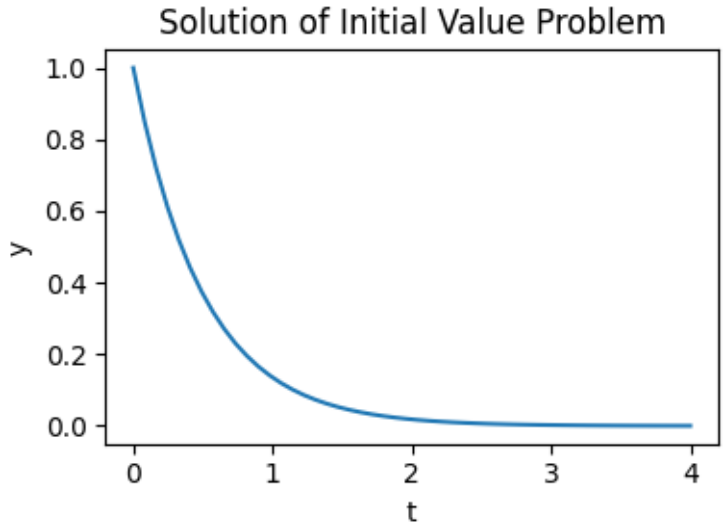
```
import matplotlib.pyplot as plt

plt.figure(figsize=(5, 4))
plt.plot(t, x, label="x")
plt.plot(t, x_detrended, label="x_detrended")
plt.legend(loc="best")
plt.show()
```



Total running time of the script: (0 minutes 0.051 seconds)

5.12.4 Integrating a simple ODE

Solve the ODE dy/dt = -2y between t = 0..4, with the initial condition y(t=0) = 1.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt


def f(t, y):
    return -2 * y
```

(continues on next page)

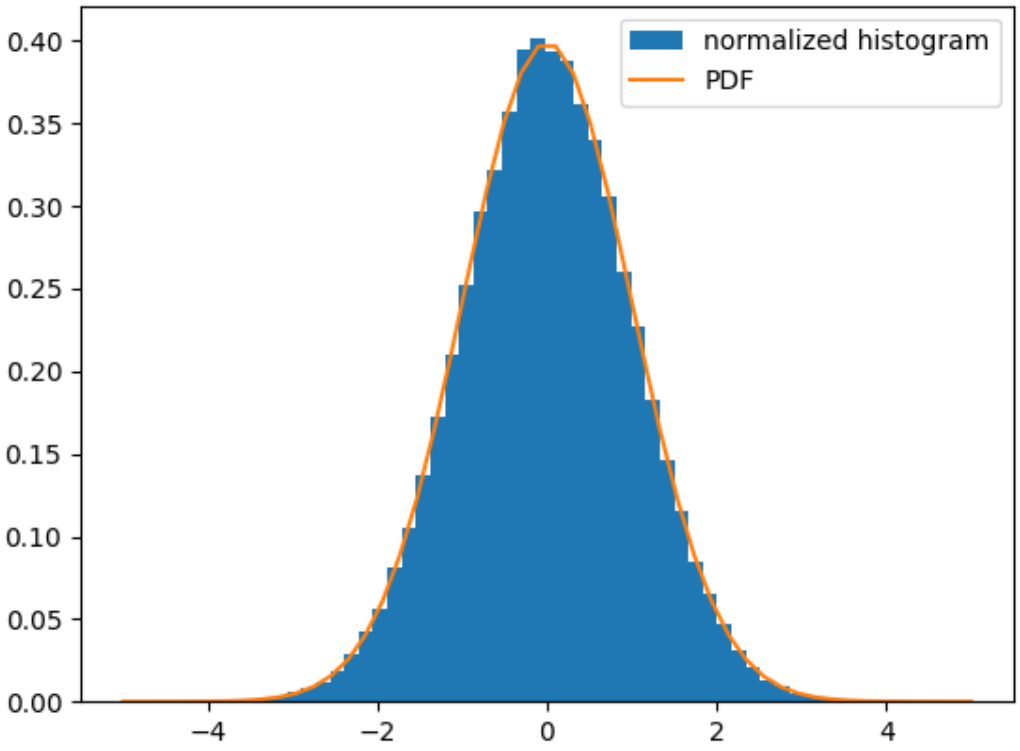
(continued from previous page)

```
t_span = (0, 4)  # time interval
t_eval = np.linspace(*t_span)  # times at which to evaluate `y`
y0 = [
    1,
]  # initial state
res = sp.integrate.solve_ivp(f, t_span=t_span, y0=y0, t_eval=t_eval)

plt.figure(figsize=(4, 3))
plt.plot(res.t, res.y[0])
plt.xlabel("t")
plt.ylabel("y")
plt.title("Solution of Initial Value Problem")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.075 seconds)

5.12.5 Normal distribution: histogram and PDF

Explore the normal distribution: a histogram built from samples and the PDF (probability density function).




```
import numpy as np
import scipy as sp
```

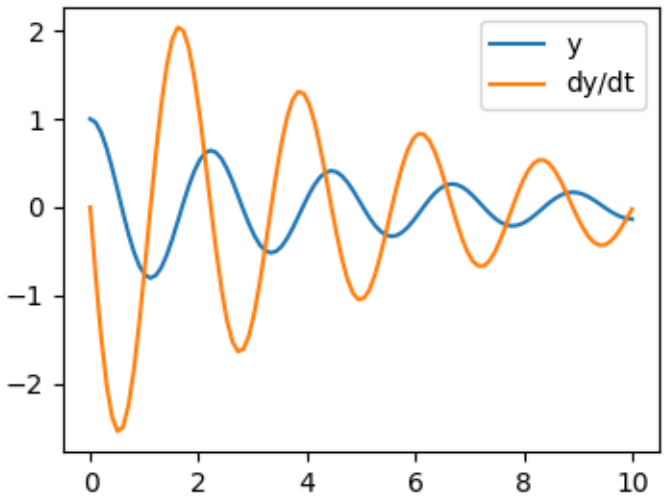
(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt

dist = sp.stats.norm(loc=0, scale=1)  # standard normal distribution
sample = dist.rvs(size=100000)  # "random variate sample"
plt.hist(
    sample,
    bins=51,  # group the observations into 50 bins
    density=True,  # normalize the frequencies
    label="normalized histogram",
)

x = np.linspace(-5, 5)  # possible values of the random variable
plt.plot(x, dist.pdf(x), label="PDF")
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.096 seconds)

5.12.6 Integrate the Damped spring-mass oscillator




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

m = 0.5  # kg
k = 4  # N/m
c = 0.4  # N s/m

zeta = c / (2 * m * np.sqrt(k / m))
omega = np.sqrt(k / m)


def f(t, z, zeta, omega):
    return (z[1], -zeta * omega * z[1] - omega**2 * z[0])
```

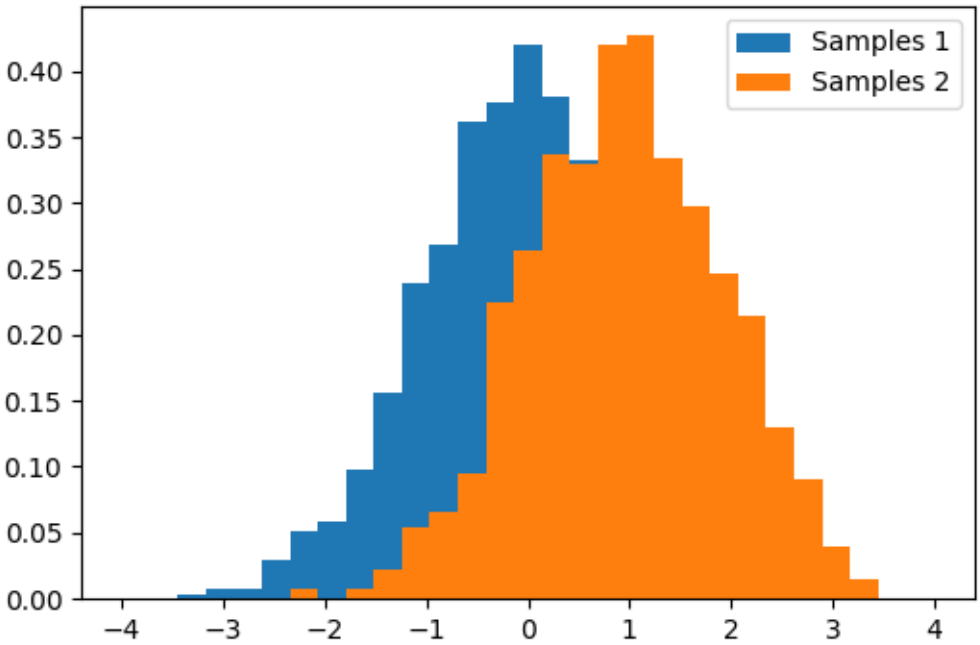
(continues on next page)

(continued from previous page)

```
t_span = (0, 10)
t_eval = np.linspace(*t_span, 100)
z0 = [1, 0]
res = sp.integrate.solve_ivp(
    f, t_span, z0, t_eval=t_eval, args=(zeta, omega), method="LSODA"
)

plt.figure(figsize=(4, 3))
plt.plot(res.t, res.y[0], label="y")
plt.plot(res.t, res.y[1], label="dy/dt")
plt.legend(loc="best")
plt.show()
```

Total running time of the script: (0 minutes 0.050 seconds)

5.12.7 Comparing 2 sets of samples from Gaussians




```
import numpy as np
import matplotlib.pyplot as plt

# Generates 2 sets of observations
rng = np.random.default_rng(27446968)
samples1 = rng.normal(0, size=1000)
samples2 = rng.normal(1, size=1000)

# Compute a histogram of the sample
bins = np.linspace(-4, 4, 30)
histogram1, bins = np.histogram(samples1, bins=bins, density=True)
```

(continues on next page)

(continued from previous page)

```
histogram2, bins = np.histogram(samples2, bins=bins, density=True)

plt.figure(figsize=(6, 4))
plt.hist(samples1, bins=bins, density=True, label="Samples 1")  # type: ignore[arg-
↪type]
plt.hist(samples2, bins=bins, density=True, label="Samples 2")  # type: ignore[arg-
↪type]
plt.legend(loc="best")
plt.show()
```

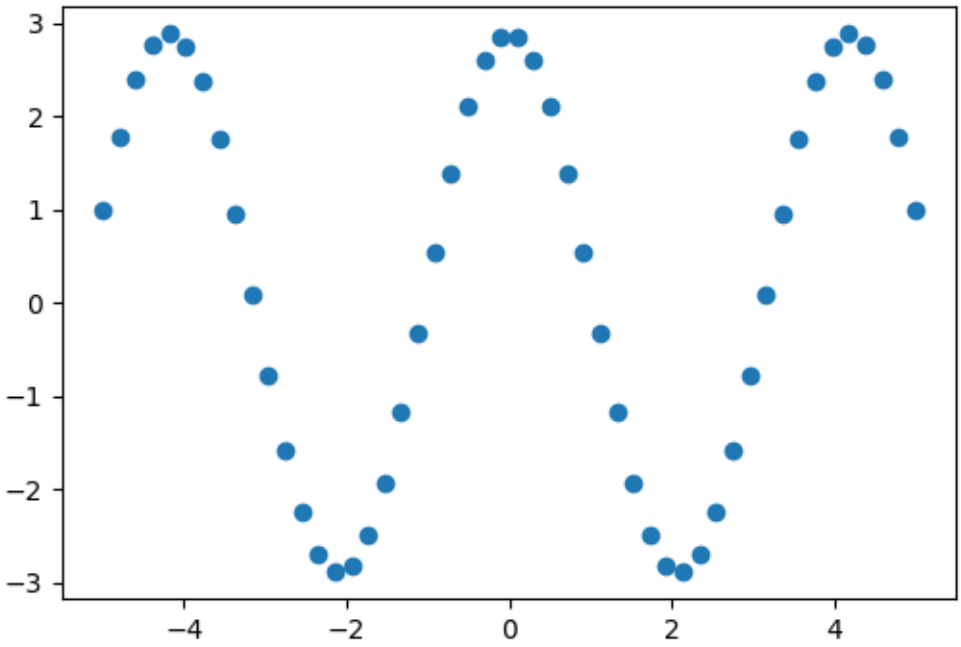
Total running time of the script: (0 minutes 0.097 seconds)

5.12.8 Curve fitting

Demos a simple curve fitting

First generate some data

```
import numpy as np

# Seed the random number generator for reproducibility
rng = np.random.default_rng(27446968)

x_data = np.linspace(-5, 5, num=50)
noise = 0.01 * np.cos(100 * x_data)
a, b = 2.9, 1.5
y_data = a * np.cos(b * x_data) + noise

# And plot it
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data)
```




```
<matplotlib.collections.PathCollection object at 0x7f3b2c2dea50>
```

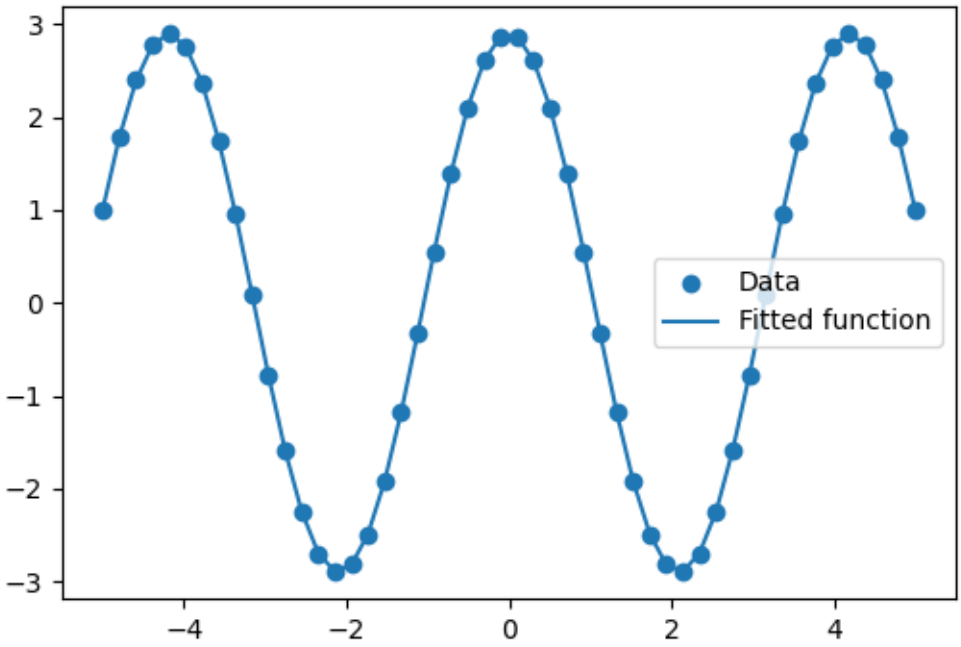
Now fit a simple sine function to the data

```
import scipy as sp


def test_func(x, a, b, c):
    return a * np.sin(b * x + c)


params, params_covariance = sp.optimize.curve_fit(
    test_func, x_data, y_data, p0=[2, 1, 3]
)

print(params)
```

```
[2.900026   1.50012043 1.57079633]
```

And plot the resulting curve on the data

```
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data, label="Data")
plt.plot(x_data, test_func(x_data, *params), label="Fitted function")

plt.legend(loc="best")

plt.show()
```



Total running time of the script: (0 minutes 0.105 seconds)

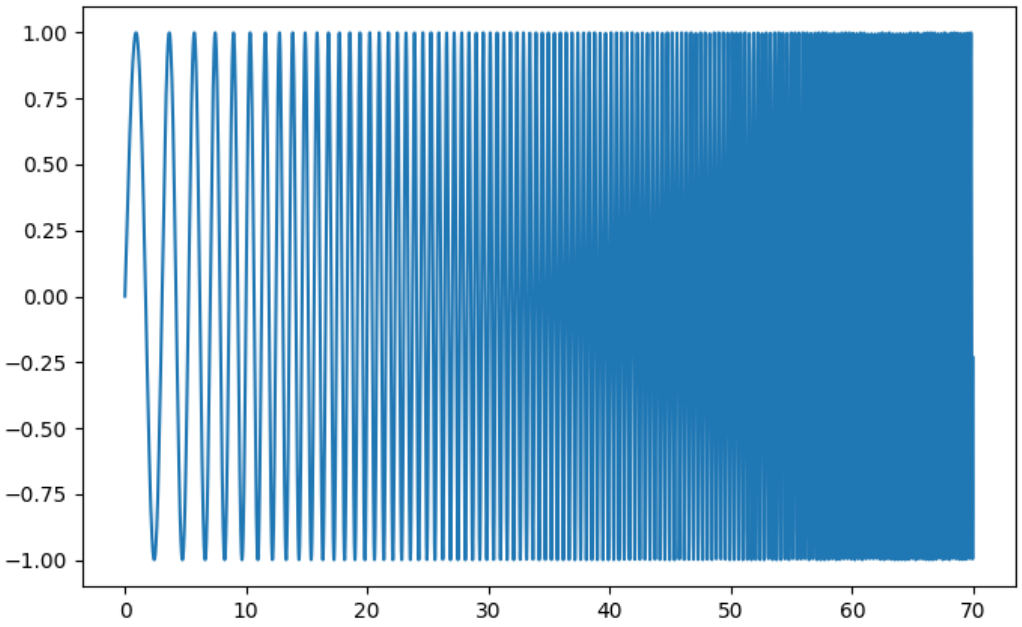
5.12.9 Spectrogram, power spectral density

Demo spectrogram and power spectral density on a frequency chirp.

```
import numpy as np
import matplotlib.pyplot as plt
```

Generate a chirp signal

```
# Seed the random number generator
np.random.seed(0)

time_step = 0.01
time_vec = np.arange(0, 70, time_step)

# A signal with a small frequency chirp
sig = np.sin(0.5 * np.pi * time_vec * (1 + 0.1 * time_vec))

plt.figure(figsize=(8, 5))
plt.plot(time_vec, sig)
```

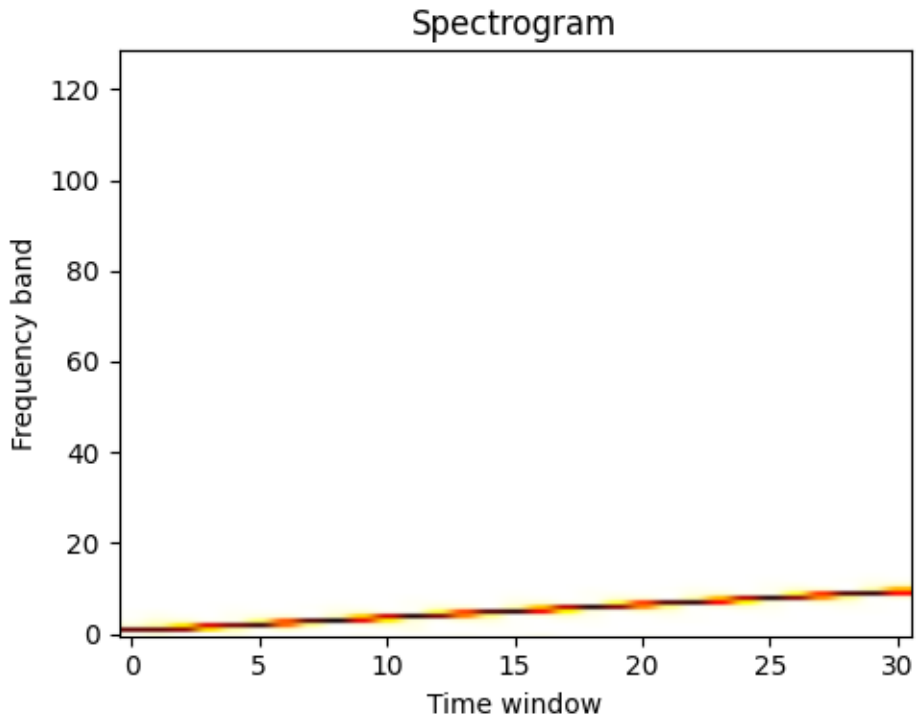



```
[<matplotlib.lines.Line2D object at 0x7f3b2bc2c050>]
```

Compute and plot the spectrogram

The spectrum of the signal on consecutive time windows

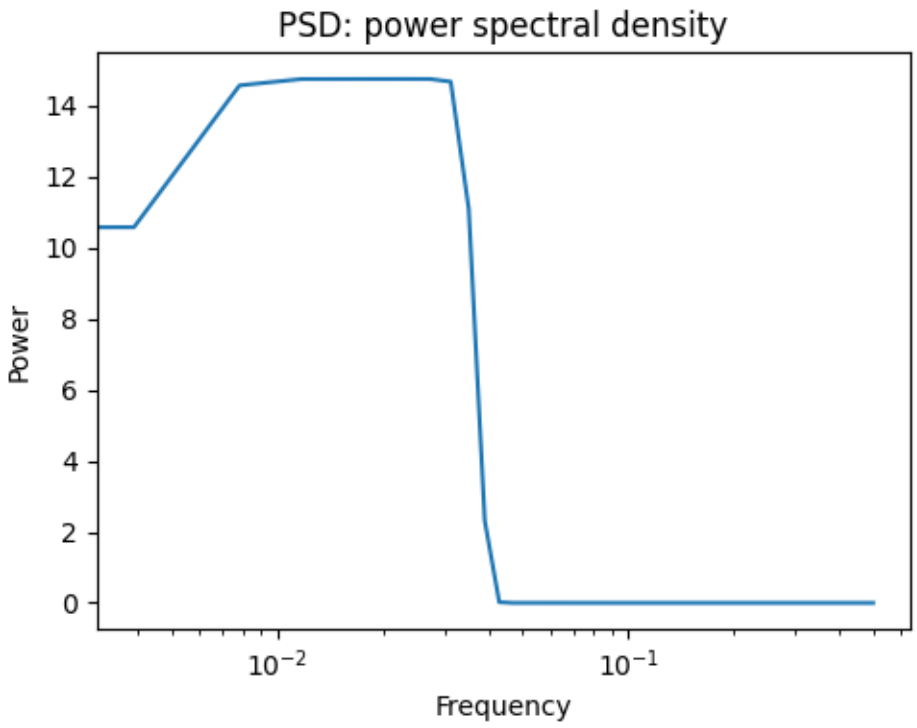
```
import scipy as sp

freqs, times, spectrogram = sp.signal.spectrogram(sig)

plt.figure(figsize=(5, 4))
plt.imshow(spectrogram, aspect="auto", cmap="hot_r", origin="lower")
plt.title("Spectrogram")
plt.ylabel("Frequency band")
plt.xlabel("Time window")
plt.tight_layout()
```



Compute and plot the power spectral density (PSD)

The power of the signal per frequency band

```
freqs, psd = sp.signal.welch(sig)

plt.figure(figsize=(5, 4))
plt.semilogx(freqs, psd)
plt.title("PSD: power spectral density")
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.tight_layout()
```

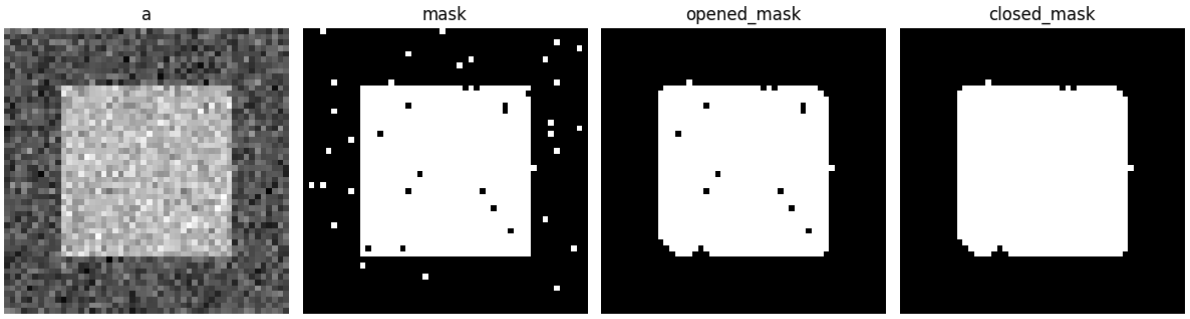



```
plt.show()
```

Total running time of the script: (0 minutes 0.327 seconds)

5.12.10 Demo mathematical morphology

A basic demo of binary opening and closing.




```
# Generate some binary data
import numpy as np

np.random.seed(0)
a = np.zeros((50, 50))
a[10:-10, 10:-10] = 1
a += 0.25 * np.random.standard_normal(a.shape)
mask = a >= 0.5

# Apply mathematical morphology
import scipy as sp
```

(continues on next page)

(continued from previous page)

```
opened_mask = sp.ndimage.binary_opening(mask)
closed_mask = sp.ndimage.binary_closing(opened_mask)

# Plot
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
plt.imshow(a, cmap="gray")
plt.axis("off")
plt.title("a")

plt.subplot(142)
plt.imshow(mask, cmap="gray")
plt.axis("off")
plt.title("mask")

plt.subplot(143)
plt.imshow(opened_mask, cmap="gray")
plt.axis("off")
plt.title("opened_mask")

plt.subplot(144)
plt.imshow(closed_mask, cmap="gray")
plt.title("closed_mask")
plt.axis("off")

plt.subplots_adjust(wspace=0.05, left=0.01, bottom=0.01, right=0.99, top=0.99)

plt.show()
```

Total running time of the script: (0 minutes 0.087 seconds)

5.12.11 Plot geometrical transformations on images

Demo geometrical transformations of images.

```
Downloading file 'face.dat' from 'https://raw.githubusercontent.com/scipy/dataset-
↪face/main/face.dat' to '/home/runner/.cache/scipy-data'.
```

```
# Load some data
import scipy as sp

face = sp.datasets.face(gray=True)

# Apply a variety of transformations
import matplotlib.pyplot as plt

shifted_face = sp.ndimage.shift(face, (50, 50))
shifted_face2 = sp.ndimage.shift(face, (50, 50), mode="nearest")
rotated_face = sp.ndimage.rotate(face, 30)
cropped_face = face[50:-50, 50:-50]
zoomed_face = sp.ndimage.zoom(face, 2)
zoomed_face.shape

plt.figure(figsize=(15, 3))
plt.subplot(151)
plt.imshow(shifted_face, cmap="gray")
plt.axis("off")

plt.subplot(152)
plt.imshow(shifted_face2, cmap="gray")
plt.axis("off")

plt.subplot(153)
plt.imshow(rotated_face, cmap="gray")
plt.axis("off")

plt.subplot(154)
plt.imshow(cropped_face, cmap="gray")
plt.axis("off")

plt.subplot(155)
plt.imshow(zoomed_face, cmap="gray")
plt.axis("off")

plt.subplots_adjust(wspace=0.05, left=0.01, bottom=0.01, right=0.99, top=0.99)

plt.show()
```

Total running time of the script: (0 minutes 1.524 seconds)

5.12.12 Demo connected components

Extracting and labeling connected components in a 2D array

```
import numpy as np
import matplotlib.pyplot as plt
```

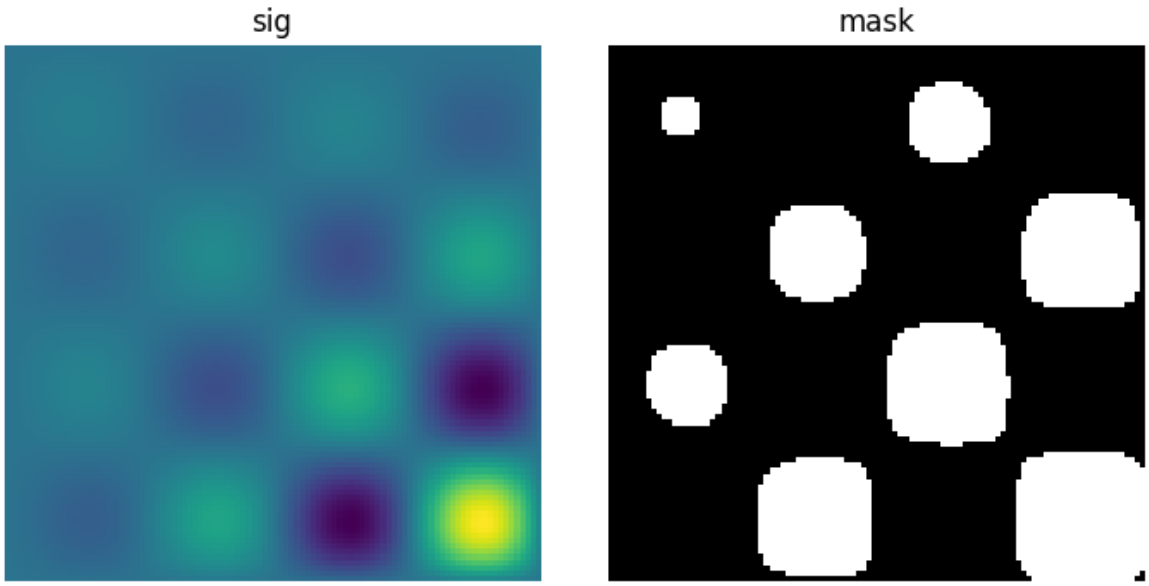
Generate some binary data

```
x, y = np.indices((100, 100))
sig = (
    np.sin(2 * np.pi * x / 50.0)
    * np.sin(2 * np.pi * y / 50.0)
    * (1 + x * y / 50.0**2) ** 2
```

(continues on next page)

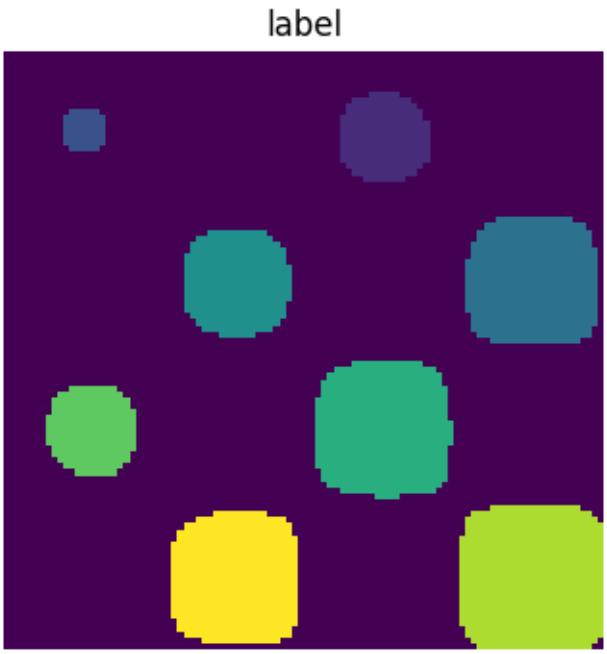
(continued from previous page)

```
)
mask = sig > 1

plt.figure(figsize=(7, 3.5))
plt.subplot(1, 2, 1)
plt.imshow(sig)
plt.axis("off")
plt.title("sig")

plt.subplot(1, 2, 2)
plt.imshow(mask, cmap="gray")
plt.axis("off")
plt.title("mask")
plt.subplots_adjust(wspace=0.05, left=0.01, bottom=0.01, right=0.99, top=0.9)
```



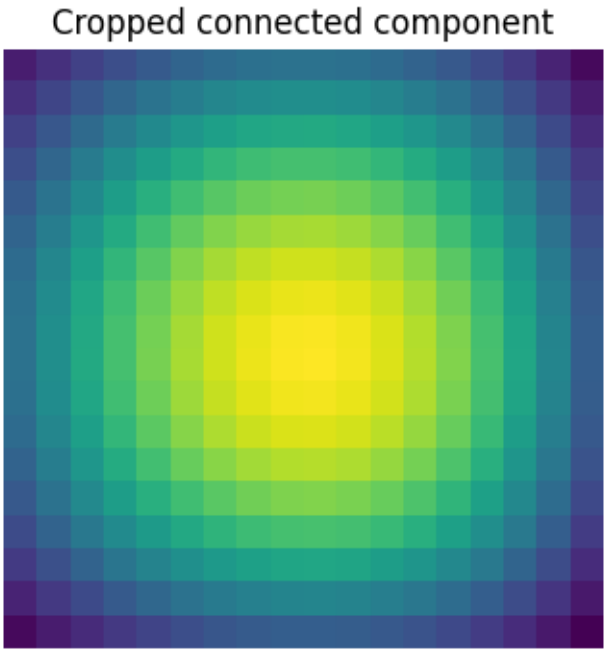
Label connected components

```
import scipy as sp

labels, nb = sp.ndimage.label(mask)

plt.figure(figsize=(3.5, 3.5))
plt.imshow(labels)
plt.title("label")
plt.axis("off")

plt.subplots_adjust(wspace=0.05, left=0.01, bottom=0.01, right=0.99, top=0.9)
```



Extract the 4th connected component, and crop the array around it

```
sl = sp.ndimage.find_objects(labels == 4)
plt.figure(figsize=(3.5, 3.5))
plt.imshow(sig[sl[0]])
plt.title("Cropped connected component")
plt.axis("off")

plt.subplots_adjust(wspace=0.05, left=0.01, bottom=0.01, right=0.99, top=0.9)

plt.show()
```



Total running time of the script: (0 minutes 0.110 seconds)

5.12.13 Minima and roots of a function

Demos finding minima and roots of a function.

Define the function

```
import numpy as np

x = np.arange(-10, 10, 0.1)


def f(x):
    return x**2 + 10 * np.sin(x)
```

Find minima

```
import scipy as sp

# Global optimization
grid = (-10, 10, 0.1)
xmin_global = sp.optimize.brute(f, (grid,))
print(f"Global minima found {xmin_global}")

# Constrain optimization
xmin_local = sp.optimize.fminbound(f, 0, 10)
print(f"Local minimum found {xmin_local}")
```

```
Global minima found [-1.30641113]
Local minimum found 3.8374671194983834
```

Root finding

```
root = sp.optimize.root(f, 1)  # our initial guess is 1
print(f"First root found {root.x}")
root2 = sp.optimize.root(f, -2.5)
print(f"Second root found {root2.x}")
```

```
First root found [0.]
Second root found [-2.47948183]
```

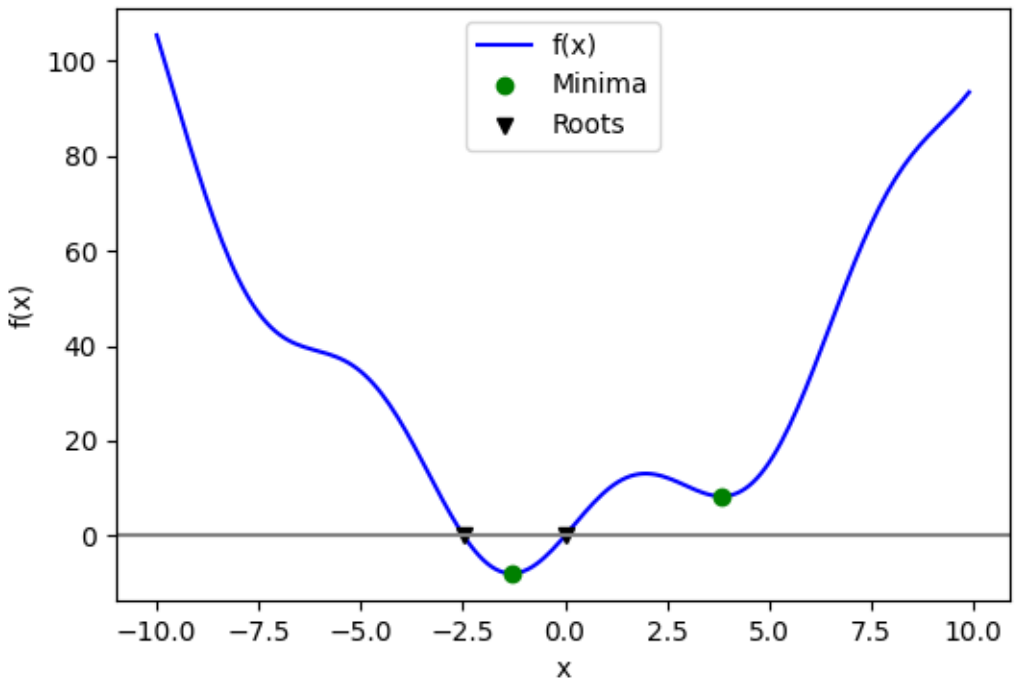
Plot function, minima, and roots

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot(111)

# Plot the function
ax.plot(x, f(x), "b-", label="f(x)")

# Plot the minima
xmins = np.array([xmin_global[0], xmin_local])
ax.plot(xmins, f(xmins), "go", label="Minima")

# Plot the roots
roots = np.array([root.x, root2.x])
```

(continues on next page)

(continued from previous page)

```
ax.plot(roots, f(roots), "kv", label="Roots")

# Decorate the figure
ax.legend(loc="best")
ax.set_xlabel("x")
ax.set_ylabel("f(x)")
ax.axhline(0, color="gray")
plt.show()
```



Total running time of the script: (0 minutes 0.066 seconds)

5.12.14 Plot filtering on images

Demo filtering for denoising of images.

```
# Load some data
import scipy as sp
```

(continues on next page)

(continued from previous page)

```
face = sp.datasets.face(gray=True)
face = face[:512, -512:]  # crop out square on right

# Apply a variety of filters
import matplotlib.pyplot as plt

import numpy as np

noisy_face = np.copy(face).astype(float)
rng = np.random.default_rng()
noisy_face += face.std() * 0.5 * rng.standard_normal(face.shape)
blurred_face = sp.ndimage.gaussian_filter(noisy_face, sigma=3)
median_face = sp.ndimage.median_filter(noisy_face, size=5)
wiener_face = sp.signal.wiener(noisy_face, (5, 5))

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
plt.imshow(noisy_face, cmap="gray")
plt.axis("off")
plt.title("noisy")

plt.subplot(142)
plt.imshow(blurred_face, cmap="gray")
plt.axis("off")
plt.title("Gaussian filter")

plt.subplot(143)
plt.imshow(median_face, cmap="gray")
plt.axis("off")
plt.title("median filter")

plt.subplot(144)
plt.imshow(wiener_face, cmap="gray")
plt.title("Wiener filter")
plt.axis("off")

plt.subplots_adjust(wspace=0.05, left=0.01, bottom=0.01, right=0.99, top=0.99)

plt.show()
```

Total running time of the script: (0 minutes 0.503 seconds)

5.12.15 Optimization of a two-parameter function

```
import numpy as np


# Define the function that we are interested in
def sixhump(x):
    return (
        (4 - 2.1 * x[0] ** 2 + x[0] ** 4 / 3) * x[0] ** 2
        + x[0] * x[1]
        + (-4 + 4 * x[1] ** 2) * x[1] ** 2
    )
```

(continues on next page)

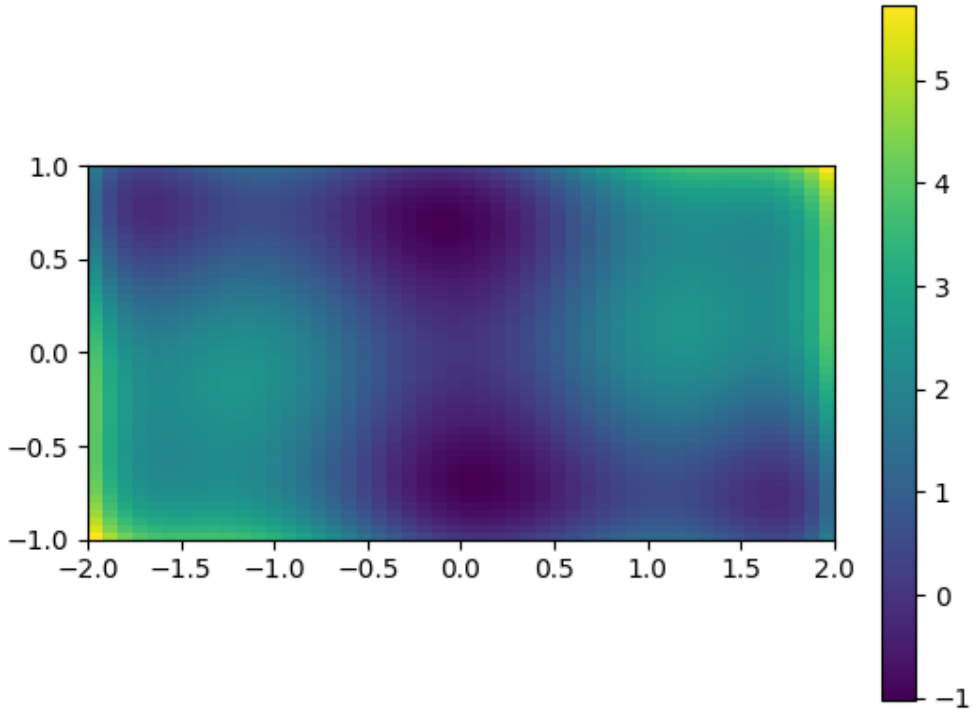
(continued from previous page)

```
# Make a grid to evaluate the function (for plotting)
xlim = [-2, 2]
ylim = [-1, 1]
x = np.linspace(*xlim)  # type: ignore[call-overload]
y = np.linspace(*ylim)  # type: ignore[call-overload]
xg, yg = np.meshgrid(x, y)
```

A 2D image plot of the function

Simple visualization in 2D

```
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(sixhump([xg, yg]), extent=xlim + ylim, origin="lower")  # type: ignore[arg-
↪type]
plt.colorbar()
```




```
<matplotlib.colorbar.Colorbar object at 0x7f3b2b823530>
```

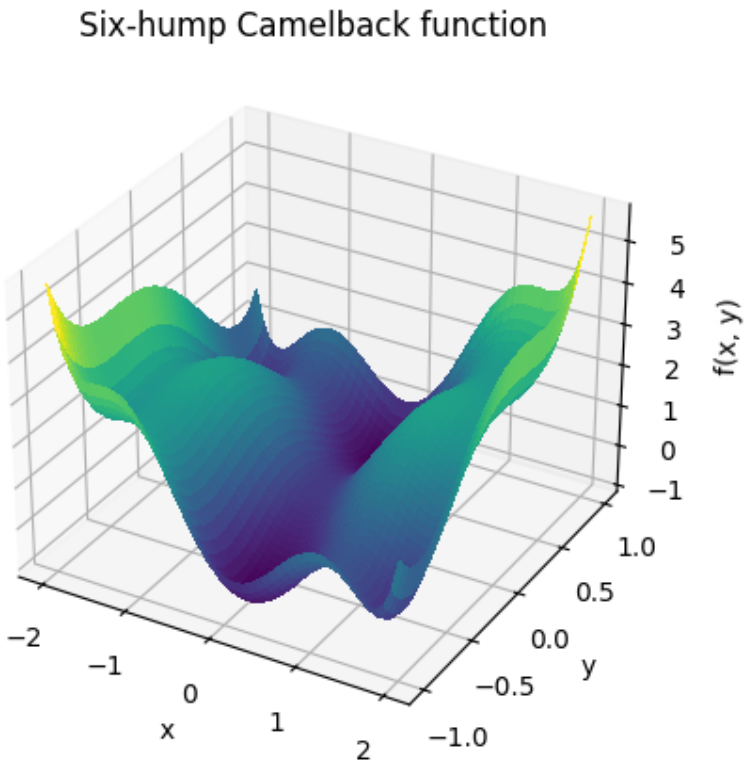
A 3D surface plot of the function

```
from mpl_toolkits.mplot3d import Axes3D
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure()
ax: Axes3D = fig.add_subplot(111, projection="3d")
surf = ax.plot_surface(
    xg,
    yg,
    sixhump([xg, yg]),
    rstride=1,
    cstride=1,
    cmap="viridis",
    linewidth=0,
    antialiased=False,
)

ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x, y)")
ax.set_title("Six-hump Camelback function")
```




```
Text(0.5, 1.0, 'Six-hump Camelback function')
```

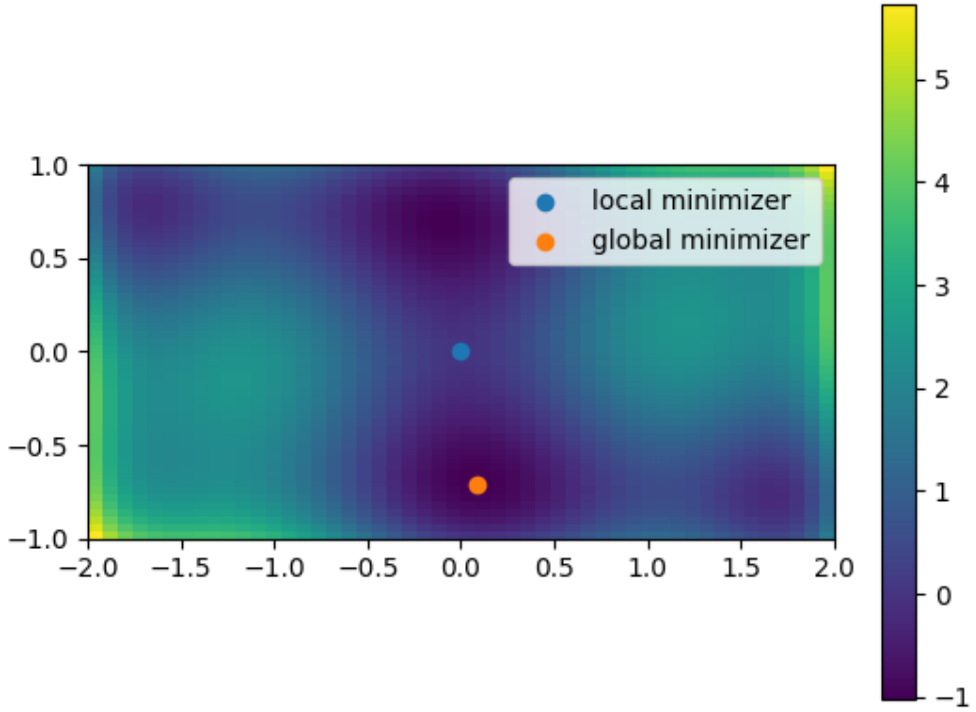
Find minima

```
import scipy as sp

# local minimization
res_local = sp.optimize.minimize(sixhump, x0=[0, 0])
```

(continues on next page)

(continued from previous page)

```
# global minimization
res_global = sp.optimize.differential_evolution(sixhump, bounds=[xlim, ylim])

plt.figure()
# Show the function in 2D
plt.imshow(sixhump([xg, yg]), extent=xlim + ylim, origin="lower")  # type: ignore[arg-
↪type]
plt.colorbar()
# Mark the minima
plt.scatter(res_local.x[0], res_local.x[1], label="local minimizer")
plt.scatter(res_global.x[0], res_global.x[1], label="global minimizer")
plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 0.398 seconds)

5.12.16 Plotting and manipulating FFTs for filtering

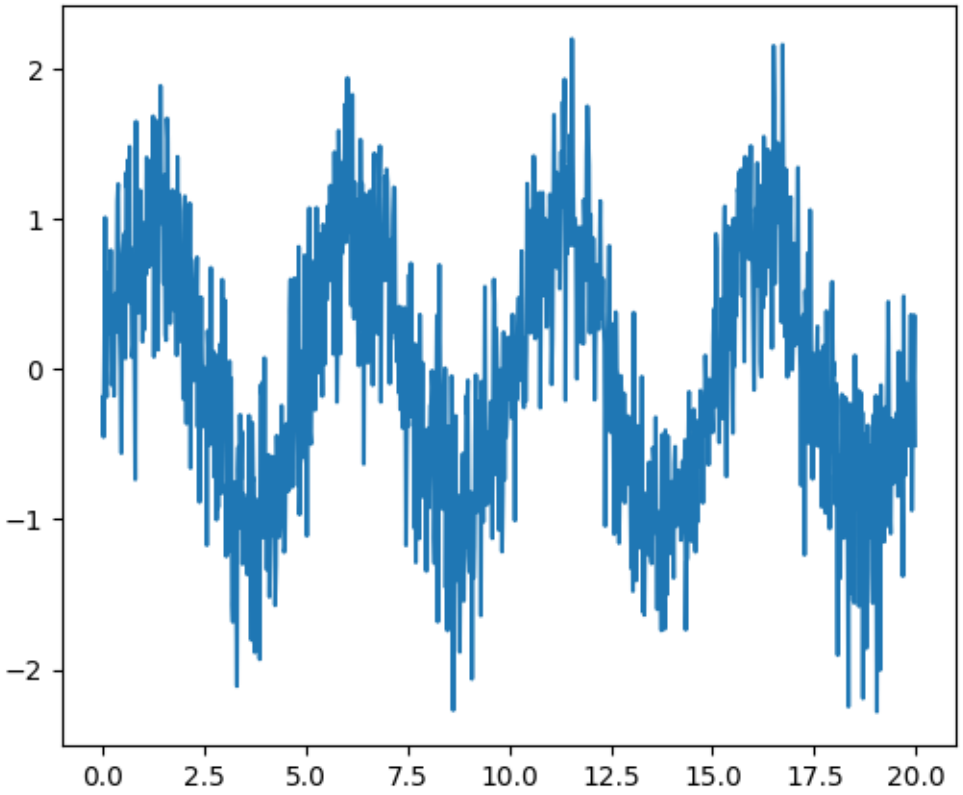
Plot the power of the FFT of a signal and inverse FFT back to reconstruct a signal.

This example demonstrate `scipy.fft.fft()`, `scipy.fft.fftfreq()` and `scipy.fft.ifft()`. It implements a basic filter that is very suboptimal, and should not be used.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

Generate the signal

```
# Seed the random number generator
rng = np.random.default_rng(27446968)

time_step = 0.02
period = 5.0

time_vec = np.arange(0, 20, time_step)
sig = np.sin(2 * np.pi / period * time_vec) + 0.5 * rng.normal(size=time_vec.size)

plt.figure(figsize=(6, 5))
plt.plot(time_vec, sig, label="Original signal")
```




```
[<matplotlib.lines.Line2D object at 0x7f3b2b9b9fa0>]
```

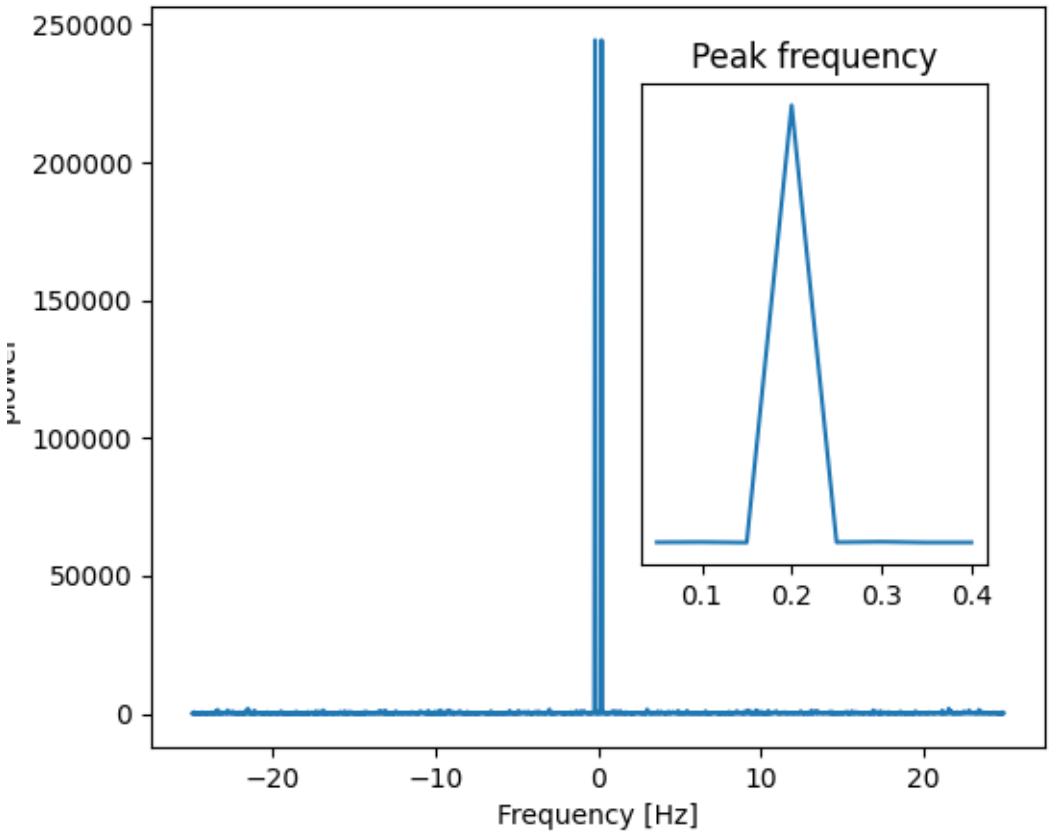
Compute and plot the power

```
# The FFT of the signal
sig_fft = sp.fft.fft(sig)

# And the power (sig_fft is of complex dtype)
power = np.abs(sig_fft) ** 2

# The corresponding frequencies
sample_freq = sp.fft.fftfreq(sig.size, d=time_step)
```

(continues on next page)

(continued from previous page)

```
# Plot the FFT power
plt.figure(figsize=(6, 5))
plt.plot(sample_freq, power)
plt.xlabel("Frequency [Hz]")
plt.ylabel("plower")

# Find the peak frequency: we can focus on only the positive frequencies
pos_mask = np.where(sample_freq > 0)
freqs = sample_freq[pos_mask]
peak_freq = freqs[power[pos_mask].argmax()]

# Check that it does indeed correspond to the frequency that we generate
# the signal with
np.allclose(peak_freq, 1.0 / period)

# An inner plot to show the peak frequency
axes = plt.axes((0.55, 0.3, 0.3, 0.5))
plt.title("Peak frequency")
plt.plot(freqs[:8], power[pos_mask][:8])
plt.setp(axes, yticks=[])

# scipy.signal.find_peaks_cwt can also be used for more advanced
# peak detection
```

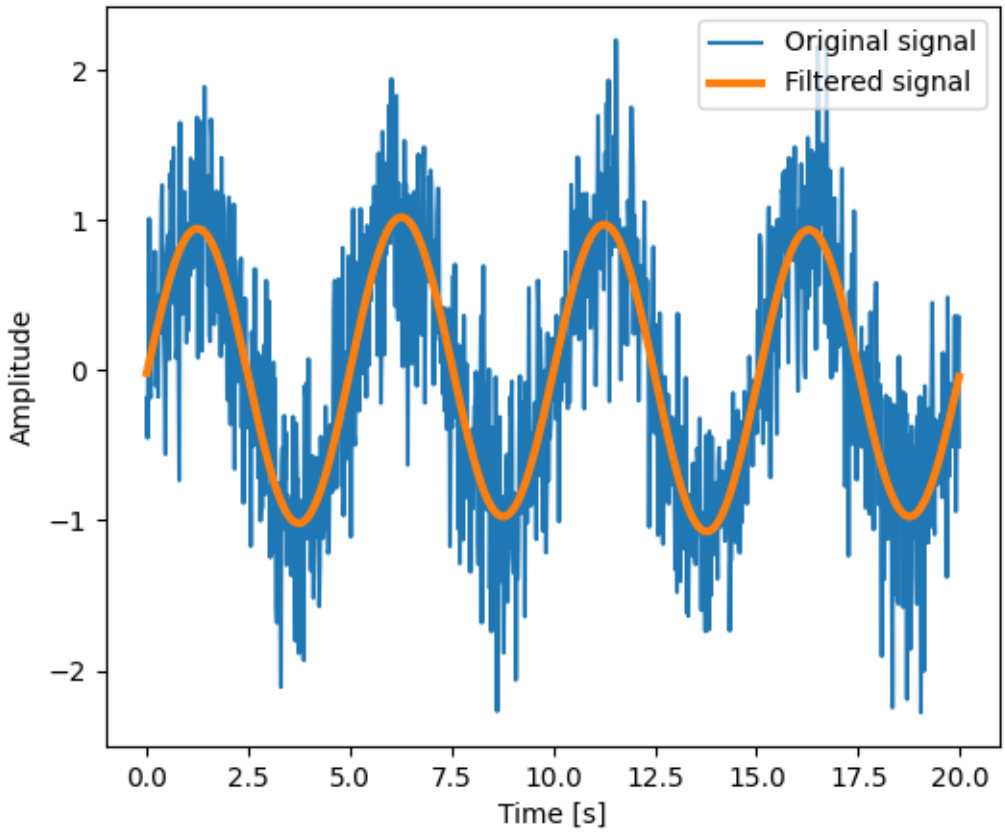


```
[]
```

Remove all the high frequencies

We now remove all the high frequencies and transform back from frequencies to signal.

```
high_freq_fft = sig_fft.copy()
high_freq_fft[np.abs(sample_freq) > peak_freq] = 0
filtered_sig = sp.fft.ifft(high_freq_fft)

plt.figure(figsize=(6, 5))
plt.plot(time_vec, sig, label="Original signal")
plt.plot(time_vec, filtered_sig, linewidth=3, label="Filtered signal")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")

plt.legend(loc="best")
```




```
/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages/matplotlib/cbook.
↪py:1719: ComplexWarning: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages/matplotlib/cbook.
↪py:1355: ComplexWarning: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)

<matplotlib.legend.Legend object at 0x7f3b2bc5bce0>
```

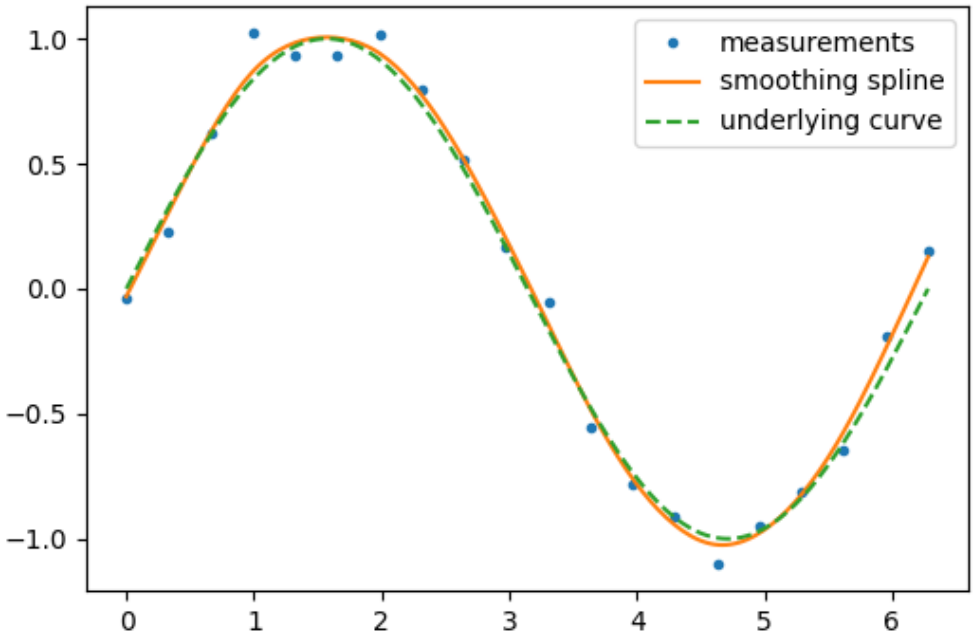
Note This is actually a bad way of creating a filter: such brutal cut-off in frequency space does not control distortion on the signal.

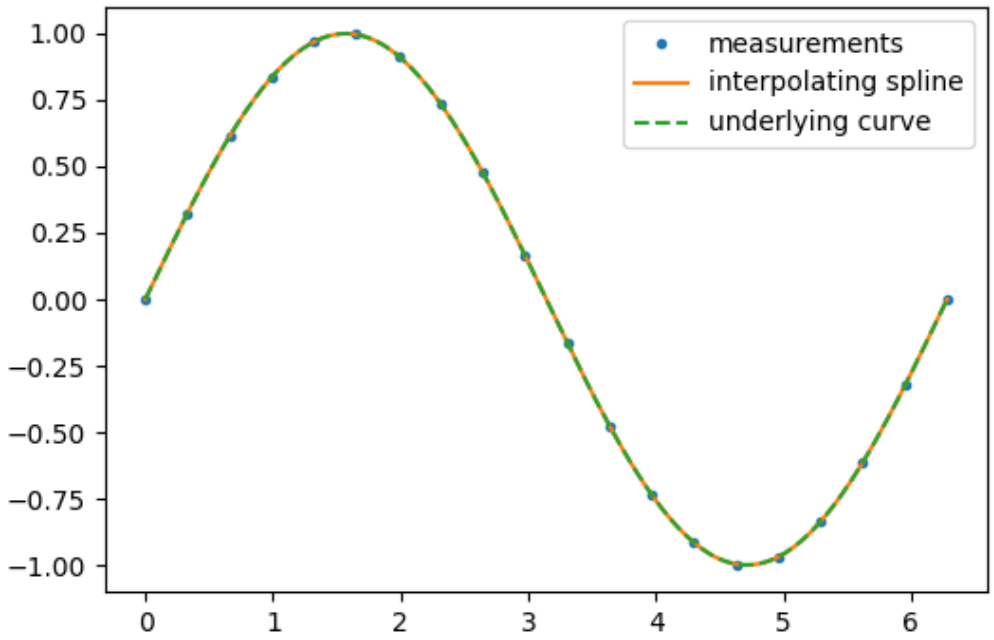
Filters should be created using the SciPy filter design code

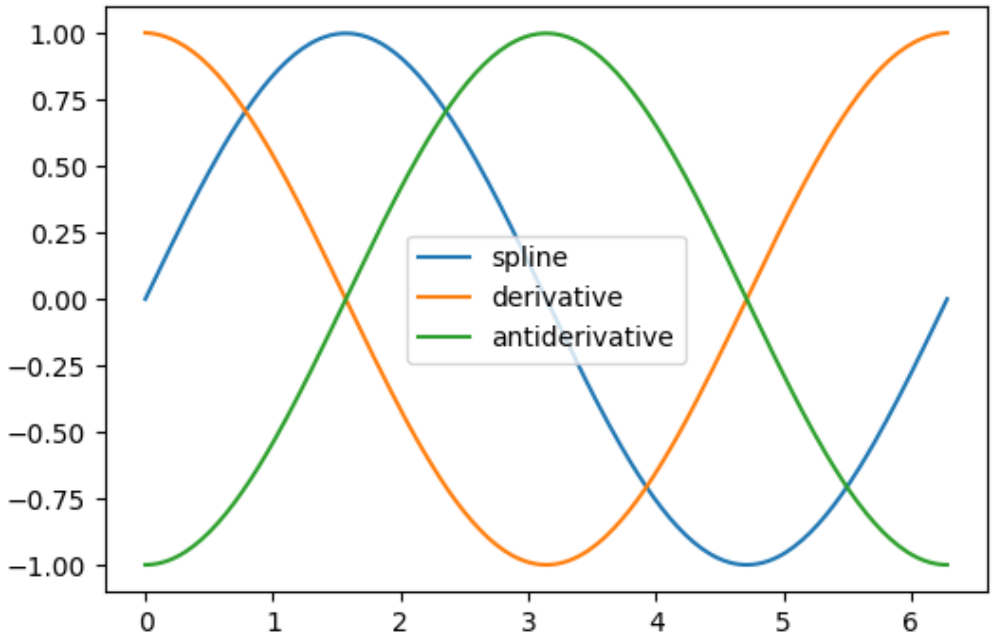
```
plt.show()
```

Total running time of the script: (0 minutes 0.192 seconds)

5.12.17 A demo of 1D interpolation





- 

-

```
# Generate data
import numpy as np

rng = np.random.default_rng(27446968)
measured_time = np.linspace(0, 2 * np.pi, 20)
function = np.sin(measured_time)
noise = rng.normal(loc=0, scale=0.1, size=20)
measurements = function + noise
```

(continues on next page)

(continued from previous page)

```
# Smooth the curve and interpolate at new times
import scipy as sp

smoothing_spline = sp.interpolate.make_smoothing_spline(measured_time, measurements)
interpolation_time = np.linspace(0, 2 * np.pi, 200)
smooth_results = smoothing_spline(interpolation_time)

# Plot the data, the interpolant, and the original function
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.plot(measured_time, measurements, ".", ms=6, label="measurements")
plt.plot(interpolation_time, smooth_results, label="smoothing spline")
plt.plot(interpolation_time, np.sin(interpolation_time), "--", label="underlying curve
↪")
plt.legend()
plt.show()

# Fit the data exactly
interp_spline = sp.interpolate.make_interp_spline(measured_time, function)
interp_results = interp_spline(interpolation_time)

# Plot the data, the interpolant, and the original function
plt.figure(figsize=(6, 4))
plt.plot(measured_time, function, ".", ms=6, label="measurements")
plt.plot(interpolation_time, interp_results, label="interpolating spline")
plt.plot(interpolation_time, np.sin(interpolation_time), "--", label="underlying curve
↪")
plt.legend()
plt.show()

# Plot interpolant, its derivative, and its antiderivative
plt.figure(figsize=(6, 4))
t = interpolation_time
plt.plot(t, interp_spline(t), label="spline")
plt.plot(t, interp_spline.derivative()(t), label="derivative")
plt.plot(t, interp_spline.antiderivative()(t) - 1, label="antiderivative")

plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.189 seconds)

5.12.18 Solutions of the exercises for SciPy

Solutions of the exercises for SciPy

Crude periodicity finding

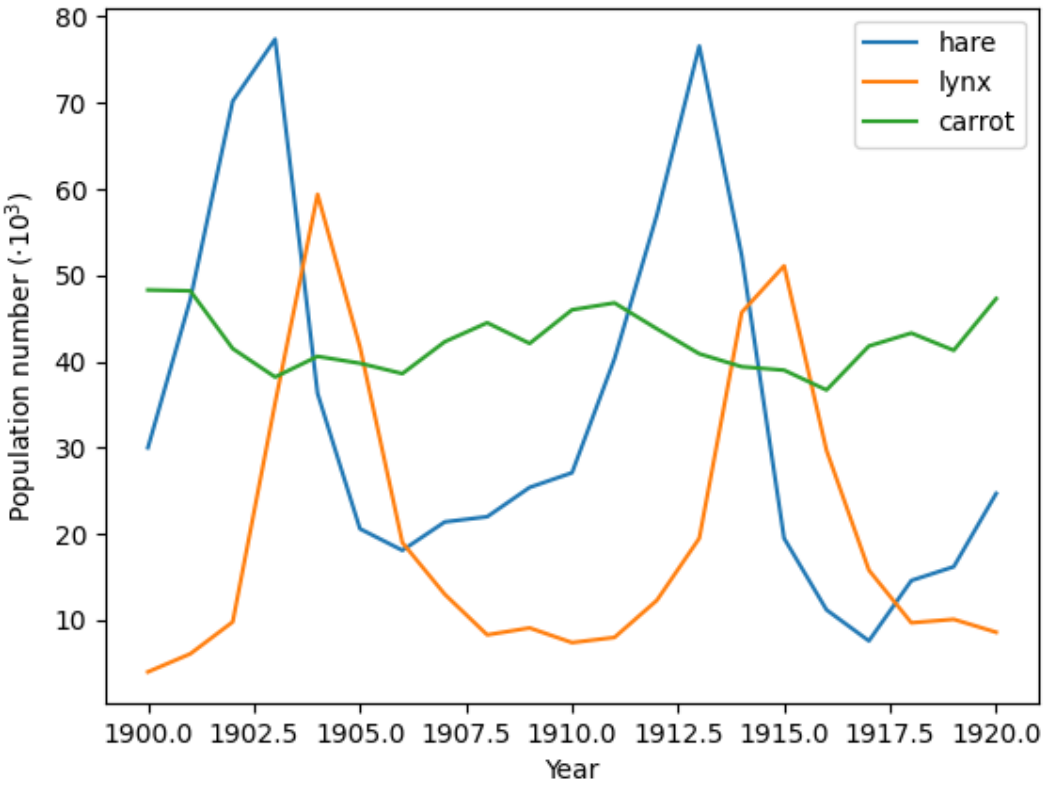
Discover the periods in evolution of animal populations (`../../../../data/populations.txt`)

Load the data

```
import numpy as np

data = np.loadtxt("../../../../data/populations.txt")
years = data[:, 0]
populations = data[:, 1:]
```

Plot the data

```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(years, populations * 1e-3)
plt.xlabel("Year")
plt.ylabel(r"Population number ($\cdot10^3$)")
plt.legend(["hare", "lynx", "carrot"], loc=1)
```




```
<matplotlib.legend.Legend object at 0x7f3b2b8c5370>
```

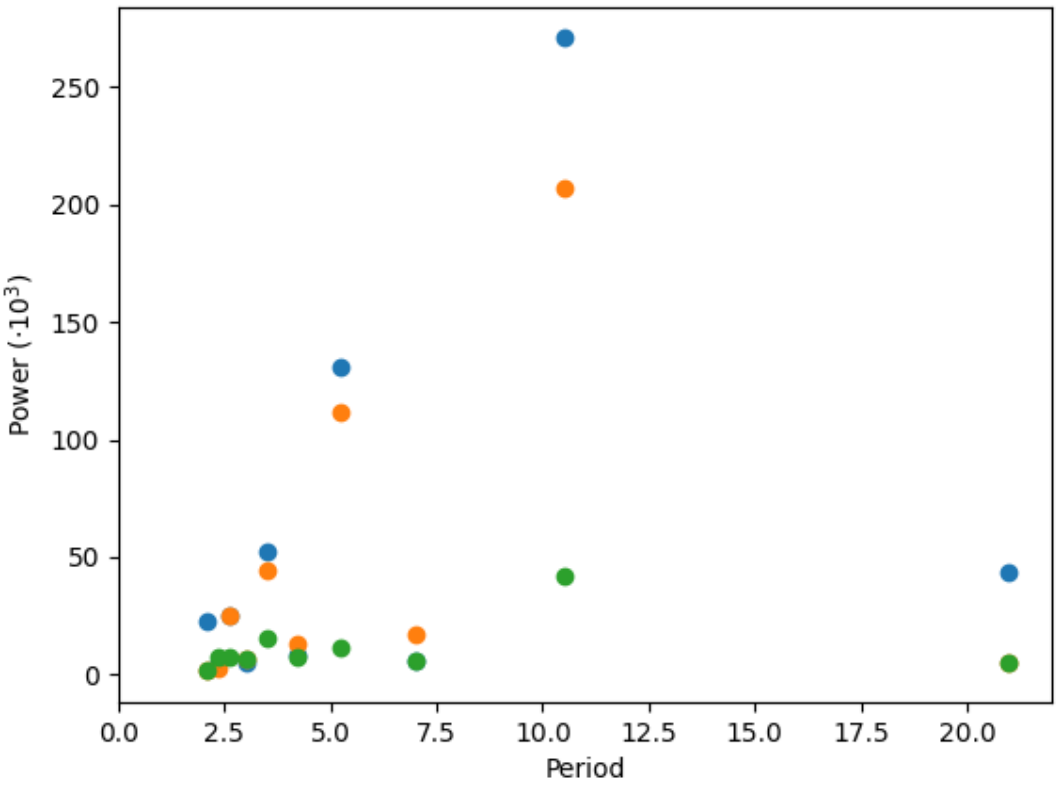
Plot its periods

```
import scipy as sp

ft_populations = sp.fft.fft(populations, axis=0)
frequencies = sp.fft.fftfreq(populations.shape[0], years[1] - years[0])
periods = 1 / frequencies
```

(continues on next page)

(continued from previous page)

```
plt.figure()
plt.plot(periods, abs(ft_populations) * 1e-3, "o")
plt.xlim(0, 22)
plt.xlabel("Period")
plt.ylabel(r"Power ($\cdot10^3$)")

plt.show()
```




```
/home/runner/work/scientific-python-lectures/scientific-python-lectures/intro/scipy/
↪examples/solutions/plot_periodicity_finder.py:39: RuntimeWarning: divide by zero␣
↪encountered in divide
  periods = 1 / frequencies
```

There's probably a period of around 10 years (obvious from the plot), but for this crude a method, there's not enough data to say much more.

Total running time of the script: (0 minutes 0.129 seconds)

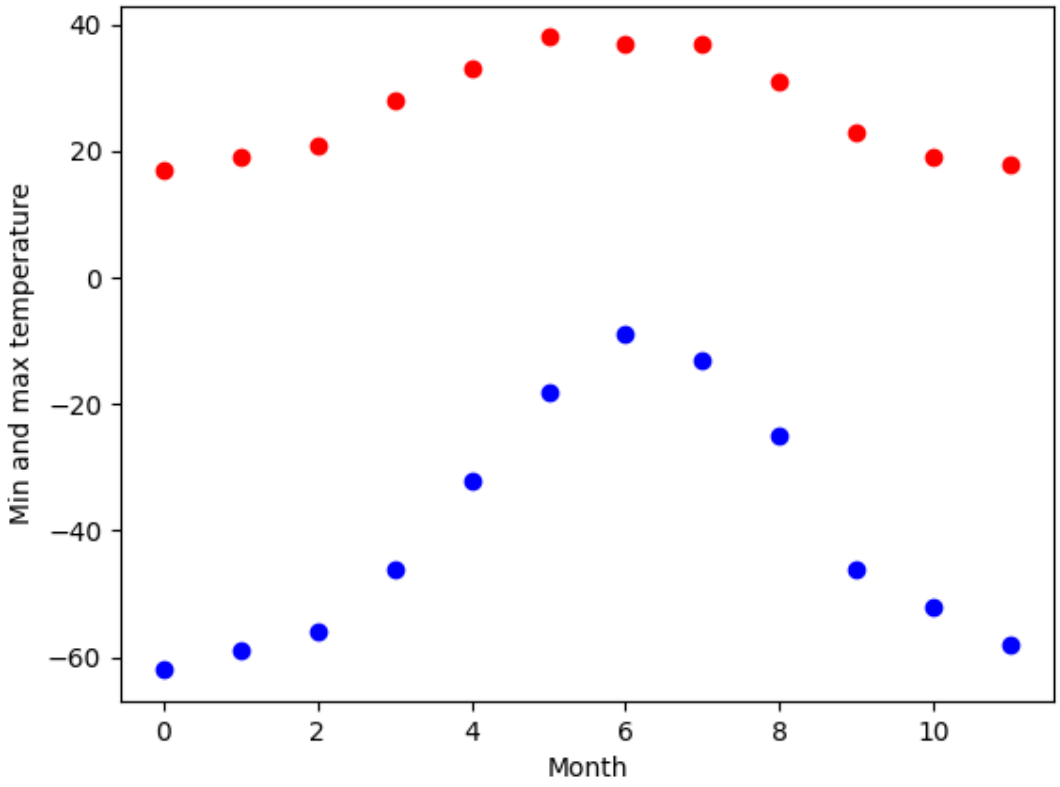
Curve fitting: temperature as a function of month of the year

We have the min and max temperatures in Alaska for each months of the year. We would like to find a function to describe this yearly evolution.

For this, we will fit a periodic function.

The data

```
import numpy as np

temp_max = np.array([17, 19, 21, 28, 33, 38, 37, 37, 31, 23, 19, 18])
temp_min = np.array([-62, -59, -56, -46, -32, -18, -9, -13, -25, -46, -52, -58])

import matplotlib.pyplot as plt

months = np.arange(12)
plt.plot(months, temp_max, "ro")
plt.plot(months, temp_min, "bo")
plt.xlabel("Month")
plt.ylabel("Min and max temperature")
```



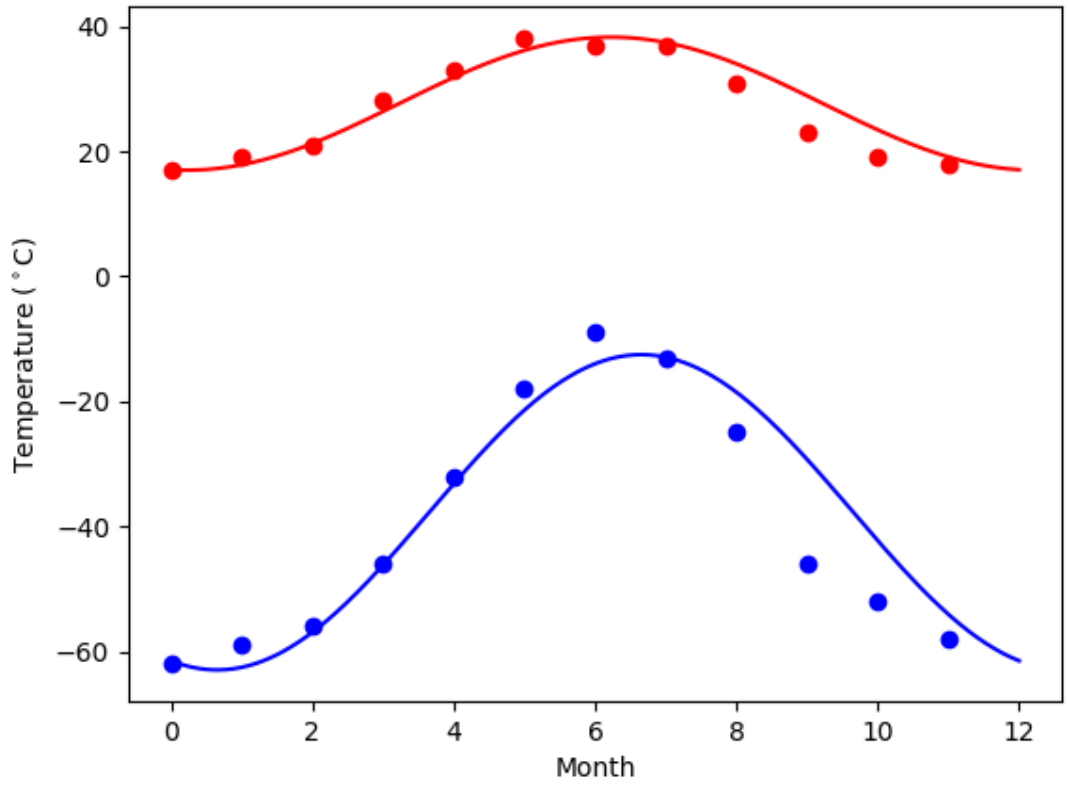

```
Text(35.472222222222214, 0.5, 'Min and max temperature')
```

Fitting it to a periodic function

```
import scipy as sp


def yearly_temps(times, avg, ampl, time_offset):
    return avg + ampl * np.cos((times + time_offset) * 2 * np.pi / times.max())


res_max, cov_max = sp.optimize.curve_fit(yearly_temps, months, temp_max, [20, 10, 0])
res_min, cov_min = sp.optimize.curve_fit(yearly_temps, months, temp_min, [-40, 20, 0])
```

Plotting the fit

```
days = np.linspace(0, 12, num=365)

plt.figure()
plt.plot(months, temp_max, "ro")
plt.plot(days, yearly_temps(days, *res_max), "r-")
plt.plot(months, temp_min, "bo")
plt.plot(days, yearly_temps(days, *res_min), "b-")
plt.xlabel("Month")
plt.ylabel(r"Temperature ($^\circ$C)")

plt.show()
```



Total running time of the script: (0 minutes 0.105 seconds)

Simple image blur by convolution with a Gaussian kernel

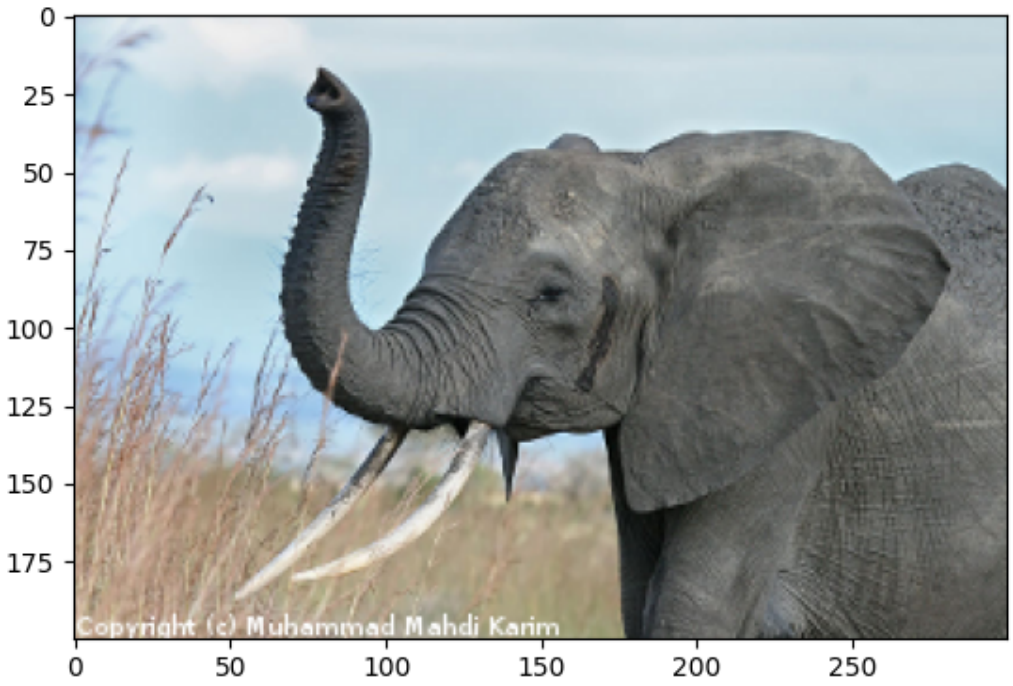
Blur an an image (`../../../../data/elephant.png`) using a Gaussian kernel.

Convolution is easy to perform with FFT: convolving two signals boils down to multiplying their FFTs (and performing an inverse FFT).

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

The original image

```
# read image
img = plt.imread("../../../../data/elephant.png")
plt.figure()
plt.imshow(img)
```




```
<matplotlib.image.AxesImage object at 0x7f3b2bc5a180>
```

Prepare an Gaussian convolution kernel

```
# First a 1-D  Gaussian
t = np.linspace(-10, 10, 30)
bump = np.exp(-0.1 * t**2)
bump /= np.trapezoid(bump)  # normalize the integral to 1

# make a 2-D kernel out of it
kernel = bump[:, np.newaxis] * bump[np.newaxis, :]
```

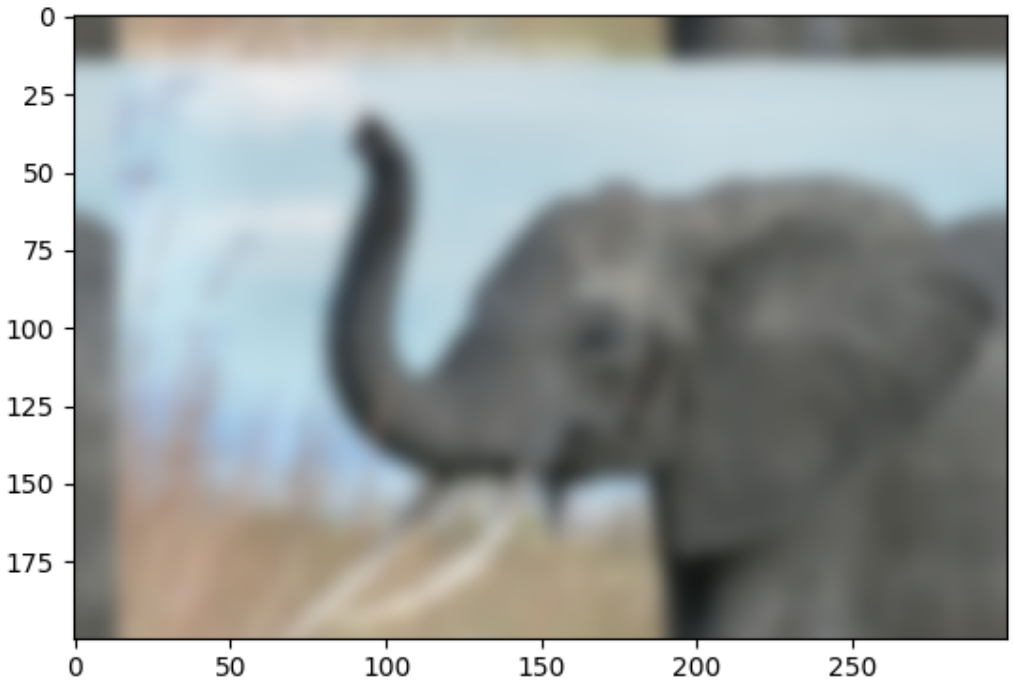
Implement convolution via FFT

```
# Padded fourier transform, with the same shape as the image
# We use :func:`scipy.fft.fft2` to have a 2D FFT
kernel_ft = sp.fft.fft2(kernel, s=img.shape[:2], axes=(0, 1))

# convolve
img_ft = sp.fft.fft2(img, axes=(0, 1))
```

(continues on next page)

(continued from previous page)

```
# the 'newaxis' is to match to color direction
img2_ft = kernel_ft[:, :, np.newaxis] * img_ft
img2 = sp.fft.ifft2(img2_ft, axes=(0, 1)).real

# clip values to range
img2 = np.clip(img2, 0, 1)

# plot output
plt.figure()
plt.imshow(img2)
```




```
<matplotlib.image.AxesImage object at 0x7f3b2b985d60>
```

Further exercise (only if you are familiar with this stuff):

A "wrapped border" appears in the upper left and top edges of the image. This is because the padding is not done correctly, and does not take the kernel size into account (so the convolution "flows out of bounds of the image"). Try to remove this artifact.

A function to do it: `scipy.signal.fftconvolve()`

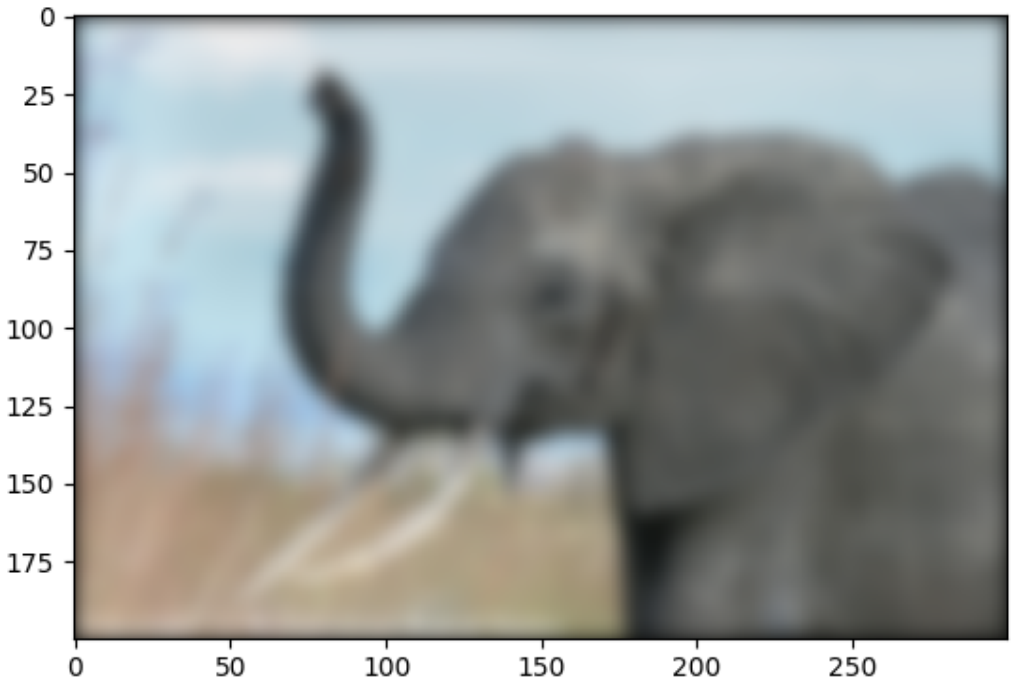
The above exercise was only for didactic reasons: there exists a function in scipy that will do this for us, and probably do a better job: `scipy.signal.fftconvolve()`

```
# mode='same' is there to enforce the same output shape as input arrays
# (ie avoid border effects)
img3 = sp.signal.fftconvolve(img, kernel[:, :, np.newaxis], mode="same")
```

(continues on next page)

(continued from previous page)

```
plt.figure()
plt.imshow(img3)
```




```
<matplotlib.image.AxesImage object at 0x7f3b2b985040>
```

Note that we still have a decay to zero at the border of the image. Using `scipy.ndimage.gaussian_filter()` would get rid of this artifact

```
plt.show()
```

Total running time of the script: (0 minutes 0.314 seconds)

Image denoising by FFT

Denoise an image (`../../../../data/moonlanding.png`) by implementing a blur with an FFT.

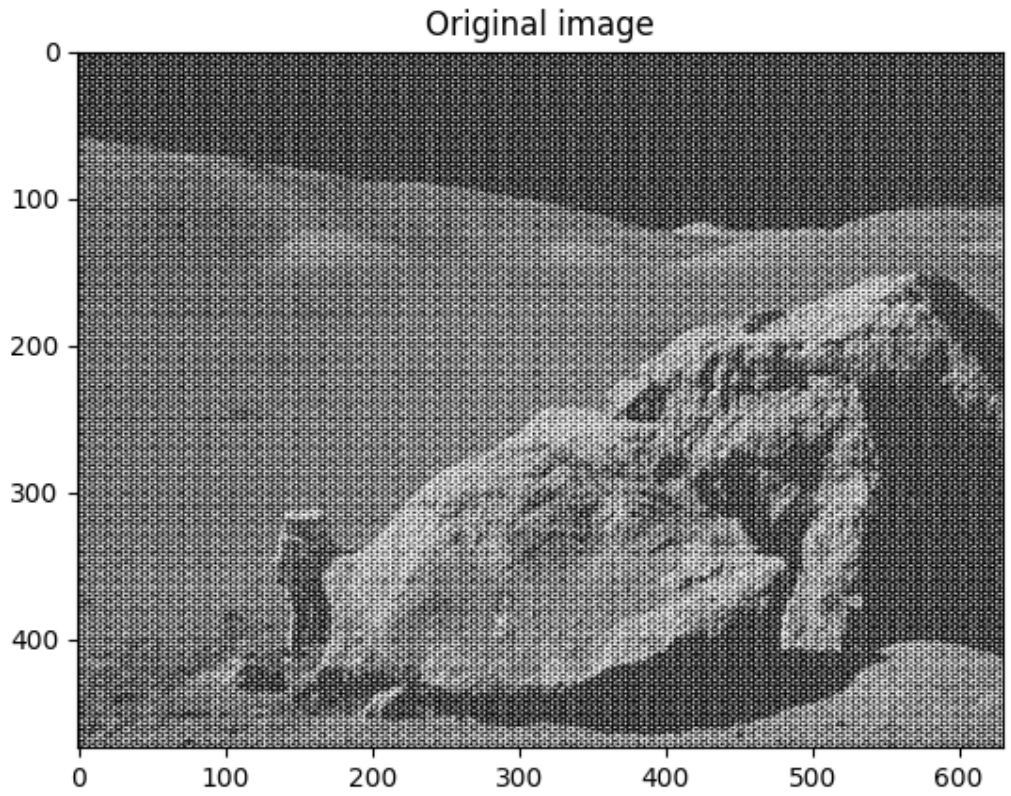
Implements, via FFT, the following convolution:

$$f_1(t) = \int dt'\, K(t - t') f_0(t')$$

$$\tilde{f}_1(\omega) = \tilde{K}(\omega) \tilde{f}_0(\omega)$$

Read and plot the image

```
import numpy as np
import matplotlib.pyplot as plt

im = plt.imread("../../../../data/moonlanding.png").astype(float)

plt.figure()
plt.imshow(im, "gray")
plt.title("Original image")
```




```
Text(0.5, 1.0, 'Original image')
```

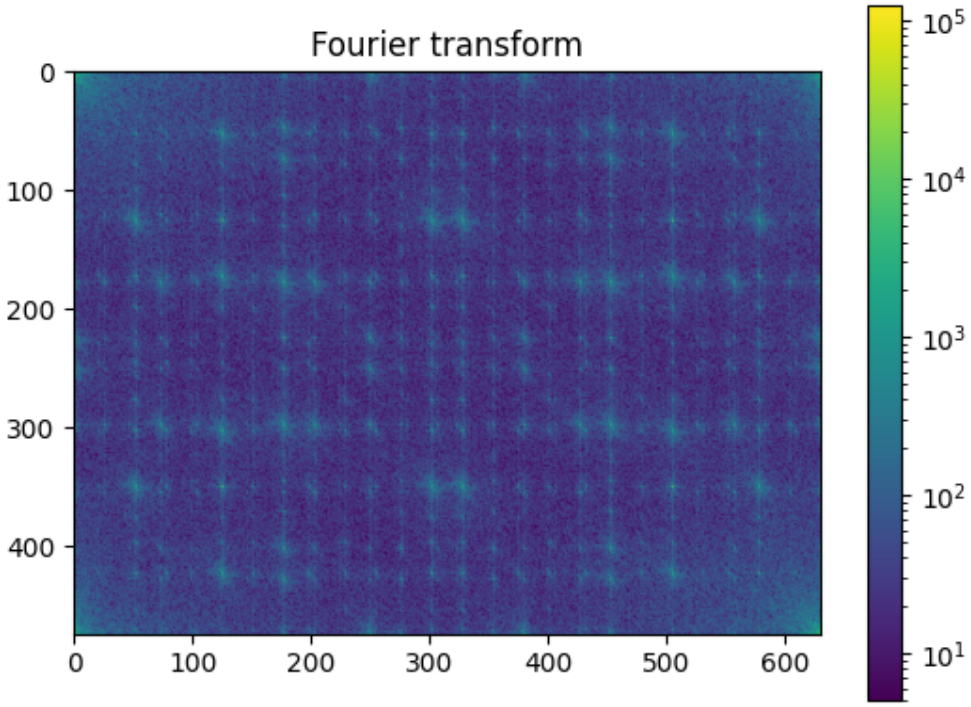
Compute the 2d FFT of the input image

```
import scipy as sp

im_fft = sp.fft.fft2(im)

# Show the results


def plot_spectrum(im_fft):
    from matplotlib.colors import LogNorm

    # A logarithmic colormap
    plt.imshow(np.abs(im_fft), norm=LogNorm(vmin=5))
```

(continues on next page)

(continued from previous page)

```
    plt.colorbar()


plt.figure()
plot_spectrum(im_fft)
plt.title("Fourier transform")
```




```
Text(0.5, 1.0, 'Fourier transform')
```

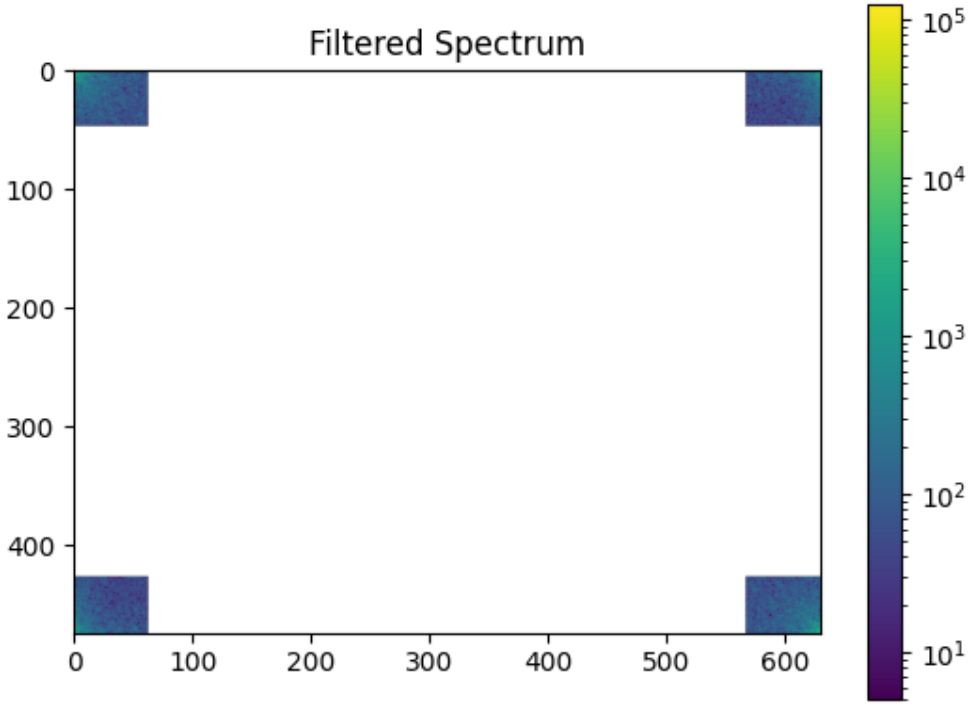
Filter in FFT

```
# In the lines following, we'll make a copy of the original spectrum and
# truncate coefficients.

# Define the fraction of coefficients (in each direction) we keep
keep_fraction = 0.1

# Call ff a copy of the original transform. NumPy arrays have a copy
# method for this purpose.
im_fft2 = im_fft.copy()

# Set r and c to be the number of rows and columns of the array.
r, c = im_fft2.shape

# Set to zero all rows with indices between r*keep_fraction and
# r*(1-keep_fraction):
```

(continues on next page)

(continued from previous page)

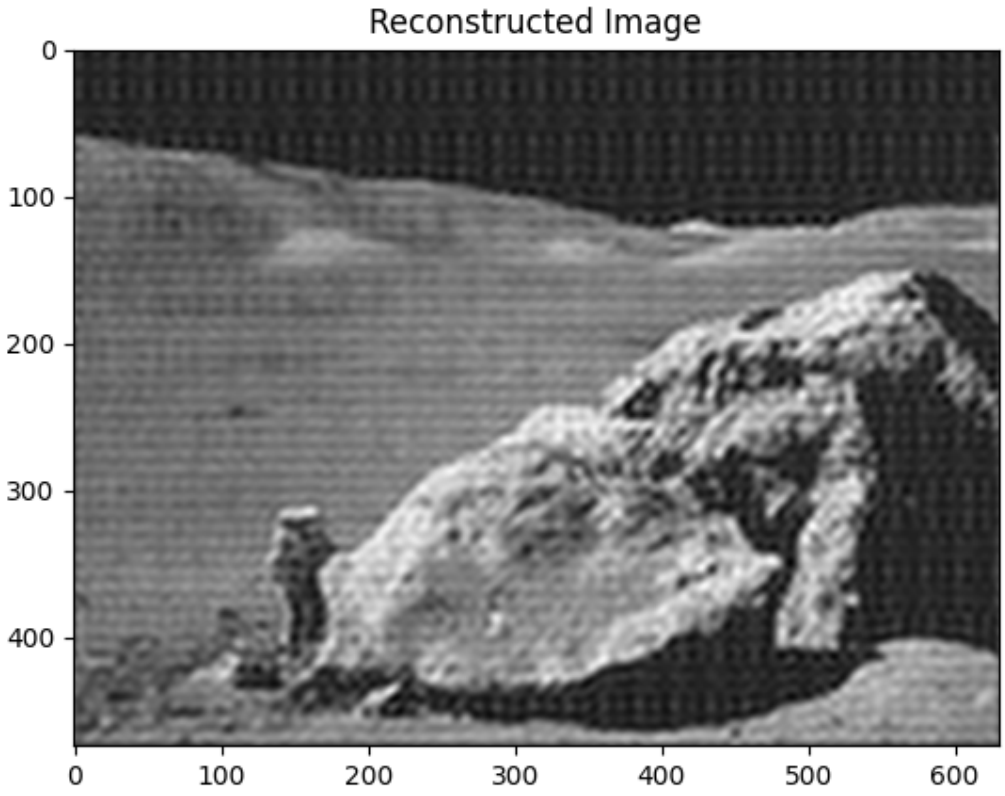
```
im_fft2[int(r * keep_fraction) : int(r * (1 - keep_fraction))] = 0

# Similarly with the columns:
im_fft2[:, int(c * keep_fraction) : int(c * (1 - keep_fraction))] = 0

plt.figure()
plot_spectrum(im_fft2)
plt.title("Filtered Spectrum")
```




```
Text(0.5, 1.0, 'Filtered Spectrum')
```

Reconstruct the final image

```
# Reconstruct the denoised image from the filtered spectrum, keep only the
# real part for display.
im_new = sp.fft.ifft2(im_fft2).real

plt.figure()
plt.imshow(im_new, "gray")
plt.title("Reconstructed Image")
```

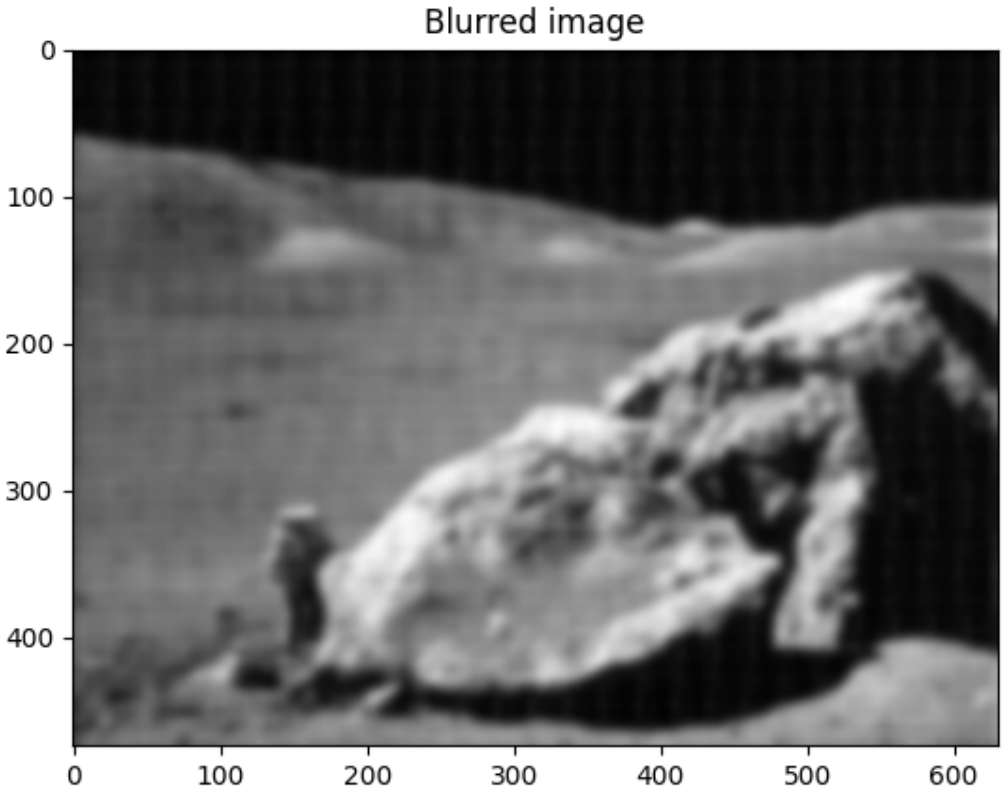



```
Text(0.5, 1.0, 'Reconstructed Image')
```

Easier and better: `scipy.ndimage.gaussian_filter()`

Implementing filtering directly with FFTs is tricky and time consuming. We can use the Gaussian filter from `scipy.ndimage`

```
im_blur = sp.ndimage.gaussian_filter(im, 4)

plt.figure()
plt.imshow(im_blur, "gray")
plt.title("Blurred image")

plt.show()
```



Total running time of the script: (0 minutes 0.761 seconds)

See also

References to go further

- Some chapters of the advanced and the packages and applications parts of the SciPy lectures
- The SciPy cookbook

CHAPTER 6

Getting help and finding documentation

Author: *Emmanuelle Gouillart*

Rather than knowing all functions in NumPy and SciPy, it is important to find rapidly information throughout the documentation and the available help. Here are some ways to get information:

- In Ipython, `help function` opens the docstring of the function. Only type the beginning of the function's name and use tab completion to display the matching functions.

```
In [1]: help(np.van<TAB>

In [2]: help(np.vander)
Help on _ArrayFunctionDispatcher in module numpy:

vander(x, N=None, increasing=False)
    Generate a Vandermonde matrix.

    The columns of the output matrix are powers of the input vector. The
    order of the powers is determined by the `increasing` boolean argument.
    Specifically, when `increasing` is False, the `i`-th output column is
    the input vector raised element-wise to the power of ``N - i - 1``. Such
    a matrix with a geometric progression in each row is named for Alexandre-
    Theophile Vandermonde.

    Parameters
    ----------
    x : array_like
        1-D input array.
    N : int, optional
        Number of columns in the output.  If `N` is not specified, a square
        array is returned (``N = len(x)``).
```

(continues on next page)

(continued from previous page)

```
    increasing : bool, optional
        Order of the powers of the columns.  If True, the powers increase
        from left to right, if False (the default) they are reversed.

    Returns
    -------
    out : ndarray
        Vandermonde matrix.  If `increasing` is False, the first column is
        ``x^(N-1)``, the second ``x^(N-2)`` and so forth. If `increasing` is
        True, the columns are ``x^0, x^1, ..., x^(N-1)``.

    See Also
    --------
    polynomial.polynomial.polyvander

    Examples
    --------
    >>> import numpy as np
    >>> x = np.array([1, 2, 3, 5])
    >>> N = 3
    >>> np.vander(x, N)
    array([[ 1,  1,  1],
           [ 4,  2,  1],
           [ 9,  3,  1],
           [25,  5,  1]])

    >>> np.column_stack([x**(N-1-i) for i in range(N)])
    array([[ 1,  1,  1],
           [ 4,  2,  1],
           [ 9,  3,  1],
           [25,  5,  1]])

    >>> x = np.array([1, 2, 3, 5])
    >>> np.vander(x)
    array([[  1,   1,   1,   1],
           [  8,   4,   2,   1],
           [ 27,   9,   3,   1],
           [125,  25,   5,   1]])
    >>> np.vander(x, increasing=True)
    array([[  1,   1,   1,   1],
           [  1,   2,   4,   8],
           [  1,   3,   9,  27],
           [  1,   5,  25, 125]])

    The determinant of a square Vandermonde matrix is the product
    of the differences between the values of the input vector:

    >>> np.linalg.det(np.vander(x))
    48.000000000000043 # may vary
    >>> (5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
    48
```

In Ipython it is not possible to open a separated window for help and documentation; however one can always open a second `Ipython` shell just to display help and docstrings...

- Numpy's and Scipy's documentations can be browsed online on https://scipy.org and https://numpy.org. The `search` button is quite useful inside the reference documentation of the two

packages.

Tutorials on various topics as well as the complete API with all docstrings are found on this website.

- Numpy's and Scipy's documentation is enriched and updated on a regular basis by users on a wiki https://numpy.org/doc/stable/. As a result, some docstrings are clearer or more detailed on the wiki, and you may want to read directly the documentation on the wiki instead of the official documentation website. Note that anyone can create an account on the wiki and write better documentation; this is an easy way to contribute to an open-source project and improve the tools you are using!
- The SciPy Cookbook https://scipy-cookbook.readthedocs.io gives recipes on many common problems frequently encountered, such as fitting data points, solving ODE, etc.
- Matplotlib's website https://matplotlib.org/ features a very nice **gallery** with a large number of plots, each of them shows both the source code and the resulting plot. This is very useful for learning by example. More standard documentation is also available.
- In Ipython, the magical function `%psearch` search for objects matching patterns. This is useful if, for example, one does not know the exact name of a function.

```
In [3]: import numpy as np
```

- If everything listed above fails (and Google doesn't have the answer)... don't despair! There is a vibrant Scientific Python community. Scientific Python is present on various platform. https://scientific-python.org/community/

  Packages like SciPy and NumPy also have their own channels. Have a look at their respective websites to find out how to engage with users and maintainers.

Part II

Advanced topics

This part of the *Scientific Python Lectures* is dedicated to advanced usage. It strives to educate the proficient Python coder to be an expert and tackles various specific topics.

CHAPTER 7

Advanced Python Constructs

Author *Zbigniew Jędrzejewski-Szmek*

This section covers some features of the Python language which can be considered advanced — in the sense that not every language has them, and also in the sense that they are more useful in more complicated programs or libraries, but not in the sense of being particularly specialized, or particularly complicated.

It is important to underline that this chapter is purely about the language itself — about features supported through special syntax complemented by functionality of the Python stdlib, which could not be implemented through clever external modules.

The process of developing the Python programming language, its syntax, is very transparent; proposed changes are evaluated from various angles and discussed via *Python Enhancement Proposals* — PEPs. As a result, features described in this chapter were added after it was shown that they indeed solve real problems and that their use is as simple as possible.

Chapter contents

- *Iterators, generator expressions and generators*
  - *Iterators*
  - *Generator expressions*
  - *Generators*
  - *Bidirectional communication*
  - *Chaining generators*
- *Decorators*

  - *Replacing or tweaking the original object*
  - *Decorators implemented as classes and as functions*
  - *Copying the docstring and other attributes of the original function*
  - *Examples in the standard library*
  - *Deprecation of functions*
  - *A `while`-loop removing decorator*
  - *A plugin registration system*
- *Context managers*
  - *Catching exceptions*
  - *Using generators to define context managers*

7.1 Iterators, generator expressions and generators

7.1.1 Iterators

Simplicity

Duplication of effort is wasteful, and replacing the various home-grown approaches with a standard feature usually ends up making things more readable, and interoperable as well.

Guido van Rossum — Adding Optional Static Typing to Python

An iterator is an object adhering to the iterator protocol — basically this means that it has a `next` method, which, when called, returns the next item in the sequence, and when there's nothing to return, raises the `StopIteration` exception.

An iterator object allows to loop just once. It holds the state (position) of a single iteration, or from the other side, each loop over a sequence requires a single iterator object. This means that we can iterate over the same sequence more than once concurrently. Separating the iteration logic from the sequence allows us to have more than one way of iteration.

Calling the `__iter__` method on a container to create an iterator object is the most straightforward way to get hold of an iterator. The `iter` function does that for us, saving a few keystrokes.

```
>>> nums = [1, 2, 3]       # note that ... varies: these are different objects
>>> iter(nums)
<...iterator object at ...>
>>> nums.__iter__()
<...iterator object at ...>
>>> nums.__reversed__()
<...reverseiterator object at ...>

>>> it = iter(nums)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
StopIteration
```

When used in a loop, `StopIteration` is swallowed and causes the loop to finish. But with explicit invocation, we can see that once the iterator is exhausted, accessing it raises an exception.

Using the for..in loop also uses the `__iter__` method. This allows us to transparently start the iteration over a sequence. But if we already have the iterator, we want to be able to use it in an `for` loop in the same way. In order to achieve this, iterators in addition to `next` are also required to have a method called `__iter__` which returns the iterator (`self`).

Support for iteration is pervasive in Python: all sequences and unordered containers in the standard library allow this. The concept is also stretched to other things: e.g. `file` objects support iteration over lines.

```
>>> with open("/etc/fstab") as f:
...     f is f.__iter__()
...
True
```

The `file` is an iterator itself and it's `__iter__` method doesn't create a separate object: only a single thread of sequential access is allowed.

7.1.2 Generator expressions

A second way in which iterator objects are created is through **generator expressions**, the basis for **list comprehensions**. To increase clarity, a generator expression must always be enclosed in parentheses or an expression. If round parentheses are used, then a generator iterator is created. If rectangular parentheses are used, the process is short-circuited and we get a `list`.

```
>>> (i for i in nums)
<generator object <genexpr> at 0x...>
>>> [i for i in nums]
[1, 2, 3]
>>> list(i for i in nums)
[1, 2, 3]
```

The list comprehension syntax also extends to **dictionary and set comprehensions**. A `set` is created when the generator expression is enclosed in curly braces. A `dict` is created when the generator expression contains "pairs" of the form `key:value`:

```
>>> {i for i in range(3)}
{0, 1, 2}
>>> {i:i**2 for i in range(3)}
{0: 0, 1: 1, 2: 4}
```

One *gotcha* should be mentioned: in old Pythons the index variable (`i`) would leak, and in versions >= 3 this is fixed.

7.1.3 Generators

Generators

A generator is a function that produces a sequence of results instead of a single value.

David Beazley — A Curious Course on Coroutines and Concurrency

A third way to create iterator objects is to call a generator function. A **generator** is a function containing the keyword yield. It must be noted that the mere presence of this keyword completely changes the nature of the function: this `yield` statement doesn't have to be invoked, or even reachable, but causes the function to be marked as a generator. When a normal function is called, the instructions contained in the body start to be executed. When a generator is called, the execution stops before the first instruction in the body. An invocation of a generator function creates a generator object, adhering to the iterator protocol. As with normal function invocations, concurrent and recursive invocations are allowed.

When `next` is called, the function is executed until the first `yield`. Each encountered `yield` statement gives a value becomes the return value of `next`. After executing the `yield` statement, the execution of this function is suspended.

```
>>> def f():
...     yield 1
...     yield 2
>>> f()
<generator object f at 0x...>
>>> gen = f()
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
StopIteration
```

Let's go over the life of the single invocation of the generator function.

```
>>> def f():
...     print("-- start --")
...     yield 3
...     print("-- finish --")
...     yield 4
>>> gen = f()
>>> next(gen)
-- start --
3
>>> next(gen)
-- finish --
4
>>> next(gen)
Traceback (most recent call last):
  ...
StopIteration
```

Contrary to a normal function, where executing `f()` would immediately cause the first `print` to be executed, `gen` is assigned without executing any statements in the function body. Only when `gen.__next__()` is invoked by `next`, the statements up to the first `yield` are executed. The second `next` prints `-- finish --` and execution halts on the second `yield`. The third `next` falls of the end of the function. Since no `yield` was reached, an exception is raised.

What happens with the function after a yield, when the control passes to the caller? The state of each generator is stored in the generator object. From the point of view of the generator function, is looks almost as if it was running in a separate thread, but this is just an illusion: execution is strictly single-threaded, but the interpreter keeps and restores the state in between the requests for the next value.

Why are generators useful? As noted in the parts about iterators, a generator function is just a different way to create an iterator object. Everything that can be done with `yield` statements, could also be

done with `next` methods. Nevertheless, using a function and having the interpreter perform its magic to create an iterator has advantages. A function can be much shorter than the definition of a class with the required `next` and `__iter__` methods. What is more important, it is easier for the author of the generator to understand the state which is kept in local variables, as opposed to instance attributes, which have to be used to pass data between consecutive invocations of `next` on an iterator object.

A broader question is why are iterators useful? When an iterator is used to power a loop, the loop becomes very simple. The code to initialise the state, to decide if the loop is finished, and to find the next value is extracted into a separate place. This highlights the body of the loop — the interesting part. In addition, it is possible to reuse the iterator code in other places.

7.1.4 Bidirectional communication

Each `yield` statement causes a value to be passed to the caller. This is the reason for the introduction of generators by **PEP 255**. But communication in the reverse direction is also useful. One obvious way would be some external state, either a global variable or a shared mutable object. Direct communication is possible thanks to **PEP 342**. It is achieved by turning the previously boring `yield` statement into an expression. When the generator resumes execution after a `yield` statement, the caller can call a method on the generator object to either pass a value **into** the generator, which then is returned by the `yield` statement, or a different method to inject an exception into the generator.

The first of the new methods is `send(value)`, which is similar to `next()`, but passes `value` into the generator to be used for the value of the `yield` expression. In fact, `g.next()` and `g.send(None)` are equivalent.

The second of the new methods is `throw(type, value=None, traceback=None)` which is equivalent to:

```
raise type, value, traceback
```

at the point of the `yield` statement.

Unlike raise (which immediately raises an exception from the current execution point), `throw()` first resumes the generator, and only then raises the exception. The word throw was picked because it is suggestive of putting the exception in another location, and is associated with exceptions in other languages.

What happens when an exception is raised inside the generator? It can be either raised explicitly or when executing some statements or it can be injected at the point of a `yield` statement by means of the `throw()` method. In either case, such an exception propagates in the standard manner: it can be intercepted by an `except` or `finally` clause, or otherwise it causes the execution of the generator function to be aborted and propagates in the caller.

For completeness' sake, it's worth mentioning that generator iterators also have a `close()` method, which can be used to force a generator that would otherwise be able to provide more values to finish immediately. It allows the generator `__del__` method to destroy objects holding the state of generator. Let's define a generator which just prints what is passed in through send and throw.

```
>>> import itertools
>>> def g():
...     print('--start--')
...     for i in itertools.count():
...         print('--yielding %i--' % i)
...         try:
...             ans = yield i
...         except GeneratorExit:
...             print('--closing--')
...             raise
...         except Exception as e:
...             print('--yield raised %r--' % e)
...         else:
...             print('--yield returned %s--' % ans)
```

(continues on next page)

(continued from previous page)

```
>>> it = g()
>>> next(it)
--start--
--yielding 0--
0
>>> it.send(11)
--yield returned 11--
--yielding 1--
1
>>> it.throw(IndexError)
--yield raised IndexError()--
--yielding 2--
2
>>> it.close()
--closing--
```

7.1.5 Chaining generators

Note

This is a preview of **PEP 380** (not yet implemented, but accepted for Python 3.3).

Let's say we are writing a generator and we want to yield a number of values generated by a second generator, a **subgenerator**. If yielding of values is the only concern, this can be performed without much difficulty using a loop such as

```
subgen = some_other_generator()
for v in subgen:
    yield v
```

However, if the subgenerator is to interact properly with the caller in the case of calls to `send()`, `throw()` and `close()`, things become considerably more difficult. The `yield` statement has to be guarded by a try..except..finally structure similar to the one defined in the previous section to "debug" the generator function. Such code is provided in **PEP 380#id13**, here it suffices to say that new syntax to properly yield from a subgenerator is being introduced in Python 3.3:

```
yield from some_other_generator()
```

This behaves like the explicit loop above, repeatedly yielding values from `some_other_generator` until it is exhausted, but also forwards `send`, `throw` and `close` to the subgenerator.

7.2 Decorators

Summary

This amazing feature appeared in the language almost apologetically and with concern that it might not be that useful.

Bruce Eckel — An Introduction to Python Decorators

Since functions and classes are objects, they can be passed around. Since they are mutable objects, they can be modified. The act of altering a function or class object after it has been constructed but before is is bound to its name is called decorating.

There are two things hiding behind the name "decorator" — one is the function which does the work of decorating, i.e. performs the real work, and the other one is the expression adhering to the decorator syntax, i.e. an at-symbol and the name of the decorating function.

Function can be decorated by using the decorator syntax for functions:

```
@decorator              # ❷
def function():         # ❶
    pass
```

- A function is defined in the standard way. ❶
- An expression starting with `@` placed before the function definition is the decorator ❷. The part after `@` must be a simple expression, usually this is just the name of a function or class. This part is evaluated first, and after the function defined below is ready, the decorator is called with the newly defined function object as the single argument. The value returned by the decorator is attached to the original name of the function.

Decorators can be applied to functions and to classes. For classes the semantics are identical — the original class definition is used as an argument to call the decorator and whatever is returned is assigned under the original name.

Before the decorator syntax was implemented (**PEP 318**), it was possible to achieve the same effect by assigning the function or class object to a temporary variable and then invoking the decorator explicitly and then assigning the return value to the name of the function. This sounds like more typing, and it is, and also the name of the decorated function doubling as a temporary variable must be used at least three times, which is prone to errors. Nevertheless, the example above is equivalent to:

```
def function():                 # ❶
    pass
function = decorator(function)  # ❷
```

Decorators can be stacked — the order of application is bottom-to-top, or inside-out. The semantics are such that the originally defined function is used as an argument for the first decorator, whatever is returned by the first decorator is used as an argument for the second decorator, ..., and whatever is returned by the last decorator is attached under the name of the original function.

The decorator syntax was chosen for its readability. Since the decorator is specified before the header of the function, it is obvious that its is not a part of the function body and its clear that it can only operate on the whole function. Because the expression is prefixed with `@` is stands out and is hard to miss ("in your face", according to the PEP :)). When more than one decorator is applied, each one is placed on a separate line in an easy to read way.

7.2.1 Replacing or tweaking the original object

Decorators can either return the same function or class object or they can return a completely different object. In the first case, the decorator can exploit the fact that function and class objects are mutable and add attributes, e.g. add a docstring to a class. A decorator might do something useful even without modifying the object, for example register the decorated class in a global registry. In the second case, virtually anything is possible: when something different is substituted for the original function or class, the new object can be completely different. Nevertheless, such behaviour is not the purpose of decorators: they are intended to tweak the decorated object, not do something unpredictable. Therefore, when a function is "decorated" by replacing it with a different function, the new function usually calls the original function, after doing some preparatory work. Likewise, when a class is "decorated" by replacing if with a new class, the new class is usually derived from the original class. When the purpose of the decorator is to do something "every time", like to log every call to a decorated function, only the second type of decorators can be used. On the other hand, if the first type is sufficient, it is better to use it, because it is simpler.

7.2.2 Decorators implemented as classes and as functions

The only *requirement* on decorators is that they can be called with a single argument. This means that decorators can be implemented as normal functions, or as classes with a `__call__` method, or in theory, even as lambda functions.

Let's compare the function and class approaches. The decorator expression (the part after `@`) can be either just a name, or a call. The bare-name approach is nice (less to type, looks cleaner, etc.), but is only possible when no arguments are needed to customise the decorator. Decorators written as functions can be used in those two cases:

```
>>> def simple_decorator(function):
...   print("doing decoration")
...   return function
>>> @simple_decorator
... def function():
...   print("inside function")
doing decoration
>>> function()
inside function

>>> def decorator_with_arguments(arg):
...   print("defining the decorator")
...   def _decorator(function):
...       # in this inner function, arg is available too
...       print("doing decoration, %r" % arg)
...       return function
...   return _decorator
>>> @decorator_with_arguments("abc")
... def function():
...   print("inside function")
defining the decorator
doing decoration, 'abc'
>>> function()
inside function
```

The two trivial decorators above fall into the category of decorators which return the original function. If they were to return a new function, an extra level of nestedness would be required. In the worst case, three levels of nested functions.

```
>>> def replacing_decorator_with_args(arg):
...   print("defining the decorator")
...   def _decorator(function):
...       # in this inner function, arg is available too
...       print("doing decoration, %r" % arg)
...       def _wrapper(*args, **kwargs):
...           print("inside wrapper, %r %r" % (args, kwargs))
...           return function(*args, **kwargs)
...       return _wrapper
...   return _decorator
>>> @replacing_decorator_with_args("abc")
... def function(*args, **kwargs):
...     print("inside function, %r %r" % (args, kwargs))
...     return 14
defining the decorator
doing decoration, 'abc'
>>> function(11, 12)
inside wrapper, (11, 12) {}
inside function, (11, 12) {}
```

(continues on next page)

(continued from previous page)

```
14
```

The `_wrapper` function is defined to accept all positional and keyword arguments. In general we cannot know what arguments the decorated function is supposed to accept, so the wrapper function just passes everything to the wrapped function. One unfortunate consequence is that the apparent argument list is misleading.

Compared to decorators defined as functions, complex decorators defined as classes are simpler. When an object is created, the `__init__` method is only allowed to return `None`, and the type of the created object cannot be changed. This means that when a decorator is defined as a class, it doesn't make much sense to use the argument-less form: the final decorated object would just be an instance of the decorating class, returned by the constructor call, which is not very useful. Therefore it's enough to discuss class-based decorators where arguments are given in the decorator expression and the decorator `__init__` method is used for decorator construction.

```
>>> class decorator_class(object):
...     def __init__(self, arg):
...         # this method is called in the decorator expression
...         print("in decorator init, %s" % arg)
...         self.arg = arg
...     def __call__(self, function):
...         # this method is called to do the job
...         print("in decorator call, %s" % self.arg)
...         return function
>>> deco_instance = decorator_class('foo')
in decorator init, foo
>>> @deco_instance
... def function(*args, **kwargs):
...     print("in function, %s %s" % (args, kwargs))
in decorator call, foo
>>> function()
in function, () {}
```

Contrary to normal rules (**PEP 8**) decorators written as classes behave more like functions and therefore their name often starts with a lowercase letter.

In reality, it doesn't make much sense to create a new class just to have a decorator which returns the original function. Objects are supposed to hold state, and such decorators are more useful when the decorator returns a new object.

```
>>> class replacing_decorator_class(object):
...     def __init__(self, arg):
...         # this method is called in the decorator expression
...         print("in decorator init, %s" % arg)
...         self.arg = arg
...     def __call__(self, function):
...         # this method is called to do the job
...         print("in decorator call, %s" % self.arg)
...         self.function = function
...         return self._wrapper
...     def _wrapper(self, *args, **kwargs):
...         print("in the wrapper, %s %s" % (args, kwargs))
...         return self.function(*args, **kwargs)
>>> deco_instance = replacing_decorator_class('foo')
in decorator init, foo
>>> @deco_instance
... def function(*args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
...   print("in function, %s %s" % (args, kwargs))
in decorator call, foo
>>> function(11, 12)
in the wrapper, (11, 12) {}
in function, (11, 12) {}
```

A decorator like this can do pretty much anything, since it can modify the original function object and mangle the arguments, call the original function or not, and afterwards mangle the return value.

7.2.3 Copying the docstring and other attributes of the original function

When a new function is returned by the decorator to replace the original function, an unfortunate consequence is that the original function name, the original docstring, the original argument list are lost. Those attributes of the original function can partially be "transplanted" to the new function by setting `__doc__` (the docstring), `__module__` and `__name__` (the full name of the function), and `__annotations__` (extra information about arguments and the return value of the function available in Python 3). This can be done automatically by using `functools.update_wrapper`.

`functools.update_wrapper(wrapper, wrapped)`

"Update a wrapper function to look like the wrapped function."

```
>>> import functools
>>> def replacing_decorator_with_args(arg):
...   print("defining the decorator")
...   def _decorator(function):
...       print("doing decoration, %r" % arg)
...       def _wrapper(*args, **kwargs):
...           print("inside wrapper, %r %r" % (args, kwargs))
...           return function(*args, **kwargs)
...       return functools.update_wrapper(_wrapper, function)
...   return _decorator
>>> @replacing_decorator_with_args("abc")
... def function():
...     "extensive documentation"
...     print("inside function")
...     return 14
defining the decorator
doing decoration, 'abc'
>>> function
<function function at 0x...>
>>> print(function.__doc__)
extensive documentation
```

One important thing is missing from the list of attributes which can be copied to the replacement function: the argument list. The default values for arguments can be modified through the `__defaults__`, `__kwdefaults__` attributes, but unfortunately the argument list itself cannot be set as an attribute. This means that `help(function)` will display a useless argument list which will be confusing for the user of the function. An effective but ugly way around this problem is to create the wrapper dynamically, using `eval`. This can be automated by using the external `decorator` module. It provides support for the `decorator` decorator, which takes a wrapper and turns it into a decorator which preserves the function signature.

To sum things up, decorators should always use `functools.update_wrapper` or some other means of copying function attributes.

7.2.4 Examples in the standard library

First, it should be mentioned that there's a number of useful decorators available in the standard library. There are three decorators which really form a part of the language:

- `classmethod` causes a method to become a "class method", which means that it can be invoked without creating an instance of the class. When a normal method is invoked, the interpreter inserts the instance object as the first positional parameter, `self`. When a class method is invoked, the class itself is given as the first parameter, often called `cls`.

  Class methods are still accessible through the class' namespace, so they don't pollute the module's namespace. Class methods can be used to provide alternative constructors:

```
class Array(object):
    def __init__(self, data):
        self.data = data

    @classmethod
    def fromfile(cls, file):
        data = numpy.load(file)
        return cls(data)
```

  This is cleaner than using a multitude of flags to `__init__`.

- `staticmethod` is applied to methods to make them "static", i.e. basically a normal function, but accessible through the class namespace. This can be useful when the function is only needed inside this class (its name would then be prefixed with `_`), or when we want the user to think of the method as connected to the class, despite an implementation which doesn't require this.

- `property` is the pythonic answer to the problem of getters and setters. A method decorated with `property` becomes a getter which is automatically called on attribute access.

```
>>> class A(object):
...   @property
...   def a(self):
...     "an important attribute"
...     return "a value"
>>> A.a
<property object at 0x...>
>>> A().a
'a value'
```

  In this example, `A.a` is an read-only attribute. It is also documented: `help(A)` includes the docstring for attribute `a` taken from the getter method. Defining `a` as a property allows it to be a calculated on the fly, and has the side effect of making it read-only, because no setter is defined.

  To have a setter and a getter, two methods are required, obviously:

```
class Rectangle(object):
    def __init__(self, edge):
        self.edge = edge

    @property
    def area(self):
        """Computed area.

        Setting this updates the edge length to the proper value.
        """
        return self.edge**2

    @area.setter
```

(continues on next page)

(continued from previous page)

```
    def area(self, area):
        self.edge = area ** 0.5
```

The way that this works, is that the `property` decorator replaces the getter method with a property object. This object in turn has three methods, `getter`, `setter`, and `deleter`, which can be used as decorators. Their job is to set the getter, setter and deleter of the property object (stored as attributes `fget`, `fset`, and `fdel`). The getter can be set like in the example above, when creating the object. When defining the setter, we already have the property object under `area`, and we add the setter to it by using the `setter` method. All this happens when we are creating the class.

Afterwards, when an instance of the class has been created, the property object is special. When the interpreter executes attribute access, assignment, or deletion, the job is delegated to the methods of the property object.

To make everything crystal clear, let's define a "debug" example:

```
>>> class D(object):
...     @property
...     def a(self):
...       print("getting 1")
...       return 1
...     @a.setter
...     def a(self, value):
...       print("setting %r" % value)
...     @a.deleter
...     def a(self):
...       print("deleting")
>>> D.a
<property object at 0x...>
>>> D.a.fget
<function ...>
>>> D.a.fset
<function ...>
>>> D.a.fdel
<function ...>
>>> d = D()                # ... varies, this is not the same `a` function
>>> d.a
getting 1
1
>>> d.a = 2
setting 2
>>> del d.a
deleting
>>> d.a
getting 1
1
```

Properties are a bit of a stretch for the decorator syntax. One of the premises of the decorator syntax — that the name is not duplicated — is violated, but nothing better has been invented so far. It is just good style to use the same name for the getter, setter, and deleter methods.

Some newer examples include:

- `functools.lru_cache` memoizes an arbitrary function maintaining a limited cache of arguments:answer pairs (Python 3.2)
- `functools.total_ordering` is a class decorator which fills in missing ordering methods (`__lt__`, `__gt__`, `__le__`, ...) based on a single available one.

7.2.5 Deprecation of functions

Let's say we want to print a deprecation warning on stderr on the first invocation of a function we don't like anymore. If we don't want to modify the function, we can use a decorator:

```
class deprecated(object):
    """Print a deprecation warning once on first use of the function.

    >>> @deprecated()                       # doctest: +SKIP
    ... def f():
    ...     pass
    >>> f()                                 # doctest: +SKIP
    f is deprecated
    """
    def __call__(self, func):
        self.func = func
        self.count = 0
        return self._wrapper
    def _wrapper(self, *args, **kwargs):
        self.count += 1
        if self.count == 1:
            print(self.func.__name__, 'is deprecated')
        return self.func(*args, **kwargs)
```

It can also be implemented as a function:

```
def deprecated(func):
    """Print a deprecation warning once on first use of the function.

    >>> @deprecated                         # doctest: +SKIP
    ... def f():
    ...     pass
    >>> f()                                 # doctest: +SKIP
    f is deprecated
    """
    count = [0]
    def wrapper(*args, **kwargs):
        count[0] += 1
        if count[0] == 1:
            print(func.__name__, 'is deprecated')
        return func(*args, **kwargs)
    return wrapper
```

7.2.6 A `while`-loop removing decorator

Let's say we have function which returns a lists of things, and this list created by running a loop. If we don't know how many objects will be needed, the standard way to do this is something like:

```
def find_answers():
    answers = []
    while True:
        ans = look_for_next_answer()
        if ans is None:
            break
        answers.append(ans)
    return answers
```

This is fine, as long as the body of the loop is fairly compact. Once it becomes more complicated, as often happens in real code, this becomes pretty unreadable. We could simplify this by using `yield`

statements, but then the user would have to explicitly call `list(find_answers())`.

We can define a decorator which constructs the list for us:

```
def vectorized(generator_func):
    def wrapper(*args, **kwargs):
        return list(generator_func(*args, **kwargs))
    return functools.update_wrapper(wrapper, generator_func)
```

Our function then becomes:

```
@vectorized
def find_answers():
    while True:
        ans = look_for_next_answer()
        if ans is None:
            break
        yield ans
```

7.2.7 A plugin registration system

This is a class decorator which doesn't modify the class, but just puts it in a global registry. It falls into the category of decorators returning the original object:

```
class WordProcessor(object):
    PLUGINS = []
    def process(self, text):
        for plugin in self.PLUGINS:
            text = plugin().cleanup(text)
        return text

    @classmethod
    def plugin(cls, plugin):
        cls.PLUGINS.append(plugin)

@WordProcessor.plugin
class CleanMdashesExtension(object):
    def cleanup(self, text):
        return text.replace('&mdash;', u'\N{em dash}')
```

Here we use a decorator to decentralise the registration of plugins. We call our decorator with a noun, instead of a verb, because we use it to declare that our class is a plugin for `WordProcessor`. Method `plugin` simply appends the class to the list of plugins.

A word about the plugin itself: it replaces HTML entity for em-dash with a real Unicode em-dash character. It exploits the unicode literal notation to insert a character by using its name in the unicode database ("EM DASH"). If the Unicode character was inserted directly, it would be impossible to distinguish it from an en-dash in the source of a program.

See also

More examples and reading

- **PEP 318** (function and method decorator syntax)
- **PEP 3129** (class decorator syntax)
- https://wiki.python.org/moin/PythonDecoratorLibrary
- https://docs.python.org/dev/library/functools.html

- https://pypi.org/project/decorator
- Bruce Eckel
  - Decorators I: Introduction to Python Decorators
  - Python Decorators II: Decorator Arguments
  - Python Decorators III: A Decorator-Based Build System

7.3 Context managers

A context manager is an object with `__enter__` and `__exit__` methods which can be used in the with statement:

```
with manager as var:
    do_something(var)
```

is in the simplest case equivalent to

```
var = manager.__enter__()
try:
    do_something(var)
finally:
    manager.__exit__()
```

In other words, the context manager protocol defined in **PEP 343** permits the extraction of the boring part of a try..except..finally structure into a separate class leaving only the interesting `do_something` block.

1. The `__enter__` method is called first. It can return a value which will be assigned to `var`. The `as`-part is optional: if it isn't present, the value returned by `__enter__` is simply ignored.
2. The block of code underneath `with` is executed. Just like with `try` clauses, it can either execute successfully to the end, or it can break, continue or return, or it can throw an exception. Either way, after the block is finished, the `__exit__` method is called. If an exception was thrown, the information about the exception is passed to `__exit__`, which is described below in the next subsection. In the normal case, exceptions can be ignored, just like in a `finally` clause, and will be rethrown after `__exit__` is finished.

Let's say we want to make sure that a file is closed immediately after we are done writing to it:

```
>>> class closing(object):
...   def __init__(self, obj):
...     self.obj = obj
...   def __enter__(self):
...     return self.obj
...   def __exit__(self, *args):
...     self.obj.close()
>>> with closing(open('/tmp/file', 'w')) as f:
...   f.write('the contents\n')
```

Here we have made sure that the `f.close()` is called when the `with` block is exited. Since closing files is such a common operation, the support for this is already present in the `file` class. It has an `__exit__` method which calls `close` and can be used as a context manager itself:

```
>>> with open('/tmp/file', 'a') as f:
...   f.write('more contents\n')
```

The common use for `try..finally` is releasing resources. Various different cases are implemented similarly: in the `__enter__` phase the resource is acquired, in the `__exit__` phase it is released, and the

exception, if thrown, is propagated. As with files, there's often a natural operation to perform after the object has been used and it is most convenient to have the support built in. With each release, Python provides support in more places:

- all file-like objects:
  - `file` ➥ automatically closed
  - `fileinput`, `tempfile`
  - `bz2.BZ2File`, `gzip.GzipFile`, `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` ➥ close connection
- locks
  - `multiprocessing.RLock` ➥ lock and unlock
  - `multiprocessing.Semaphore`
  - `memoryview` ➥ automatically release
- `decimal.localcontext` ➥ modify precision of computations temporarily
- `_winreg.PyHKEY` ➥ open and close hive key
- `warnings.catch_warnings` ➥ kill warnings temporarily
- `contextlib.closing` ➥ the same as the example above, call `close`
- parallel programming
  - `concurrent.futures.ThreadPoolExecutor` ➥ invoke in parallel then kill thread pool
  - `concurrent.futures.ProcessPoolExecutor` ➥ invoke in parallel then kill process pool
  - `nogil` ➥ solve the GIL problem temporarily (cython only :( )

7.3.1 Catching exceptions

When an exception is thrown in the `with`-block, it is passed as arguments to `__exit__`. Three arguments are used, the same as returned by `sys.exc_info()`: type, value, traceback. When no exception is thrown, `None` is used for all three arguments. The context manager can "swallow" the exception by returning a true value from `__exit__`. Exceptions can be easily ignored, because if `__exit__` doesn't use `return` and just falls of the end, `None` is returned, a false value, and therefore the exception is rethrown after `__exit__` is finished.

The ability to catch exceptions opens interesting possibilities. A classic example comes from unit-tests — we want to make sure that some code throws the right kind of exception:

```
class assert_raises(object):
    # based on pytest and unittest.TestCase
    def __init__(self, type):
        self.type = type
    def __enter__(self):
        pass
    def __exit__(self, type, value, traceback):
        if type is None:
            raise AssertionError('exception expected')
        if issubclass(type, self.type):
            return True # swallow the expected exception
        raise AssertionError('wrong exception type')

with assert_raises(KeyError):
    {}['foo']
```

7.3.2 Using generators to define context managers

When discussing *generators*, it was said that we prefer generators to iterators implemented as classes because they are shorter, sweeter, and the state is stored as local, not instance, variables. On the other hand, as described in *Bidirectional communication*, the flow of data between the generator and its caller can be bidirectional. This includes exceptions, which can be thrown into the generator. We would like to implement context managers as special generator functions. In fact, the generator protocol was designed to support this use case.

```
@contextlib.contextmanager
def some_generator(<arguments>):
    <setup>
    try:
        yield <value>
    finally:
        <cleanup>
```

The `contextlib.contextmanager` helper takes a generator and turns it into a context manager. The generator has to obey some rules which are enforced by the wrapper function — most importantly it must `yield` exactly once. The part before the `yield` is executed from `__enter__`, the block of code protected by the context manager is executed when the generator is suspended in `yield`, and the rest is executed in `__exit__`. If an exception is thrown, the interpreter hands it to the wrapper through `__exit__` arguments, and the wrapper function then throws it at the point of the `yield` statement. Through the use of generators, the context manager is shorter and simpler.

Let's rewrite the `closing` example as a generator:

```
@contextlib.contextmanager
def closing(obj):
    try:
        yield obj
    finally:
        obj.close()
```

Let's rewrite the `assert_raises` example as a generator:

```
@contextlib.contextmanager
def assert_raises(type):
    try:
        yield
    except type:
        return
    except Exception as value:
        raise AssertionError('wrong exception type')
    else:
        raise AssertionError('exception expected')
```

Here we use a decorator to turn generator functions into context managers!

CHAPTER 8

Advanced NumPy

Author: *Pauli Virtanen*

NumPy is at the base of Python's scientific stack of tools. Its purpose to implement efficient operations on many items in a block of memory. Understanding how it works in detail helps in making efficient use of its flexibility, taking useful shortcuts.

This section covers:

- Anatomy of NumPy arrays, and its consequences. Tips and tricks.
- Universal functions: what, why, and what to do if you want a new one.
- Integration with other tools: NumPy offers several ways to wrap any data in an ndarray, without unnecessary copies.
- Recently added features, and what's in them: PEP 3118 buffers, generalized ufuncs, ...

Prerequisites

- NumPy
- Cython
- Pillow (Python imaging library, used in a couple of examples)

Chapter contents

- *Life of ndarray*
  - *It's...*

- *Block of memory*
- *Data types*
- *Indexing scheme: strides*
- *Findings in dissection*

- *Universal functions*
  - *What they are?*
  - *Exercise: building an ufunc from scratch*
  - *Solution: building an ufunc from scratch*
  - *Generalized ufuncs*
- *Interoperability features*
  - *Sharing multidimensional, typed data*
  - *The old buffer protocol*
  - *The old buffer protocol*
  - *Array interface protocol*
- *Array siblings:* `chararray`, `maskedarray`
  - `chararray`*: vectorized string operations*
  - `masked_array` *missing data*
  - `recarray`*: purely convenience*
- *Summary*
- *Contributing to NumPy/SciPy*
  - *Why*
  - *Reporting bugs*
  - *Contributing documentation*
  - *Contributing features*
  - *How to help, in general*

Tip

In this section, NumPy will be imported as follows:

```
>>> import numpy as np
```

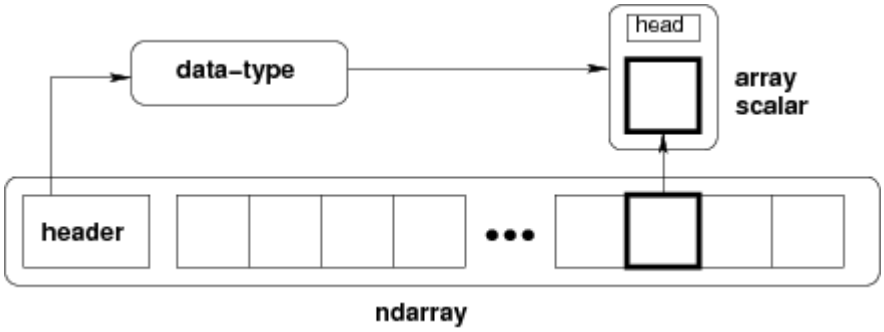
8.1 Life of ndarray

8.1.1 It's...

ndarray =

block of memory + indexing scheme + data type descriptor

- raw data
- how to locate an element
- how to interpret an element




```
typedef struct PyArrayObject {
        PyObject_HEAD

        /* Block of memory */
        char *data;

        /* Data type descriptor */
        PyArray_Descr *descr;

        /* Indexing scheme */
        int nd;
        npy_intp *dimensions;
        npy_intp *strides;

        /* Other stuff */
        PyObject *base;
        int flags;
        PyObject *weakreflist;
} PyArrayObject;
```

8.1.2 Block of memory

```
>>> x = np.array([1, 2, 3], dtype=np.int32)
>>> x.data
<... at ...>
>>> bytes(x.data)
b'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
```

Memory address of the data:

```
>>> x.__array_interface__['data'][0]
64803824
```

The whole `__array_interface__`:

```
>>> x.__array_interface__
{'data': (..., False), 'strides': None, 'descr': [('', '<i4')], 'typestr': '<i4',
↪'shape': (3,), 'version': 3}
```

Reminder: two `ndarrays` may share the same memory:

```
>>> x = np.array([1, 2, 3, 4])
>>> y = x[:-1]
>>> x[0] = 9
>>> y
array([9, 2, 3])
```

Memory does not need to be owned by an `ndarray`:

```
>>> x = b'1234'
```

x is a string (in Python 3 a bytes), we can represent its data as an array of ints:

```
>>> y = np.frombuffer(x, dtype=np.int8)
>>> y.data
<... at ...>
>>> y.base is x
True

>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : False
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

The `owndata` and `writeable` flags indicate status of the memory block.

See also

array interface

8.1.3 Data types

The descriptor

`dtype` describes a single item in the array:

type	**scalar type** of the data, one of: int8, int16, float64, *et al.* (fixed size) str, unicode, void (flexible size)
itemsize	**size** of the data block
byte-order	**byte order**: big-endian `>` / little-endian `<` / not applicable `|`
fields	sub-dtypes, if it's a **structured data type**
shape	shape of the array, if it's a **sub-array**

```
>>> np.dtype(int).type
<class 'numpy.int64'>
>>> np.dtype(int).itemsize
8
>>> np.dtype(int).byteorder
'='
```

Example: reading `.wav` files

The `.wav` file header:

chunk_id	"RIFF"
chunk_size	4-byte unsigned little-endian integer
format	"WAVE"
fmt_id	"fmt "
fmt_size	4-byte unsigned little-endian integer
audio_fmt	2-byte unsigned little-endian integer
num_channels	2-byte unsigned little-endian integer
sample_rate	4-byte unsigned little-endian integer
byte_rate	4-byte unsigned little-endian integer
block_align	2-byte unsigned little-endian integer
bits_per_sample	2-byte unsigned little-endian integer
data_id	"data"
data_size	4-byte unsigned little-endian integer

- 44-byte block of raw data (in the beginning of the file)
- ... followed by `data_size` bytes of actual sound data.

The `.wav` file header as a NumPy *structured* data type:

```
>>> wav_header_dtype = np.dtype([
...     ("chunk_id", (bytes, 4)), # flexible-sized scalar type, item size 4
...     ("chunk_size", "<u4"),    # little-endian unsigned 32-bit integer
...     ("format", "S4"),         # 4-byte string
...     ("fmt_id", "S4"),
...     ("fmt_size", "<u4"),
...     ("audio_fmt", "<u2"),     #
...     ("num_channels", "<u2"),  # .. more of the same ...
...     ("sample_rate", "<u4"),   #
...     ("byte_rate", "<u4"),
...     ("block_align", "<u2"),
...     ("bits_per_sample", "<u2"),
...     ("data_id", ("S1", (2, 2))), # sub-array, just for fun!
...     ("data_size", "u4"),
...     #
...     # the sound data itself cannot be represented here:
...     # it does not have a fixed size
...    ])
```

See also

wavreader.py

```
>>> wav_header_dtype['format']
dtype('S4')
>>> wav_header_dtype.fields
mappingproxy({'chunk_id': (dtype('S4'), 0), 'chunk_size': (dtype('uint32'), 4),
↪'format': (dtype('S4'), 8), 'fmt_id': (dtype('S4'), 12), 'fmt_size': (dtype('uint32
↪'), 16), 'audio_fmt': (dtype('uint16'), 20), 'num_channels': (dtype('uint16'), 22),
↪'sample_rate': (dtype('uint32'), 24), 'byte_rate': (dtype('uint32'), 28), 'block_
↪align': (dtype('uint16'), 32), 'bits_per_sample': (dtype('uint16'), 34), 'data_id':␣
↪(dtype(('S1', (2, 2))), 36), 'data_size': (dtype('uint32'), 40)})
>>> wav_header_dtype.fields['format']
(dtype('S4'), 8)
```

- The first element is the sub-dtype in the structured data, corresponding to the name `format`

- The second one is its offset (in bytes) from the beginning of the item

Exercise

Mini-exercise, make a "sparse" dtype by using offsets, and only some of the fields:

```
>>> wav_header_dtype = np.dtype(dict(
...   names=['format', 'sample_rate', 'data_id'],
...   offsets=[offset_1, offset_2, offset_3], # counted from start of structure in␣
↪bytes
...   formats=list of dtypes for each of the fields,
... ))
```

and use that to read the sample rate, and `data_id` (as sub-array).

```
>>> f = open('data/test.wav', 'r')
>>> wav_header = np.fromfile(f, dtype=wav_header_dtype, count=1)
>>> f.close()
>>> print(wav_header)
[ ('RIFF', 17402L, 'WAVE', 'fmt ', 16L, 1, 1, 16000L, 32000L, 2, 16, [['d', 'a'], ['t
↪', 'a']], 17366L)]
>>> wav_header['sample_rate']
array([16000], dtype=uint32)
```

Let's try accessing the sub-array:

```
>>> wav_header['data_id']
array([[['d', 'a'],
        ['t', 'a']]],
      dtype='|S1')
>>> wav_header.shape
(1,)
>>> wav_header['data_id'].shape
(1, 2, 2)
```

When accessing sub-arrays, the dimensions get added to the end!

Note

There are existing modules such as `wavfile`, `audiolab`, etc. for loading sound data...

Casting and re-interpretation/views

casting

- on assignment
- on array construction
- on arithmetic
- etc.
- and manually: `.astype(dtype)`

data re-interpretation

- manually: `.view(dtype)`

Casting

- Casting in arithmetic, in nutshell:
  - only type (not value!) of operands matters
  - largest "safe" type able to represent both is picked
  - scalars can "lose" to arrays in some situations
- Casting in general copies data:

```
>>> x = np.array([1, 2, 3, 4], dtype=float)
>>> x
array([1.,  2.,  3.,  4.])
>>> y = x.astype(np.int8)
>>> y
array([1, 2, 3, 4], dtype=int8)
>>> y + 1
array([2, 3, 4, 5], dtype=int8)
>>> y + 256
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: Python integer 256 out of bounds for int8
>>> y + 256.0
array([257.,  258.,  259.,  260.])
>>> y + np.array([256], dtype=np.int32)
array([257, 258, 259, 260], dtype=int32)
```

- Casting on setitem: dtype of the array is not changed on item assignment:

```
>>> y[:] = y + 1.5
>>> y
array([2, 3, 4, 5], dtype=int8)
```

Note

Exact rules: see NumPy documentation

Re-interpretation / viewing

- Data block in memory (4 bytes)

0x01	0x02	0x03	0x04

  - 4 of uint8, OR,
  - 4 of int8, OR,
  - 2 of int16, OR,
  - 1 of int32, OR,
  - 1 of float32, OR,
  - ...

  How to switch from one to another?

1. Switch the dtype:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.uint8)
>>> x.dtype = "<i2"
>>> x
array([ 513, 1027], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
```

0x01	0x02	||	0x03	0x04

Note

little-endian: least significant byte is on the *left* in memory

2. Create a new view of type `uint32`, shorthand `i4`:

```
>>> y = x.view("<i4")
>>> y
array([67305985], dtype=int32)
>>> 0x04030201
67305985
```

0x01	0x02	0x03	0x04

Note

- `.view()` makes *views*, does not copy (or alter) the memory block
- only changes the dtype (and adjusts array shape):

```
>>> x[1] = 5
>>> y
array([328193], dtype=int32)
>>> y.base is x
True
```

Mini-exercise: data re-interpretation

See also

view-colors.py

You have RGBA data in an array:

```
>>> x = np.zeros((10, 10, 4), dtype=np.int8)
>>> x[:, :, 0] = 1
>>> x[:, :, 1] = 2
>>> x[:, :, 2] = 3
>>> x[:, :, 3] = 4
```

where the last three dimensions are the R, B, and G, and alpha channels.

How to make a (10, 10) structured array with field names 'r', 'g', 'b', 'a' without copying data?

```
>>> y = ...

>>> assert (y['r'] == 1).all()
>>> assert (y['g'] == 2).all()
>>> assert (y['b'] == 3).all()
>>> assert (y['a'] == 4).all()
```

Solution

```
>>> y = x.view([('r', 'i1'),
...             ('g', 'i1'),
...             ('b', 'i1'),
...             ('a', 'i1')]
...              )[:, :, 0]
```

Warning

Another two arrays, each occupying exactly 4 bytes of memory:

```
>>> x = np.array([[1, 3], [2, 4]], dtype=np.uint8)
>>> x
array([[1, 3],
       [2, 4]], dtype=uint8)
>>> y = x.transpose()
>>> y
array([[1, 2],
       [3, 4]], dtype=uint8)
```

We view the elements of `x` (1 byte each) as `int16` (2 bytes each):

```
>>> x.view(np.int16)
array([[ 769],
       [1026]], dtype=int16)
```

What is happening here? Take a look at the bytes stored in memory by `x`:

```
>>> x.tobytes()
b'\x01\x03\x02\x04'
```

The `\x` stands for heXadecimal, so what we are seeing is:

```
0x01 0x03 0x02 0x04
```

We ask NumPy to interpret these bytes as elements of dtype `int16`—each of which occupies *two* bytes in memory. Therefore, `0x01 0x03` becomes the first `uint16` and `0x02 0x04` the second.

You may then expect to see `0x0103` (259, when converting from hexadecimal to decimal) as the first result. But your computer likely stores most significant bytes first, and as such reads the number as `0x0301` or 769 (go on and type *0x0301* into your Python terminal to verify).

We can do the same on a copy of `y` (why doesn't it work on `y` directly?):

```
>>> y.copy().view(np.int16)
array([[ 513],
       [1027]], dtype=int16)
```

Can you explain these numbers, 513 and 1027, as well as the output shape of the resulting array?

8.1.4 Indexing scheme: strides

Main point

The question:

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]], dtype=np.int8)
>>> x.tobytes('A')
b'\x01\x02\x03\x04\x05\x06\x07\x08\t'

At which byte in ``x.data`` does the item ``x[1, 2]`` begin?
```

The answer (in NumPy)

- **strides**: the number of bytes to jump to find the next element
- 1 stride per dimension

```
>>> x.strides
(3, 1)
>>> byte_offset = 3 * 1 + 1 * 2  # to find x[1, 2]
>>> x.flat[byte_offset]
np.int8(6)
>>> x[1, 2]
np.int8(6)
```

simple, **flexible**

C and Fortran order

> **Note**
>
> The Python built-in `bytes` returns bytes in C-order by default which can cause confusion when trying to inspect memory layout. We use `numpy.ndarray.tobytes()` with `order=A` instead, which preserves the C or F ordering of the bytes in memory.

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6]], dtype=np.int16, order='C')
>>> x.strides
(6, 2)
>>> x.tobytes('A')
b'\x01\x00\x02\x00\x03\x00\x04\x00\x05\x00\x06\x00'
```

- Need to jump 6 bytes to find the next row
- Need to jump 2 bytes to find the next column

```
>>> y = np.array(x, order='F')
>>> y.strides
(2, 4)
>>> y.tobytes('A')
b'\x01\x00\x04\x00\x02\x00\x05\x00\x03\x00\x06\x00'
```

- Need to jump 2 bytes to find the next row
- Need to jump 4 bytes to find the next column
- Similarly to higher dimensions:

- C: last dimensions vary fastest (= smaller strides)
- F: first dimensions vary fastest

$$\text{shape} = (d_1, d_2, ..., d_n)$$
$$\text{strides} = (s_1, s_2, ..., s_n)$$
$$s_j^C = d_{j+1} d_{j+2} ... d_n \times \text{itemsize}$$
$$s_j^F = d_1 d_2 ... d_{j-1} \times \text{itemsize}$$

Note

Now we can understand the behavior of `.view()`:

```
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
>>> x = y.copy()
```

Transposition does not affect the memory layout of the data, only strides

```
>>> x.strides
(2, 1)
>>> y.strides
(1, 2)
```

```
>>> x.tobytes('A')
b'\x01\x02\x03\x04'
>>> y.tobytes('A')
b'\x01\x03\x02\x04'
```

- the results are different when interpreted as 2 of int16
- `.copy()` creates new arrays in the C order (by default)

Note

In-place operations with views

Prior to NumPy version 1.13, in-place operations with views could result in **incorrect** results for large arrays. Since version 1.13, NumPy includes checks for *memory overlap* to guarantee that results are consistent with the non in-place version (e.g. `a = a + a.T` produces the same result as `a += a.T`). Note however that this may result in the data being copied (as if using `a += a.T.copy()`), ultimately resulting in more memory being used than might otherwise be expected for in-place operations!

Slicing with integers

- *Everything* can be represented by changing only `shape`, `strides`, and possibly adjusting the `data` pointer!
- Never makes copies of the data

```
>>> x = np.array([1, 2, 3, 4, 5, 6], dtype=np.int32)
>>> y = x[::-1]
>>> y
array([6, 5, 4, 3, 2, 1], dtype=int32)
>>> y.strides
(-4,)

>>> y = x[2:]
>>> y.__array_interface__['data'][0] - x.__array_interface__['data'][0]
```

(continues on next page)

(continued from previous page)

```
8

>>> x = np.zeros((10, 10, 10), dtype=float)
>>> x.strides
(800, 80, 8)
>>> x[::2,::3,::4].strides
(1600, 240, 32)
```

- Similarly, transposes never make copies (it just swaps strides):

```
>>> x = np.zeros((10, 10, 10), dtype=float)
>>> x.strides
(800, 80, 8)
>>> x.T.strides
(8, 80, 800)
```

But: not all reshaping operations can be represented by playing with strides:

```
>>> a = np.arange(6, dtype=np.int8).reshape(3, 2)
>>> b = a.T
>>> b.strides
(1, 2)
```

So far, so good. However:

```
>>> bytes(a.data)
b'\x00\x01\x02\x03\x04\x05'
>>> b
array([[0, 2, 4],
       [1, 3, 5]], dtype=int8)
>>> c = b.reshape(3*2)
>>> c
array([0, 2, 4, 1, 3, 5], dtype=int8)
```

Here, there is no way to represent the array `c` given one stride and the block of memory for `a`. Therefore, the `reshape` operation needs to make a copy here.

Example: fake dimensions with strides

Stride manipulation

```
>>> from numpy.lib.stride_tricks import as_strided
>>> help(as_strided)
Help on function as_strided in module numpy.lib.stride_tricks:
...
```

Warning

`as_strided` does **not** check that you stay inside the memory block bounds...

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> as_strided(x, strides=(2*2, ), shape=(2, ))
array([1, 3], dtype=int16)
>>> x[::2]
array([1, 3], dtype=int16)
```

See also

stride-fakedims.py

Exercise

```
array([1, 2, 3, 4], dtype=np.int8)

-> array([[1, 2, 3, 4],
          [1, 2, 3, 4],
          [1, 2, 3, 4]], dtype=np.int8)
```

using only `as_strided`.:

```
Hint: byte_offset = stride[0]*index[0] + stride[1]*index[1] + ...
```

Spoiler

Stride can also be *0*:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int8)
>>> y = as_strided(x, strides=(0, 1), shape=(3, 4))
>>> y
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int8)
>>> y.base.base is x
True
```

Broadcasting

- Doing something useful with it: outer product of `[1, 2, 3, 4]` and `[5, 6, 7]`

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> x2 = as_strided(x, strides=(0, 1*2), shape=(3, 4))
>>> x2
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int16)
```

```
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> y2 = as_strided(y, strides=(1*2, 0), shape=(3, 4))
>>> y2
array([[5, 5, 5, 5],
       [6, 6, 6, 6],
       [7, 7, 7, 7]], dtype=int16)
```

```
>>> x2 * y2
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

... seems somehow familiar ...

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> y = np.array([5, 6, 7], dtype=np.int16)
```

(continues on next page)

(continued from previous page)

```
>>> x[np.newaxis,:] * y[:,np.newaxis]
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

- Internally, array **broadcasting** is indeed implemented using 0-strides.

More tricks: diagonals

See also

stride-diagonals.py

Challenge

- Pick diagonal entries of the matrix: (assume C memory order):

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]], dtype=np.int32)

>>> x_diag = as_strided(x, shape=(3,), strides=(???,))
```

- Pick the first super-diagonal entries `[2, 6]`.
- And the sub-diagonals?

(Hint to the last two: slicing first moves the point where striding starts from.)

Solution

Pick diagonals:

```
>>> x_diag = as_strided(x, shape=(3, ), strides=((3+1)*x.itemsize, ))
>>> x_diag
array([1, 5, 9], dtype=int32)
```

Slice first, to adjust the data pointer:

```
>>> as_strided(x[0, 1:], shape=(2, ), strides=((3+1)*x.itemsize, ))
array([2, 6], dtype=int32)

>>> as_strided(x[1:, 0], shape=(2, ), strides=((3+1)*x.itemsize, ))
array([4, 8], dtype=int32)
```

Note

Using np.diag

```
>>> y = np.diag(x, k=1)
>>> y
array([2, 6], dtype=int32)
```

However,

```
>>> y.flags.owndata
False
```

See also

stride-diagonals.py

Challenge

Compute the tensor trace:

```
>>> x = np.arange(5*5*5*5).reshape(5, 5, 5, 5)
>>> s = 0
>>> for i in range(5):
...     for j in range(5):
...         s += x[j, i, j, i]
```

by striding, and using `sum()` on the result.

```
>>> y = as_strided(x, shape=(5, 5), strides=(TODO, TODO))
>>> s2 = ...
>>> assert s == s2
```

Solution

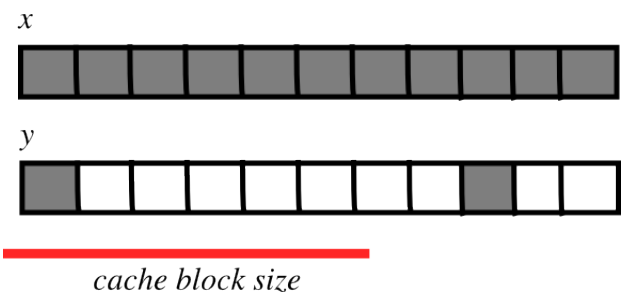
```
>>> y = as_strided(x, shape=(5, 5), strides=((5*5*5 + 5)*x.itemsize,
...                                          (5*5 + 1)*x.itemsize))
>>> s2 = y.sum()
```

CPU cache effects

Memory layout can affect performance:

```
In [1]: x = np.zeros((20000,))

In [2]: y = np.zeros((20000*67,))[::67]

In [3]: x.shape, y.shape
Out[3]: ((20000,), (20000,))

In [4]: %timeit x.sum()
5.31 us +- 8.49 ns per loop (mean +- std. dev. of 7 runs, 100,000 loops each)

In [5]: %timeit y.sum()
20.3 us +- 1.1 us per loop (mean +- std. dev. of 7 runs, 100,000 loops each)

In [6]: x.strides, y.strides
Out[6]: ((8,), (536,))
```

Smaller strides are faster?



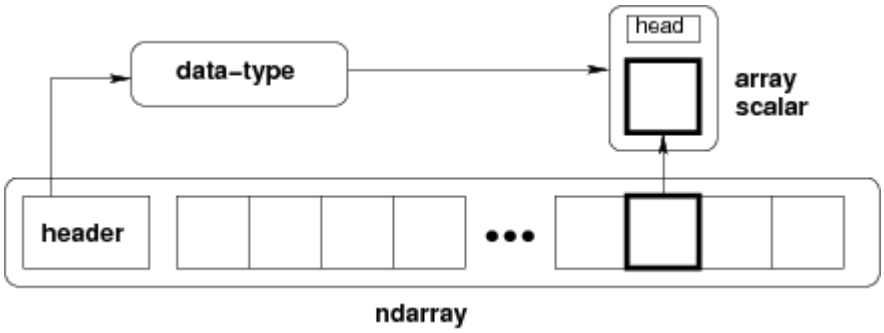
- CPU pulls data from main memory to its cache in blocks

- If many array items consecutively operated on fit in a single block (small stride):
  - ⇒ fewer transfers needed
  - ⇒ faster

See also

- numexpr is designed to mitigate cache effects when evaluating array expressions.
- numba is a compiler for Python code, that is aware of numpy arrays.

8.1.5 Findings in dissection



- *memory block*: may be shared, `.base`, `.data`
- *data type descriptor*: structured data, sub-arrays, byte order, casting, viewing, `.astype()`, `.view()`
- *strided indexing*: strides, C/F-order, slicing w/ integers, `as_strided`, broadcasting, stride tricks, `diag`, CPU cache coherence

8.2 Universal functions

8.2.1 What they are?

- Ufunc performs and elementwise operation on all elements of an array.

  Examples:

```
np.add, np.subtract, scipy.special.*, ...
```

- Automatically support: broadcasting, casting, ...
- The author of an ufunc only has to supply the elementwise operation, NumPy takes care of the rest.
- The elementwise operation needs to be implemented in C (or, e.g., Cython)

Parts of an Ufunc

1. Provided by user

```
void ufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    /*
     * int8 output = elementwise_function(int8 input_1, int8 input_2)
     *
     * This function must compute the ufunc for many values at once,
     * in the way shown below.
```

(continues on next page)

(continued from previous page)

```
     */
    char *input_1 = (char*)args[0];
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];
    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        *output = elementwise_function(*input_1, *input_2);
        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

2. The NumPy part, built by

```
char types[3]

types[0] = NPY_BYTE   /* type of first input arg */
types[1] = NPY_BYTE   /* type of second input arg */
types[2] = NPY_BYTE   /* type of third input arg */

PyObject *python_ufunc = PyUFunc_FromFuncAndData(
    ufunc_loop,
    NULL,
    types,
    1, /* ntypes */
    2, /* num_inputs */
    1, /* num_outputs */
    identity_element,
    name,
    docstring,
    unused)
```

- A ufunc can also support multiple different input-output type combinations.

Making it easier

3. `ufunc_loop` is of very generic form, and NumPy provides pre-made ones

`PyUfunc_f_f`	`float elementwise_func(float input_1)`
`PyUfunc_ff_f`	`float elementwise_func(float input_1, float input_2)`
`PyUfunc_d_d`	`double elementwise_func(double input_1)`
`PyUfunc_dd_d`	`double elementwise_func(double input_1, double input_2)`
`PyUfunc_D_D`	`elementwise_func(npy_cdouble *input, npy_cdouble* output)`
`PyUfunc_DD_D`	`elementwise_func(npy_cdouble *in1, npy_cdouble *in2, npy_cdouble* out)`

- Only `elementwise_func` needs to be supplied
- ... except when your elementwise function is not in one of the above forms

8.2.2 Exercise: building an ufunc from scratch

The Mandelbrot fractal is defined by the iteration

$$z \leftarrow z^2 + c$$

where $c = x + iy$ is a complex number. This iteration is repeated – if z stays finite no matter how long the iteration runs, c belongs to the Mandelbrot set.

- Make ufunc called `mandel(z0, c)` that computes:

  ```
  z = z0
  for k in range(iterations):
      z = z*z + c
  ```

  say, 100 iterations or until `z.real**2 + z.imag**2 > 1000`. Use it to determine which c are in the Mandelbrot set.

- Our function is a simple one, so make use of the `PyUFunc_*` helpers.
- Write it in Cython

See also

mandel.pyx, mandelplot.py

```
#
# Fix the parts marked by TODO
#

#
# Compile this file by (Cython >= 0.12 required because of the complex vars)
#
#    cython mandel.pyx
#    python setup.py build_ext -i
#
# and try it out with, in this directory,
#
#    >>> import mandel
#    >>> mandel.mandel(0, 1 + 2j)
#
#

# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    #
    # The Mandelbrot iteration
    #

    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
    #
    # - Note also that this function receives *pointers* to the data
```

(continues on next page)

(continued from previous page)

```
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    #
    # TODO: write the Mandelbrot iteration for one point here,
    #       as you would write it in Python.
    #
    #       Say, use 100 as the maximum number of iterations, and 1000
    #       as the cutoff for z.real**2 + z.imag**2.
    #

    TODO: mandelbrot iteration should go here

    # Return the answer for this point
    z_out[0] = z


# Boilerplate Cython definitions
#
# Pulls definitions from the NumPy C headers.
# -------------------------------------------

from numpy cimport import_array, import_ufunc
from numpy cimport (PyUFunc_FromFuncAndData,
                    PyUFuncGenericFunction)
from numpy cimport NPY_CDOUBLE, NP_DOUBLE, NPY_LONG

# Import all pre-defined loop functions
# you won't need all of them - keep the relevant ones

from numpy cimport (
    PyUFunc_f_f_As_d_d,
    PyUFunc_d_d,
    PyUFunc_f_f,
    PyUFunc_g_g,
    PyUFunc_F_F_As_D_D,
    PyUFunc_F_F,
    PyUFunc_D_D,
    PyUFunc_G_G,
    PyUFunc_ff_f_As_dd_d,
    PyUFunc_ff_f,
    PyUFunc_dd_d,
    PyUFunc_gg_g,
    PyUFunc_FF_F_As_DD_D,
    PyUFunc_DD_D,
    PyUFunc_FF_F,
    PyUFunc_GG_G)


# Required module initialization
# ------------------------------

import_array()
```

(continues on next page)

(continued from previous page)

```
import_ufunc()


# The actual ufunc declaration
# ----------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

#
# Reminder: some pre-made Ufunc loops:
#
# ================  ======================================================
# ``PyUfunc_f_f``   ``float elementwise_func(float input_1)``
# ``PyUfunc_ff_f``  ``float elementwise_func(float input_1, float input_2)``
# ``PyUfunc_d_d``   ``double elementwise_func(double input_1)``
# ``PyUfunc_dd_d``  ``double elementwise_func(double input_1, double input_2)``
# ``PyUfunc_D_D``   ``elementwise_func(complex_double *input, complex_double* complex_
↪double)``
# ``PyUfunc_DD_D``  ``elementwise_func(complex_double *in1, complex_double *in2,␣
↪complex_double* out)``
# ================  ======================================================
#
# The full list is above.
#
#
# Type codes:
#
# NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
# NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
# NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETIME,
# NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID
#

loop_func[0] = ... TODO: suitable PyUFunc_* ...
input_output_types[0] = ... TODO ...
... TODO: fill in rest of input_output_types ...

# This thing is passed as the ``data`` parameter for the generic
# PyUFunc_* loop, to let it know which function it should call.
elementwise_funcs[0] = <void*>mandel_single_point

# Construct the ufunc:

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    TODO, # number of input args
    TODO, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes z*z + c", # docstring
    0 # unused
```

(continues on next page)

(continued from previous page)

```
    )
```

Reminder: some pre-made Ufunc loops:

```
PyUfunc_f_f  float elementwise_func(float input_1)
PyUfunc_ff_f float elementwise_func(float input_1, float input_2)
PyUfunc_d_d  double elementwise_func(double input_1)
PyUfunc_dd_d double elementwise_func(double input_1, double input_2)
PyUfunc_D_D  elementwise_func(complex_double *input, complex_double* output)
PyUfunc_DD_D elementwise_func(complex_double *in1, complex_double *in2,
             complex_double* out)
```

Type codes:

```
NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETIME,
NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID
```

8.2.3 Solution: building an ufunc from scratch

```
# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) noexcept nogil:
    #
    # The Mandelbrot iteration
    #

    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
    #
    # - Note also that this function receives *pointers* to the data;
    #   the "traditional" solution to passing complex variables around
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    # Straightforward iteration

    for k in range(100):
        z = z*z + c
        if z.real**2 + z.imag**2 > 1000:
            break
```

(continues on next page)

(continued from previous page)

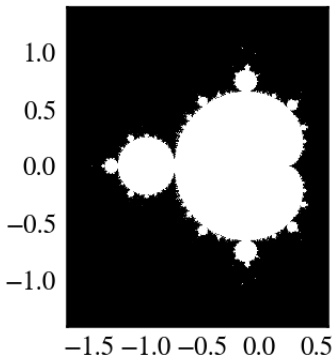
```
    # Return the answer for this point
    z_out[0] = z


# Boilerplate Cython definitions
#
# Pulls definitions from the NumPy C headers.
# -------------------------------------------

from numpy cimport import_array, import_ufunc
from numpy cimport (PyUFunc_FromFuncAndData,
                    PyUFuncGenericFunction)
from numpy cimport NPY_CDOUBLE
from numpy cimport PyUFunc_DD_D

# Required module initialization
# ------------------------------

import_array()
import_ufunc()


# The actual ufunc declaration
# ----------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

loop_func[0] = PyUFunc_DD_D

input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE

elementwise_funcs[0] = <void*>mandel_single_point

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

```
"""
Plot Mandelbrot
===============

Plot the Mandelbrot ensemble.
```

(continues on next page)

(continued from previous page)

```
"""

import numpy as np
import mandel

x = np.linspace(-1.7, 0.6, 1000)
y = np.linspace(-1.4, 1.4, 1000)
c = x[None, :] + 1j * y[:, None]
z = mandel.mandel(c, c)

import matplotlib.pyplot as plt

plt.imshow(abs(z) ** 2 < 1000, extent=[-1.7, 0.6, -1.4, 1.4])
plt.gray()
plt.show()
```



Note

Most of the boilerplate could be automated by these Cython modules:

https://github.com/cython/cython/wiki/MarkLodato-CreatingUfuncs

Several accepted input types

E.g. supporting both single- and double-precision versions

```
cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
   ...

cdef void mandel_single_point_singleprec(float complex *z_in,
                                         float complex *c_in,
                                         float complex *z_out) nogil:
   ...

cdef PyUFuncGenericFunction loop_funcs[2]
cdef char input_output_types[3*2]
cdef void *elementwise_funcs[1*2]
```

(continues on next page)

(continued from previous page)

```
loop_funcs[0] = PyUFunc_DD_D
input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE
elementwise_funcs[0] = <void*>mandel_single_point

loop_funcs[1] = PyUFunc_FF_F
input_output_types[3] = NPY_CFLOAT
input_output_types[4] = NPY_CFLOAT
input_output_types[5] = NPY_CFLOAT
elementwise_funcs[1] = <void*>mandel_single_point_singleprec

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    2, # number of supported input types   <----------------
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

8.2.4 Generalized ufuncs

ufunc

`output = elementwise_function(input)`

Both `output` and `input` can be a single array element only.

generalized ufunc

`output` and `input` can be arrays with a fixed number of dimensions

For example, matrix trace (sum of diag elements):

```
input shape = (n, n)
output shape = ()      i.e.  scalar

(n, n) -> ()
```

Matrix product:

```
input_1 shape = (m, n)
input_2 shape = (n, p)
output shape  = (m, p)

(m, n), (n, p) -> (m, p)
```

- This is called the *"signature"* of the generalized ufunc
- The dimensions on which the g-ufunc acts, are *"core dimensions"*

Status in NumPy

- g-ufuncs are in NumPy already ...
- new ones can be created with `PyUFunc_FromFuncAndDataAndSignature`
- most linear-algebra functions are implemented as g-ufuncs to enable working with stacked arrays:

```
>>> import numpy as np
>>> rng = np.random.default_rng(27446968)
>>> np.linalg.det(rng.random((3, 5, 5)))
array([ 0.01829761, -0.0077266 , -0.05336566])
>>> np.linalg._umath_linalg.det.signature
'(m,m)->()'
```

- matrix multiplication this way could be useful for operating on many small matrices at once
- Also see `tensordot` and `einsum`

Generalized ufunc loop

Matrix multiplication `(m,n),(n,p) -> (m,p)`

```
void gufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    char *input_1 = (char*)args[0];  /* these are as previously */
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];

    int input_1_stride_m = steps[3];  /* strides for the core dimensions */
    int input_1_stride_n = steps[4];  /* are added after the non-core */
    int input_2_strides_n = steps[5]; /* steps */
    int input_2_strides_p = steps[6];
    int output_strides_n = steps[7];
    int output_strides_p = steps[8];

    int m = dimension[1]; /* core dimensions are added after */
    int n = dimension[2]; /* the main dimension; order as in */
    int p = dimension[3]; /* signature */

    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        matmul_for_strided_matrices(input_1, input_2, output,
                                    strides for each array...);

        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

8.3 Interoperability features

8.3.1 Sharing multidimensional, typed data

Suppose you

1. Write a library than handles (multidimensional) binary data,
2. Want to make it easy to manipulate the data with NumPy, or whatever other library,

3. ... but would **not** like to have NumPy as a dependency.

Currently, 3 solutions:

1. the "old" buffer interface
2. the array interface
3. the "new" buffer interface (**PEP 3118**)

8.3.2 The old buffer protocol

- Only 1-D buffers
- No data type information
- C-level interface; `PyBufferProcs tp_as_buffer` in the type object
- But it's integrated into Python (e.g. strings support it)

Mini-exercise using Pillow (Python Imaging Library):

See also

pilbuffer.py

```
>>> from PIL import Image
>>> data = np.zeros((200, 200, 4), dtype=np.uint8)
>>> data[:, :] = [255, 0, 0, 255] # Red
>>> # In PIL, RGBA images consist of 32-bit integers whose bytes are [RR,GG,BB,AA]
>>> data = data.view(np.int32).squeeze()
>>> img = Image.frombuffer("RGBA", (200, 200), data, "raw", "RGBA", 0, 1)
>>> img.save('test.png')
```

Q:

Check what happens if `data` is now modified, and `img` saved again.

8.3.3 The old buffer protocol

```
"""
From buffer
============

Show how to exchange data between numpy and a library that only knows
the buffer interface.
"""

import numpy as np
from PIL import Image

# Let's make a sample image, RGBA format

x = np.zeros((200, 200, 4), dtype=np.uint8)

x[:, :, 0] = 255  # red
x[:, :, 3] = 255  # opaque

data = x.view(np.int32)  # Check that you understand why this is OK!
```

(continues on next page)

(continued from previous page)

```
img = Image.frombuffer("RGBA", (200, 200), data)
img.save("test.png")

# Modify the original data, and save again.

x[:, :, 1] = 255
img.save("test2.png")
```

8.3.4 Array interface protocol

- Multidimensional buffers
- Data type information present
- NumPy-specific approach; slowly deprecated (but not going away)
- Not integrated in Python otherwise

See also

Documentation: https://numpy.org/doc/stable/reference/arrays.interface.html

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x.__array_interface__
{'data': (171694552, False),      # memory address of data, is readonly?
 'descr': [('', '<i4')],          # data type descriptor
 'typestr': '<i4',                # same, in another form
 'strides': None,                 # strides; or None if in C-order
 'shape': (2, 2),
```

(continues on next page)

(continued from previous page)

```
 'version': 3,
}
```

::

```
>>> from PIL import Image
>>> img = Image.open('data/test.png')
>>> img.__array_interface__
{'version': 3,
 'data': ...,
 'shape': (200, 200, 4),
 'typestr': '|u1'}
>>> x = np.asarray(img)
>>> x.shape
(200, 200, 4)
```

Note

A more C-friendly variant of the array interface is also defined.

8.4 Array siblings: `chararray`, `maskedarray`

8.4.1 `chararray`: vectorized string operations

```
>>> x = np.char.asarray(['a', '  bbb', '  ccc'])
>>> x
chararray(['a', '  bbb', '  ccc'], dtype='<U5')
>>> x.upper()
chararray(['A', '  BBB', '  CCC'], dtype='<U5')
```

8.4.2 `masked_array` missing data

Masked arrays are arrays that may have missing or invalid entries.

For example, suppose we have an array where the fourth entry is invalid:

```
>>> x = np.array([1, 2, 3, -99, 5])
```

One way to describe this is to create a masked array:

```
>>> mx = np.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx
masked_array(data=[1, 2, 3, --, 5],
             mask=[False, False, False,  True, False],
       fill_value=999999)
```

Masked mean ignores masked data:

```
>>> mx.mean()
np.float64(2.75)
>>> np.mean(mx)
np.float64(2.75)
```

Warning

Not all NumPy functions respect masks, for instance `np.dot`, so check the return types.

The `masked_array` returns a **view** to the original array:

```
>>> mx[1] = 9
>>> x
array([  1,   9,   3, -99,   5])
```

The mask

You can modify the mask by assigning:

```
>>> mx[1] = np.ma.masked
>>> mx
masked_array(data=[1, --, 3, --, 5],
             mask=[False,  True, False,  True, False],
       fill_value=999999)
```

The mask is cleared on assignment:

```
>>> mx[1] = 9
>>> mx
masked_array(data=[1, 9, 3, --, 5],
             mask=[False, False, False,  True, False],
       fill_value=999999)
```

The mask is also available directly:

```
>>> mx.mask
array([False, False, False,  True, False])
```

The masked entries can be filled with a given value to get an usual array back:

```
>>> x2 = mx.filled(-1)
>>> x2
array([ 1,  9,  3, -1,  5])
```

The mask can also be cleared:

```
>>> mx.mask = np.ma.nomask
>>> mx
masked_array(data=[1, 9, 3, -99, 5],
             mask=[False, False, False, False, False],
       fill_value=999999)
```

Domain-aware functions

The masked array package also contains domain-aware functions:

```
>>> np.ma.log(np.array([1, 2, -1, -2, 3, -5]))
masked_array(data=[0.0, 0.693147180559..., --, --, 1.098612288668..., --],
             mask=[False, False,  True,  True, False,  True],
       fill_value=1e+20)
```

Note

Streamlined and more seamless support for dealing with missing data in arrays is making its way into NumPy 1.7. Stay tuned!

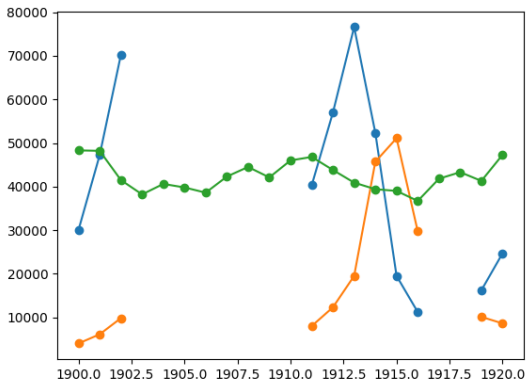
Example: Masked statistics

Canadian rangers were distracted when counting hares and lynxes in 1903-1910 and 1917-1918, and got the numbers are wrong. (Carrot farmers stayed alert, though.) Compute the mean populations over time, ignoring the invalid numbers.

```
>>> data = np.loadtxt('data/populations.txt')
>>> populations = np.ma.masked_array(data[:,1:])
>>> year = data[:, 0]

>>> bad_years = (((year >= 1903) & (year <= 1910))
...              | ((year >= 1917) & (year <= 1918)))
>>> # '&' means 'and' and '|' means 'or'
>>> populations[bad_years, 0] = np.ma.masked
>>> populations[bad_years, 1] = np.ma.masked

>>> populations.mean(axis=0)
masked_array(data=[40472.72727272727, 18627.272727272728, 42400.0],
             mask=[False, False, False],
       fill_value=1e+20)

>>> populations.std(axis=0)
masked_array(data=[21087.656489006717, 15625.799814240254, 3322.5062255844787],
             mask=[False, False, False],
       fill_value=1e+20)
```

Note that Matplotlib knows about masked arrays:

```
>>> plt.plot(year, populations, 'o-')
[<matplotlib.lines.Line2D object at ...>, ...]
```



8.4.3 `recarray`: purely convenience

```
>>> arr = np.array([('a', 1), ('b', 2)], dtype=[('x', 'S1'), ('y', int)])
>>> arr2 = arr.view(np.recarray)
>>> arr2.x
```

(continues on next page)

(continued from previous page)

```
array([b'a', b'b'], dtype='|S1')
>>> arr2.y
array([1, 2])
```

8.5 Summary

- Anatomy of the ndarray: data, dtype, strides.
- Universal functions: elementwise operations, how to make new ones
- Ndarray subclasses
- Various buffer interfaces for integration with other tools
- Recent additions: PEP 3118, generalized ufuncs

8.6 Contributing to NumPy/SciPy

Get this tutorial: https://www.euroscipy.org/talk/882

8.6.1 Why

- "There's a bug?"
- "I don't understand what this is supposed to do?"
- "I have this fancy code. Would you like to have it?"
- "I'd like to help! What can I do?"

8.6.2 Reporting bugs

- Bug tracker (prefer **this**)
  - https://github.com/numpy/numpy/issues
  - https://github.com/scipy/scipy/issues
  - Click the "Sign up" link to get an account
- Mailing lists (https://numpy.org/community/)
  - If you're unsure
  - No replies in a week or so? Just file a bug ticket.

Good bug report

```
Title: numpy.random.permutations fails for non-integer arguments

I'm trying to generate random permutations, using numpy.random.permutations

When calling numpy.random.permutation with non-integer arguments
it fails with a cryptic error message::

    >>> rng.permutation(12)
    array([ 2,  6,  4,  1,  8, 11, 10,  5,  9,  3,  7,  0])
    >>> rng.permutation(12.) #doctest: +SKIP
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "_generator.pyx", line 4844, in numpy.random._generator.Generator.permutation
```

(continues on next page)

(continued from previous page)

```
    numpy.exceptions.AxisError: axis 0 is out of bounds for array of dimension 0

This also happens with long arguments, and so
np.random.permutation(X.shape[0]) where X is an array fails on 64
bit windows (where shape is a tuple of longs).

It would be great if it could cast to integer or at least raise a
proper error for non-integer types.

I'm using NumPy 1.4.1, built from the official tarball, on Windows
64 with Visual studio 2008, on Python.org 64-bit Python.
```

0. What are you trying to do?
1. **Small code snippet reproducing the bug** (if possible)
   - What actually happens
   - What you'd expect
2. Platform (Windows / Linux / OSX, 32/64 bits, x86/PPC, ...)
3. Version of NumPy/SciPy

```
>>> print(np.__version__)
2...
```

Check that the following is what you expect

```
>>> print(np.__file__)
/...
```

In case you have old/broken NumPy installations lying around.

If unsure, try to remove existing NumPy installations, and reinstall...

8.6.3 Contributing to documentation

1. Documentation editor
   - https://numpy.org/doc/stable/
   - Registration
     - Register an account
     - Subscribe to `scipy-dev` mailing list (subscribers-only)
     - Problem with mailing lists: you get mail
       * But: **you can turn mail delivery off**
       * "change your subscription options", at the bottom of https://mail.python.org/mailman3/lists/scipy-dev.python.org/
     - Send a mail @ `scipy-dev` mailing list; ask for activation:

```
To: scipy-dev@scipy.org

Hi,

I'd like to edit NumPy/SciPy docstrings. My account is XXXXX
```

(continues on next page)

(continued from previous page)

```
Cheers,
N. N.
```

- Check the style guide:
  - https://numpy.org/doc/stable/
  - Don't be intimidated; to fix a small thing, just fix it
- Edit

2. Edit sources and send patches (as for bugs)
3. Complain on the mailing list

8.6.4 Contributing features

The contribution of features is documented on https://numpy.org/doc/stable/dev/

8.6.5 How to help, in general

- Bug fixes always welcome!
  - What irks you most
  - Browse the tracker
- Documentation work
  - API docs: improvements to docstrings
    - Know some SciPy module well?
  - *User guide*
    - https://numpy.org/doc/stable/user/
- Ask on communication channels:
  - `numpy-discussion` list
  - `scipy-dev` list

CHAPTER 9

Debugging code

Author: *Gaël Varoquaux*

This section explores tools to understand better your code base: debugging, to find and fix bugs.

It is not specific to the scientific Python community, but the strategies that we will employ are tailored to its needs.

Prerequisites

- NumPy
- IPython
- nosetests
- pyflakes
- gdb for the C-debugging part.

Chapter contents

- *Avoiding bugs*
  - *Coding best practices to avoid getting in trouble*
  - *pyflakes: fast static analysis*
- *Debugging workflow*
- *Using the Python debugger*
  - *Invoking the debugger*

  - *Debugger commands and interaction*
- *Debugging segmentation faults using gdb*

9.1 Avoiding bugs

9.1.1 Coding best practices to avoid getting in trouble

Brian Kernighan

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

- We all write buggy code. Accept it. Deal with it.
- Write your code with testing and debugging in mind.
- Keep It Simple, Stupid (KISS).
  - What is the simplest thing that could possibly work?
- Don't Repeat Yourself (DRY).
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
  - Constants, algorithms, etc...
- Try to limit interdependencies of your code. (Loose Coupling)
- Give your variables, functions and modules meaningful names (not mathematics names)

9.1.2 pyflakes: fast static analysis

They are several static analysis tools in Python; to name a few:

- pylint
- pychecker
- pyflakes
- flake8

Here we focus on *pyflakes*, which is the simplest tool.

- **Fast, simple**
- Detects syntax errors, missing imports, typos on names.

Another good recommendation is the *flake8* tool which is a combination of pyflakes and pep8. Thus, in addition to the types of errors that pyflakes catches, flake8 detects violations of the recommendation in PEP8 style guide.

Integrating pyflakes (or flake8) in your editor or IDE is highly recommended, it **does yield productivity gains**.

Running pyflakes on the current edited file

You can bind a key to run pyflakes in the current buffer.

- **In kate** Menu: 'settings -> configure kate
  - In plugins enable 'external tools'

  - In external Tools', add *pyflakes*:

```
kdialog --title "pyflakes %filename" --msgbox "$(pyflakes %filename)"
```

- **In TextMate**

  Menu: TextMate -> Preferences -> Advanced -> Shell variables, add a shell variable:

```
TM_PYCHECKER = /Library/Frameworks/Python.framework/Versions/Current/bin/pyflakes
```

  Then *Ctrl-Shift-V* is binded to a pyflakes report

- **In vim** In your *.vimrc* (binds F5 to *pyflakes*):

```
autocmd FileType python let &mp = 'echo "*** running % ***" ; pyflakes %'
autocmd FileType tex,mp,rst,python imap <Esc>[15~ <C-O>:make!^M
autocmd FileType tex,mp,rst,python map  <Esc>[15~ :make!^M
autocmd FileType tex,mp,rst,python set autowrite
```

- **In emacs** In your *.emacs* (binds F5 to *pyflakes*):

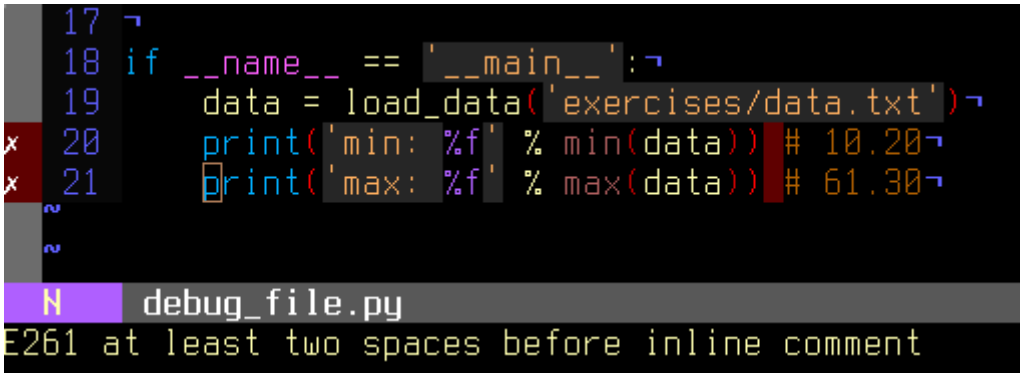
```
(defun pyflakes-thisfile () (interactive)
       (compile (format "pyflakes %s" (buffer-file-name)))
)

(define-minor-mode pyflakes-mode
    "Toggle pyflakes mode.
    With no argument, this command toggles the mode.
    Non-null prefix argument turns on the mode.
    Null prefix argument turns off the mode."
    ;; The initial value.
    nil
    ;; The indicator for the mode line.
    " Pyflakes"
    ;; The minor mode bindings.
    '( ([f5] . pyflakes-thisfile) )
)

(add-hook 'python-mode-hook (lambda () (pyflakes-mode t)))
```

A type-as-go spell-checker like integration

- **In vim**
  - Use the pyflakes.vim plugin:
    1. download the zip file from https://www.vim.org/scripts/script.php?script_id=2441
    2. extract the files in `~/.vim/ftplugin/python`
    3. make sure your vimrc has `filetype plugin indent on`

```
869
870     def _compute_log_likelihood(obs):
871         return self._log_emissionprob[:, obs].T
872
```

  - Alternatively: use the syntastic plugin. This can be configured to use `flake8` too and also handles on-the-fly checking for many other languages.



- **In emacs**

  Use the flymake mode with pyflakes, documented on https://www.emacswiki.org/emacs/FlyMake and included in Emacs 26 and more recent. To activate it, use `M-x` (meta-key then x) and enter *flymake-mode* at the prompt. To enable it automatically when opening a Python file, add the following line to your .emacs file:

```
(add-hook 'python-mode-hook '(lambda () (flymake-mode)))
```

9.2 Debugging workflow

If you do have a non trivial bug, this is when debugging strategies kick in. There is no silver bullet. Yet, strategies help:

> **For debugging a given problem, the favorable situation is when the problem is isolated in a small number of lines of code, outside framework or application code, with short modify-run-fail cycles**

1. Make it fail reliably. Find a test case that makes the code fail every time.
2. Divide and Conquer. Once you have a failing test case, isolate the failing code.
   - Which module.
   - Which function.
   - Which line of code.

   => isolate a small reproducible failure: a test case
3. Change one thing at a time and re-run the failing test case.
4. Use the debugger to understand what is going wrong.
5. Take notes and be patient. It may take a while.

> **Note**
>
> Once you have gone through this process: isolated a tight piece of code reproducing the bug and fix the bug using this piece of code, add the corresponding code to your test suite.

9.3 Using the Python debugger

The python debugger, `pdb`: https://docs.python.org/3/library/pdb.html, allows you to inspect your code interactively.

Specifically it allows you to:

- View the source code.

- Walk up and down the call stack.
- Inspect values of variables.
- Modify values of variables.
- Set breakpoints.

print

Yes, `print` statements do work as a debugging tool. However to inspect runtime, it is often more efficient to use the debugger.

9.3.1 Invoking the debugger

Ways to launch the debugger:

1. Postmortem, launch debugger after module errors.
2. Launch the module with the debugger.
3. Call the debugger inside the module

Postmortem

Situation: You're working in IPython and you get a traceback.

Here we debug the file `index_error.py`. When running it, an `IndexError` is raised. Type `%debug` and drop into the debugger.

```
In [1]: %run index_error.py
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
File ~/src/scientific-python-lectures/advanced/debugging/index_error.py:10
      6     print(lst[len(lst)])
      9 if __name__ == "__main__":
---> 10     index_error()

File ~/src/scientific-python-lectures/advanced/debugging/index_error.py:6, in index_
↪error()
      4 def index_error():
      5     lst = list("foobar")
----> 6     print(lst[len(lst)])

IndexError: list index out of range

In [2]: %debug
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/index_error.
↪py(6)index_error()
      4 def index_error():
      5     lst = list("foobar")
----> 6     print(lst[len(lst)])
      7
      8

ipdb> list
      1 """Small snippet to raise an IndexError."""
      2
      3
      4 def index_error():
```

(continues on next page)

(continued from previous page)

```
      5     lst = list("foobar")
----> 6     print(lst[len(lst)])
      7
      8
      9 if __name__ == "__main__":
     10     index_error()

ipdb> len(lst)
6
ipdb> print(lst[len(lst) - 1])
r
ipdb> quit
```

Post-mortem debugging without IPython

In some situations you cannot use IPython, for instance to debug a script that wants to be called from the command line. In this case, you can call the script with `python -m pdb script.py`:

```
$ python -m pdb index_error.py
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/index_error.py(1)
↪<module>()
-> """Small snippet to raise an IndexError."""
(Pdb) continue
Traceback (most recent call last):
  File "/usr/lib64/python3.11/pdb.py", line 1793, in main
    pdb._run(target)
  File "/usr/lib64/python3.11/pdb.py", line 1659, in _run
    self.run(target.code)
  File "/usr/lib64/python3.11/bdb.py", line 600, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "/home/jarrod/src/scientific-python-lectures/advanced/debugging/index_error.
↪py", line 10, in <module>
    index_error()
  File "/home/jarrod/src/scientific-python-lectures/advanced/debugging/index_error.
↪py", line 6, in index_error
    print(lst[len(lst)])
          ~~~^^^^^^^^^^
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/index_error.
↪py(6)index_error()
-> print(lst[len(lst)])
(Pdb)
```

Step-by-step execution

Situation: You believe a bug exists in a module but are not sure where.

For instance we are trying to debug `wiener_filtering.py`. Indeed the code runs, but the filtering does not work well.

- Run the script in IPython with the debugger using `%run -d wiener_filtering.py` :

```
In [1]: %run -d wiener_filtering.py
*** Blank or comment
*** Blank or comment
*** Blank or comment
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_
↪filtering.py(1)<module>()
----> 1 """Wiener filtering a noisy raccoon face: this module is buggy"""
      2
      3 import numpy as np
      4 import scipy as sp
      5 import matplotlib.pyplot as plt
```

- Set a break point at line 29 using `b 29`:

```
ipdb> n
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_
↪filtering.py(3)<module>()
      1 """Wiener filtering a noisy raccoon face: this module is buggy"""
      2
----> 3 import numpy as np
      4 import scipy as sp
      5 import matplotlib.pyplot as plt

ipdb> b 29
Breakpoint 1 at /home/jarrod/src/scientific-python-lectures/advanced/debugging/
↪wiener_filtering.py:29
```

- Continue execution to next breakpoint with `c(ont(inue))`:

```
ipdb> c
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_
↪filtering.py(29)iterated_wiener()
     27     Do not use this: this is crappy code to demo bugs!
     28     """
1--> 29     noisy_img = noisy_img
     30     denoised_img = local_mean(noisy_img, size=size)
     31     l_var = local_var(noisy_img, size=size)
```

- Step into code with `n(ext)` and `s(tep)`: `next` jumps to the next statement in the current execution context, while `step` will go across execution contexts, i.e. enable exploring inside function calls:

```
ipdb> s
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_
↪filtering.py(30)iterated_wiener()
     28     """
1    29     noisy_img = noisy_img
---> 30     denoised_img = local_mean(noisy_img, size=size)
     31     l_var = local_var(noisy_img, size=size)
     32     for i in range(3):

ipdb> n
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_
↪filtering.py(31)iterated_wiener()
1    29     noisy_img = noisy_img
     30     denoised_img = local_mean(noisy_img, size=size)
---> 31     l_var = local_var(noisy_img, size=size)
```

(continues on next page)

(continued from previous page)

```
     32     for i in range(3):
     33         res = noisy_img - denoised_img
```

- Step a few lines and explore the local variables:

```
ipdb> n
> /home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_
↪filtering.py(32)iterated_wiener()
     30     denoised_img = local_mean(noisy_img, size=size)
     31     l_var = local_var(noisy_img, size=size)
---> 32     for i in range(3):
     33         res = noisy_img - denoised_img
     34         noise = (res**2).sum() / res.size

ipdb> print(l_var)
[[2571 2782 3474 ... 3008 2922 3141]
 [2105  708  475 ...  469  354 2884]
 [1697  420  645 ...  273  236 2517]
 ...
 [2437  345  432 ...  413  387 4188]
 [2598  179  247 ...  367  441 3909]
 [2808 2525 3117 ... 4413 4454 4385]]
ipdb> print(l_var.min())
0
```

Oh dear, nothing but integers, and 0 variation. Here is our bug, we are doing integer arithmetic.

Raising exception on numerical errors

When we run the `wiener_filtering.py` file, the following warnings are raised:

```
In [2]: %run wiener_filtering.py
/home/jarrod/src/scientific-python-lectures/advanced/debugging/wiener_filtering.
↪py:35: RuntimeWarning: divide by zero encountered in divide
  noise_level = 1 - noise / l_var
```

We can turn these warnings in exception, which enables us to do post-mortem debugging on them, and find our problem more quickly:

```
In [3]: np.seterr(all='raise')
Out[3]: {'divide': 'warn', 'over': 'warn', 'under': 'ignore', 'invalid': 'warn'}

In [4]:  %run wiener_filtering.py
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
File ~/src/scientific-python-lectures/advanced/debugging/wiener_filtering.py:52
     49 plt.matshow(face[cut], cmap=plt.cm.gray)
     50 plt.matshow(noisy_face[cut], cmap=plt.cm.gray)
---> 52 denoised_face = iterated_wiener(noisy_face)
     53 plt.matshow(denoised_face[cut], cmap=plt.cm.gray)
     55 plt.show()

File ~/src/scientific-python-lectures/advanced/debugging/wiener_filtering.py:35,␣
↪in iterated_wiener(noisy_img, size)
     33 res = noisy_img - denoised_img
     34 noise = (res**2).sum() / res.size
---> 35 noise_level = 1 - noise / l_var
     36 noise_level[noise_level < 0] = 0
     37 denoised_img = np.int64(noise_level * res)

FloatingPointError: divide by zero encountered in divide
```

Other ways of starting a debugger

- **Raising an exception as a poor man break point**

  If you find it tedious to note the line number to set a break point, you can simply raise an exception at the point that you want to inspect and use IPython's `%debug`. Note that in this case you cannot step or continue the execution.

- **Debugging test failures using nosetests**

  You can run `nosetests --pdb` to drop in post-mortem debugging on exceptions, and `nosetests --pdb-failure` to inspect test failures using the debugger.

  In addition, you can use the IPython interface for the debugger in nose by installing the nose plugin ipdbplugin. You can than pass `--ipdb` and `--ipdb-failure` options to nosetests.

- **Calling the debugger explicitly**

  Insert the following line where you want to drop in the debugger:

```
import pdb; pdb.set_trace()
```

Warning

When running `nosetests`, the output is captured, and thus it seems that the debugger does not work. Simply run the nosetests with the `-s` flag.

Graphical debuggers and alternatives

- pudb is a good semi-graphical debugger with a text user interface in the console.
- The Visual Studio Code integrated development environment includes a debugging mode.
- The Mu editor is a simple Python editor that includes a debugging mode.

9.3.2 Debugger commands and interaction

`l(list)`	Lists the code at the current position
`u(p)`	Walk up the call stack
`d(own)`	Walk down the call stack
`n(ext)`	Execute the next line (does not go down in new functions)
`s(tep)`	Execute the next statement (goes down in new functions)
`bt`	Print the call stack
`a`	Print the local variables
`!command`	Execute the given **Python** command (by opposition to pdb commands

Warning

Debugger commands are not Python code

You cannot name the variables the way you want. For instance, if in you cannot override the variables in the current frame with the same name: **use different names than your local variable when typing code in the debugger**.

Getting help when in the debugger

Type `h` or `help` to access the interactive help:

```
ipdb> help

Documented commands (type help <topic>):
========================================
EOF    commands   enable      ll        pp       s                until
a      condition  exceptions  longlist  psource  skip_hidden      up
alias  cont       exit        n         q        skip_predicates  w
args   context    h           next      quit     source           whatis
b      continue   help        p         r        step             where
break  d          ignore      pdef      restart  tbreak
bt     debug      j           pdoc      return   u
c      disable    jump        pfile     retval   unalias
cl     display    l           pinfo     run      undisplay
clear  down       list        pinfo2    rv       unt

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
interact
```

9.4 Debugging segmentation faults using gdb

If you have a segmentation fault, you cannot debug it with pdb, as it crashes the Python interpreter before it can drop in the debugger. Similarly, if you have a bug in C code embedded in Python, pdb is useless. For this we turn to the gnu debugger, gdb, available on Linux.

Before we start with gdb, let us add a few Python-specific tools to it. For this we add a few macros to our `~/.gdbinit`. The optimal choice of macro depends on your Python version and your gdb version. I have added a simplified version in `gdbinit`, but feel free to read DebuggingWithGdb.

To debug with gdb the Python script `segfault.py`, we can run the script in gdb as follows

```
$ gdb python
...
(gdb) run segfault.py
Starting program: /usr/bin/python segfault.py
[Thread debugging using libthread_db enabled]

Program received signal SIGSEGV, Segmentation fault.
_strided_byte_copy (dst=0x8537478 "\360\343G", outstrides=4, src=
    0x86c0690 <Address 0x86c0690 out of bounds>, instrides=32, N=3,
    elsize=4)
        at numpy/core/src/multiarray/ctors.c:365
365             _FAST_MOVE(Int32);
(gdb)
```

We get a segfault, and gdb captures it for post-mortem debugging in the C level stack (not the Python call stack). We can debug the C call stack using gdb's commands:

```
(gdb) up
#1  0x004af4f5 in _copy_from_same_shape (dest=<value optimized out>,
```

(continues on next page)

(continued from previous page)

```
    src=<value optimized out>, myfunc=0x496780 <_strided_byte_copy>,
    swap=0)
at numpy/core/src/multiarray/ctors.c:748
748         myfunc(dit->dataptr, dest->strides[maxaxis],
```

As you can see, right now, we are in the C code of numpy. We would like to know what is the Python code that triggers this segfault, so we go up the stack until we hit the Python execution loop:

```
(gdb) up
#8  0x080ddd23 in call_function (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/
↪core/arrayprint.py, line 156, in _leading_trailing (a=<numpy.ndarray at remote␣
↪0x85371b0>, _nc=<module at remote 0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:3750
3750    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c

(gdb) up
#9  PyEval_EvalFrameEx (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/
↪core/arrayprint.py, line 156, in _leading_trailing (a=<numpy.ndarray at remote␣
↪0x85371b0>, _nc=<module at remote 0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:2412
2412    in ../Python/ceval.c
(gdb)
```

Once we are in the Python execution loop, we can use our special Python helper function. For instance we can find the corresponding Python code:

```
(gdb) pyframe
/home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.py (158): _
↪leading_trailing
(gdb)
```

This is numpy code, we need to go up until we find code that we have written:

```
(gdb) up
...
(gdb) up
#34 0x080dc97a in PyEval_EvalFrameEx (f=
    Frame 0x82f064c, for file segfault.py, line 11, in print_big_array (small_array=
↪<numpy.ndarray at remote 0x853ecf0>, big_array=<numpy.ndarray at remote 0x853ed20>),
↪ throwflag=0) at ../Python/ceval.c:1630
1630    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c
(gdb) pyframe
segfault.py (12): print_big_array
```

The corresponding code is:

```
def make_big_array(small_array):
    big_array = stride_tricks.as_strided(
        small_array, shape=(int(2e6), int(2e6)), strides=(32, 32)
    )
    return big_array
```

Thus the segfault happens when printing `big_array[-10:]`. The reason is simply that `big_array` has been allocated with its end outside the program memory.

Note

For a list of Python-specific commands defined in the *gdbinit*, read the source of this file.

Wrap up exercise

The following script is well documented and hopefully legible. It seeks to answer a problem of actual interest for numerical computing, but it does not work... Can you debug it?

Python source code: `to_debug.py`

CHAPTER 10

Optimizing code

Donald Knuth

"Premature optimization is the root of all evil"

Author: *Gaël Varoquaux*

This chapter deals with strategies to make Python code go faster.

Prerequisites

- line_profiler

Chapters contents

- *Optimization workflow*
- *Profiling Python code*
  - *Timeit*
  - *Profiler*
  - *Line-profiler*
- *Making code go faster*
  - *Algorithmic optimization*
    - *Example of the SVD*

- *Writing faster numerical code*
  - *Additional Links*

10.1 Optimization workflow

1. Make it work: write the code in a simple **legible** ways.
2. Make it work reliably: write automated test cases, make really sure that your algorithm is right and that if you break it, the tests will capture the breakage.
3. Optimize the code by profiling simple use-cases to find the bottlenecks and speeding up these bottleneck, finding a better algorithm or implementation. Keep in mind that a trade off should be found between profiling on a realistic example and the simplicity and speed of execution of the code. For efficient work, it is best to work with profiling runs lasting around 10s.

10.2 Profiling Python code

No optimization without measuring!

- **Measure:** profiling, timing
- You'll have surprises: the fastest code is not always what you think

10.2.1 Timeit

In IPython, use `timeit` (https://docs.python.org/3/library/timeit.html) to time elementary operations:

```
In [1]: import numpy as np

In [2]: a = np.arange(1000)

In [3]: %timeit a ** 2
944 ns +- 3.08 ns per loop (mean +- std. dev. of 7 runs, 1,000,000 loops each)

In [4]: %timeit a ** 2.1
16 us +- 103 ns per loop (mean +- std. dev. of 7 runs, 100,000 loops each)

In [5]: %timeit a * a
1.06 us +- 4.72 ns per loop (mean +- std. dev. of 7 runs, 1,000,000 loops each)
```

Use this to guide your choice between strategies.

Note

For long running calls, using `%time` instead of `%timeit`; it is less precise but faster

10.2.2 Profiler

Useful when you have a large program to profile, for example the `following file`:

```
# For this example to run, you also need the 'ica.py' file

import numpy as np
```

(continues on next page)

(continued from previous page)

```
import scipy as sp

from ica import fastica


# @profile  # uncomment this line to run with line_profiler
def test():
    rng = np.random.default_rng()
    data = rng.random((5000, 100))
    u, s, v = sp.linalg.svd(data)
    pca = u[:, :10].T @ data
    results = fastica(pca.T, whiten=False)


if __name__ == "__main__":
    test()
```

Note

This is a combination of two unsupervised learning techniques, principal component analysis (PCA) and independent component analysis (ICA). PCA is a technique for dimensionality reduction, i.e. an algorithm to explain the observed variance in your data using less dimensions. ICA is a source separation technique, for example to unmix multiple signals that have been recorded through multiple sensors. Doing a PCA first and then an ICA can be useful if you have more sensors than signals. For more information see: the FastICA example from scikits-learn.

To run it, you also need to download the `ica module`. In IPython we can time the script:

```
In [6]: %run -t demo.py
IPython CPU timings (estimated):
    User   :    14.3929 s.
    System:   0.256016 s.
```

and profile it:

```
In [7]: %run -p demo.py
      916 function calls in 14.551 CPU seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno (function)
     1   14.457   14.457   14.479   14.479 decomp.py:849 (svd)
     1    0.054    0.054    0.054    0.054 {method 'random_sample' of 'mtrand.
↪RandomState' objects}
     1    0.017    0.017    0.021    0.021 function_base.py:645 (asarray_chkfinite)
    54    0.011    0.000    0.011    0.000 {numpy.core._dotblas.dot}
     2    0.005    0.002    0.005    0.002 {method 'any' of 'numpy.ndarray' objects}
     6    0.001    0.000    0.001    0.000 ica.py:195 (gprime)
     6    0.001    0.000    0.001    0.000 ica.py:192 (g)
    14    0.001    0.000    0.001    0.000 {numpy.linalg.lapack_lite.dsyevd}
    19    0.001    0.000    0.001    0.000 twodim_base.py:204 (diag)
     1    0.001    0.001    0.008    0.008 ica.py:69 (_ica_par)
     1    0.001    0.001   14.551   14.551 {execfile}
   107    0.000    0.000    0.001    0.000 defmatrix.py:239 (__array_finalize__)
     7    0.000    0.000    0.004    0.001 ica.py:58 (_sym_decorrelation)
     7    0.000    0.000    0.002    0.000 linalg.py:841 (eigh)
   172    0.000    0.000    0.000    0.000 {isinstance}
```

(continues on next page)

(continued from previous page)

```
     1    0.000    0.000   14.551   14.551 demo.py:1 (<module>)
    29    0.000    0.000    0.000    0.000 numeric.py:180 (asarray)
    35    0.000    0.000    0.000    0.000 defmatrix.py:193 (__new__)
    35    0.000    0.000    0.001    0.000 defmatrix.py:43 (asmatrix)
    21    0.000    0.000    0.001    0.000 defmatrix.py:287 (__mul__)
    41    0.000    0.000    0.000    0.000 {numpy.core.multiarray.zeros}
    28    0.000    0.000    0.000    0.000 {method 'transpose' of 'numpy.ndarray'␣
↪objects}
     1    0.000    0.000    0.008    0.008 ica.py:97 (fastica)
     ...
```

Clearly the `svd` (in *decomp.py*) is what takes most of our time, a.k.a. the bottleneck. We have to find a way to make this step go faster, or to avoid this step (algorithmic optimization). Spending time on the rest of the code is useless.

Profiling outside of IPython, running ``cProfile``

Similar profiling can be done outside of IPython, simply calling the built-in Python profilers `cProfile` and `profile`.

```
$ python -m cProfile -o demo.prof demo.py
```

Using the `-o` switch will output the profiler results to the file `demo.prof` to view with an external tool. This can be useful if you wish to process the profiler output with a visualization tool.

10.2.3 Line-profiler

The profiler tells us which function takes most of the time, but not where it is called.

For this, we use the line_profiler: in the source file, we decorate a few functions that we want to inspect with `@profile` (no need to import it)

```
@profile
def test():
    rng = np.random.default_rng()
    data = rng.random((5000, 100))
    u, s, v = linalg.svd(data)
    pca = u[:, :10] @ data
    results = fastica(pca.T, whiten=False)
```

Then we run the script using the kernprof command, with switches `-l, --line-by-line` and `-v, --view` to use the line-by-line profiler and view the results in addition to saving them:

```
$ kernprof -l -v demo.py

Wrote profile results to demo.py.lprof
Timer unit: 1e-06 s

Total time: 1.27874 s
File: demo.py
Function: test at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           def test():
    11         1         69.0     69.0      0.0      rng = np.random.default_rng()
```

(continues on next page)

(continued from previous page)

```
    12         1       2453.0   2453.0      0.2      data = rng.random((5000, 100))
    13         1    1274715.0 1274715.0     99.7       u, s, v = sp.linalg.svd(data)
    14         1        413.0    413.0      0.0      pca = u[:, :10].T @ data
    15         1       1094.0   1094.0      0.1      results = fastica(pca.T,␣
↪whiten=False)
```

The SVD is taking all the time. We need to optimise this line.

10.3 Making code go faster

Once we have identified the bottlenecks, we need to make the corresponding code go faster.

10.3.1 Algorithmic optimization

The first thing to look for is algorithmic optimization: are there ways to compute less, or better?

For a high-level view of the problem, a good understanding of the maths behind the algorithm helps. However, it is not uncommon to find simple changes, like **moving computation or memory allocation outside a for loop**, that bring in big gains.

Example of the SVD

In both examples above, the SVD - Singular Value Decomposition - is what takes most of the time. Indeed, the computational cost of this algorithm is roughly n^3 in the size of the input matrix.

However, in both of these example, we are not using all the output of the SVD, but only the first few rows of its first return argument. If we use the `svd` implementation of SciPy, we can ask for an incomplete version of the SVD. Note that implementations of linear algebra in SciPy are richer then those in NumPy and should be preferred.

```
In [8]: %timeit np.linalg.svd(data)
1 loops, best of 3: 14.5 s per loop

In [9]: import scipy as sp

In [10]: %timeit sp.linalg.svd(data)
1 loops, best of 3: 14.2 s per loop

In [11]: %timeit sp.linalg.svd(data, full_matrices=False)
1 loops, best of 3: 295 ms per loop

In [12]: %timeit np.linalg.svd(data, full_matrices=False)
1 loops, best of 3: 293 ms per loop
```

We can then use this insight to `optimize the previous code`:

```
def test():
    rng = np.random.default_rng()
    data = rng.random((5000, 100))
    u, s, v = sp.linalg.svd(data, full_matrices=False)
    pca = u[:, :10].T @ data
    results = fastica(pca.T, whiten=False)
```

```
In [13]: import demo

In [14]: %timeit demo.
demo.fastica   demo.np        demo.prof.pdf  demo.py        demo.pyc
```

(continues on next page)

(continued from previous page)

```
demo.linalg    demo.prof      demo.prof.png  demo.py.lprof  demo.test

In [15]: %timeit demo.test()
ica.py:65: RuntimeWarning: invalid value encountered in sqrt
  W = (u * np.diag(1.0/np.sqrt(s)) * u.T) * W  # W = (W * W.T) ^{-1/2} * W
1 loops, best of 3: 17.5 s per loop

In [16]: import demo_opt

In [17]: %timeit demo_opt.test()
1 loops, best of 3: 208 ms per loop
```

Real incomplete SVDs, e.g. computing only the first 10 eigenvectors, can be computed with arpack, available in `scipy.sparse.linalg.eigsh`.

Computational linear algebra

For certain algorithms, many of the bottlenecks will be linear algebra computations. In this case, using the right function to solve the right problem is key. For instance, an eigenvalue problem with a symmetric matrix is easier to solve than with a general matrix. Also, most often, you can avoid inverting a matrix and use a less costly (and more numerically stable) operation.

Know your computational linear algebra. When in doubt, explore `scipy.linalg`, and use `%timeit` to try out different alternatives on your data.

10.4 Writing faster numerical code

A complete discussion on advanced use of NumPy is found in chapter *Advanced NumPy*, or in the article The NumPy array: a structure for efficient numerical computation by van der Walt et al. Here we discuss only some commonly encountered tricks to make code faster.

- **Vectorizing for loops**

  Find tricks to avoid for loops using NumPy arrays. For this, masks and indices arrays can be useful.

- **Broadcasting**

  Use *broadcasting* to do operations on arrays as small as possible before combining them.

- **In place operations**

```
In [18]: a = np.zeros(1e7)

In [19]: %timeit global a ; a = 0*a
10 loops, best of 3: 111 ms per loop

In [20]: %timeit global a ; a *= 0
10 loops, best of 3: 48.4 ms per loop
```

  note: we need *global a* in the timeit so that it work, as it is assigning to *a*, and thus considers it as a local variable.

- **Be easy on the memory: use views, and not copies**

  Copying big arrays is as costly as making simple numerical operations on them:

```
In [21]: a = np.zeros(1e7)
```

(continues on next page)

(continued from previous page)

```
In [22]: %timeit a.copy()
10 loops, best of 3: 124 ms per loop

In [23]: %timeit a + 1
10 loops, best of 3: 112 ms per loop
```

- **Beware of cache effects**

  Memory access is cheaper when it is grouped: accessing a big array in a continuous way is much faster than random access. This implies amongst other things that **smaller strides are faster** (see *CPU cache effects*):

```
In [24]: c = np.zeros((1e4, 1e4), order='C')

In [25]: %timeit c.sum(axis=0)
1 loops, best of 3: 3.89 s per loop

In [26]: %timeit c.sum(axis=1)
1 loops, best of 3: 188 ms per loop

In [27]: c.strides
Out[27]: (80000, 8)
```

  This is the reason why Fortran ordering or C ordering may make a big difference on operations:

```
In [28]: rng = np.random.default_rng()

In [29]: a = rng.random((20, 2**18))

In [30]: b = rng.random((20, 2**18))

In [31]: %timeit b @ a.T
9.39 ms +- 11.3 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [32]: c = np.ascontiguousarray(a.T)

In [33]: %timeit b @ c
10.4 ms +- 32.6 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

  Note that copying the data to work around this effect may not be worth it:

```
In [34]: %timeit c = np.ascontiguousarray(a.T)
15.5 ms +- 61.2 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

  Using numexpr can be useful to automatically optimize code for such effects.

- **Use compiled code**

  The last resort, once you are sure that all the high-level optimizations have been explored, is to transfer the hot spots, i.e. the few lines or functions in which most of the time is spent, to compiled code. For compiled code, the preferred option is to use Cython: it is easy to transform exiting Python code in compiled code, and with a good use of the NumPy support yields efficient code on NumPy arrays, for instance by unrolling loops.

Warning

For all the above: profile and time your choices. Don't base your optimization on theoretical considerations.

10.4.1 Additional Links

- If you need to profile memory usage, you could try the memory_profiler
- If you need to profile down into C extensions, you could try using gperftools from Python with yep.
- If you would like to track performance of your code across time, i.e. as you make new commits to your repository, you could try: asv
- If you need some interactive visualization why not try RunSnakeRun

CHAPTER 11

Sparse Arrays in SciPy

Author: *Robert Cimrman*

11.1 Introduction

(dense) matrix is:

- mathematical object
- data structure for storing a 2D array of values

important features:

- **memory allocated once for all items**
  - usually a contiguous chunk, think NumPy ndarray
- *fast* access to individual items (*)

11.1.1 Why Sparse Matrices?

- the memory grows like *n**2* for dense matrix
- small example (double precision matrix):

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 1e6, 10)
>>> plt.plot(x, 8.0 * (x**2) / 1e6, lw=5)
```

(continues on next page)

(continued from previous page)

```
[<matplotlib.lines.Line2D object at ...>]
>>> plt.xlabel('size n')
Text(...'size n')
>>> plt.ylabel('memory [MB]')
Text(...'memory [MB]')
```

11.1.2 Sparse Matrices vs. Sparse Matrix Storage Schemes

- sparse matrix is a matrix, which is *almost empty*
- storing all the zeros is wasteful -> store only nonzero items
- think **compression**
- pros: huge memory savings
- cons: slow access to individual items, but it depends on actual storage scheme.

11.1.3 Typical Applications

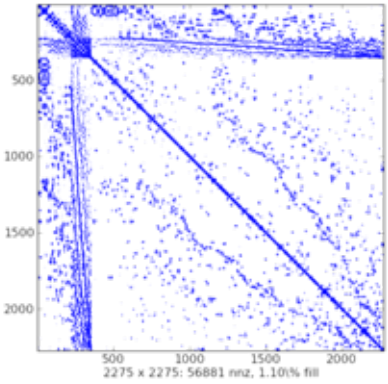
- **solution of partial differential equations (PDEs)**
  - the *finite element method*
  - mechanical engineering, electrotechnics, physics, ...
- **graph theory**
  - nonzero at *(i, j)* means that node i is connected to node j
- **natural language processing**
  - nonzero at *(i, j)* means that the document i contains the word j
- ...

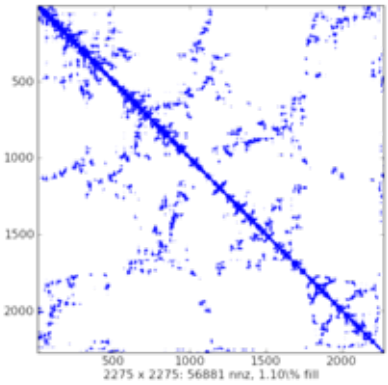
11.1.4 Prerequisites

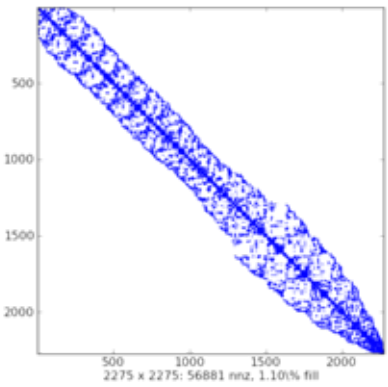
- *numpy*
- *scipy*
- *matplotlib (optional)*
- *ipython (the enhancements come handy)*

11.1.5 Sparsity Structure Visualization

- `spy()` from `matplotlib`
- example plots:







11.2 Storage Schemes

- **seven sparse array types in scipy.sparse:**
    1. csr_array: Compressed Sparse Row format
    2. csc_array: Compressed Sparse Column format
    3. bsr_array: Block Sparse Row format
    4. lil_array: List of Lists format
    5. dok_array: Dictionary of Keys format
    6. coo_array: COOrdinate format (aka IJV, triplet format)

7. dia_array: DIAgonal format

- each suitable for some tasks
- many employ sparsetools C++ module by Nathan Bell
- assume the following is imported:

```
>>> import numpy as np
>>> import scipy as sp
>>> import matplotlib.pyplot as plt
```

- **warning for Numpy users:**
  - passing a sparse array object to NumPy functions that expect ndarray/matrix does not work. Use sparse functions.
  - the older csr_matrix classes use '*' for matrix multiplication (dot product) and 'A.multiply(B)' for elementwise multiplication.
  - the newer csr_array uses '@' for dot product and '*' for elementwise multiplication
  - sparse arrays can be 1D or 2D, but not nD for n > 2 (unlike Numpy arrays).

11.2.1 Common Methods

- **all scipy.sparse array classes are subclasses of `sparray`**
  - **default implementation of arithmetic operations**
    - always converts to CSR
    - subclasses override for efficiency
  - shape, data type, set/get
  - indices of nonzero values in the array
  - format conversion, interaction with NumPy (*toarray()*)
  - ...
- **attributes:**
  - *mtx.T* - transpose (same as mtx.transpose())
  - *mtx.real* - real part of complex matrix
  - *mtx.imag* - imaginary part of complex matrix
  - *mtx.size* - the number of nonzeros (same as self.getnnz())
  - *mtx.shape* - the number of rows and columns (tuple)
- data and indices usually stored in 1D NumPy arrays

11.2.2 Sparse Array Classes

Diagonal Format (DIA)

- very simple scheme
- **diagonals in dense NumPy array of shape *(n_diag, length)***
  - fixed length -> waste space a bit when far from main diagonal
  - subclass of `_data_matrix` (sparse array classes with *.data* attribute)
- **offset for each diagonal**
  - 0 is the main diagonal

  - negative offset = below
  - positive offset = above
- fast matrix * vector (sparsetools)
- **fast and easy item-wise operations**
  - manipulate data array directly (fast NumPy machinery)
- **constructor accepts:**
  - dense array/matrix
  - sparse array/matrix
  - shape tuple (create empty array)
  - *(data, offsets)* tuple
- no slicing, no individual item access
- **use:**
  - rather specialized
  - solving PDEs by finite differences
  - with an iterative solver

Examples

- create some DIA arrays:

```
>>> data = np.array([[1, 2, 3, 4]]).repeat(3, axis=0)
>>> data
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])
>>> offsets = np.array([0, -1, 2])
>>> mtx = sp.sparse.dia_array((data, offsets), shape=(4, 4))
>>> mtx
<DIAgonal sparse array of dtype 'int64'
    with 9 stored elements (3 diagonals) and shape (4, 4)>
>>> mtx.toarray()
array([[1, 0, 3, 0],
       [1, 2, 0, 4],
       [0, 2, 3, 0],
       [0, 0, 3, 4]])

>>> data = np.arange(12).reshape((3, 4)) + 1
>>> data
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> mtx = sp.sparse.dia_array((data, offsets), shape=(4, 4))
>>> mtx.data
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> mtx.offsets
array([ 0, -1,  2], dtype=int32)
>>> print(mtx)
<DIAgonal sparse array of dtype 'int64'
```

(continues on next page)

(continued from previous page)

```
    with 9 stored elements (3 diagonals) and shape (4, 4)>
  Coords    Values
  (0, 0)    1
  (1, 1)    2
  (2, 2)    3
  (3, 3)    4
  (1, 0)    5
  (2, 1)    6
  (3, 2)    7
  (0, 2)    11
  (1, 3)    12
>>> mtx.toarray()
array([[ 1,  0, 11,  0],
       [ 5,  2,  0, 12],
       [ 0,  6,  3,  0],
       [ 0,  0,  7,  4]])
```

- explanation with a scheme:

```
offset: row

     2:  9
     1:  --10------
     0:  1  . 11  .
    -1:  5  2  . 12
    -2:  .  6  3  .
    -3:  .  .  7  4
           ---------8
```

- matrix-vector multiplication

```
>>> vec = np.ones((4, ))
>>> vec
array([1.,  1.,  1.,  1.])
>>> mtx @ vec
array([12.,  19.,   9.,  11.])
>>> (mtx * vec).toarray()
array([[ 1.,   0.,  11.,   0.],
       [ 5.,   2.,   0.,  12.],
       [ 0.,   6.,   3.,   0.],
       [ 0.,   0.,   7.,   4.]])
```

List of Lists Format (LIL)

- **row-based linked list**
  - each row is a Python list (sorted) of column indices of non-zero elements
  - rows stored in a NumPy array (*dtype=np.object*)
  - non-zero values data stored analogously
- efficient for constructing sparse arrays incrementally
- **constructor accepts:**
  - dense array/matrix
  - sparse array/matrix

  - shape tuple (create empty array)
- flexible slicing, changing sparsity structure is efficient
- slow arithmetic, slow column slicing due to being row-based
- **use:**
  - when sparsity pattern is not known apriori or changes
  - example: reading a sparse array from a text file

Examples

- create an empty LIL array:

```
>>> mtx = sp.sparse.lil_array((4, 5))
```

- prepare random data:

```
>>> rng = np.random.default_rng(27446968)
>>> data = np.round(rng.random((2, 3)))
>>> data
array([[1.,  0.,  1.],
       [0.,  0.,  1.]])
```

- assign the data using fancy indexing:

```
>>> mtx[:2, [1, 2, 3]] = data
>>> mtx
<List of Lists sparse array of dtype 'float64'
        with 3 stored elements and shape (4, 5)>
>>> print(mtx)
<List of Lists sparse array of dtype 'float64'
        with 3 stored elements and shape (4, 5)>
  Coords    Values
  (0, 1)    1.0
  (0, 3)    1.0
  (1, 3)    1.0
>>> mtx.toarray()
array([[0., 1., 0., 1., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
>>> mtx.toarray()
array([[0., 1., 0., 1., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

- more slicing and indexing:

```
>>> mtx = sp.sparse.lil_array([[0, 1, 2, 0], [3, 0, 1, 0], [1, 0, 0, 1]])
>>> mtx.toarray()
array([[0, 1, 2, 0],
       [3, 0, 1, 0],
       [1, 0, 0, 1]]...)
>>> print(mtx)
<List of Lists sparse array of dtype 'int64'
        with 6 stored elements and shape (3, 4)>
  Coords    Values
```

(continues on next page)

(continued from previous page)

```
  (0, 1)    1
  (0, 2)    2
  (1, 0)    3
  (1, 2)    1
  (2, 0)    1
  (2, 3)    1
>>> mtx[:2, :]
<List of Lists sparse array of dtype 'int64'
        with 4 stored elements and shape (2, 4)>
>>> mtx[:2, :].toarray()
array([[0, 1, 2, 0],
       [3, 0, 1, 0]]...)
>>> mtx[1:2, [0,2]].toarray()
array([[3, 1]]...)
>>> mtx.toarray()
array([[0, 1, 2, 0],
       [3, 0, 1, 0],
       [1, 0, 0, 1]]...)
```

Dictionary of Keys Format (DOK)

- **subclass of Python dict**
  - keys are *(row, column)* index tuples (no duplicate entries allowed)
  - values are corresponding non-zero values
- efficient for constructing sparse arrays incrementally
- **constructor accepts:**
  - dense array/matrix
  - sparse array/matrix
  - shape tuple (create empty array)
- efficient O(1) access to individual elements
- flexible slicing, changing sparsity structure is efficient
- can be efficiently converted to a coo_array once constructed
- slow arithmetic (*for* loops with *dict.items()*)
- **use:**
  - when sparsity pattern is not known apriori or changes

Examples

- create a DOK array element by element:

```
>>> mtx = sp.sparse.dok_array((5, 5), dtype=np.float64)
>>> mtx
<Dictionary Of Keys sparse array of dtype 'float64'
        with 0 stored elements and shape (5, 5)>
>>> for ir in range(5):
...     for ic in range(5):
...         mtx[ir, ic] = 1.0 * (ir != ic)
>>> mtx
```

(continues on next page)

(continued from previous page)

```
<Dictionary Of Keys sparse array of dtype 'float64'
        with 20 stored elements and shape (5, 5)>
>>> mtx.toarray()
array([[0.,  1.,  1.,  1.,  1.],
       [1.,  0.,  1.,  1.,  1.],
       [1.,  1.,  0.,  1.,  1.],
       [1.,  1.,  1.,  0.,  1.],
       [1.,  1.,  1.,  1.,  0.]])
```

- slicing and indexing:

```
>>> mtx[1, 1]
np.float64(0.0)
>>> mtx[[1], 1:3]
<Dictionary Of Keys sparse array of dtype 'float64'
        with 1 stored elements and shape (1, 2)>
>>> mtx[[1], 1:3].toarray()
array([[0.,  1.]])
>>> mtx[[2, 1], 1:3].toarray()
array([[1.,  0.],
       [0.,  1.]])
```

Coordinate Format (COO)

- **also known as the 'ijv' or 'triplet' format**
  - three NumPy arrays: *row*, *col*, *data*.
  - attribute *coords* is the tuple *(row, col)*
  - *data[i]* is value at *(row[i], col[i])* position
  - permits duplicate entries
  - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- fast format for constructing sparse arrays
- **constructor accepts:**
  - dense array/matrix
  - sparse array/matrix
  - shape tuple (create empty matrix)
  - *(data, coords)* tuple
- very fast conversion to and from CSR/CSC formats
- fast matrix * vector (sparsetools)
- **fast and easy item-wise operations**
  - manipulate data array directly (fast NumPy machinery)
- no slicing, no arithmetic (directly, converts to CSR)
- **use:**
  - facilitates fast conversion among sparse formats
  - when converting to other format (usually CSR or CSC), duplicate entries are summed together

    * facilitates efficient construction of finite element matrices

Examples

- create empty COO array:

```
>>> mtx = sp.sparse.coo_array((3, 4), dtype=np.int8)
>>> mtx.toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

- create using *(data, ij)* tuple:

```
>>> row = np.array([0, 3, 1, 0])
>>> col = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> mtx = sp.sparse.coo_array((data, (row, col)), shape=(4, 4))
>>> mtx
<COOrdinate sparse array of dtype 'int64'
       with 4 stored elements and shape (4, 4)>
>>> mtx.toarray()
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])
```

- duplicates entries are summed together:

```
>>> row = np.array([0, 0, 1, 3, 1, 0, 0])
>>> col = np.array([0, 2, 1, 3, 1, 0, 0])
>>> data = np.array([1, 1, 1, 1, 1, 1, 1])
>>> mtx = sp.sparse.coo_array((data, (row, col)), shape=(4, 4))
>>> mtx.toarray()
array([[3, 0, 1, 0],
       [0, 2, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 1]])
```

- no slicing...:

```
>>> mtx[2, 3]
Traceback (most recent call last):
...
TypeError: 'coo_array' object ...
```

Compressed Sparse Row Format (CSR)

- **row oriented**
    - **three NumPy arrays: *indices*, *indptr*, *data***
        * *indices* is array of column indices
        * *data* is array of corresponding nonzero values
        * *indptr* points to row starts in *indices* and *data*
        * length of *indptr* is *n_row + 1*, last item = number of values = length of both *indices* and *data*

        * nonzero values of the *i*-th row are *data[indptr[i]:indptr[i + 1]]* with column indices *indices[indptr[i]:indptr[i + 1]]*
        * item *(i, j)* can be accessed as *data[indptr[i] + k]*, where *k* is position of *j* in *indices[indptr[i]:indptr[i + 1]]*
    - **subclass of `_cs_matrix` (common CSR/CSC functionality)**
        * subclass of `_data_matrix` (sparse array classes with *.data* attribute)
- fast matrix vector products and other arithmetic (sparsetools)
- **constructor accepts:**
    - dense array/matrix
    - sparse array/matrix
    - shape tuple (create empty array)
    - *(data, coords)* tuple
    - *(data, indices, indptr)* tuple
- efficient row slicing, row-oriented operations
- slow column slicing, expensive changes to the sparsity structure
- **use:**
    - actual computations (most linear solvers support this format)

Examples

- create empty CSR array:

```
>>> mtx = sp.sparse.csr_array((3, 4), dtype=np.int8)
>>> mtx.toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

- create using *(data, coords)* tuple:

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> mtx = sp.sparse.csr_array((data, (row, col)), shape=(3, 3))
>>> mtx
<Compressed Sparse Row sparse array of dtype 'int64'
        with 6 stored elements and shape (3, 3)>
>>> mtx.toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]]...)
>>> mtx.data
array([1, 2, 3, 4, 5, 6]...)
>>> mtx.indices
array([0, 2, 2, 0, 1, 2])
>>> mtx.indptr
array([0, 2, 3, 6])
```

- create using *(data, indices, indptr)* tuple:

```
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> indptr = np.array([0, 2, 3, 6])
>>> mtx = sp.sparse.csr_array((data, indices, indptr), shape=(3, 3))
>>> mtx.toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

Compressed Sparse Column Format (CSC)

- **column oriented**
  - **three NumPy arrays: *indices*, *indptr*, *data***
    - *indices* is array of row indices
    - *data* is array of corresponding nonzero values
    - *indptr* points to column starts in *indices* and *data*
    - length is *n_col + 1*, last item = number of values = length of both *indices* and *data*
    - nonzero values of the *i*-th column are *data[indptr[i]:indptr[i+1]]* with row indices *indices[indptr[i]:indptr[i+1]]*
    - item *(i, j)* can be accessed as *data[indptr[j]+k]*, where *k* is position of *i* in *indices[indptr[j]:indptr[j+1]]*
  - **subclass of `_cs_matrix` (common CSR/CSC functionality)**
    - subclass of `_data_matrix` (sparse array classes with *.data* attribute)
- fast matrix vector products and other arithmetic (sparsetools)
- **constructor accepts:**
  - dense array/matrix
  - sparse array/matrix
  - shape tuple (create empty array)
  - *(data, coords)* tuple
  - *(data, indices, indptr)* tuple
- efficient column slicing, column-oriented operations
- slow row slicing, expensive changes to the sparsity structure
- **use:**
  - actual computations (most linear solvers support this format)

Examples

- create empty CSC array:

```
>>> mtx = sp.sparse.csc_array((3, 4), dtype=np.int8)
>>> mtx.toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

- create using *(data, coords)* tuple:

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> mtx = sp.sparse.csc_array((data, (row, col)), shape=(3, 3))
>>> mtx
<Compressed Sparse Column sparse array of dtype 'int64'
        with 6 stored elements and shape (3, 3)>
>>> mtx.toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]]...)
>>> mtx.data
array([1, 4, 5, 2, 3, 6]...)
>>> mtx.indices
array([0, 2, 2, 0, 1, 2])
>>> mtx.indptr
array([0, 2, 3, 6])
```

- create using *(data, indices, indptr)* tuple:

```
>>> data = np.array([1, 4, 5, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> indptr = np.array([0, 2, 3, 6])
>>> mtx = sp.sparse.csc_array((data, indices, indptr), shape=(3, 3))
>>> mtx.toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

Block Compressed Row Format (BSR)

- **basically a CSR with dense sub-matrices of fixed shape instead of scalar items**
  - block size *(R, C)* must evenly divide the shape of the matrix *(M, N)*
  - **three NumPy arrays: *indices*, *indptr*, *data***
    - *indices* is array of column indices for each block
    - *data* is array of corresponding nonzero values of shape *(nnz, R, C)*
    - ...
  - **subclass of `_cs_matrix` (common CSR/CSC functionality)**
    - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- fast matrix vector products and other arithmetic (sparsetools)
- **constructor accepts:**
  - dense array/matrix
  - sparse array/matrix
  - shape tuple (create empty array)
  - *(data, coords)* tuple
  - *(data, indices, indptr)* tuple

- many arithmetic operations considerably more efficient than CSR for sparse matrices with dense sub-matrices
- **use:**
  - like CSR
  - vector-valued finite element discretizations

Examples

- create empty BSR array with (1, 1) block size (like CSR...):

```
>>> mtx = sp.sparse.bsr_array((3, 4), dtype=np.int8)
>>> mtx
<Block Sparse Row sparse array of dtype 'int8'
        with 0 stored elements (blocksize=1x1) and shape (3, 4)>
>>> mtx.toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

- create empty BSR array with (3, 2) block size:

```
>>> mtx = sp.sparse.bsr_array((3, 4), blocksize=(3, 2), dtype=np.int8)
>>> mtx
<Block Sparse Row sparse array of dtype 'int8'
        with 0 stored elements (blocksize=3x2) and shape (3, 4)>
>>> mtx.toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

  - a bug?

- create using *(data, coords)* tuple with (1, 1) block size (like CSR...):

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> mtx = sp.sparse.bsr_array((data, (row, col)), shape=(3, 3))
>>> mtx
<Block Sparse Row sparse array of dtype 'int64'
        with 6 stored elements (blocksize=1x1) and shape (3, 3)>
>>> mtx.toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]]...)
>>> mtx.data
array([[[1]],

       [[2]],

       [[3]],

       [[4]],

       [[5]],

       [[6]]]...)
```

(continues on next page)

(continued from previous page)

```
>>> mtx.indices
array([0, 2, 2, 0, 1, 2])
>>> mtx.indptr
array([0, 2, 3, 6])
```

- create using *(data, indices, indptr)* tuple with (2, 2) block size:

```
>>> indptr = np.array([0, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6]).repeat(4).reshape(6, 2, 2)
>>> mtx = sp.sparse.bsr_array((data, indices, indptr), shape=(6, 6))
>>> mtx.toarray()
array([[1, 1, 0, 0, 2, 2],
       [1, 1, 0, 0, 2, 2],
       [0, 0, 0, 0, 3, 3],
       [0, 0, 0, 0, 3, 3],
       [4, 4, 5, 5, 6, 6],
       [4, 4, 5, 5, 6, 6]])
>>> data
array([[[1, 1],
        [1, 1]],

       [[2, 2],
        [2, 2]],

       [[3, 3],
        [3, 3]],

       [[4, 4],
        [4, 4]],

       [[5, 5],
        [5, 5]],

       [[6, 6],
        [6, 6]]])
```

11.2.3 Summary

Table 1: Summary of storage schemes.

format	matrix * vector	get item	fancy get	set item	fancy set	solvers	note
CSR	sparse-tools	yes	yes	slow	.	any	has data array, fast row-wise ops
CSC	sparse-tools	yes	yes	slow	.	any	has data array, fast column-wise ops
BSR	sparse-tools	.	.	.	.	special-ized	has data array, specialized
COO	sparse-tools	.	.	.	.	itera-tive	has data array, facilitates fast conversion
DIA	sparse-tools	.	.	.	.	itera-tive	has data array, specialized
LIL	via CSR	yes	yes	yes	yes	itera-tive	arithmetic via CSR, incre-mental construction
DOK	python	yes	one axis only	yes	yes	itera-tive	O(1) item access, incremen-tal construction, slow arith-metic

11.3 Linear System Solvers

- sparse matrix/eigenvalue problem solvers live in `scipy.sparse.linalg`
- **the submodules:**
  - `dsolve`: direct factorization methods for solving linear systems
  - `isolve`: iterative methods for solving linear systems
  - `eigen`: sparse eigenvalue problem solvers
- all solvers are accessible from:

```
>>> import scipy as sp
>>> sp.sparse.linalg.__all__
['ArpackError', 'ArpackNoConvergence', ..., 'use_solver']
```

11.3.1 Sparse Direct Solvers

- **default solver: SuperLU**
  - included in SciPy
  - real and complex systems
  - both single and double precision
- **optional: umfpack**
  - real and complex systems
  - double precision only
  - recommended for performance
  - wrappers now live in `scikits.umfpack`
  - check-out the new `scikits.suitesparse` by Nathaniel Smith

Examples

- import the whole module, and see its docstring:

```
>>> help(sp.sparse.linalg.spsolve)
Help on function spsolve in module scipy.sparse.linalg._dsolve.linsolve:
...
```

- both superlu and umfpack can be used (if the latter is installed) as follows:
  - prepare a linear system:

```
>>> import numpy as np
>>> mtx = sp.sparse.spdiags([[1, 2, 3, 4, 5], [6, 5, 8, 9, 10]], [0, 1], 5,␣
↪5, "csc")
>>> mtx.toarray()
array([[ 1,  5,  0,  0,  0],
       [ 0,  2,  8,  0,  0],
       [ 0,  0,  3,  9,  0],
       [ 0,  0,  0,  4, 10],
       [ 0,  0,  0,  0,  5]])
>>> rhs = np.array([1, 2, 3, 4, 5], dtype=np.float32)
```

  - solve as single precision real:

```
>>> mtx1 = mtx.astype(np.float32)
>>> x = sp.sparse.linalg.spsolve(mtx1, rhs, use_umfpack=False)
>>> print(x)
[106.   -21.     5.5   -1.5    1. ]
>>> print("Error: %s" % (mtx1 * x - rhs))
Error:  [0.  0.  0.  0.  0.]
```

  - solve as double precision real:

```
>>> mtx2 = mtx.astype(np.float64)
>>> x = sp.sparse.linalg.spsolve(mtx2, rhs, use_umfpack=True)
>>> print(x)
[106.   -21.     5.5   -1.5    1. ]
>>> print("Error: %s" % (mtx2 * x - rhs))
Error:  [0.  0.  0.  0.  0.]
```

  - solve as single precision complex:

```
>>> mtx1 = mtx.astype(np.complex64)
>>> x = sp.sparse.linalg.spsolve(mtx1, rhs, use_umfpack=False)
>>> print(x)
[106. +0.j  -21. +0.j    5.5+0.j   -1.5+0.j    1. +0.j]
>>> print("Error: %s" % (mtx1 * x - rhs))
Error:  [0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

  - solve as double precision complex:

```
>>> mtx2 = mtx.astype(np.complex128)
>>> x = sp.sparse.linalg.spsolve(mtx2, rhs, use_umfpack=True)
>>> print(x)
[106. +0.j  -21. +0.j    5.5+0.j   -1.5+0.j    1. +0.j]
>>> print("Error: %s" % (mtx2 * x - rhs))
Error:  [0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

```
"""
Solve a linear system
=====================

Construct a 1000x1000 lil_array and add some values to it, convert it
to CSR format and solve A x = b for x:and solve a linear system with a
direct solver.
"""

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

mtx = sp.sparse.lil_array((1000, 1000), dtype=np.float64)
mtx[0, :100] = rng.random(100)
mtx[1, 100:200] = mtx[[0], :100]
mtx.setdiag(rng.random(1000))

plt.clf()
plt.spy(mtx, marker=".", markersize=2)
plt.show()

mtx = mtx.tocsr()
rhs = rng.random(1000)

x = sp.sparse.linalg.spsolve(mtx, rhs)

print(f"residual: {np.linalg.norm(mtx @ x - rhs)!r}")
```

- `examples/direct_solve.py`

11.3.2 Iterative Solvers

- **the `isolve` module contains the following solvers:**
  - `bicg` (BIConjugate Gradient)
  - `bicgstab` (BIConjugate Gradient STABilized)
  - `cg` (Conjugate Gradient) - symmetric positive definite matrices only
  - `cgs` (Conjugate Gradient Squared)
  - `gmres` (Generalized Minimal RESidual)
  - `minres` (MINimum RESidual)
  - `qmr` (Quasi-Minimal Residual)

Common Parameters

- mandatory:

  A
  : [{sparse array/matrix, dense array/matrix, LinearOperator}] The N-by-N matrix of the linear system.

  b
  : [{array, matrix}] Right hand side of the linear system. Has shape (N,) or (N,1).

- optional:

x0
[{array, matrix}] Starting guess for the solution.

tol
[float] Relative tolerance to achieve before terminating.

maxiter
[integer] Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M
[{sparse array/matrix, dense array/matrix, LinearOperator}] Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback
[function] User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

LinearOperator Class

- common interface for performing matrix vector products
- useful abstraction that enables using dense and sparse matrices within the solvers, as well as *matrix-free* solutions
- has *shape* and *matvec()* (+ some optional parameters)
- example:

```
>>> import numpy as np
>>> import scipy as sp
>>> def mv(v):
...     return np.array([2 * v[0], 3 * v[1]])
...
>>> A = sp.sparse.linalg.LinearOperator((2, 2), matvec=mv)
>>> A
<2x2 _CustomLinearOperator with dtype=int8>
>>> A.matvec(np.ones(2))
array([2.,  3.])
>>> A * np.ones(2)
array([2.,  3.])
```

A Few Notes on Preconditioning

- problem specific
- often hard to develop
- **if not sure, try ILU**
  - available in `scipy.sparse.linalg` as `spilu()`

11.3.3 Eigenvalue Problem Solvers

The `eigen` module

- `arpack` * a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems
- `lobpcg` (Locally Optimal Block Preconditioned Conjugate Gradient Method) * works very well in combination with PyAMG * example by Nathan Bell:

```
"""
Compute eigenvectors and eigenvalues using a preconditioned eigensolver
=======================================================================

In this example Smoothed Aggregation (SA) is used to precondition
the LOBPCG eigensolver on a two-dimensional Poisson problem with
Dirichlet boundary conditions.
"""

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

from pyamg import smoothed_aggregation_solver
from pyamg.gallery import poisson

N = 100
K = 9
A = poisson((N, N), format="csr")

# create the AMG hierarchy
ml = smoothed_aggregation_solver(A)

# initial approximation to the K eigenvectors
X = np.random.random((A.shape[0], K))

# preconditioner based on ml
M = ml.aspreconditioner()

# compute eigenvalues and eigenvectors with LOBPCG
W, V = sp.sparse.linalg.lobpcg(A, X, M=M, tol=1e-8, largest=False)


# plot the eigenvectors
plt.figure(figsize=(9, 9))

for i in range(K):
    plt.subplot(3, 3, i + 1)
    plt.title(f"Eigenvector {i}")
    plt.pcolor(V[:, i].reshape(N, N))
    plt.axis("equal")
    plt.axis("off")
plt.show()
```

  - examples/pyamg_with_lobpcg.py

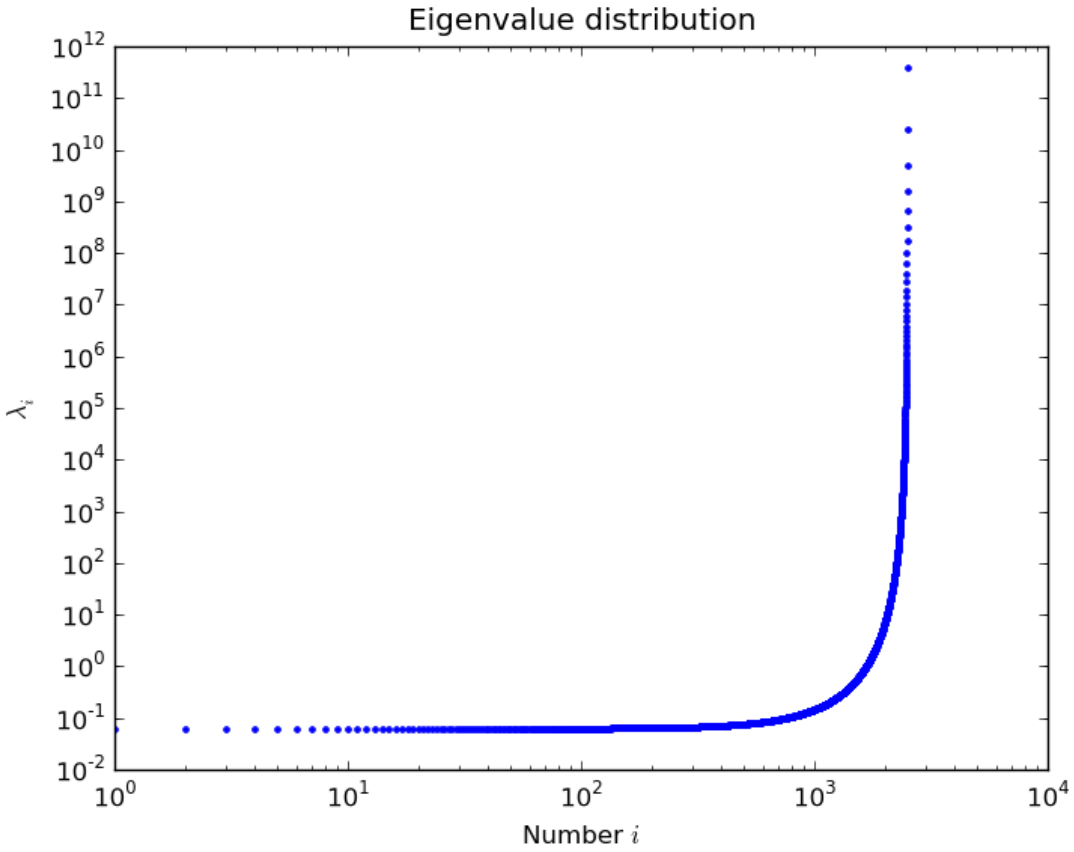
- example by Nils Wagner:
  - examples/lobpcg_sakurai.py
- output:

```
$ python examples/lobpcg_sakurai.py
Results by LOBPCG for n=2500

[ 0.06250083  0.06250028  0.06250007]

Exact eigenvalues
```

(continues on next page)

(continued from previous page)

```
[ 0.06250005  0.0625002   0.06250044]

Elapsed time 7.01
```



11.4 Other Interesting Packages

- **PyAMG**
  - algebraic multigrid solvers
  - https://github.com/pyamg/pyamg
- **Pysparse**
  - own sparse matrix classes
  - matrix and eigenvalue problem solvers
  - https://pysparse.sourceforge.net/

CHAPTER 12

Image manipulation and processing using NumPy and SciPy

Authors: *Emmanuelle Gouillart, Gaël Varoquaux*

This section addresses basic image manipulation and processing using the core scientific modules NumPy and SciPy. Some of the operations covered by this tutorial may be useful for other kinds of multidimensional array processing than image processing. In particular, the submodule `scipy.ndimage` provides functions operating on n-dimensional NumPy arrays.

See also

For more advanced image processing and image-specific routines, see the tutorial *scikit-image: image processing*, dedicated to the `skimage` module.

Image = 2-D numerical array

(or 3-D: CT, MRI, 2D + time; 4-D, ...)

Here, **image == NumPy array** `np.array`

Tools used in this tutorial:

- `numpy`: basic array manipulation
- `scipy`: `scipy.ndimage` submodule dedicated to image processing (n-dimensional images). See the documentation:

```
>>> import scipy as sp
```

Common tasks in image processing:

- Input/Output, displaying images
- Basic manipulations: cropping, flipping, rotating, ...
- Image filtering: denoising, sharpening
- Image segmentation: labeling pixels corresponding to different objects
- Classification
- Feature extraction
- Registration
- ...

Chapters contents

- *Opening and writing to image files*
- *Displaying images*
- *Basic manipulations*
  - *Statistical information*
  - *Geometrical transformations*
- *Image filtering*
  - *Blurring/smoothing*
  - *Sharpening*
  - *Denoising*
  - *Mathematical morphology*
- *Feature extraction*
  - *Edge detection*
  - *Segmentation*
- *Measuring objects properties:* `scipy.ndimage.measurements`
- *Full code examples*
- *Examples for the image processing chapter*

12.1 Opening and writing to image files

Writing an array to a file:

```
import scipy as sp
import imageio.v3 as iio

f = sp.datasets.face()
iio.imwrite("face.png", f)  # uses the Image module (PIL)

import matplotlib.pyplot as plt

plt.imshow(f)
plt.show()
```

Creating a NumPy array from an image file:

```
>>> import imageio.v3 as iio
>>> face = sp.datasets.face()
>>> iio.imwrite('face.png', face)  # First we need to create the PNG file

>>> face = iio.imread('face.png')
>>> type(face)
<class 'numpy.ndarray'>
>>> face.shape, face.dtype
((768, 1024, 3), dtype('uint8'))
```

dtype is uint8 for 8-bit images (0-255)

Opening raw files (camera, 3-D images)

```
>>> face.tofile('face.raw') # Create raw file
>>> face_from_raw = np.fromfile('face.raw', dtype=np.uint8)
>>> face_from_raw.shape
(2359296,)
>>> face_from_raw.shape = (768, 1024, 3)
```

Need to know the shape and dtype of the image (how to separate data bytes).

For large data, use `np.memmap` for memory mapping:

```
>>> face_memmap = np.memmap('face.raw', dtype=np.uint8, shape=(768, 1024, 3))
```

(data are read from the file, and not loaded into memory)

Working on a list of image files

```
>>> rng = np.random.default_rng(27446968)
>>> for i in range(10):
...     im = rng.integers(0, 256, 10000, dtype=np.uint8).reshape((100, 100))
```

(continues on next page)

(continued from previous page)

```
...     iio.imwrite(f'random_{i:02d}.png', im)
>>> from glob import glob
>>> filelist = glob('random*.png')
>>> filelist.sort()
```

12.2 Displaying images

Use `matplotlib` and `imshow` to display an image inside a `matplotlib figure`:

```
>>> f = sp.datasets.face(gray=True)  # retrieve a grayscale image
>>> import matplotlib.pyplot as plt
>>> plt.imshow(f, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>
```

Increase contrast by setting min and max values:

```
>>> plt.imshow(f, cmap=plt.cm.gray, vmin=30, vmax=200)
<matplotlib.image.AxesImage object at 0x...>
>>> # Remove axes and ticks
>>> plt.axis('off')
(np.float64(-0.5), np.float64(1023.5), np.float64(767.5), np.float64(-0.5))
```

Draw contour lines:

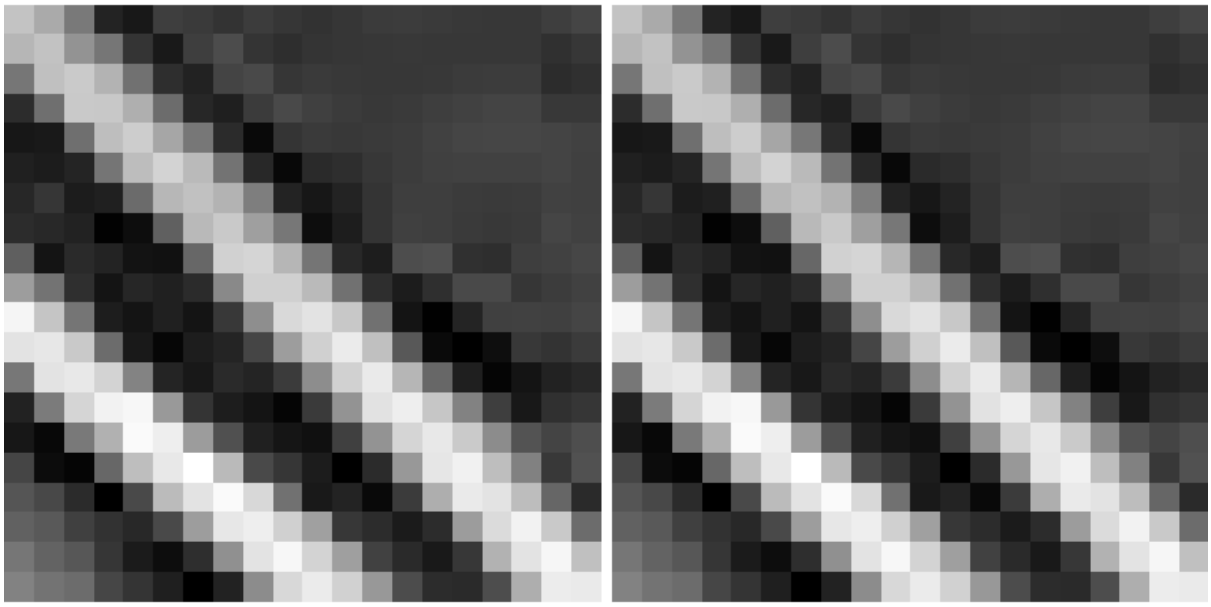
```
>>> plt.contour(f, [50, 200])
<matplotlib.contour.QuadContourSet ...>
```

For smooth intensity variations, use `interpolation='bilinear'`. For fine inspection of intensity variations, use `interpolation='nearest'`:

```
>>> plt.imshow(f[320:340, 510:530], cmap=plt.cm.gray, interpolation='bilinear')
<matplotlib.image.AxesImage object at 0x...>
>>> plt.imshow(f[320:340, 510:530], cmap=plt.cm.gray, interpolation='nearest')
<matplotlib.image.AxesImage object at 0x...>
```

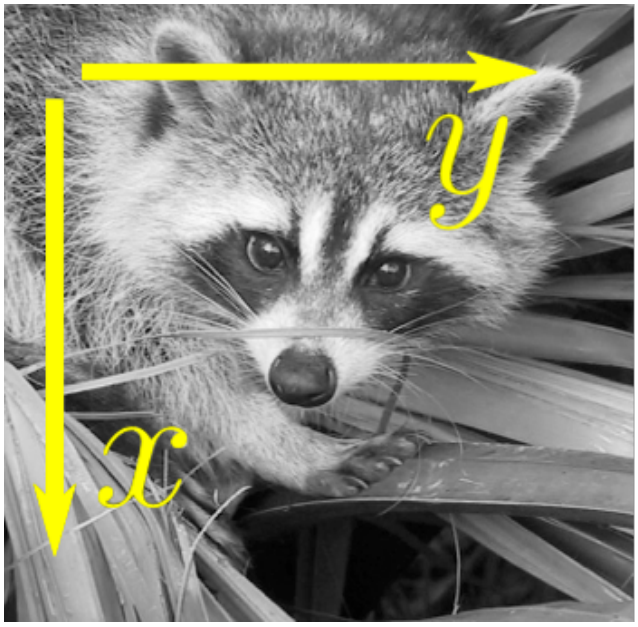
See also

More interpolation methods are in Matplotlib's examples.



12.3 Basic manipulations

Images are arrays: use the whole `numpy` machinery.



0	1	2
3	4	5
6	7	8

```
>>> face = sp.datasets.face(gray=True)
>>> face[0, 40]
np.uint8(127)
>>> # Slicing
>>> face[10:13, 20:23]
array([[141, 153, 145],
       [133, 134, 125],
       [ 96,  92,  94]], dtype=uint8)
>>> face[100:120] = 255
>>>
>>> lx, ly = face.shape
>>> X, Y = np.ogrid[0:lx, 0:ly]
>>> mask = (X - lx / 2) ** 2 + (Y - ly / 2) ** 2 > lx * ly / 4
```

(continues on next page)

(continued from previous page)

```
>>> # Masks
>>> face[mask] = 0
>>> # Fancy indexing
>>> face[range(400), range(400)] = 255
```

12.3.1 Statistical information

```
>>> face = sp.datasets.face(gray=True)
>>> face.mean()
np.float64(113.48026784261067)
>>> face.max(), face.min()
(np.uint8(250), np.uint8(0))
```

`np.histogram`

Exercise

- Open as an array the `scikit-image` logo (https://scikit-image.org/_static/img/logo.png), or an image that you have on your computer.
- Crop a meaningful part of the image, for example the python circle in the logo.
- Display the image array using `matplotlib`. Change the interpolation method and zoom to see the difference.
- Transform your image to greyscale
- Increase the contrast of the image by changing its minimum and maximum values. **Optional**: use `scipy.stats.scoreatpercentile` (read the docstring!) to saturate 5% of the darkest pixels and 5% of the lightest pixels.
- Save the array to two different file formats (png, jpg, tiff)

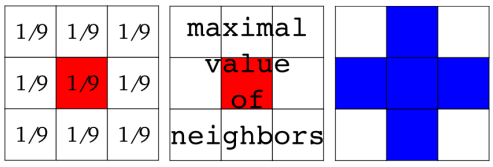
12.3.2 Geometrical transformations

```
>>> face = sp.datasets.face(gray=True)
>>> lx, ly = face.shape
>>> # Cropping
>>> crop_face = face[lx // 4: - lx // 4, ly // 4: - ly // 4]
>>> # up <-> down flip
>>> flip_ud_face = np.flipud(face)
>>> # rotation
>>> rotate_face = sp.ndimage.rotate(face, 45)
>>> rotate_face_noreshape = sp.ndimage.rotate(face, 45, reshape=False)
```

12.4 Image filtering

Local filters: replace the value of pixels by a function of the values of neighboring pixels.

Neighbourhood: square (choose size), disk, or more complicated *structuring element*.



12.4.1 Blurring/smoothing

Gaussian filter from `scipy.ndimage`:

```
>>> face = sp.datasets.face(gray=True)
>>> blurred_face = sp.ndimage.gaussian_filter(face, sigma=3)
>>> very_blurred = sp.ndimage.gaussian_filter(face, sigma=5)
```

Uniform filter

```
>>> local_mean = sp.ndimage.uniform_filter(face, size=11)
```

12.4.2 Sharpening

Sharpen a blurred image:

```
>>> face = sp.datasets.face(gray=True).astype(float)
>>> blurred_f = sp.ndimage.gaussian_filter(face, 3)
```

increase the weight of edges by adding an approximation of the Laplacian:

```
>>> filter_blurred_f = sp.ndimage.gaussian_filter(blurred_f, 1)
>>> alpha = 30
>>> sharpened = blurred_f + alpha * (blurred_f - filter_blurred_f)
```

12.4.3 Denoising

Noisy face:

```
>>> f = sp.datasets.face(gray=True)
>>> f = f[230:290, 220:320]
>>> rng = np.random.default_rng()
>>> noisy = f + 0.4 * f.std() * rng.random(f.shape)
```

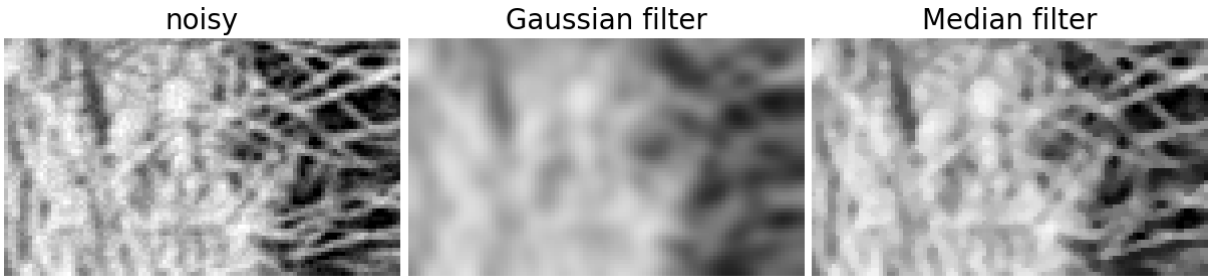
A **Gaussian filter** smoothes the noise out... and the edges as well:

```
>>> gauss_denoised = sp.ndimage.gaussian_filter(noisy, 2)
```

Most local linear isotropic filters blur the image (`scipy.ndimage.uniform_filter`)

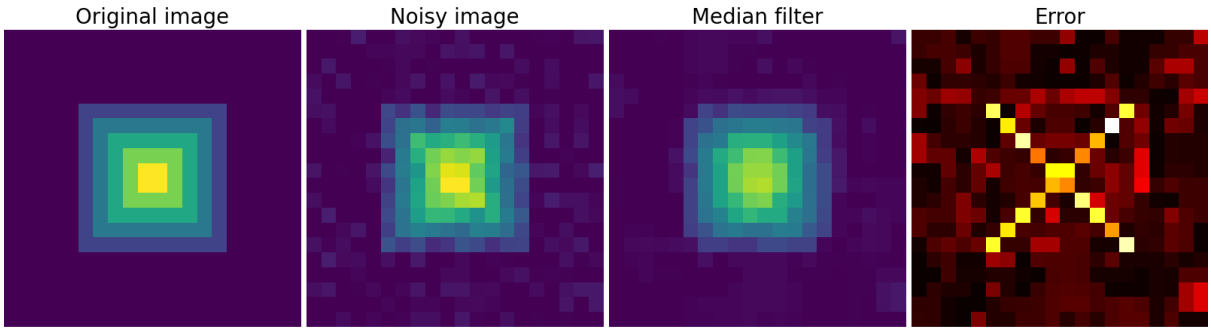
A **median filter** preserves better the edges:

```
>>> med_denoised = sp.ndimage.median_filter(noisy, 3)
```



Median filter: better result for straight boundaries (**low curvature**):

```
>>> im = np.zeros((20, 20))
>>> im[5:-5, 5:-5] = 1
>>> im = sp.ndimage.distance_transform_bf(im)
>>> rng = np.random.default_rng()
>>> im_noise = im + 0.2 * rng.standard_normal(im.shape)
>>> im_med = sp.ndimage.median_filter(im_noise, 3)
```



Other rank filter: `scipy.ndimage.maximum_filter`, `scipy.ndimage.percentile_filter`

Other local non-linear filters: Wiener (`scipy.signal.wiener`), etc.

Non-local filters

Exercise: denoising

- Create a binary image (of 0s and 1s) with several objects (circles, ellipses, squares, or random shapes).
- Add some noise (e.g., 20% of noise)
- Try two different denoising methods for denoising the image: gaussian filtering and median filtering.
- Compare the histograms of the two different denoised images. Which one is the closest to the histogram of the original (noise-free) image?

See also

More denoising filters are available in `skimage.denoising`, see the *scikit-image: image processing* tutorial.

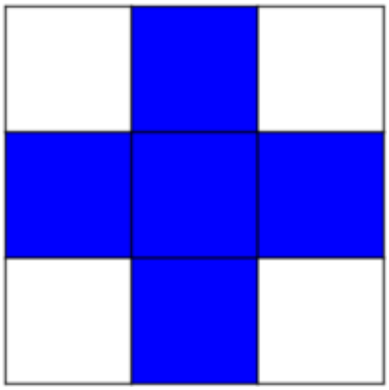
12.4.4 Mathematical morphology

See wikipedia for a definition of mathematical morphology.

Probe an image with a simple shape (a **structuring element**), and modify this image according to how the shape locally fits or misses the image.

Structuring element:

```
>>> el = sp.ndimage.generate_binary_structure(2, 1)
>>> el
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]])
>>> el.astype(int)
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```



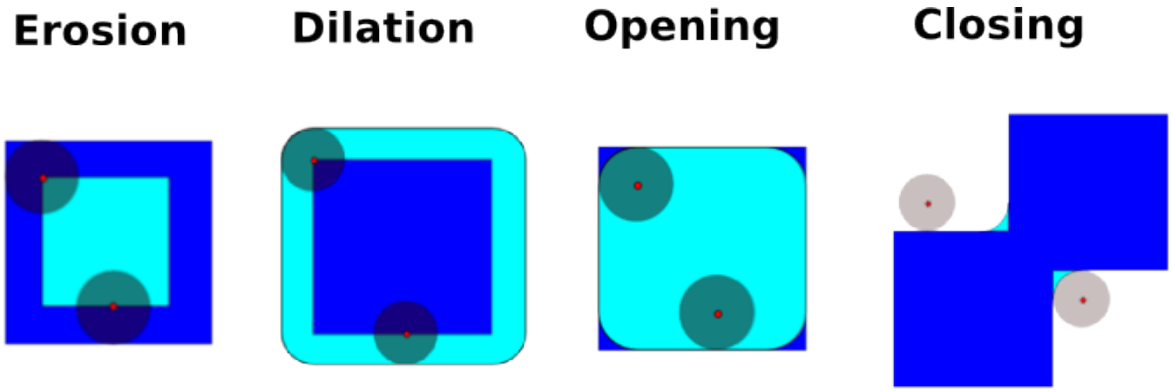
Erosion = minimum filter. Replace the value of a pixel by the minimal value covered by the structuring element.:

```
>>> a = np.zeros((7,7), dtype=int)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> sp.ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> # Erosion removes objects smaller than the structure
>>> sp.ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

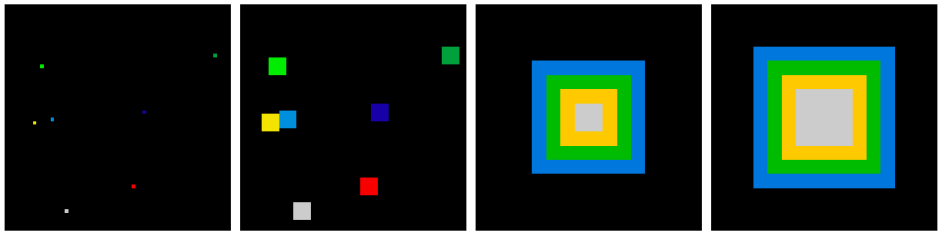


Dilation: maximum filter:

```
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.]])
>>> sp.ndimage.binary_dilation(a).astype(a.dtype)
array([[0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  1.,  1.,  1.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.]])
```

Also works for grey-valued images:

```
>>> rng = np.random.default_rng(27446968)
>>> im = np.zeros((64, 64))
>>> x, y = (63*rng.random((2, 8))).astype(int)
>>> im[x, y] = np.arange(8)

>>> bigger_points = sp.ndimage.grey_dilation(im, size=(5, 5), structure=np.ones((5,␣
↪5)))

>>> square = np.zeros((16, 16))
>>> square[4:-4, 4:-4] = 1
>>> dist = sp.ndimage.distance_transform_bf(square)
>>> dilate_dist = sp.ndimage.grey_dilation(dist, size=(3, 3), \
...          structure=np.ones((3, 3)))
```



Opening: erosion + dilation:

```
>>> a = np.zeros((5,5), dtype=int)
>>> a[1:4, 1:4] = 1; a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
>>> sp.ndimage.binary_opening(a, structure=np.ones((3,3))).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> sp.ndimage.binary_opening(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

Application: remove noise:

```
>>> square = np.zeros((32, 32))
>>> square[10:-10, 10:-10] = 1
>>> rng = np.random.default_rng(27446968)
>>> x, y = (32*rng.random((2, 20))).astype(int)
>>> square[x, y] = 1

>>> open_square = sp.ndimage.binary_opening(square)

>>> eroded_square = sp.ndimage.binary_erosion(square)
>>> reconstruction = sp.ndimage.binary_propagation(eroded_square, mask=square)
```

Closing: dilation + erosion

Many other mathematical morphology operations: hit and miss transform, tophat, etc.

12.5 Feature extraction

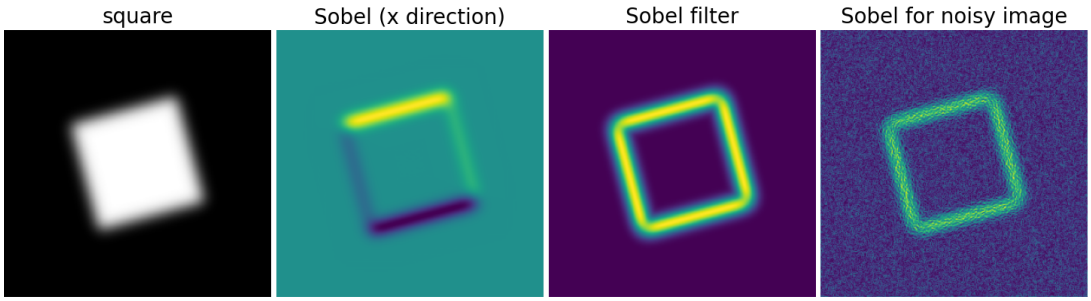
12.5.1 Edge detection

Synthetic data:

```
>>> im = np.zeros((256, 256))
>>> im[64:-64, 64:-64] = 1
>>>
>>> im = sp.ndimage.rotate(im, 15, mode='constant')
>>> im = sp.ndimage.gaussian_filter(im, 8)
```

Use a **gradient operator** (**Sobel**) to find high intensity variations:

```
>>> sx = sp.ndimage.sobel(im, axis=0, mode='constant')
>>> sy = sp.ndimage.sobel(im, axis=1, mode='constant')
>>> sob = np.hypot(sx, sy)
```



12.5.2 Segmentation

- **Histogram-based** segmentation (no spatial information)

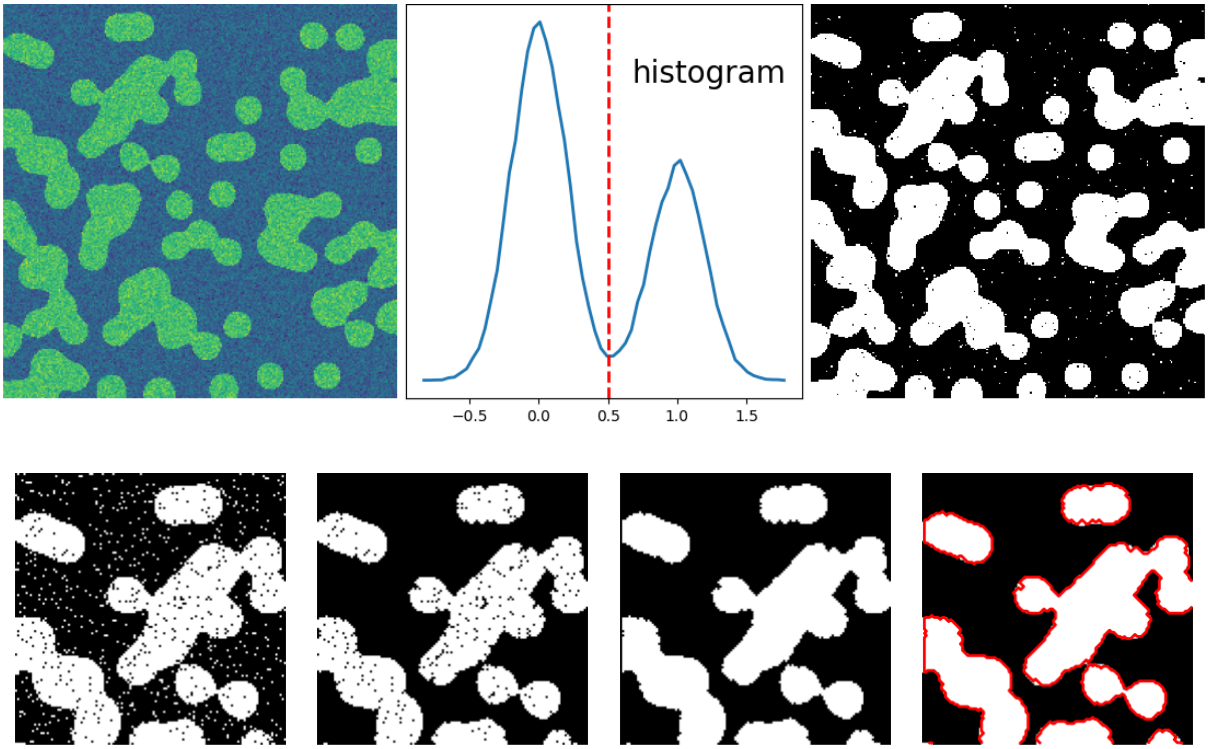
```
>>> n = 10
>>> l = 256
>>> im = np.zeros((l, l))
>>> rng = np.random.default_rng(27446968)
>>> points = l*rng.random((2, n**2))
>>> im[(points[0]).astype(int), (points[1]).astype(int)] = 1
>>> im = sp.ndimage.gaussian_filter(im, sigma=l/(4.*n))

>>> mask = (im > im.mean()).astype(float)
>>> mask += 0.1 * im
>>> img = mask + 0.2*rng.standard_normal(mask.shape)

>>> hist, bin_edges = np.histogram(img, bins=60)
>>> bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])

>>> binary_img = img > 0.5
```

Use mathematical morphology to clean up the result:

```
>>> # Remove small white regions
>>> open_img = sp.ndimage.binary_opening(binary_img)
>>> # Remove small black hole
>>> close_img = sp.ndimage.binary_closing(open_img)
```

Exercise

Check that reconstruction operations (erosion + propagation) produce a better result than open-



ing/closing:

```
>>> eroded_img = sp.ndimage.binary_erosion(binary_img)
>>> reconstruct_img = sp.ndimage.binary_propagation(eroded_img, mask=binary_img)
>>> tmp = np.logical_not(reconstruct_img)
>>> eroded_tmp = sp.ndimage.binary_erosion(tmp)
>>> reconstruct_final = np.logical_not(sp.ndimage.binary_propagation(eroded_tmp,␣
↪mask=tmp))
>>> np.abs(mask - close_img).mean()
np.float64(0.00640699...)
>>> np.abs(mask - reconstruct_final).mean()
np.float64(0.00082232...)
```

Exercise

Check how a first denoising step (e.g. with a median filter) modifies the histogram, and check that the resulting histogram-based segmentation is more accurate.

See also

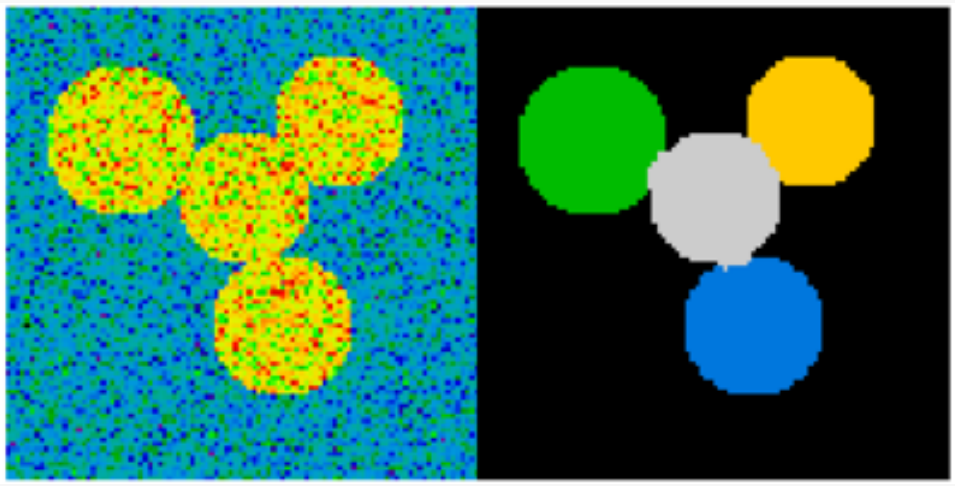
More advanced segmentation algorithms are found in the `scikit-image`: see *scikit-image: image processing*.

See also

Other Scientific Packages provide algorithms that can be useful for image processing. In this example, we use the spectral clustering function of the `scikit-learn` in order to segment glued objects.

```
>>> from sklearn.feature_extraction import image
```

```
>>> from sklearn.cluster import spectral_clustering

>>> l = 100
>>> x, y = np.indices((l, l))

>>> center1 = (28, 24)
>>> center2 = (40, 50)
>>> center3 = (67, 58)
>>> center4 = (24, 70)
>>> radius1, radius2, radius3, radius4 = 16, 14, 15, 14

>>> circle1 = (x - center1[0])**2 + (y - center1[1])**2 < radius1**2
>>> circle2 = (x - center2[0])**2 + (y - center2[1])**2 < radius2**2
>>> circle3 = (x - center3[0])**2 + (y - center3[1])**2 < radius3**2
>>> circle4 = (x - center4[0])**2 + (y - center4[1])**2 < radius4**2

>>> # 4 circles
>>> img = circle1 + circle2 + circle3 + circle4
>>> mask = img.astype(bool)
>>> img = img.astype(float)

>>> rng = np.random.default_rng()
>>> img += 1 + 0.2*rng.standard_normal(img.shape)
>>> # Convert the image into a graph with the value of the gradient on
>>> # the edges.
>>> graph = image.img_to_graph(img, mask=mask)

>>> # Take a decreasing function of the gradient: we take it weakly
>>> # dependent from the gradient the segmentation is close to a voronoi
>>> graph.data = np.exp(-graph.data/graph.data.std())

>>> labels = spectral_clustering(graph, n_clusters=4, eigen_solver='arpack')
>>> label_im = -np.ones(mask.shape)
>>> label_im[mask] = labels
```



12.6 Measuring objects properties: `scipy.ndimage.measurements`

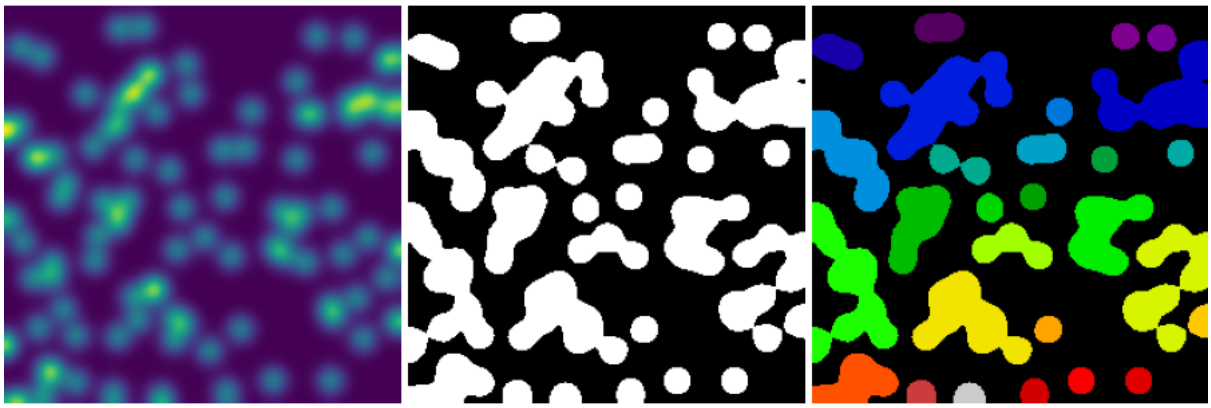
Synthetic data:

```
>>> n = 10
>>> l = 256
>>> im = np.zeros((l, l))
>>> rng = np.random.default_rng(27446968)
>>> points = l * rng.random((2, n**2))
>>> im[(points[0]).astype(int), (points[1]).astype(int)] = 1
>>> im = sp.ndimage.gaussian_filter(im, sigma=l/(4.*n))
>>> mask = im > im.mean()
```

- **Analysis of connected components**

Label connected components: `scipy.dimage.label`:

```
>>> label_im, nb_labels = sp.ndimage.label(mask)
>>> nb_labels # how many regions?
28
>>> plt.imshow(label_im)
<matplotlib.image.AxesImage object at 0x...>
```



Compute size, mean_value, etc. of each region:

```
>>> sizes = sp.ndimage.sum(mask, label_im, range(nb_labels + 1))
>>> mean_vals = sp.ndimage.sum(im, label_im, range(1, nb_labels + 1))
```

Clean up small connect components:

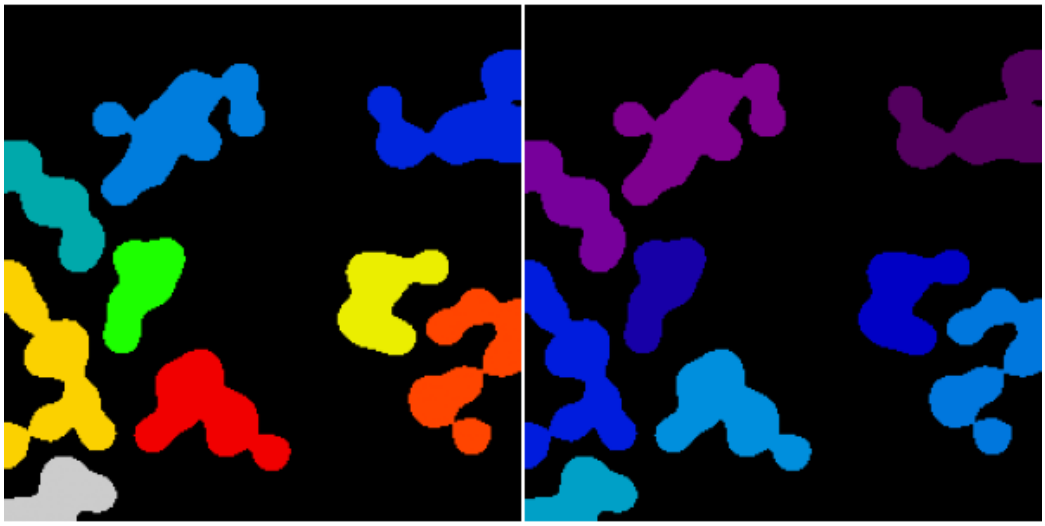
```
>>> mask_size = sizes < 1000
>>> remove_pixel = mask_size[label_im]
>>> remove_pixel.shape
(256, 256)
>>> label_im[remove_pixel] = 0
>>> plt.imshow(label_im)
<matplotlib.image.AxesImage object at 0x...>
```

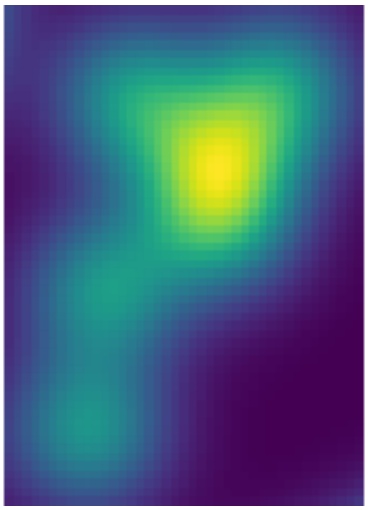
Now reassign labels with `np.searchsorted`:

```
>>> labels = np.unique(label_im)
>>> label_im = np.searchsorted(labels, label_im)
```

Find region of interest enclosing object:

```
>>> slice_x, slice_y = sp.ndimage.find_objects(label_im)[3]
>>> roi = im[slice_x, slice_y]
>>> plt.imshow(roi)
<matplotlib.image.AxesImage object at 0x...>
```





Other spatial measures: `scipy.ndimage.center_of_mass`, `scipy.ndimage.maximum_position`, etc.

Can be used outside the limited scope of segmentation applications.

Example: block mean:

```
>>> f = sp.datasets.face(gray=True)
>>> sx, sy = f.shape
>>> X, Y = np.ogrid[0:sx, 0:sy]
>>> regions = (sy//6) * (X//4) + (Y//6)  # note that we use broadcasting
>>> block_mean = sp.ndimage.mean(f, labels=regions, index=np.arange(1,
...     regions.max() +1))
>>> block_mean.shape = (sx // 4, sy // 6)
```

When regions are regular blocks, it is more efficient to use stride tricks (*Example: fake dimensions with strides*).

Non-regularly-spaced blocks: radial mean:

```
>>> sx, sy = f.shape
>>> X, Y = np.ogrid[0:sx, 0:sy]
>>> r = np.hypot(X - sx/2, Y - sy/2)
>>> rbin = (20* r/r.max()).astype(int)
>>> radial_mean = sp.ndimage.mean(f, labels=rbin, index=np.arange(1, rbin.max() +1))
```

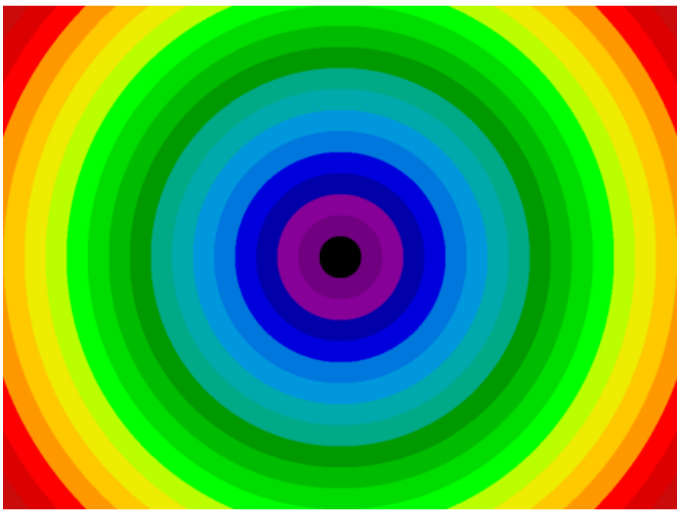
- **Other measures**

Correlation function, Fourier/wavelet spectrum, etc.

One example with mathematical morphology: granulometry

```
>>> def disk_structure(n):
...     struct = np.zeros((2 * n + 1, 2 * n + 1))
...     x, y = np.indices((2 * n + 1, 2 * n + 1))
...     mask = (x - n)**2 + (y - n)**2 <= n**2
...     struct[mask] = 1
...     return struct.astype(bool)
...
```

(continues on next page)



(continued from previous page)

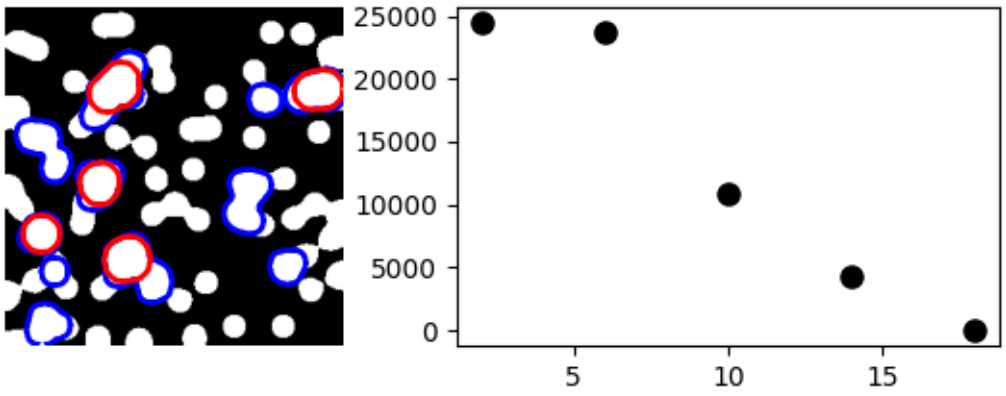
```
>>>
>>> def granulometry(data, sizes=None):
...     s = max(data.shape)
...     if sizes is None:
...         sizes = range(1, s/2, 2)
...     granulo = [sp.ndimage.binary_opening(data, \
...         structure=disk_structure(n)).sum() for n in sizes]
...     return granulo
...
>>>
>>> rng = np.random.default_rng(27446968)
>>> n = 10
>>> l = 256
>>> im = np.zeros((l, l))
>>> points = l*rng.random((2, n**2))
>>> im[(points[0]).astype(int), (points[1]).astype(int)] = 1
>>> im = sp.ndimage.gaussian_filter(im, sigma=l/(4.*n))
>>>
>>> mask = im > im.mean()
>>>
>>> granulo = granulometry(mask, sizes=np.arange(2, 19, 4))
```

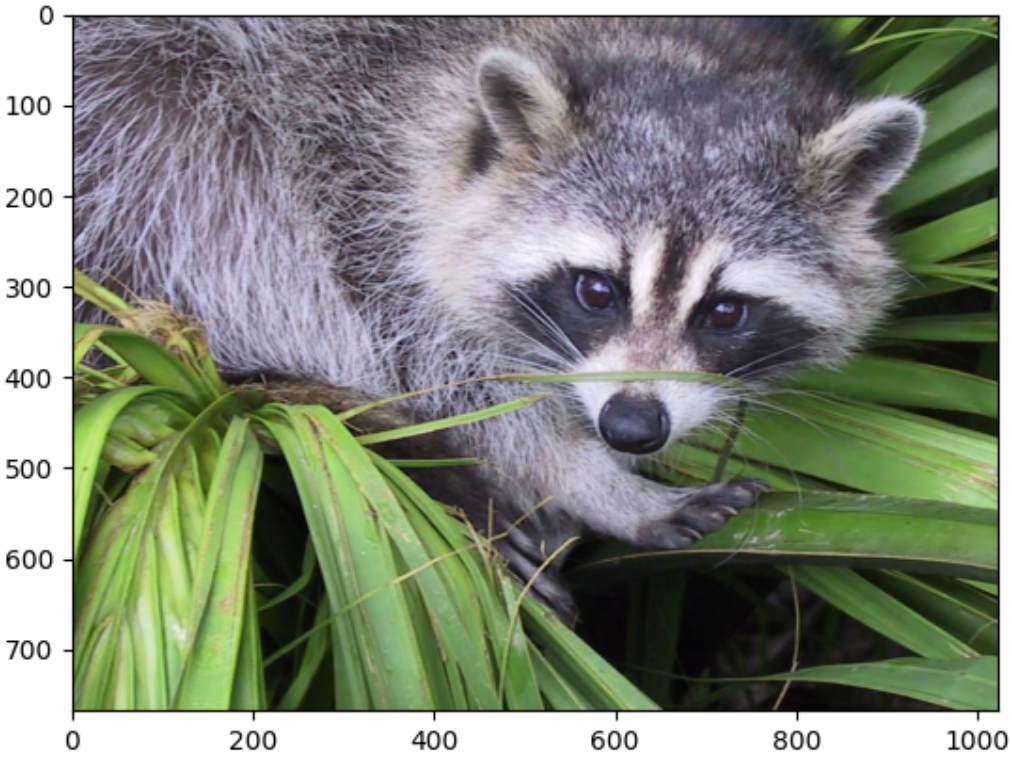
12.7 Full code examples

12.8 Examples for the image processing chapter

12.8.1 Displaying a Raccoon Face

Small example to plot a raccoon face.



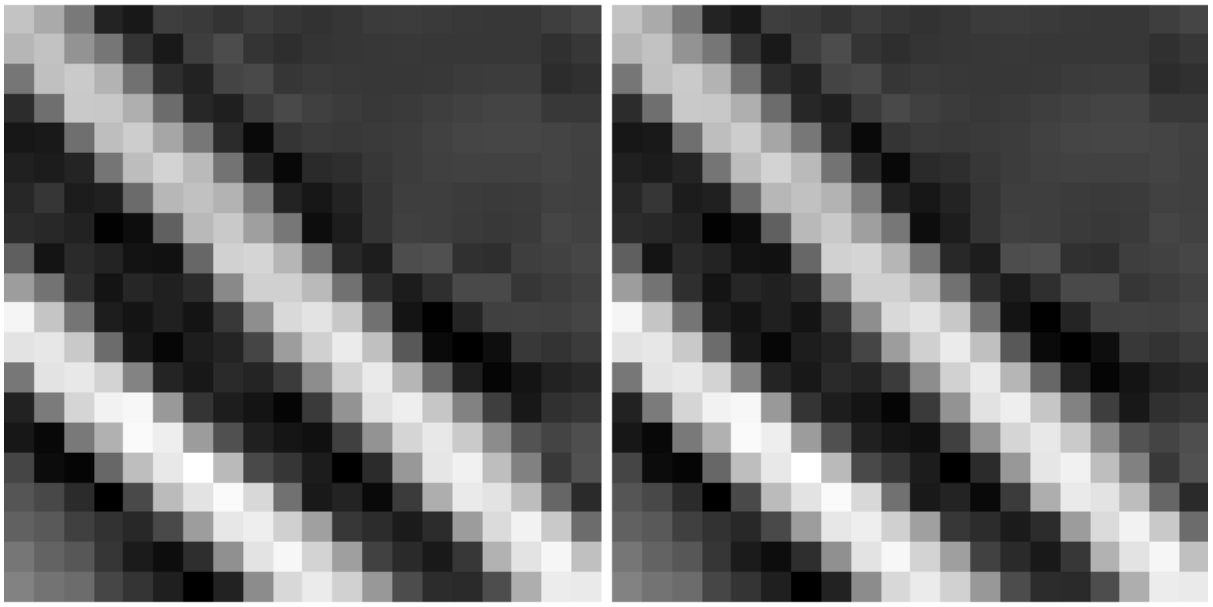



```
import scipy as sp
import imageio.v3 as iio

f = sp.datasets.face()
iio.imwrite("face.png", f)  # uses the Image module (PIL)

import matplotlib.pyplot as plt

plt.imshow(f)
plt.show()
```

Total running time of the script: (0 minutes 0.547 seconds)

12.8.2 Image interpolation

The example demonstrates image interpolation on a Raccoon face.



```
import scipy as sp
import matplotlib.pyplot as plt

f = sp.datasets.face(gray=True)

plt.figure(figsize=(8, 4))

plt.subplot(1, 2, 1)
plt.imshow(f[320:340, 510:530], cmap="gray")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(f[320:340, 510:530], cmap="gray", interpolation="nearest")
plt.axis("off")

plt.subplots_adjust(wspace=0.02, hspace=0.02, top=1, bottom=0, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.176 seconds)

12.8.3 Plot the block mean of an image

An example showing how to use broad-casting to plot the mean of blocks of an image.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

f = sp.datasets.face(gray=True)
sx, sy = f.shape
X, Y = np.ogrid[0:sx, 0:sy]

regions = sy // 6 * (X // 4) + Y // 6
block_mean = sp.ndimage.mean(f, labels=regions, index=np.arange(1, regions.max() + 1))
block_mean.shape = (sx // 4, sy // 6)

plt.figure(figsize=(5, 5))
plt.imshow(block_mean, cmap="gray")
plt.axis("off")

plt.show()
```

Total running time of the script: (0 minutes 0.199 seconds)

12.8.4 Image manipulation and NumPy arrays

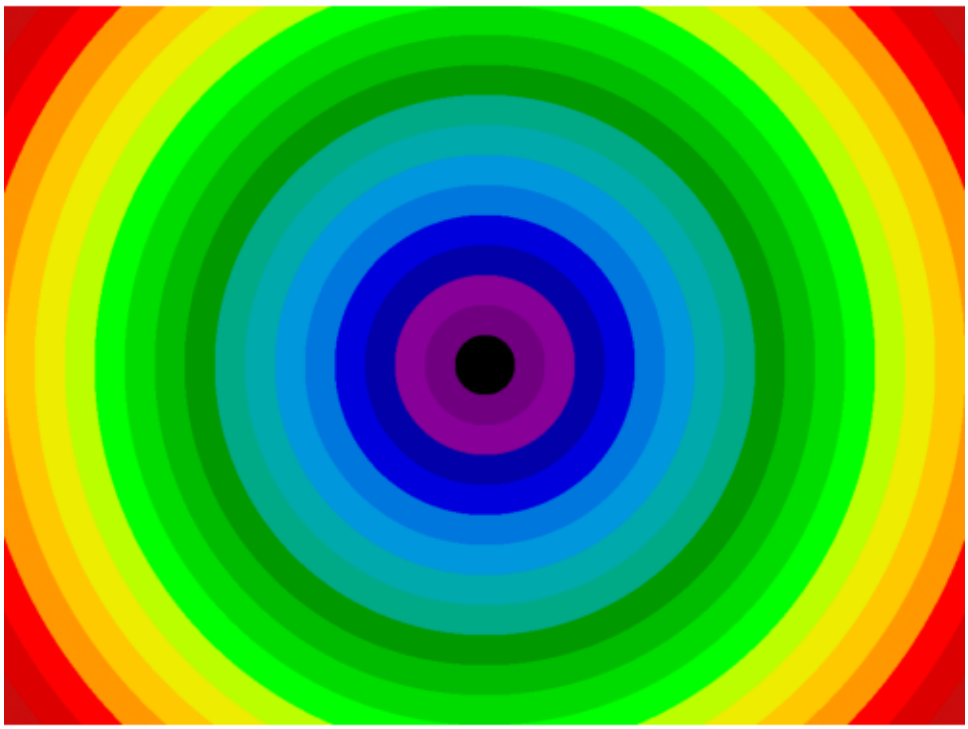
This example shows how to do image manipulation using common NumPy arrays tricks.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

face = sp.datasets.face(gray=True)
face[10:13, 20:23]
face[100:120] = 255

lx, ly = face.shape
X, Y = np.ogrid[0:lx, 0:ly]
mask = (X - lx / 2) ** 2 + (Y - ly / 2) ** 2 > lx * ly / 4
face[mask] = 0
face[range(400), range(400)] = 255

plt.figure(figsize=(3, 3))
plt.axes((0, 0, 1, 1))
plt.imshow(face, cmap="gray")
plt.axis("off")

plt.show()
```

Total running time of the script: (0 minutes 0.203 seconds)

12.8.5 Radial mean

This example shows how to do a radial mean with scikit-image.



```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

f = sp.datasets.face(gray=True)
sx, sy = f.shape
X, Y = np.ogrid[0:sx, 0:sy]


r = np.hypot(X - sx / 2, Y - sy / 2)

rbin = (20 * r / r.max()).astype(int)
radial_mean = sp.ndimage.mean(f, labels=rbin, index=np.arange(1, rbin.max() + 1))

plt.figure(figsize=(5, 5))
plt.axes((0, 0, 1, 1))
plt.imshow(rbin, cmap="nipy_spectral")
plt.axis("off")

plt.show()
```

Total running time of the script: (0 minutes 0.227 seconds)

12.8.6 Display a Raccoon Face

An example that displays a raccoon face with matplotlib.

```
import scipy as sp
import matplotlib.pyplot as plt

f = sp.datasets.face(gray=True)

plt.figure(figsize=(10, 3.6))

plt.subplot(131)
plt.imshow(f, cmap="gray")

plt.subplot(132)
plt.imshow(f, cmap="gray", vmin=30, vmax=200)
plt.axis("off")

plt.subplot(133)
plt.imshow(f, cmap="gray")
plt.contour(f, [50, 200])
plt.axis("off")

plt.subplots_adjust(wspace=0, hspace=0.0, top=0.99, bottom=0.01, left=0.05, right=0.
↪99)
plt.show()
```

Total running time of the script: (0 minutes 0.418 seconds)

12.8.7 Image sharpening

This example shows how to sharpen an image in noiseless situation by applying the filter inverse to the blur.

```
import scipy as sp
import matplotlib.pyplot as plt

f = sp.datasets.face(gray=True).astype(float)
blurred_f = sp.ndimage.gaussian_filter(f, 3)

filter_blurred_f = sp.ndimage.gaussian_filter(blurred_f, 1)

alpha = 30
sharpened = blurred_f + alpha * (blurred_f - filter_blurred_f)

plt.figure(figsize=(12, 4))

plt.subplot(131)
plt.imshow(f, cmap="gray")
plt.axis("off")
plt.subplot(132)
plt.imshow(blurred_f, cmap="gray")
plt.axis("off")
plt.subplot(133)
plt.imshow(sharpened, cmap="gray")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.550 seconds)

12.8.8 Blurring of images

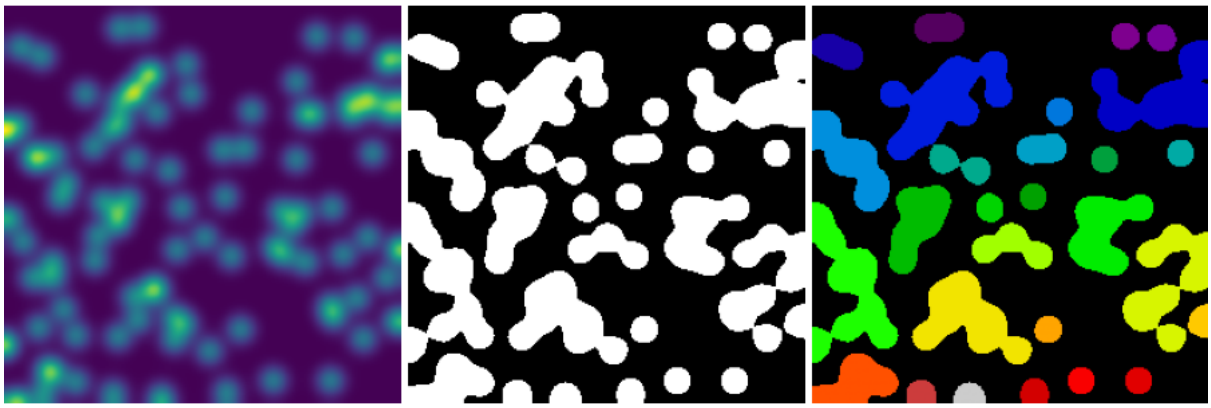
An example showing various processes that blur an image.

```
import scipy as sp
import matplotlib.pyplot as plt

face = sp.datasets.face(gray=True)
blurred_face = sp.ndimage.gaussian_filter(face, sigma=3)
very_blurred = sp.ndimage.gaussian_filter(face, sigma=5)
local_mean = sp.ndimage.uniform_filter(face, size=11)

plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.imshow(blurred_face, cmap="gray")
plt.axis("off")
plt.subplot(132)
plt.imshow(very_blurred, cmap="gray")
plt.axis("off")
plt.subplot(133)
plt.imshow(local_mean, cmap="gray")
plt.axis("off")

plt.subplots_adjust(wspace=0, hspace=0.0, top=0.99, bottom=0.01, left=0.01, right=0.
↪99)

plt.show()
```

Total running time of the script: (0 minutes 0.369 seconds)

12.8.9 Synthetic data

The example generates and displays simple synthetic data.

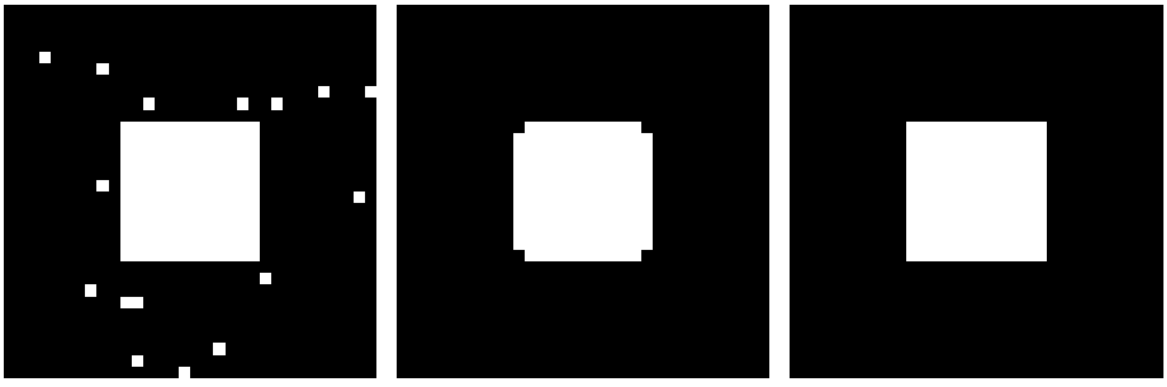


```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = im > im.mean()

label_im, nb_labels = sp.ndimage.label(mask)

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.imshow(im)
plt.axis("off")
plt.subplot(132)
plt.imshow(mask, cmap="gray")
plt.axis("off")
plt.subplot(133)
plt.imshow(label_im, cmap="nipy_spectral")
plt.axis("off")

plt.subplots_adjust(wspace=0.02, hspace=0.02, top=1, bottom=0, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.081 seconds)

12.8.10 Opening, erosion, and propagation

This example shows simple operations of mathematical morphology.



```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

square = np.zeros((32, 32))
square[10:-10, 10:-10] = 1
```

(continues on next page)

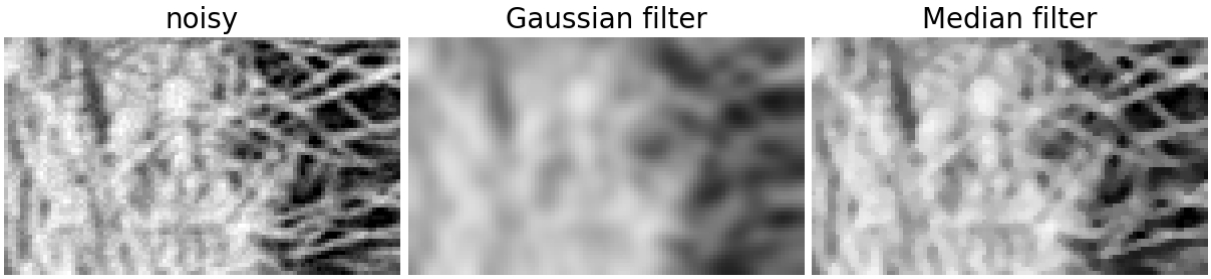
(continued from previous page)

```
rng = np.random.default_rng(27446968)
x, y = (32 * rng.random((2, 20))).astype(int)
square[x, y] = 1

open_square = sp.ndimage.binary_opening(square)

eroded_square = sp.ndimage.binary_erosion(square)
reconstruction = sp.ndimage.binary_propagation(eroded_square, mask=square)

plt.figure(figsize=(9.5, 3))
plt.subplot(131)
plt.imshow(square, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(132)
plt.imshow(open_square, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(133)
plt.imshow(reconstruction, cmap="gray", interpolation="nearest")
plt.axis("off")

plt.subplots_adjust(wspace=0, hspace=0.02, top=0.99, bottom=0.01, left=0.01, right=0.
↪99)
plt.show()
```

Total running time of the script: (0 minutes 0.037 seconds)

12.8.11 Image denoising

This example demoes image denoising on a Raccoon face.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

f = sp.datasets.face(gray=True)
f = f[230:290, 220:320]

noisy = f + 0.4 * f.std() * rng.random(f.shape)

gauss_denoised = sp.ndimage.gaussian_filter(noisy, 2)
med_denoised = sp.ndimage.median_filter(noisy, 3)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(12, 2.8))

plt.subplot(131)
plt.imshow(noisy, cmap="gray", vmin=40, vmax=220)
plt.axis("off")
plt.title("noisy", fontsize=20)
plt.subplot(132)
plt.imshow(gauss_denoised, cmap="gray", vmin=40, vmax=220)
plt.axis("off")
plt.title("Gaussian filter", fontsize=20)
plt.subplot(133)
plt.imshow(med_denoised, cmap="gray", vmin=40, vmax=220)
plt.axis("off")
plt.title("Median filter", fontsize=20)

plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.9, bottom=0, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.221 seconds)

12.8.12 Geometrical transformations

This examples demos some simple geometrical transformations on a Raccoon face.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

face = sp.datasets.face(gray=True)
lx, ly = face.shape
# Cropping
crop_face = face[lx // 4 : -lx // 4, ly // 4 : -ly // 4]
# up <-> down flip
flip_ud_face = np.flipud(face)
# rotation
rotate_face = sp.ndimage.rotate(face, 45)
rotate_face_noreshape = sp.ndimage.rotate(face, 45, reshape=False)

plt.figure(figsize=(12.5, 2.5))


plt.subplot(151)
plt.imshow(face, cmap="gray")
plt.axis("off")
plt.subplot(152)
plt.imshow(crop_face, cmap="gray")
plt.axis("off")
```

(continues on next page)

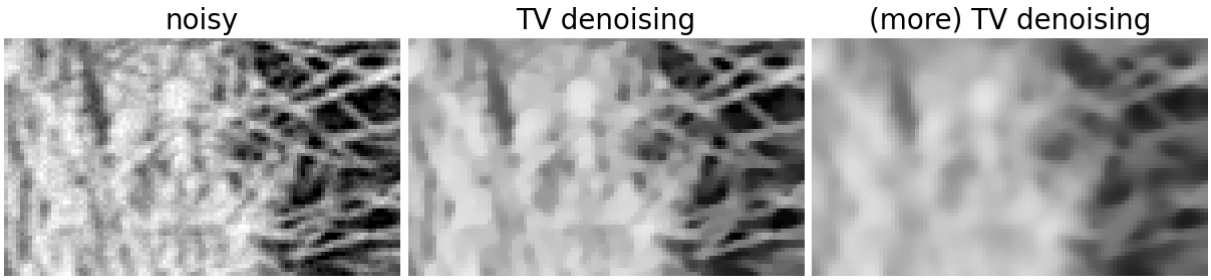
(continued from previous page)

```
plt.subplot(153)
plt.imshow(flip_ud_face, cmap="gray")
plt.axis("off")
plt.subplot(154)
plt.imshow(rotate_face, cmap="gray")
plt.axis("off")
plt.subplot(155)
plt.imshow(rotate_face_noreshape, cmap="gray")
plt.axis("off")

plt.subplots_adjust(wspace=0.02, hspace=0.3, top=1, bottom=0.1, left=0, right=1)

plt.show()
```

Total running time of the script: (0 minutes 0.609 seconds)

12.8.13 Total Variation denoising

This example demoes Total-Variation (TV) denoising on a Raccoon face.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

from skimage.restoration import denoise_tv_chambolle

rng = np.random.default_rng(27446968)

f = sp.datasets.face(gray=True)
f = f[230:290, 220:320]

noisy = f + 0.4 * f.std() * rng.random(f.shape)

tv_denoised = denoise_tv_chambolle(noisy, weight=10)


plt.figure(figsize=(12, 2.8))

plt.subplot(131)
plt.imshow(noisy, cmap="gray", vmin=40, vmax=220)
plt.axis("off")
plt.title("noisy", fontsize=20)
plt.subplot(132)
plt.imshow(tv_denoised, cmap="gray", vmin=40, vmax=220)
plt.axis("off")
```

(continues on next page)

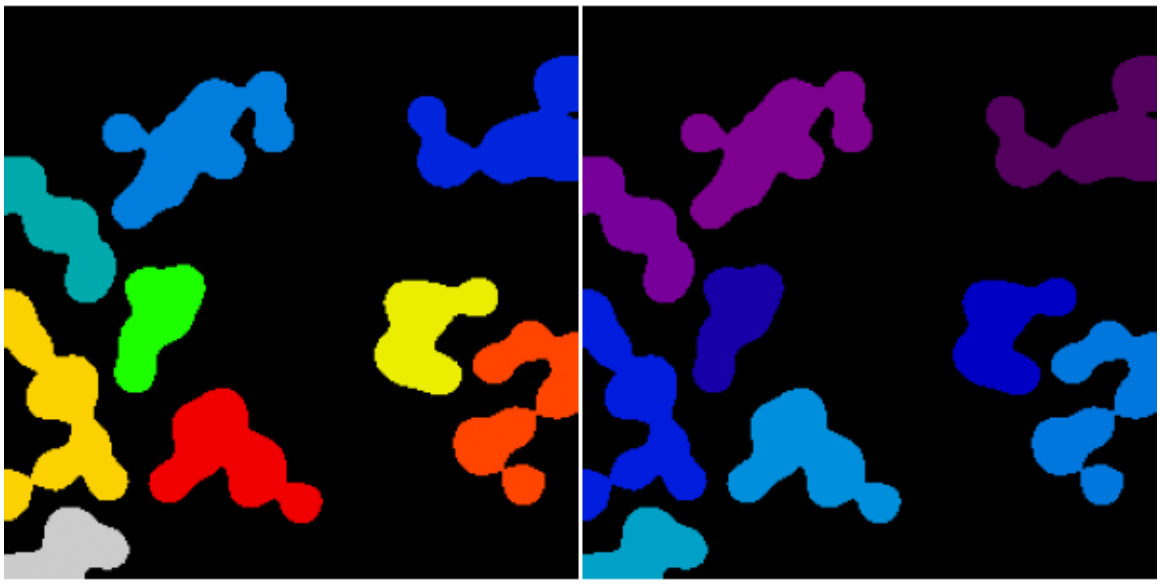
(continued from previous page)

```
plt.title("TV denoising", fontsize=20)

tv_denoised = denoise_tv_chambolle(noisy, weight=50)
plt.subplot(133)
plt.imshow(tv_denoised, cmap="gray", vmin=40, vmax=220)
plt.axis("off")
plt.title("(more) TV denoising", fontsize=20)

plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.9, bottom=0, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.236 seconds)

12.8.14 Measurements from images

This examples shows how to measure quantities from various images.



```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = im > im.mean()

label_im, nb_labels = sp.ndimage.label(mask)

sizes = sp.ndimage.sum(mask, label_im, range(nb_labels + 1))
mask_size = sizes < 1000
remove_pixel = mask_size[label_im]
```

(continues on next page)

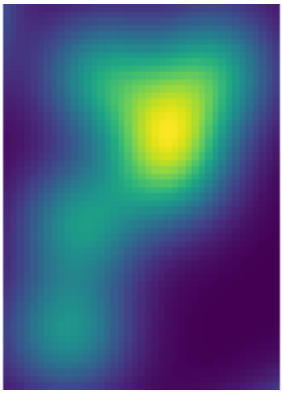
(continued from previous page)

```
label_im[remove_pixel] = 0
labels = np.unique(label_im)
label_clean = np.searchsorted(labels, label_im)


plt.figure(figsize=(6, 3))

plt.subplot(121)
plt.imshow(label_im, cmap="nipy_spectral")
plt.axis("off")
plt.subplot(122)
plt.imshow(label_clean, vmax=nb_labels, cmap="nipy_spectral")
plt.axis("off")

plt.subplots_adjust(wspace=0.01, hspace=0.01, top=1, bottom=0, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.043 seconds)

12.8.15 Find the bounding box of an object

This example shows how to extract the bounding box of the largest object



```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = im > im.mean()

label_im, nb_labels = sp.ndimage.label(mask)

# Find the largest connected component
sizes = sp.ndimage.sum(mask, label_im, range(nb_labels + 1))
mask_size = sizes < 1000
```

(continues on next page)

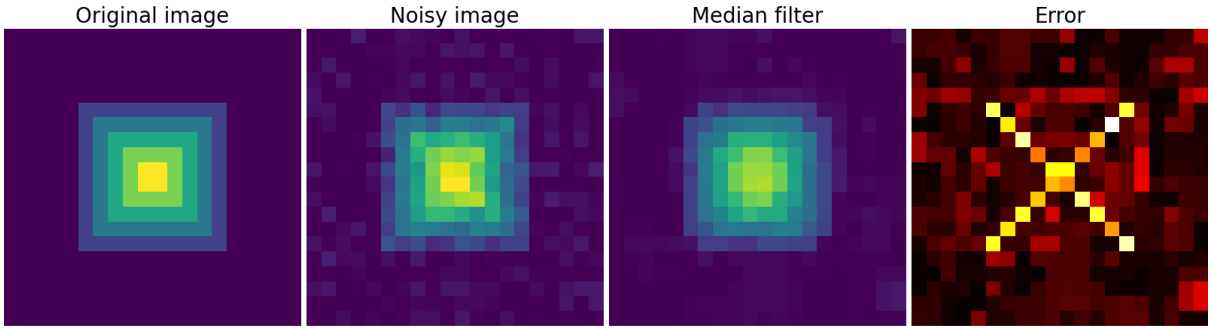
(continued from previous page)

```
remove_pixel = mask_size[label_im]
label_im[remove_pixel] = 0
labels = np.unique(label_im)
label_im = np.searchsorted(labels, label_im)

# Now that we have only one connected component, extract it's bounding box
slice_x, slice_y = sp.ndimage.find_objects(label_im == 4)[0]
roi = im[slice_x, slice_y]

plt.figure(figsize=(4, 2))
plt.axes((0, 0, 1, 1))
plt.imshow(roi)
plt.axis("off")

plt.show()
```

Total running time of the script: (0 minutes 0.018 seconds)

12.8.16 Denoising an image with the median filter

This example shows the original image, the noisy image, the denoised one (with the median filter) and the difference between the two.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

im = np.zeros((20, 20))
im[5:-5, 5:-5] = 1
im = sp.ndimage.distance_transform_bf(im)
im_noise = im + 0.2 * rng.normal(size=im.shape)

im_med = sp.ndimage.median_filter(im_noise, 3)

plt.figure(figsize=(16, 5))

plt.subplot(141)
plt.imshow(im, interpolation="nearest")
plt.axis("off")
plt.title("Original image", fontsize=20)
```

(continues on next page)

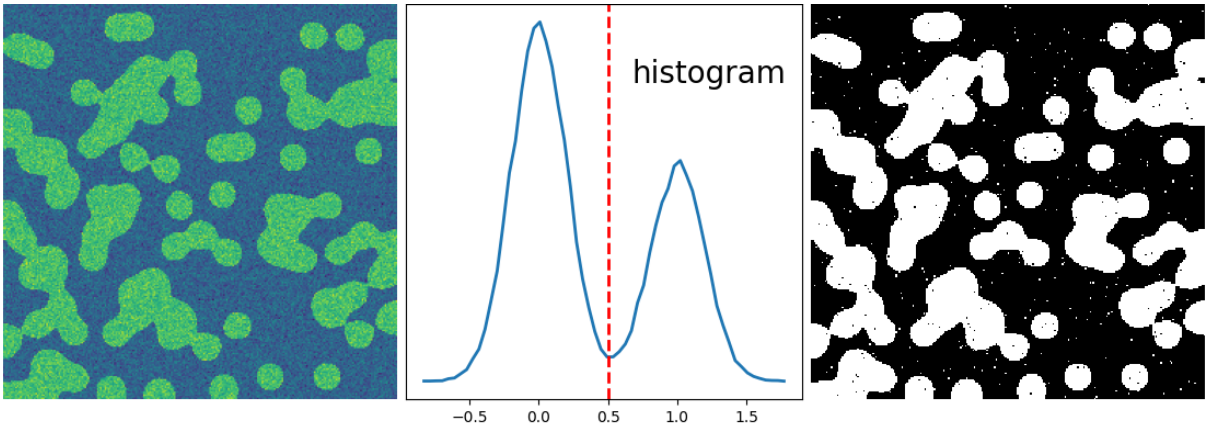
(continued from previous page)

```
plt.subplot(142)
plt.imshow(im_noise, interpolation="nearest", vmin=0, vmax=5)
plt.axis("off")
plt.title("Noisy image", fontsize=20)
plt.subplot(143)
plt.imshow(im_med, interpolation="nearest", vmin=0, vmax=5)
plt.axis("off")
plt.title("Median filter", fontsize=20)
plt.subplot(144)
plt.imshow(np.abs(im - im_med), cmap="hot", interpolation="nearest")
plt.axis("off")
plt.title("Error", fontsize=20)


plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.9, bottom=0, left=0, right=1)

plt.show()
```

Total running time of the script: (0 minutes 0.119 seconds)

12.8.17 Histogram segmentation

This example does simple histogram analysis to perform segmentation.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = (im > im.mean()).astype(float)

mask += 0.1 * im
```

(continues on next page)

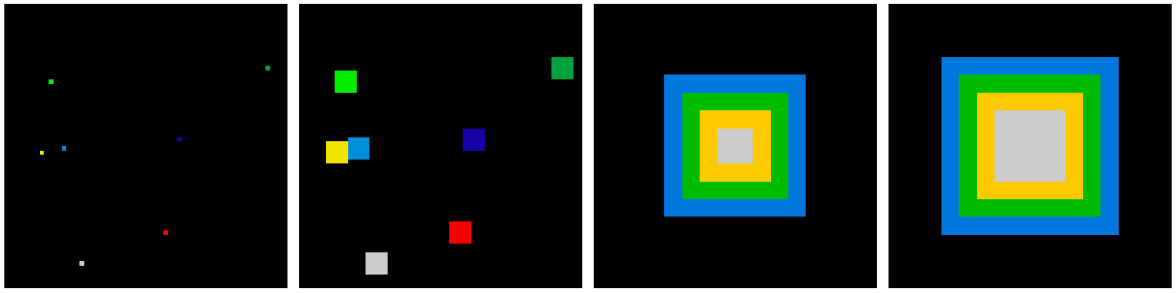
(continued from previous page)

```
img = mask + 0.2 * rng.normal(size=mask.shape)

hist, bin_edges = np.histogram(img, bins=60)
bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

binary_img = img > 0.5

plt.figure(figsize=(11, 4))

plt.subplot(131)
plt.imshow(img)
plt.axis("off")
plt.subplot(132)
plt.plot(bin_centers, hist, lw=2)
plt.axvline(0.5, color="r", ls="--", lw=2)
plt.text(0.57, 0.8, "histogram", fontsize=20, transform=plt.gca().transAxes)
plt.yticks([])
plt.subplot(133)
plt.imshow(binary_img, cmap="gray", interpolation="nearest")
plt.axis("off")

plt.subplots_adjust(wspace=0.02, hspace=0.3, top=1, bottom=0.1, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.104 seconds)

12.8.18 Greyscale dilation

This example illustrates greyscale mathematical morphology.



```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

im = np.zeros((64, 64))
rng = np.random.default_rng(27446968)
x, y = (63 * rng.random((2, 8))).astype(int)
im[x, y] = np.arange(8)

bigger_points = sp.ndimage.grey_dilation(im, size=(5, 5), structure=np.ones((5, 5)))

square = np.zeros((16, 16))
square[4:-4, 4:-4] = 1
dist = sp.ndimage.distance_transform_bf(square)
dilate_dist = sp.ndimage.grey_dilation(dist, size=(3, 3), structure=np.ones((3, 3)))
```

(continues on next page)

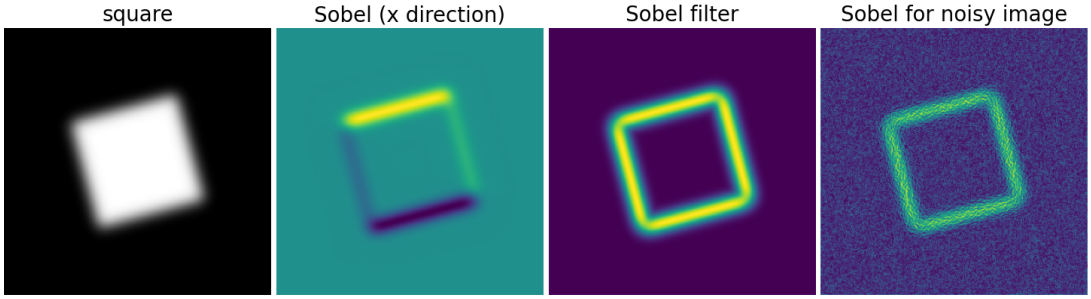
(continued from previous page)

```
plt.figure(figsize=(12.5, 3))
plt.subplot(141)
plt.imshow(im, interpolation="nearest", cmap="nipy_spectral")
plt.axis("off")
plt.subplot(142)
plt.imshow(bigger_points, interpolation="nearest", cmap="nipy_spectral")
plt.axis("off")
plt.subplot(143)
plt.imshow(dist, interpolation="nearest", cmap="nipy_spectral")
plt.axis("off")
plt.subplot(144)
plt.imshow(dilate_dist, interpolation="nearest", cmap="nipy_spectral")
plt.axis("off")

plt.subplots_adjust(wspace=0, hspace=0.02, top=0.99, bottom=0.01, left=0.01, right=0.
↪99)
plt.show()
```

Total running time of the script: (0 minutes 0.048 seconds)

12.8.19 Finding edges with Sobel filters

The Sobel filter is one of the simplest way of finding edges.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

im = np.zeros((256, 256))
im[64:-64, 64:-64] = 1

im = sp.ndimage.rotate(im, 15, mode="constant")
im = sp.ndimage.gaussian_filter(im, 8)

sx = sp.ndimage.sobel(im, axis=0, mode="constant")
sy = sp.ndimage.sobel(im, axis=1, mode="constant")
sob = np.hypot(sx, sy)

plt.figure(figsize=(16, 5))
```

(continues on next page)

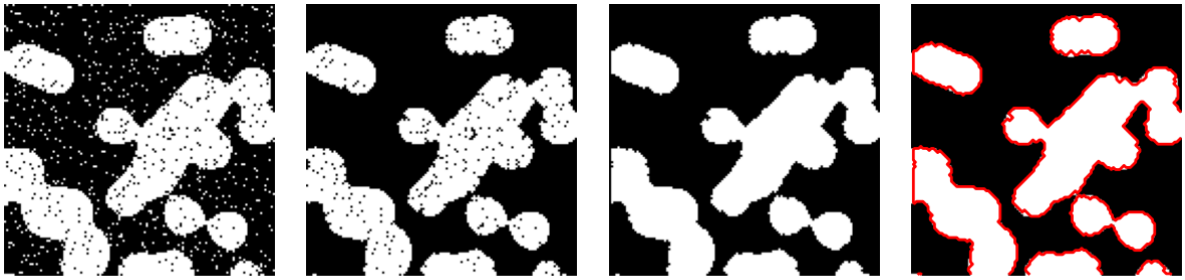
(continued from previous page)

```
plt.subplot(141)
plt.imshow(im, cmap="gray")
plt.axis("off")
plt.title("square", fontsize=20)
plt.subplot(142)
plt.imshow(sx)
plt.axis("off")
plt.title("Sobel (x direction)", fontsize=20)
plt.subplot(143)
plt.imshow(sob)
plt.axis("off")
plt.title("Sobel filter", fontsize=20)

im += 0.07 * rng.random(im.shape)

sx = sp.ndimage.sobel(im, axis=0, mode="constant")
sy = sp.ndimage.sobel(im, axis=1, mode="constant")
sob = np.hypot(sx, sy)

plt.subplot(144)
plt.imshow(sob)
plt.axis("off")
plt.title("Sobel for noisy image", fontsize=20)


plt.subplots_adjust(wspace=0.02, hspace=0.02, top=1, bottom=0, left=0, right=0.9)

plt.show()
```

Total running time of the script: (0 minutes 0.204 seconds)

12.8.20 Cleaning segmentation with mathematical morphology

An example showing how to clean segmentation with mathematical morphology: removing small regions and holes.



```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
```

(continues on next page)

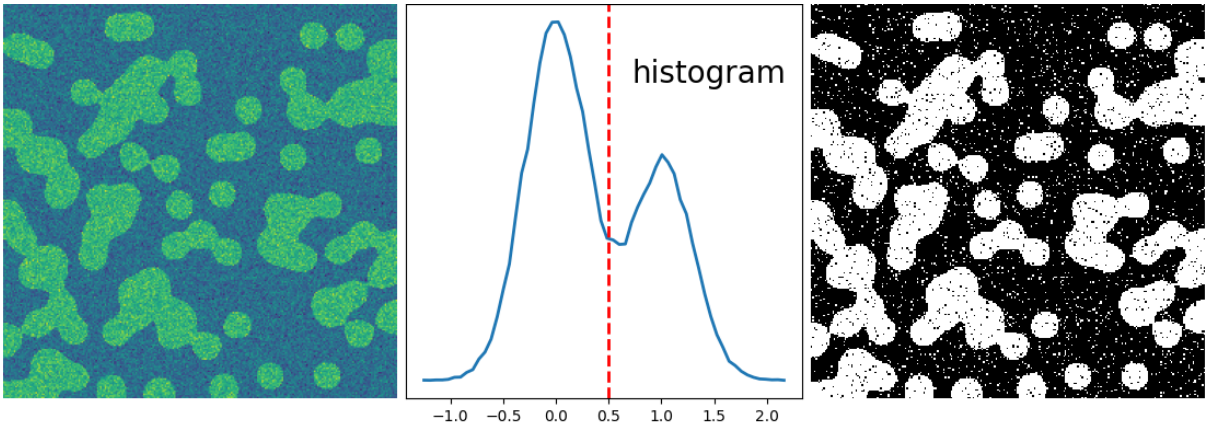
(continued from previous page)

```
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = (im > im.mean()).astype(float)


img = mask + 0.3 * rng.normal(size=mask.shape)

binary_img = img > 0.5

# Remove small white regions
open_img = sp.ndimage.binary_opening(binary_img)
# Remove small black hole
close_img = sp.ndimage.binary_closing(open_img)

plt.figure(figsize=(12, 3))

l = 128

plt.subplot(141)
plt.imshow(binary_img[:l, :l], cmap="gray")
plt.axis("off")
plt.subplot(142)
plt.imshow(open_img[:l, :l], cmap="gray")
plt.axis("off")
plt.subplot(143)
plt.imshow(close_img[:l, :l], cmap="gray")
plt.axis("off")
plt.subplot(144)
plt.imshow(mask[:l, :l], cmap="gray")
plt.contour(close_img[:l, :l], [0.5], linewidths=2, colors="r")
plt.axis("off")

plt.subplots_adjust(wspace=0.02, hspace=0.3, top=1, bottom=0.1, left=0, right=1)

plt.show()
```

Total running time of the script: (0 minutes 0.077 seconds)

12.8.21 Segmentation with Gaussian mixture models

This example performs a Gaussian mixture model analysis of the image histogram to find the right thresholds for separating foreground from background.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture

rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = (im > im.mean()).astype(float)


img = mask + 0.3 * rng.normal(size=mask.shape)

hist, bin_edges = np.histogram(img, bins=60)
bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

classif = GaussianMixture(n_components=2)
classif.fit(img.reshape((img.size, 1)))

threshold = np.mean(classif.means_)
binary_img = img > threshold


plt.figure(figsize=(11, 4))

plt.subplot(131)
plt.imshow(img)
plt.axis("off")
plt.subplot(132)
plt.plot(bin_centers, hist, lw=2)
plt.axvline(0.5, color="r", ls="--", lw=2)
plt.text(0.57, 0.8, "histogram", fontsize=20, transform=plt.gca().transAxes)
plt.yticks([])
plt.subplot(133)
plt.imshow(binary_img, cmap="gray", interpolation="nearest")
plt.axis("off")

plt.subplots_adjust(wspace=0.02, hspace=0.3, top=1, bottom=0.1, left=0, right=1)
```

(continues on next page)

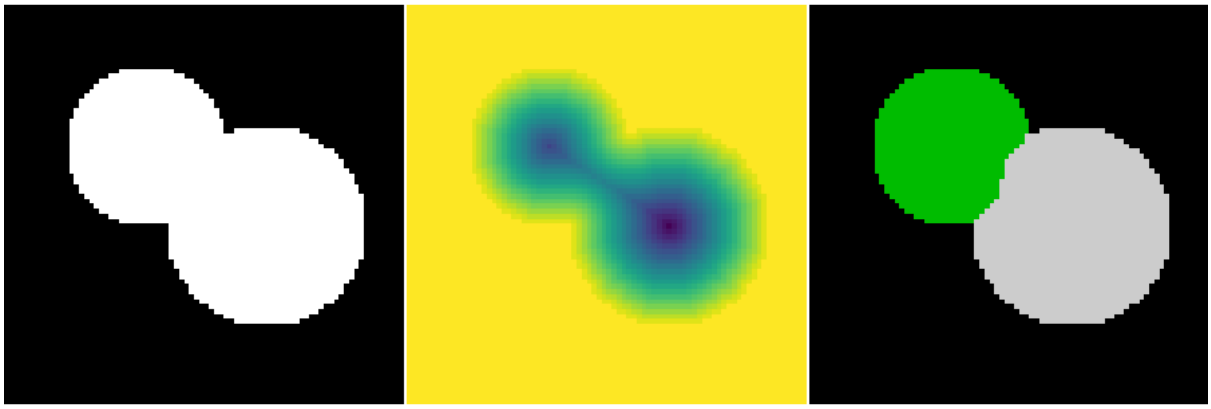
(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 0.736 seconds)

12.8.22 Watershed segmentation

This example shows how to do segmentation with watershed.



```
import numpy as np
from skimage.segmentation import watershed
from skimage.feature import peak_local_max
import matplotlib.pyplot as plt
import scipy as sp

# Generate an initial image with two overlapping circles
x, y = np.indices((80, 80))
x1, y1, x2, y2 = 28, 28, 44, 52
r1, r2 = 16, 20
mask_circle1 = (x - x1) ** 2 + (y - y1) ** 2 < r1**2
mask_circle2 = (x - x2) ** 2 + (y - y2) ** 2 < r2**2
image = np.logical_or(mask_circle1, mask_circle2)
# Now we want to separate the two objects in image
# Generate the markers as local maxima of the distance
# to the background
distance = sp.ndimage.distance_transform_edt(image)
peak_idx = peak_local_max(distance, footprint=np.ones((3, 3)), labels=image)
peak_mask = np.zeros_like(distance, dtype=bool)
peak_mask[tuple(peak_idx.T)] = True
markers = sp.ndimage.label(peak_mask)[0]
labels = watershed(-distance, markers, mask=image)

plt.figure(figsize=(9, 3.5))
plt.subplot(131)
plt.imshow(image, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(132)
plt.imshow(-distance, interpolation="nearest")
plt.axis("off")
plt.subplot(133)
```

(continues on next page)

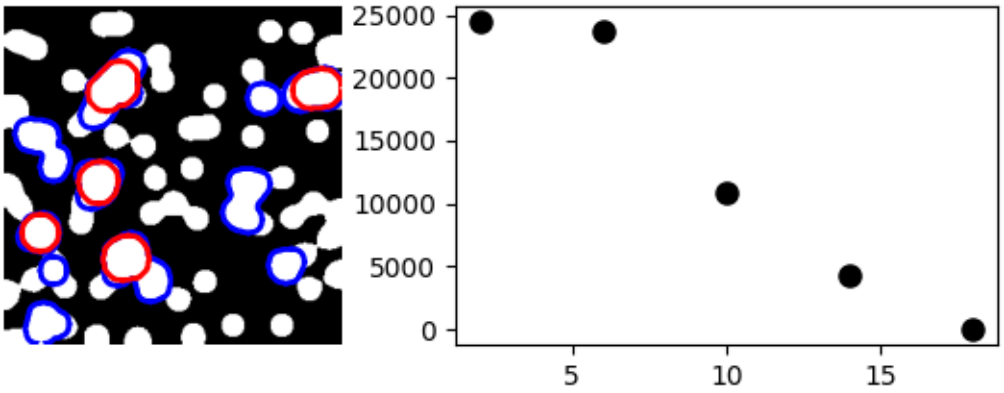
(continued from previous page)

```
plt.imshow(labels, cmap="nipy_spectral", interpolation="nearest")
plt.axis("off")

plt.subplots_adjust(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0, right=1)
plt.show()
```

Total running time of the script: (0 minutes 0.061 seconds)

12.8.23 Granulometry

This example performs a simple granulometry analysis.




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt


def disk_structure(n):
    struct = np.zeros((2 * n + 1, 2 * n + 1))
    x, y = np.indices((2 * n + 1, 2 * n + 1))
    mask = (x - n) ** 2 + (y - n) ** 2 <= n**2
    struct[mask] = 1
    return struct.astype(bool)


def granulometry(data, sizes=None):
    s = max(data.shape)
    if sizes is None:
        sizes = range(1, s / 2, 2)
    granulo = [
        sp.ndimage.binary_opening(data, structure=disk_structure(n)).sum()
        for n in sizes
    ]
    return granulo


rng = np.random.default_rng(27446968)
n = 10
l = 256
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
```

(continues on next page)

(continued from previous page)

```
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = sp.ndimage.gaussian_filter(im, sigma=l / (4.0 * n))

mask = im > im.mean()

granulo = granulometry(mask, sizes=np.arange(2, 19, 4))

plt.figure(figsize=(6, 2.2))

plt.subplot(121)
plt.imshow(mask, cmap="gray")
opened = sp.ndimage.binary_opening(mask, structure=disk_structure(10))
opened_more = sp.ndimage.binary_opening(mask, structure=disk_structure(14))
plt.contour(opened, [0.5], colors="b", linewidths=2)
plt.contour(opened_more, [0.5], colors="r", linewidths=2)
plt.axis("off")
plt.subplot(122)
plt.plot(np.arange(2, 19, 4), granulo, "ok", ms=8)


plt.subplots_adjust(wspace=0.02, hspace=0.15, top=0.95, bottom=0.15, left=0, right=0.
↪95)
plt.show()
```

Total running time of the script: (0 minutes 0.244 seconds)

12.8.24 Segmentation with spectral clustering

This example uses spectral clustering to do segmentation.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering
```

```
l = 100
x, y = np.indices((l, l))

center1 = (28, 24)
center2 = (40, 50)
center3 = (67, 58)
center4 = (24, 70)

radius1, radius2, radius3, radius4 = 16, 14, 15, 14

circle1 = (x - center1[0]) ** 2 + (y - center1[1]) ** 2 < radius1**2
circle2 = (x - center2[0]) ** 2 + (y - center2[1]) ** 2 < radius2**2
circle3 = (x - center3[0]) ** 2 + (y - center3[1]) ** 2 < radius3**2
circle4 = (x - center4[0]) ** 2 + (y - center4[1]) ** 2 < radius4**2
```

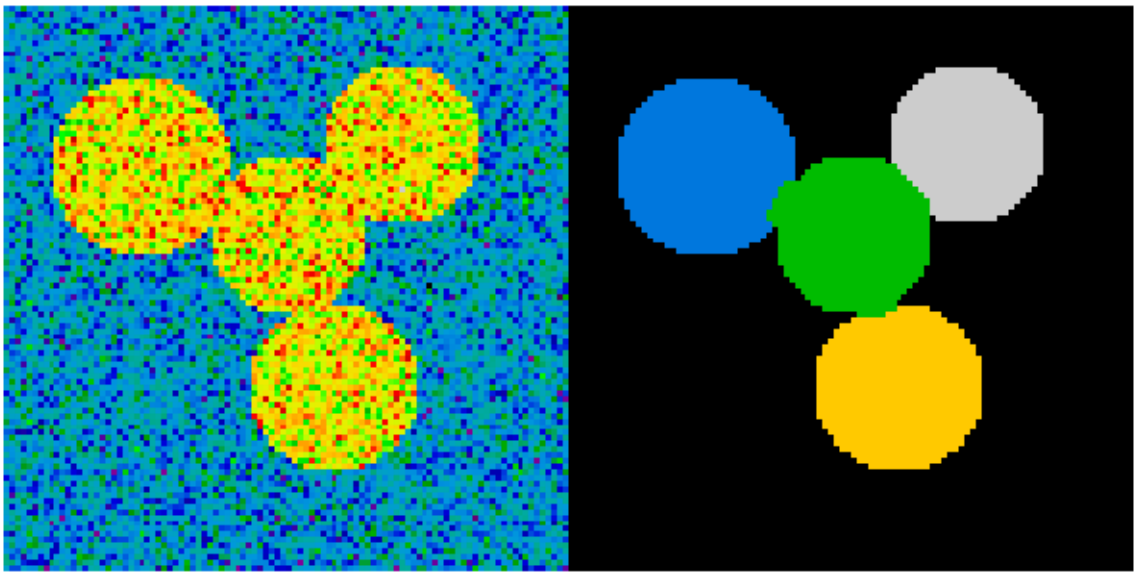
4 circles

```
img = circle1 + circle2 + circle3 + circle4
mask = img.astype(bool)
```

(continues on next page)

(continued from previous page)

```
img = img.astype(float)

rng = np.random.default_rng(27446968)
img += 1 + 0.2 * rng.normal(size=img.shape)

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(img, mask=mask)

# Take a decreasing function of the gradient: we take it weakly
# dependent from the gradient the segmentation is close to a voronoi
graph.data = np.exp(-graph.data / graph.data.std())

# Force the solver to be arpack, since amg is numerically
# unstable on this example
labels = spectral_clustering(graph, n_clusters=4)
label_im = -np.ones(mask.shape)
label_im[mask] = labels

plt.figure(figsize=(6, 3))
plt.subplot(121)
plt.imshow(img, cmap="nipy_spectral", interpolation="nearest")
plt.axis("off")
plt.subplot(122)
plt.imshow(label_im, cmap="nipy_spectral", interpolation="nearest")
plt.axis("off")

plt.subplots_adjust(wspace=0, hspace=0.0, top=0.99, bottom=0.01, left=0.01, right=0.
↪99)
plt.show()
```



Total running time of the script: (0 minutes 0.264 seconds)

See also

More on image-processing:

- The chapter on *Scikit-image*
- Other, more powerful and complete modules: OpenCV (Python bindings), CellProfiler, ITK with Python bindings

CHAPTER 13

Mathematical optimization: finding minima of functions

Authors: *Gaël Varoquaux*

Mathematical optimization deals with the problem of finding numerically minimums (or maximums or zeros) of a function. In this context, the function is called *cost function*, or *objective function*, or *energy*.

Here, we are interested in using `scipy.optimize` for black-box optimization: we do not rely on the mathematical expression of the function that we are optimizing. Note that this expression can often be used for more efficient, non black-box, optimization.

Prerequisites

- *NumPy*
- *SciPy*
- *Matplotlib*

See also

References

Mathematical optimization is very . . . mathematical. If you want performance, it really pays to read the books:

- Convex Optimization by Boyd and Vandenberghe (pdf available free online).
- Numerical Optimization, by Nocedal and Wright. Detailed reference on gradient descent methods.

- Practical Methods of Optimization by Fletcher: good at hand-waving explanations.

Chapters contents

- *Knowing your problem*
  - *Convex versus non-convex optimization*
  - *Smooth and non-smooth problems*
  - *Noisy versus exact cost functions*
  - *Constraints*
- *A review of the different optimizers*
  - *Getting started: 1D optimization*
  - *Gradient based methods*
  - *Newton and quasi-newton methods*
- *Full code examples*
- *Examples for the mathematical optimization chapter*
  - *Gradient-less methods*
  - *Global optimizers*
- *Practical guide to optimization with SciPy*
  - *Choosing a method*
  - *Making your optimizer faster*
  - *Computing gradients*
  - *Synthetic exercises*
- *Special case: non-linear least-squares*
  - *Minimizing the norm of a vector function*
  - *Curve fitting*
- *Optimization with constraints*
  - *Box bounds*
  - *General constraints*
- *Full code examples*
- *Examples for the mathematical optimization chapter*

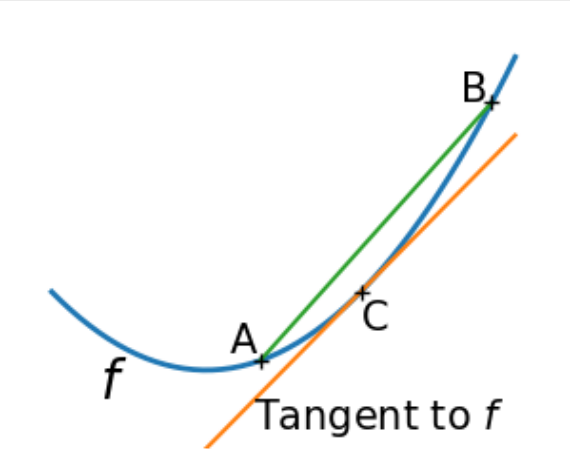
13.1 Knowing your problem

Not all optimization problems are equal. Knowing your problem enables you to choose the right tool.

Dimensionality of the problem

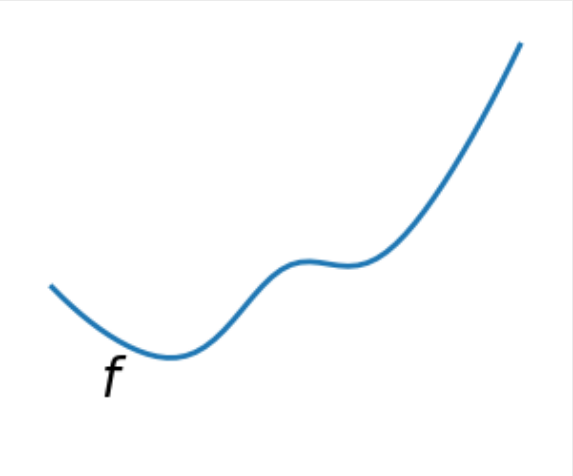
The scale of an optimization problem is pretty much set by the *dimensionality of the problem*, i.e. the number of scalar variables on which the search is performed.

13.1.1 Convex versus non-convex optimization



A convex function:

- f is above all its tangents.
- equivalently, for two point A, B, f(C) lies below the segment [f(A), f(B])], if A < C < B

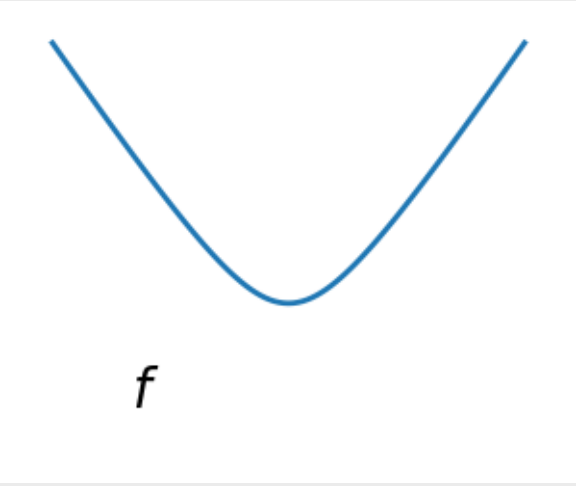


A non-convex function

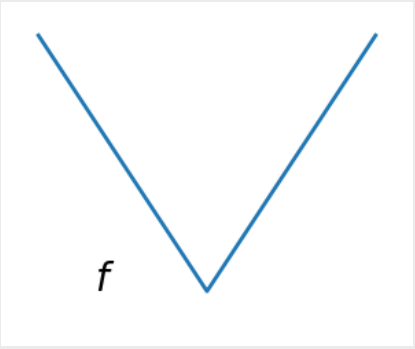
Optimizing convex functions is easy. Optimizing non-convex functions can be very hard.

> **Note**
>
> It can be proven that for a convex function a local minimum is also a global minimum. Then, in some sense, the minimum is unique.

13.1.2 Smooth and non-smooth problems



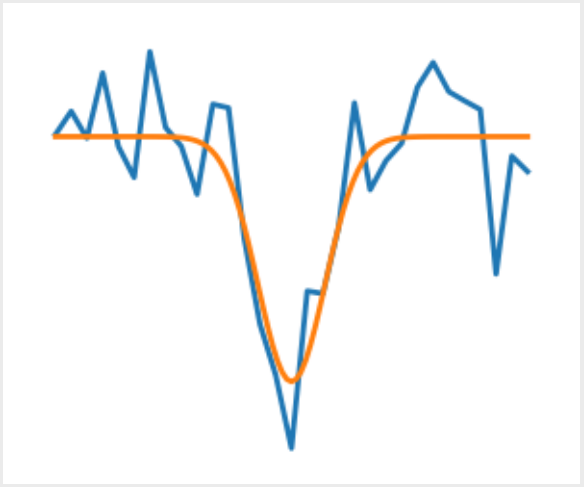
A smooth function:
The gradient is defined everywhere, and is a continuous function



A non-smooth function

Optimizing smooth functions is easier (true in the context of *black-box* optimization, otherwise Linear Programming is an example of methods which deal very efficiently with piece-wise linear functions).

13.1.3 Noisy versus exact cost functions

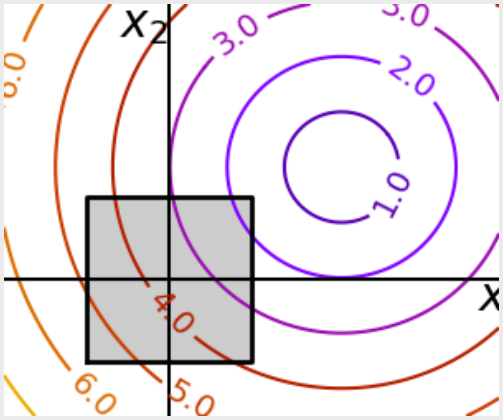


Noisy (blue) and non-noisy (green) functions

Noisy gradients

Many optimization methods rely on gradients of the objective function. If the gradient function is not given, they are computed numerically, which induces errors. In such situation, even if the objective function is not noisy, a gradient-based optimization may be a noisy optimization.

13.1.4 Constraints



Optimizations under constraints
Here:
$-1 < x_1 < 1$
$-1 < x_2 < 1$

13.2 A review of the different optimizers

13.2.1 Getting started: 1D optimization

Let's get started by finding the minimum of the scalar function $f(x) = \exp[(x-0.5)^2]$. `scipy.optimize.minimize_scalar()` uses Brent's method to find the minimum of a function:

```
>>> import numpy as np
>>> import scipy as sp
>>> def f(x):
```

(continues on next page)

(continued from previous page)

```
...     return -np.exp(-(x - 0.5)**2)
>>> result = sp.optimize.minimize_scalar(f)
>>> result.success # check if solver was successful
True
>>> x_min = result.x
>>> x_min
np.float64(0.50...)
>>> x_min - 0.5
np.float64(5.8...e-09)
```

Table 1: **Brent's method on a quadratic function**: it converges in 3 iterations, as the quadratic approximation is then exact.

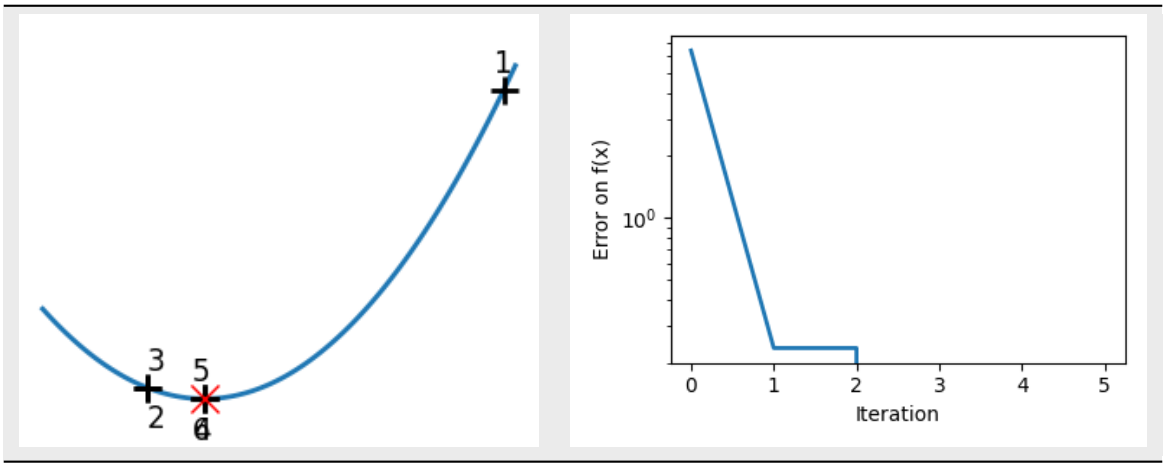
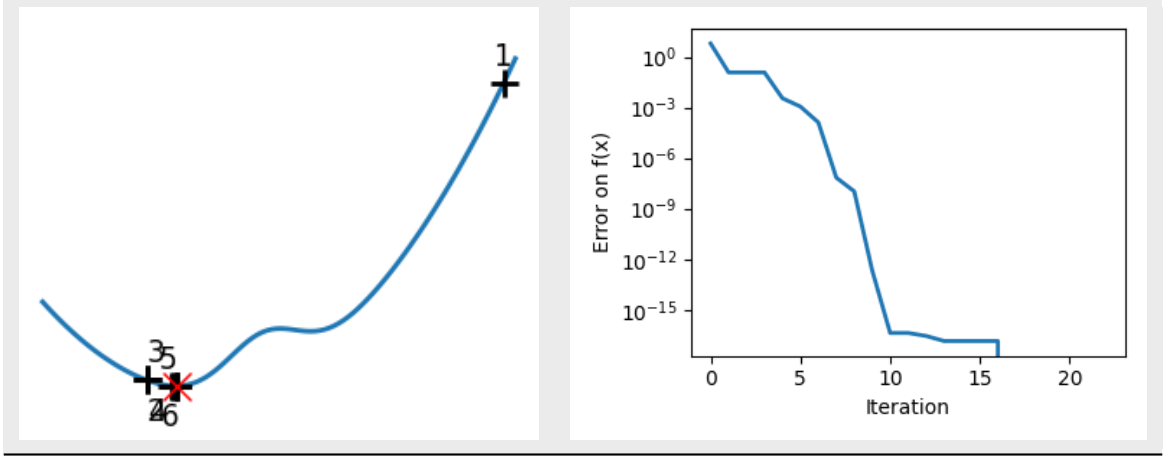


Table 2: **Brent's method on a non-convex function**: note that the fact that the optimizer avoided the local minimum is a matter of luck.



Note

You can use different solvers using the parameter `method`.

Note

`scipy.optimize.minimize_scalar()` can also be used for optimization constrained to an interval using the parameter `bounds`.

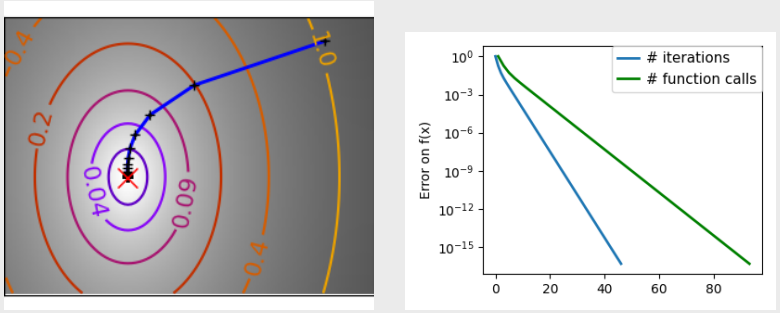
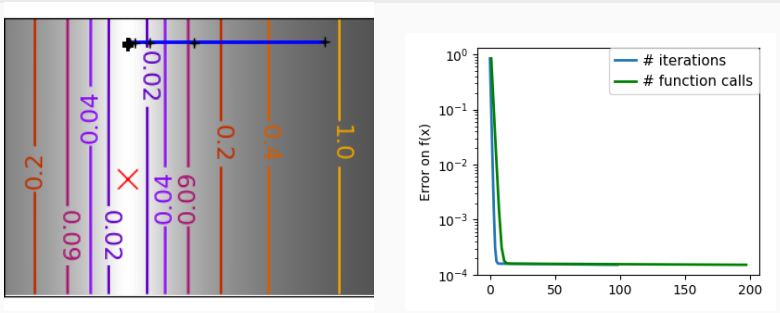
13.2.2 Gradient based methods

Some intuitions about gradient descent

Here we focus on **intuitions**, not code. Code will follow.

Gradient descent basically consists in taking small steps in the direction of the gradient, that is the direction of the *steepest descent*.

Table 3: **Fixed step gradient descent**

A well-conditioned quadratic function.	
An ill-conditioned quadratic function. The core problem of gradient-methods on ill-conditioned problems is that the gradient tends not to point in the direction of the minimum.	

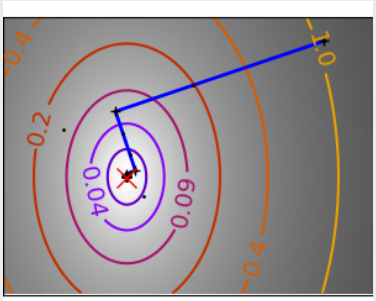
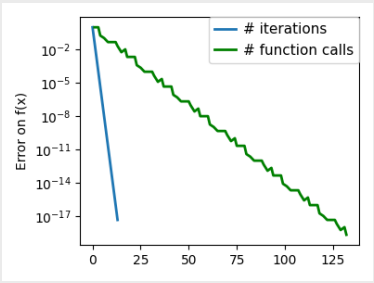
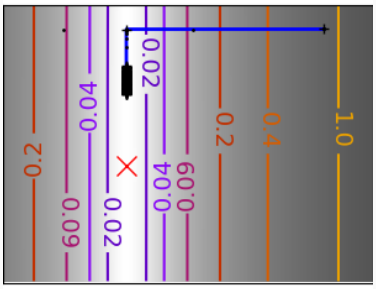
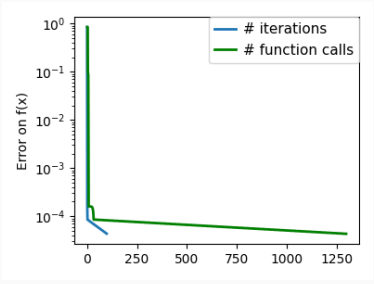
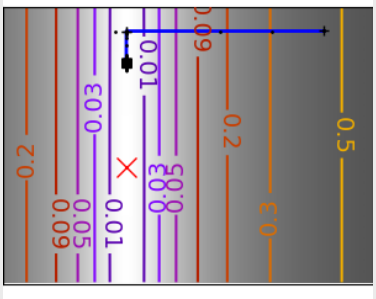
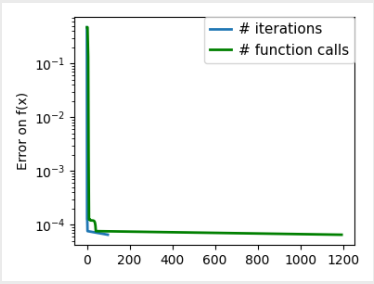
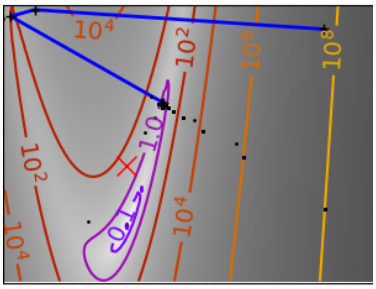
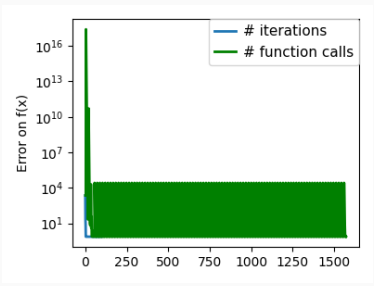
We can see that very anisotropic (ill-conditioned) functions are harder to optimize.

Take home message: conditioning number and preconditioning

If you know natural scaling for your variables, prescale them so that they behave similarly. This is related to preconditioning.

Also, it clearly can be advantageous to take bigger steps. This is done in gradient descent code using a line search.

Table 4: **Adaptive step gradient descent**

A well-conditioned quadratic function.		
An ill-conditioned quadratic function.		
An ill-conditioned non-quadratic function.		
An ill-conditioned very non-quadratic function.		

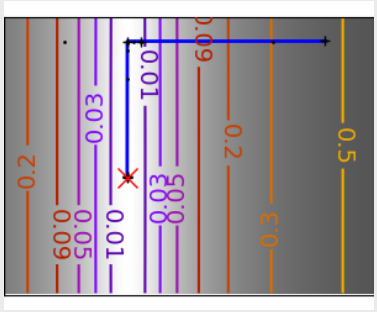
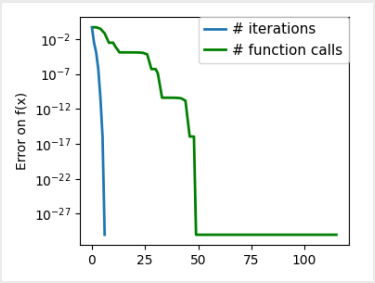
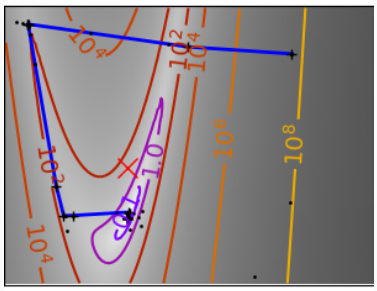
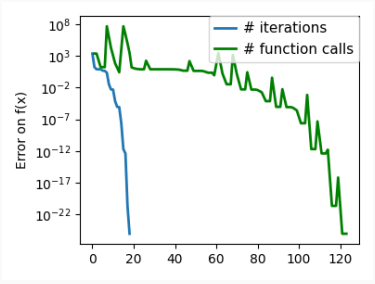
The more a function looks like a quadratic function (elliptic iso-curves), the easier it is to optimize.

Conjugate gradient descent

The gradient descent algorithms above are toys not to be used on real problems.

As can be seen from the above experiments, one of the problems of the simple gradient descent algorithms, is that it tends to oscillate across a valley, each time following the direction of the gradient, that makes it cross the valley. The conjugate gradient solves this problem by adding a *friction* term: each step depends on the two last values of the gradient and sharp turns are reduced.

Table 5: **Conjugate gradient descent**

An ill-conditioned non-quadratic function.		
An ill-conditioned very non-quadratic function.		

SciPy provides `scipy.optimize.minimize()` to find the minimum of scalar functions of one or more variables. The simple conjugate gradient method can be used by setting the parameter `method` to CG

```
>>> def f(x):    # The rosenbrock function
...     return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2
>>> sp.optimize.minimize(f, [2, -1], method="CG")
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 1.650...e-11
       x: [ 1.000e+00  1.000e+00]
     nit: 13
     jac: [-6.15...e-06  2.53...e-07]
    nfev: 81
    njev: 27
```

Gradient methods need the Jacobian (gradient) of the function. They can compute it numerically, but will perform better if you can pass them the gradient:

```
>>> def jacobian(x):
...     return np.array((-2*.5*(1 - x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] -␣
↪x[0]**2)))
>>> sp.optimize.minimize(f, [2, 1], method="CG", jac=jacobian)
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 2.95786...e-14
       x: [ 1.000e+00  1.000e+00]
     nit: 8
     jac: [ 7.183e-07 -2.990e-07]
    nfev: 16
    njev: 16
```

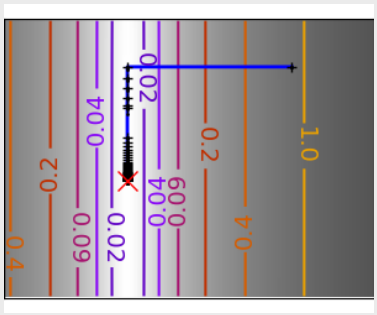
Note that the function has only been evaluated 27 times, compared to 108 without the gradient.

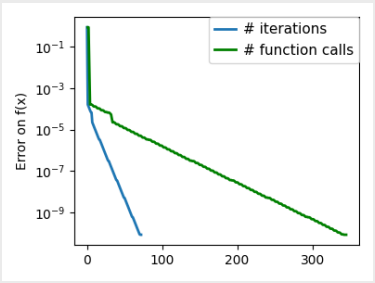
13.2.3 Newton and quasi-newton methods

Newton methods: using the Hessian (2nd differential)

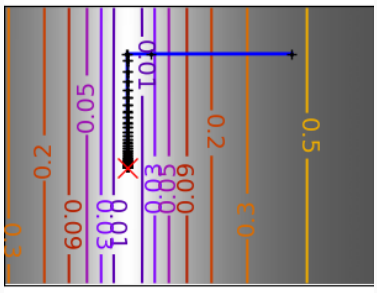
Newton methods use a local quadratic approximation to compute the jump direction. For this purpose, they rely on the 2 first derivative of the function: the *gradient* and the Hessian.

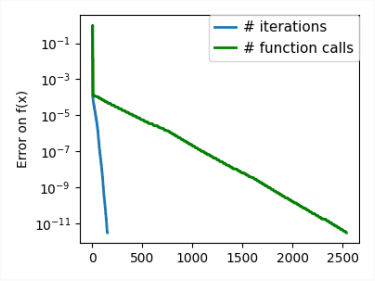
An ill-conditioned quadratic function:
Note that, as the quadratic approximation is exact, the Newton method is blazing fast



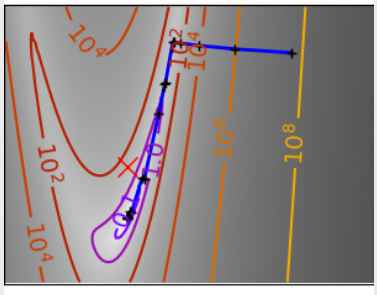


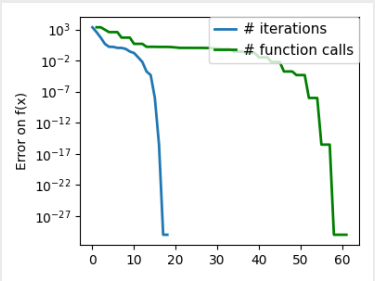
An ill-conditioned non-quadratic function:
Here we are optimizing a Gaussian, which is always below its quadratic approximation. As a result, the Newton method overshoots and leads to oscillations.





An ill-conditioned very non-quadratic function:





In SciPy, you can use the Newton method by setting `method` to Newton-CG in `scipy.optimize.minimize()`. Here, CG refers to the fact that an internal inversion of the Hessian is performed by conjugate gradient

```
>>> def f(x):    # The rosenbrock function
...     return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2
```

(continues on next page)

(continued from previous page)

```
>>> def jacobian(x):
...     return np.array((-2*.5*(1 - x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] -␣
↪x[0]**2)))
>>> sp.optimize.minimize(f, [2,-1], method="Newton-CG", jac=jacobian)
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 1.5601357400786612e-15
       x: [ 1.000e+00  1.000e+00]
     nit: 10
     jac: [ 1.058e-07 -7.483e-08]
    nfev: 11
    njev: 33
    nhev: 0
```

Note that compared to a conjugate gradient (above), Newton's method has required less function evaluations, but more gradient evaluations, as it uses it to approximate the Hessian. Let's compute the Hessian and pass it to the algorithm:

```
>>> def hessian(x): # Computed with sympy
...     return np.array(((1 - 4*x[1] + 12*x[0]**2, -4*x[0]), (-4*x[0], 2)))
>>> sp.optimize.minimize(f, [2,-1], method="Newton-CG", jac=jacobian, hess=hessian)
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 1.6277298383706738e-15
       x: [ 1.000e+00  1.000e+00]
     nit: 10
     jac: [ 1.110e-07 -7.781e-08]
    nfev: 11
    njev: 11
    nhev: 10
```

Note

At very high-dimension, the inversion of the Hessian can be costly and unstable (large scale > 250).

Note

Newton optimizers should not to be confused with Newton's root finding method, based on the same principles, `scipy.optimize.newton()`.

Quasi-Newton methods: approximating the Hessian on the fly

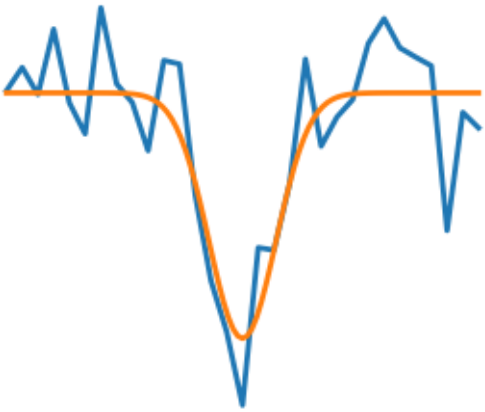
BFGS: BFGS (Broyden-Fletcher-Goldfarb-Shanno algorithm) refines at each step an approximation of the Hessian.

13.3 Full code examples

13.4 Examples for the mathematical optimization chapter

13.4.1 Noisy optimization problem

Draws a figure explaining noisy vs non-noisy optimization

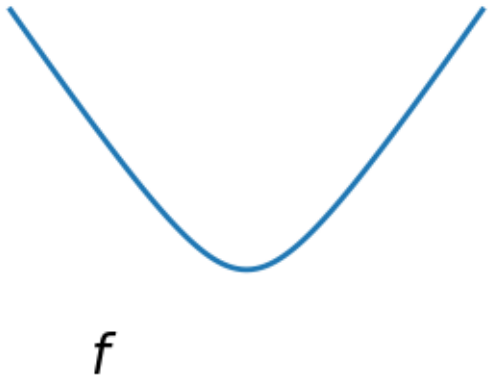


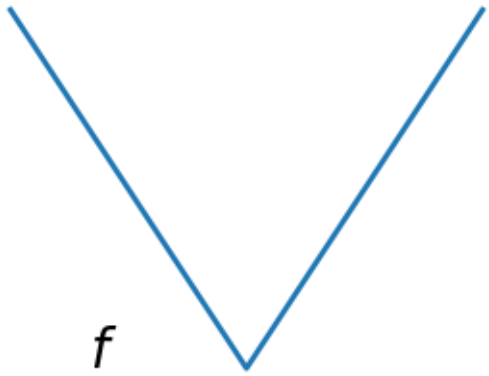
```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

x = np.linspace(-5, 5, 101)
x_ = np.linspace(-5, 5, 31)


def f(x):
    return -np.exp(-(x**2))


# A smooth function
plt.figure(1, figsize=(3, 2.5))
plt.clf()

plt.plot(x_, f(x_) + 0.2 * np.random.normal(size=31), linewidth=2)
plt.plot(x, f(x), linewidth=2)

plt.ylim(ymin=-1.3)
plt.axis("off")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.015 seconds)

13.4.2 Smooth vs non-smooth

Draws a figure to explain smooth versus non smooth optimization.



- 

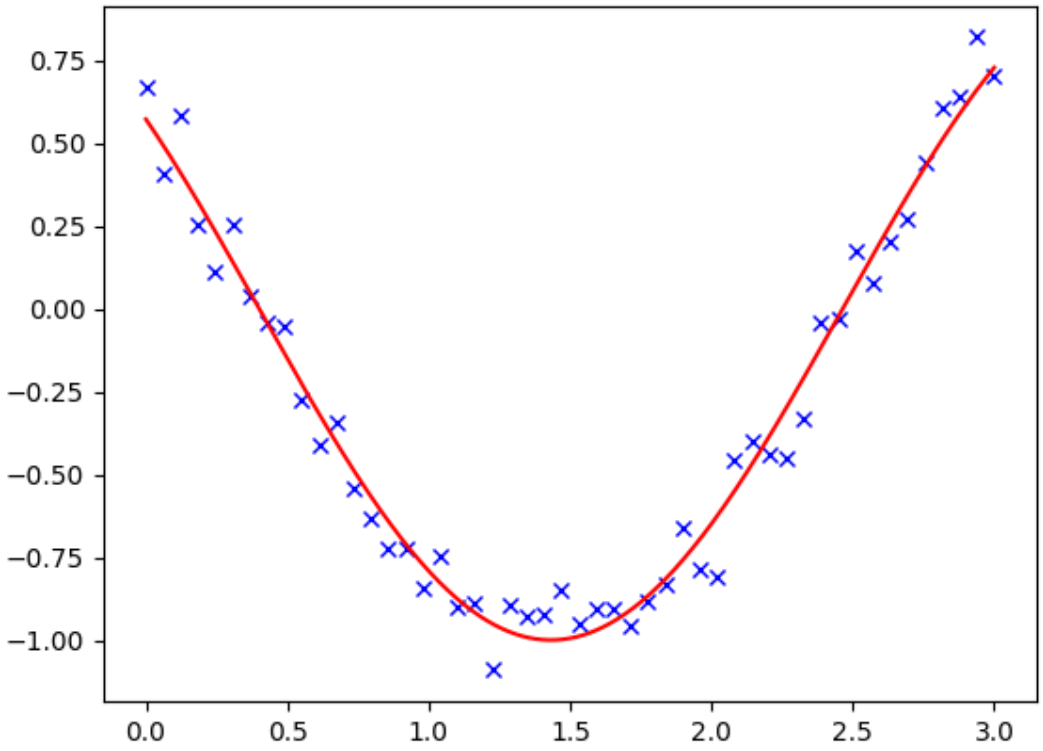
-

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1.5, 1.5, 101)

# A smooth function
plt.figure(1, figsize=(3, 2.5))
plt.clf()

plt.plot(x, np.sqrt(0.2 + x**2), linewidth=2)
plt.text(-1, 0, "$f$", size=20)

plt.ylim(ymin=-0.2)
plt.axis("off")
plt.tight_layout()

# A non-smooth function
plt.figure(2, figsize=(3, 2.5))
plt.clf()
plt.plot(x, np.abs(x), linewidth=2)
plt.text(-1, 0, "$f$", size=20)

plt.ylim(ymin=-0.2)
plt.axis("off")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.038 seconds)

13.4.3 Curve fitting

A curve fitting example




```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)


# Our test function
def f(t, omega, phi):
    return np.cos(omega * t + phi)


# Our x and y data
x = np.linspace(0, 3, 50)
y = f(x, 1.5, 1) + 0.1 * np.random.normal(size=50)

# Fit the model: the parameters omega and phi can be found in the
# `params` vector
params, params_cov = sp.optimize.curve_fit(f, x, y)

# plot the data and the fitted curve
```

(continues on next page)

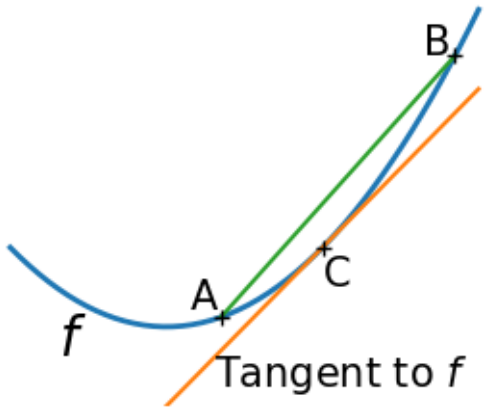
(continued from previous page)

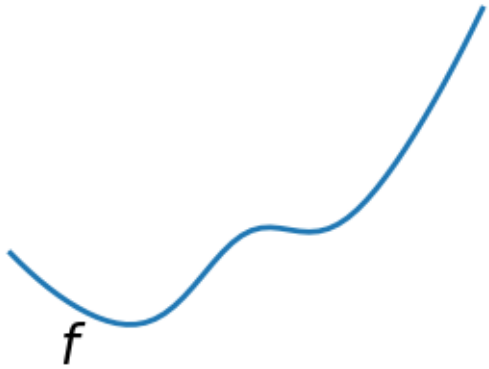
```
t = np.linspace(0, 3, 1000)

plt.figure(1)
plt.clf()
plt.plot(x, y, "bx")
plt.plot(t, f(t, *params), "r-")
plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

13.4.4 Convex function

A figure showing the definition of a convex function

- 

- 


```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1, 2)

plt.figure(1, figsize=(3, 2.5))
plt.clf()

# A convex function
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, x**2, linewidth=2)
plt.text(-0.7, -(0.6**2), "$f$", size=20)

# The tangent in one point
plt.plot(x, 2 * x - 1)
plt.plot(1, 1, "k+")
plt.text(0.3, -0.75, "Tangent to $f$", size=15)
plt.text(1, 1 - 0.5, "C", size=15)

# Convexity as barycenter
plt.plot([0.35, 1.85], [0.35**2, 1.85**2])
plt.plot([0.35, 1.85], [0.35**2, 1.85**2], "k+")
plt.text(0.35 - 0.2, 0.35**2 + 0.1, "A", size=15)
plt.text(1.85 - 0.2, 1.85**2, "B", size=15)

plt.ylim(ymin=-1)
plt.axis("off")
plt.tight_layout()

# Convexity as barycenter
plt.figure(2, figsize=(3, 2.5))
plt.clf()
plt.plot(x, x**2 + np.exp(-5 * (x - 0.5) ** 2), linewidth=2)
plt.text(-0.7, -(0.6**2), "$f$", size=20)

plt.ylim(ymin=-1)
plt.axis("off")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.045 seconds)

13.4.5 Finding a minimum in a flat neighborhood

An exercise of finding minimum. This exercise is hard because the function is very flat around the minimum (all its derivatives are zero). Thus gradient information is unreliable.

The function admits a minimum in [0, 0]. The challenge is to get within 1e-7 of this minimum, starting at x0 = [1, 1].

The solution that we adopt here is to give up on using gradient or information based on local differences, and to rely on the Powell algorithm. With 162 function evaluations, we get to 1e-8 of the solution.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt


def f(x):
    return np.exp(-1 / (0.01 * x[0] ** 2 + x[1] ** 2))


# A well-conditionned version of f:
def g(x):
    return f([10 * x[0], x[1]])
```

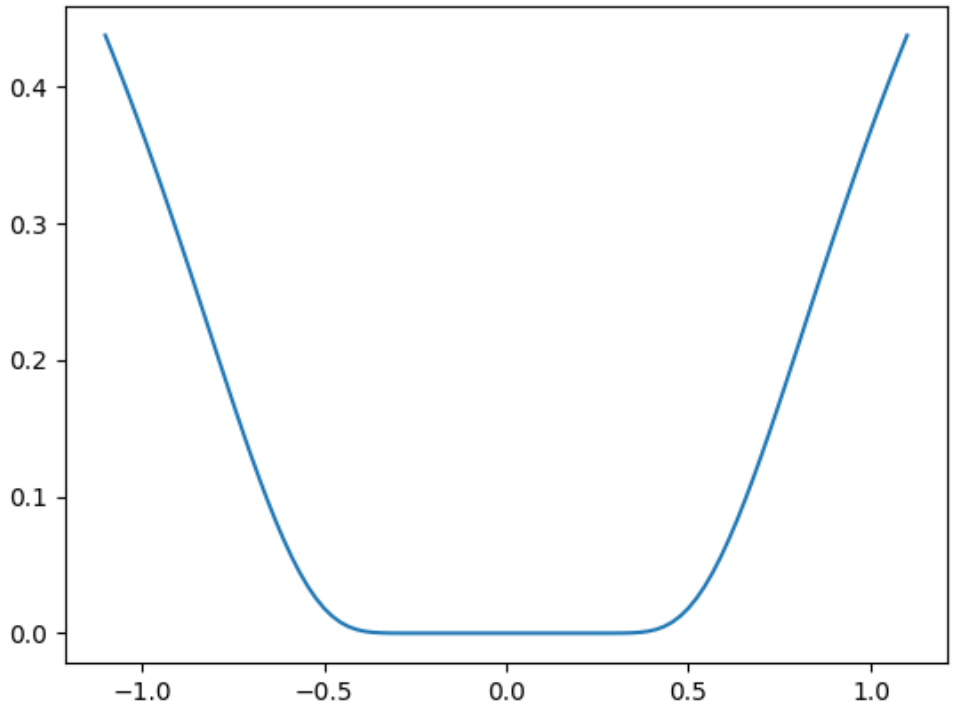
(continues on next page)

(continued from previous page)

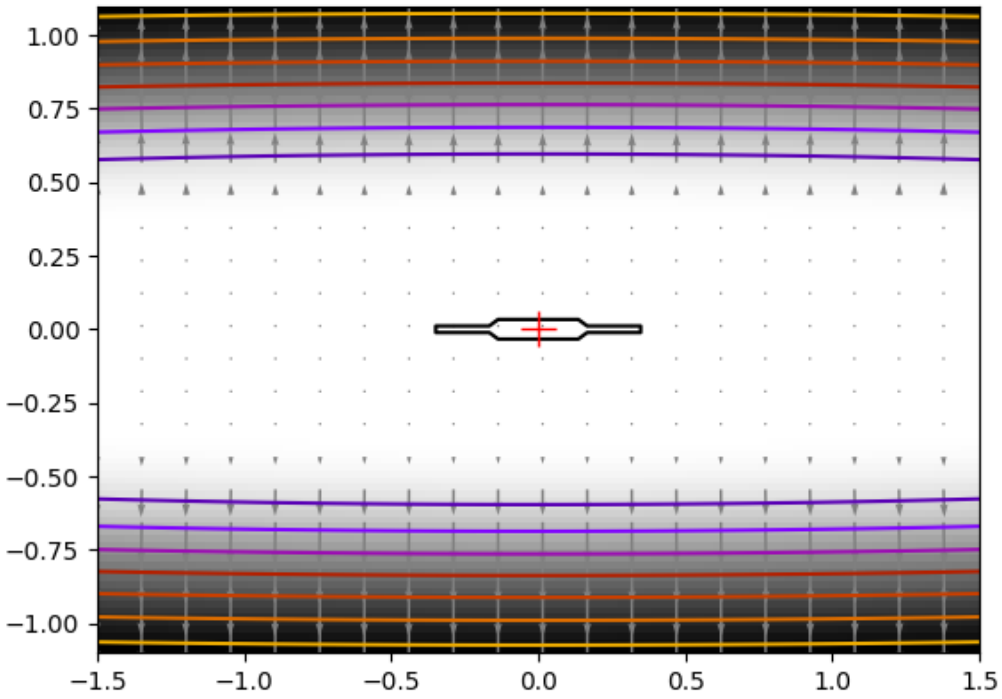
```
# The gradient of g. We won't use it here for the optimization.
def g_prime(x):
    r = np.sqrt(x[0] ** 2 + x[1] ** 2)
    return 2 / r**3 * g(x) * x / r


result = sp.optimize.minimize(g, [1, 1], method="Powell", tol=1e-10)
x_min = result.x
```

Some pretty plotting

```
plt.figure(0)
plt.clf()
t = np.linspace(-1.1, 1.1, 100)
plt.plot(t, f([0, t]))

plt.figure(1)
plt.clf()
X, Y = np.mgrid[-1.5:1.5:100j, -1.1:1.1:100j]  # type: ignore[misc]
plt.imshow(f([X, Y]).T, cmap="gray_r", extent=(-1.5, 1.5, -1.1, 1.1), origin="lower")
plt.contour(X, Y, f([X, Y]), cmap="gnuplot")

# Plot the gradient
dX, dY = g_prime([0.1 * X[::5, ::5], Y[::5, ::5]])
# Adjust for our preconditioning
dX *= 0.1
plt.quiver(X[::5, ::5], Y[::5, ::5], dX, dY, color=".5")

# Plot our solution
plt.plot(x_min[0], x_min[1], "r+", markersize=15)

plt.show()
```



-

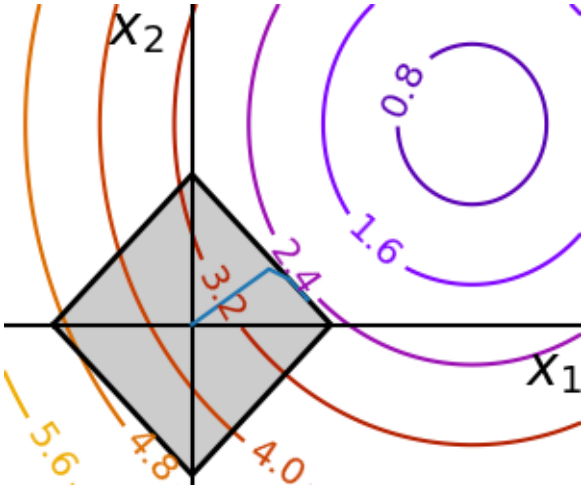


-

Total running time of the script: (0 minutes 0.114 seconds)

13.4.6 Optimization with constraints

An example showing how to do optimization with general constraints using SLSQP and cobyla.




```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp

x, y = np.mgrid[-2.03:4.2:0.04, -1.6:3.2:0.04]  # type: ignore[misc]
x = x.T
y = y.T

plt.figure(1, figsize=(3, 2.5))
plt.clf()
plt.axes((0, 0, 1, 1))

contours = plt.contour(
    np.sqrt((x - 3) ** 2 + (y - 2) ** 2),
    extent=[-2.03, 4.2, -1.6, 3.2],
    cmap="gnuplot",
)
plt.clabel(contours, inline=1, fmt="%1.1f", fontsize=14)
plt.plot([-1.5, 0, 1.5, 0, -1.5], [0, 1.5, 0, -1.5, 0], "k", linewidth=2)
plt.fill_between([-1.5, 0, 1.5], [0, -1.5, 0], [0, 1.5, 0], color=".8")
plt.axvline(0, color="k")
plt.axhline(0, color="k")

plt.text(-0.9, 2.8, "$x_2$", size=20)
plt.text(3.6, -0.6, "$x_1$", size=20)
plt.axis("tight")
plt.axis("off")

# And now plot the optimization path
accumulator = []


def f(x):
    # Store the list of function calls
    accumulator.append(x)
    return np.sqrt((x[0] - 3) ** 2 + (x[1] - 2) ** 2)
```

(continues on next page)

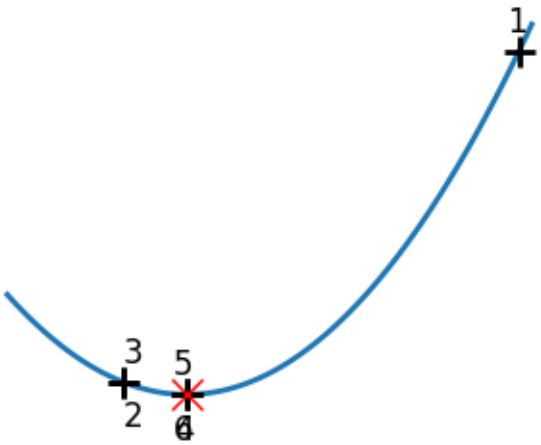
(continued from previous page)

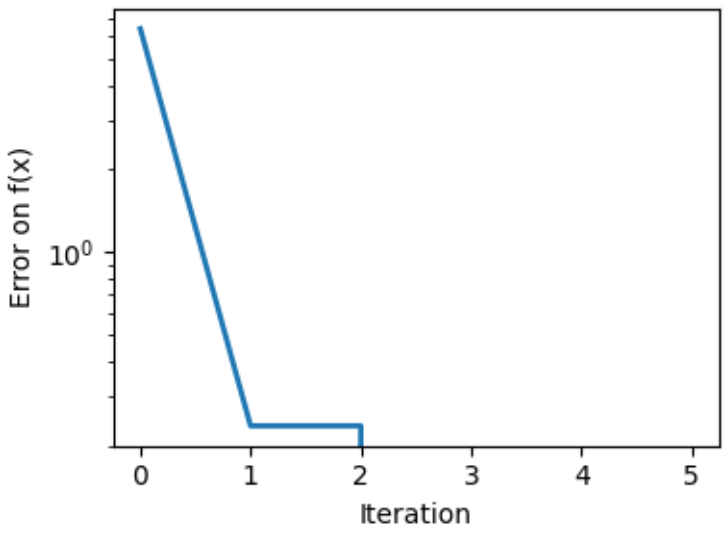
```
def constraint(x):
    return np.atleast_1d(1.5 - np.sum(np.abs(x)))


sp.optimize.minimize(
    f, np.array([0, 0]), method="SLSQP", constraints={"fun": constraint, "type": "ineq
↪"}
)

accumulated = np.array(accumulator)
plt.plot(accumulated[:, 0], accumulated[:, 1])

plt.show()
```

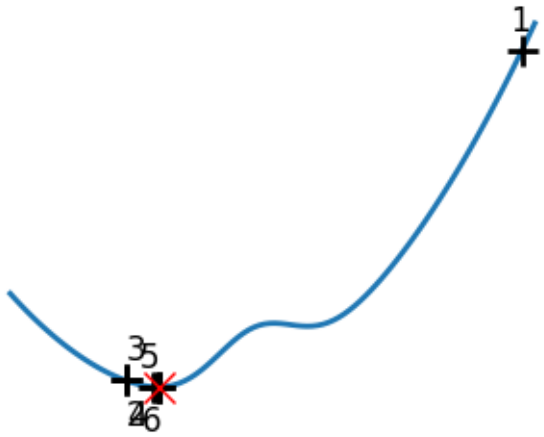
Total running time of the script: (0 minutes 0.048 seconds)

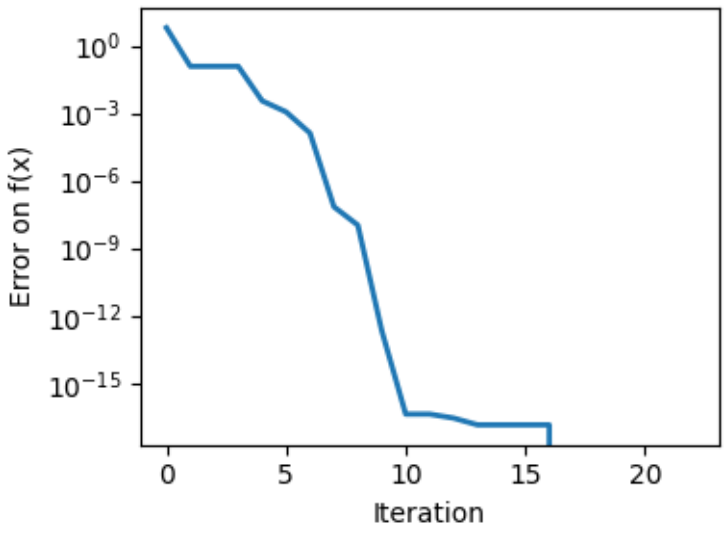
13.4.7 Brent's method

Illustration of 1D optimization: Brent's method

- 

- 

- 

- 


```
Converged at  6
Converged at  23
```

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp

x = np.linspace(-1, 3, 100)
x_0 = np.exp(-1)


def f(x):
    return (x - x_0) ** 2 + epsilon * np.exp(-5 * (x - 0.5 - x_0) ** 2)


for epsilon in (0, 1):
    plt.figure(figsize=(3, 2.5))
    plt.axes((0, 0, 1, 1))
```

(continues on next page)

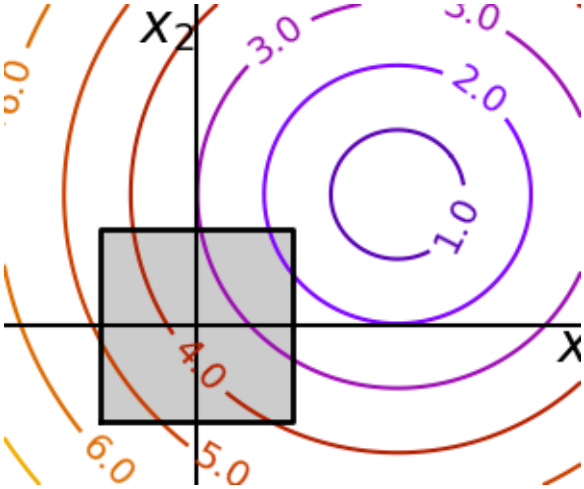
(continued from previous page)

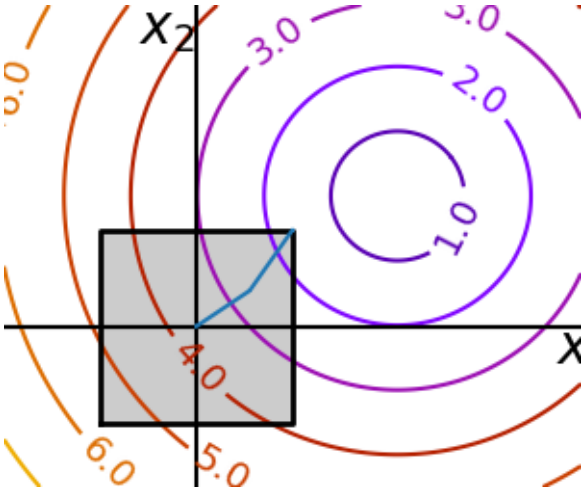
```
    # A convex function
    plt.plot(x, f(x), linewidth=2)

    # Apply brent method. To have access to the iteration, do this in an
    # artificial way: allow the algorithm to iter only once
    all_x = []
    all_y = []
    for iter in range(30):
        result = sp.optimize.minimize_scalar(
            f,
            bracket=(-5, 2.9, 4.5),
            method="Brent",
            options={"maxiter": iter},
            tol=np.finfo(1.0).eps,
        )
        if result.success:
            print("Converged at ", iter)
            break

        this_x = result.x
        all_x.append(this_x)
        all_y.append(f(this_x))
        if iter < 6:
            plt.text(
                this_x - 0.05 * np.sign(this_x) - 0.05,
                f(this_x) + 1.2 * (0.3 - iter % 2),
                str(iter + 1),
                size=12,
            )

    plt.plot(all_x[:10], all_y[:10], "k+", markersize=12, markeredgewidth=2)

    plt.plot(all_x[-1], all_y[-1], "rx", markersize=12)
    plt.axis("off")
    plt.ylim(ymin=-1, ymax=8)

    plt.figure(figsize=(4, 3))
    plt.semilogy(np.abs(all_y - all_y[-1]), linewidth=2)
    plt.ylabel("Error on f(x)")
    plt.xlabel("Iteration")
    plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 0.226 seconds)

13.4.8 Constraint optimization: visualizing the geometry

A small figure explaining optimization with constraints

- 

- 


```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp

x, y = np.mgrid[-2.9:5.8:0.05, -2.5:5:0.05]  # type: ignore[misc]
x = x.T
y = y.T

for i in (1, 2):
    # Create 2 figure: only the second one will have the optimization
    # path
    plt.figure(i, figsize=(3, 2.5))
    plt.clf()
    plt.axes((0, 0, 1, 1))

    contours = plt.contour(
        np.sqrt((x - 3) ** 2 + (y - 2) ** 2),
        extent=[-3, 6, -2.5, 5],
        cmap="gnuplot",
    )
    plt.clabel(contours, inline=1, fmt="%1.1f", fontsize=14)
    plt.plot(
        [-1.5, -1.5, 1.5, 1.5, -1.5], [-1.5, 1.5, 1.5, -1.5, -1.5], "k", linewidth=2
    )
    plt.fill_between([-1.5, 1.5], [-1.5, -1.5], [1.5, 1.5], color=".8")
    plt.axvline(0, color="k")
```

(continues on next page)

(continued from previous page)

```
    plt.axhline(0, color="k")

    plt.text(-0.9, 4.4, "$x_2$", size=20)
    plt.text(5.6, -0.6, "$x_1$", size=20)
    plt.axis("equal")
    plt.axis("off")

# And now plot the optimization path
accumulator = []


def f(x):
    # Store the list of function calls
    accumulator.append(x)
    return np.sqrt((x[0] - 3) ** 2 + (x[1] - 2) ** 2)


# We don't use the gradient, as with the gradient, L-BFGS is too fast,
# and finds the optimum without showing us a pretty path
def f_prime(x):
    r = np.sqrt((x[0] - 3) ** 2 + (x[0] - 2) ** 2)
    return np.array(((x[0] - 3) / r, (x[0] - 2) / r))


sp.optimize.minimize(
    f, np.array([0, 0]), method="L-BFGS-B", bounds=((-1.5, 1.5), (-1.5, 1.5))
)

accumulated = np.array(accumulator)
plt.plot(accumulated[:, 0], accumulated[:, 1])

plt.show()
```

Total running time of the script: (0 minutes 0.088 seconds)

13.4.9 Alternating optimization

The challenge here is that Hessian of the problem is a very ill-conditioned matrix. This can easily be seen, as the Hessian of the first term in simply 2 * K.T @ K. Thus the conditioning of the problem can be judged from looking at the conditioning of K.

```
import time

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

rng = np.random.default_rng(27446968)

K = rng.normal(size=(100, 100))


def f(x):
    return np.sum((K @ (x - 1)) ** 2) + np.sum(x**2) ** 2
```

(continues on next page)

(continued from previous page)

```
def f_prime(x):
    return 2 * K.T @ K @ (x - 1) + 4 * np.sum(x**2) * x


def hessian(x):
    H = 2 * K.T @ K + 4 * 2 * x * x[:, np.newaxis]
    return H + 4 * np.eye(H.shape[0]) * np.sum(x**2)
```

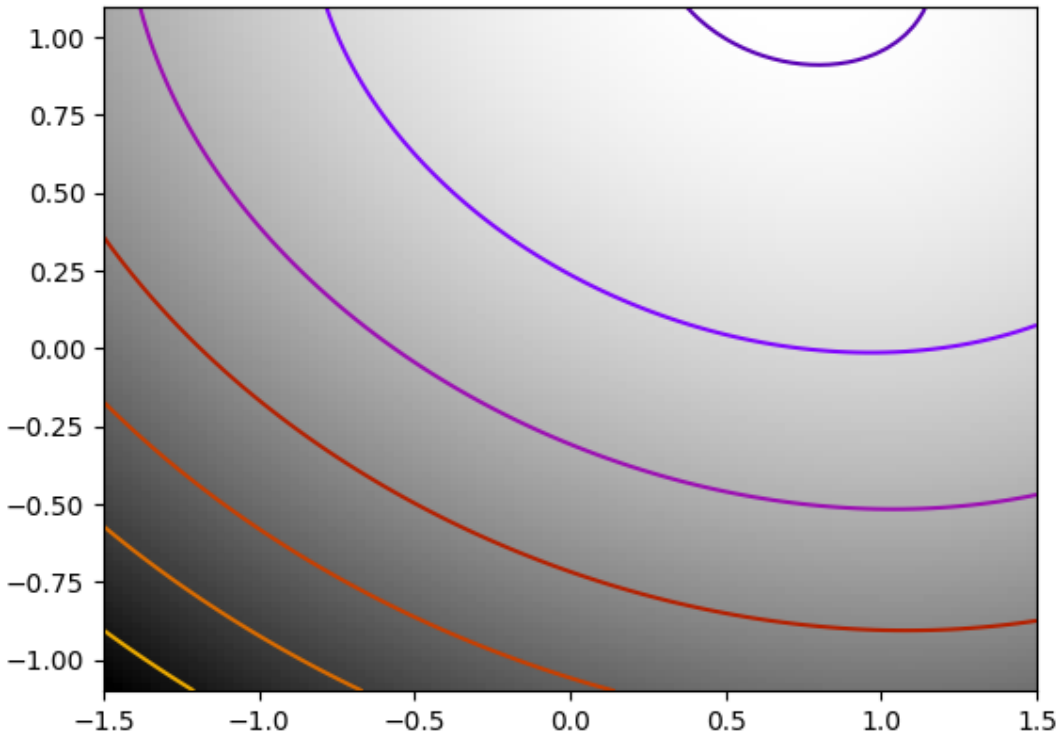
Some pretty plotting

```
plt.figure(1)
plt.clf()
Z = X, Y = np.mgrid[-1.5:1.5:100j, -1.1:1.1:100j]  # type: ignore[misc]
# Complete in the additional dimensions with zeros
Z = np.reshape(Z, (2, -1)).copy()
Z.resize((100, Z.shape[-1]))
Z = np.apply_along_axis(f, 0, Z)
Z = np.reshape(Z, X.shape)
plt.imshow(Z.T, cmap="gray_r", extent=(-1.5, 1.5, -1.1, 1.1), origin="lower")
plt.contour(X, Y, Z, cmap="gnuplot")

# A reference but slow solution:
t0 = time.time()
x_ref = sp.optimize.minimize(f, K[0], method="Powell").x
print(f"     Powell: time {time.time() - t0:.2f}s")
f_ref = f(x_ref)

# Compare different approaches
t0 = time.time()
x_bfgs = sp.optimize.minimize(f, K[0], method="BFGS").x
print(
    f"       BFGS: time {time.time() - t0:.2f}s, x error {np.sqrt(np.sum((x_bfgs -␣
↪x_ref) ** 2)):.2f}, f error {f(x_bfgs) - f_ref:.2f}"
)

t0 = time.time()
x_l_bfgs = sp.optimize.minimize(f, K[0], method="L-BFGS-B").x
print(
    f"     L-BFGS: time {time.time() - t0:.2f}s, x error {np.sqrt(np.sum((x_l_bfgs -
↪ x_ref) ** 2)):.2f}, f error {f(x_l_bfgs) - f_ref:.2f}"
)


t0 = time.time()
x_bfgs = sp.optimize.minimize(f, K[0], jac=f_prime, method="BFGS").x
print(
    f"  BFGS w f': time {time.time() - t0:.2f}s, x error {np.sqrt(np.sum((x_bfgs -␣
↪x_ref) ** 2)):.2f}, f error {f(x_bfgs) - f_ref:.2f}"
)

t0 = time.time()
x_l_bfgs = sp.optimize.minimize(f, K[0], jac=f_prime, method="L-BFGS-B").x
print(
    f"L-BFGS w f': time {time.time() - t0:.2f}s, x error {np.sqrt(np.sum((x_l_bfgs -
↪ x_ref) ** 2)):.2f}, f error {f(x_l_bfgs) - f_ref:.2f}"
```

(continues on next page)

(continued from previous page)

```
)

t0 = time.time()
x_newton = sp.optimize.minimize(
    f, K[0], jac=f_prime, hess=hessian, method="Newton-CG"
).x
print(
    f"     Newton: time {time.time() - t0:.2f}s, x error {np.sqrt(np.sum((x_newton -
↪ x_ref) ** 2)):.2f}, f error {f(x_newton) - f_ref:.2f}"
)

plt.show()
```

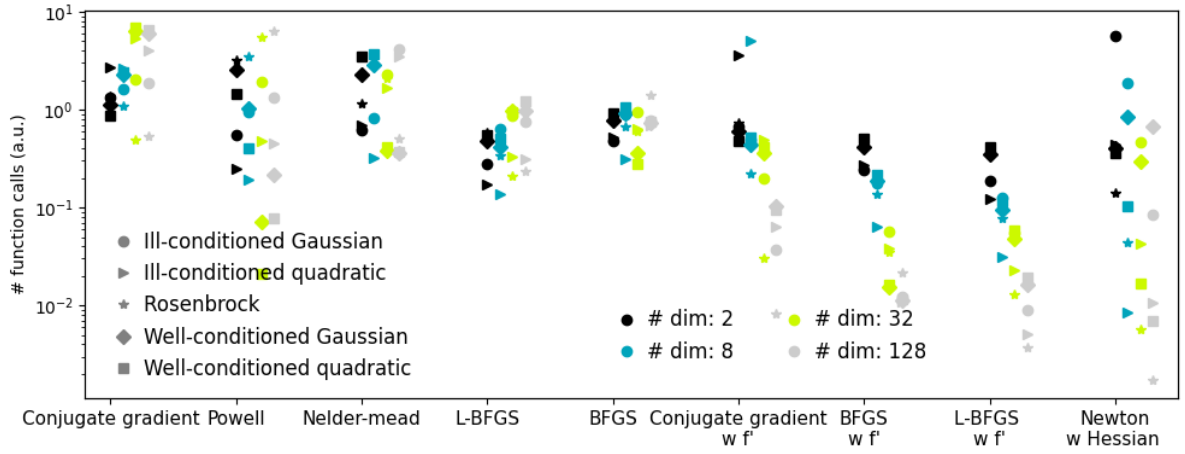



```
     Powell: time 0.13s
       BFGS: time 0.58s, x error 0.02, f error -0.03
     L-BFGS: time 0.09s, x error 0.02, f error -0.03
  BFGS w f': time 0.14s, x error 0.02, f error -0.03
L-BFGS w f': time 0.07s, x error 0.02, f error -0.03
     Newton: time 0.01s, x error 0.02, f error -0.03
```

Total running time of the script: (0 minutes 1.290 seconds)

13.4.10 Plotting the comparison of optimizers

Plots the results from the comparison of optimizers.




```
import pickle
import sys

import numpy as np

import matplotlib
import matplotlib.pyplot as plt

results = pickle.load(
    open(f"helper/compare_optimizers_py{sys.version_info[0]}.pkl", "rb")
)
n_methods = len(list(results.values())[0]["Rosenbrock  "])
n_dims = len(results)

symbols = "o>*Ds"

plt.figure(1, figsize=(10, 4))
plt.clf()

nipy_spectral = matplotlib.colormaps["nipy_spectral"]
colors = nipy_spectral(np.linspace(0, 1, n_dims))[:, :3]

method_names = list(list(results.values())[0]["Rosenbrock  "].keys())
method_names.sort(key=lambda x: x[::-1], reverse=True)

for n_dim_index, ((n_dim, n_dim_bench), color) in enumerate(
    zip(sorted(results.items()), colors, strict=True)
):
    for (cost_name, cost_bench), symbol in zip(
        sorted(n_dim_bench.items()), symbols, strict=True
    ):
        for (
            method_index,
            method_name,
        ) in enumerate(method_names):
            this_bench = cost_bench[method_name]
            bench = np.mean(this_bench)
            plt.semilogy(
```

(continues on next page)

(continued from previous page)

```
                [
                    method_index + 0.1 * n_dim_index,
                ],
                [
                    bench,
                ],
                marker=symbol,
                color=color,
            )

# Create a legend for the problem type
for cost_name, symbol in zip(sorted(n_dim_bench.keys()), symbols, strict=True):
    plt.semilogy(
        [
            -10,
        ],
        [
            0,
        ],
        symbol,
        color=".5",
        label=cost_name,
    )

plt.xticks(np.arange(n_methods), method_names, size=11)
plt.xlim(-0.2, n_methods - 0.5)
plt.legend(loc="best", numpoints=1, handletextpad=0, prop={"size": 12}, frameon=False)
plt.ylabel("# function calls (a.u.)")

# Create a second legend for the problem dimensionality
plt.twinx()

for n_dim, color in zip(sorted(results.keys()), colors, strict=True):
    plt.plot(
        [
            -10,
        ],
        [
            0,
        ],
        "o",
        color=color,
        label=f"# dim: {n_dim}",
    )
plt.legend(
    loc=(0.47, 0.07),
    numpoints=1,
    handletextpad=0,
    prop={"size": 12},
    frameon=False,
    ncol=2,
)
plt.xlim(-0.2, n_methods - 0.5)

plt.xticks(np.arange(n_methods), method_names)
plt.yticks(())
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.575 seconds)

13.4.11 Gradient descent

An example demoing gradient descent by creating figures that trace the evolution of the optimizer.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp

import collections
import sys
import os

sys.path.append(os.path.abspath("helper"))
from cost_functions import (
    mk_quad,
    mk_gauss,
    rosenbrock,
    rosenbrock_prime,
    rosenbrock_hessian,
    LoggingFunction,
    CountingFunction,
)

x_min, x_max = -1, 2
y_min, y_max = 2.25 / 3 * x_min - 0.2, 2.25 / 3 * x_max - 0.2
```

A formatter to print values on contours

```
def super_fmt(value):
    if value > 1:
        if np.abs(int(value) - value) < 0.1:
            out = f"$10^{{{int(value):d}}}$"
        else:
            out = f"$10^{{{value:.1f}}}$"
    else:
        value = np.exp(value - 0.01)
        if value > 0.1:
            out = f"{value:1.1f}"
        elif value > 0.01:
            out = f"{value:.2f}"
        else:
            out = f"{value:.2e}"
    return out
```

A gradient descent algorithm do not use: its a toy, use scipy's optimize.fmin_cg

```
def gradient_descent(x0, f, f_prime, hessian=None, adaptative=False):
    x_i, y_i = x0
    all_x_i = []
```

(continues on next page)

(continued from previous page)

```
    all_y_i = []
    all_f_i = []

    for i in range(1, 100):
        all_x_i.append(x_i)
        all_y_i.append(y_i)
        all_f_i.append(f([x_i, y_i]))
        dx_i, dy_i = f_prime(np.asarray([x_i, y_i]))
        if adaptative:
            # Compute a step size using a line_search to satisfy the Wolf
            # conditions
            step = sp.optimize.line_search(
                f,
                f_prime,
                np.r_[x_i, y_i],
                -np.r_[dx_i, dy_i],
                np.r_[dx_i, dy_i],
                c2=0.05,
            )
            step = step[0]
            if step is None:
                step = 0
        else:
            step = 1
        x_i += -step * dx_i
        y_i += -step * dy_i
        if np.abs(all_f_i[-1]) < 1e-16:
            break
    return all_x_i, all_y_i, all_f_i


def gradient_descent_adaptative(x0, f, f_prime, hessian=None):
    return gradient_descent(x0, f, f_prime, adaptative=True)


def conjugate_gradient(x0, f, f_prime, hessian=None):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    sp.optimize.minimize(
        f, x0, jac=f_prime, method="CG", callback=store, options={"gtol": 1e-12}
    )
    return all_x_i, all_y_i, all_f_i


def newton_cg(x0, f, f_prime, hessian):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]
```

(continues on next page)

(continued from previous page)

```
    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    sp.optimize.minimize(
        f,
        x0,
        method="Newton-CG",
        jac=f_prime,
        hess=hessian,
        callback=store,
        options={"xtol": 1e-12},
    )
    return all_x_i, all_y_i, all_f_i


def bfgs(x0, f, f_prime, hessian=None):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    sp.optimize.minimize(
        f, x0, method="BFGS", jac=f_prime, callback=store, options={"gtol": 1e-12}
    )
    return all_x_i, all_y_i, all_f_i


def powell(x0, f, f_prime, hessian=None):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    sp.optimize.minimize(
        f, x0, method="Powell", callback=store, options={"ftol": 1e-12}
    )
    return all_x_i, all_y_i, all_f_i


def nelder_mead(x0, f, f_prime, hessian=None):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
```

(continues on next page)

(continued from previous page)

```
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    sp.optimize.minimize(
        f, x0, method="Nelder-Mead", callback=store, options={"ftol": 1e-12}
    )
    return all_x_i, all_y_i, all_f_i
```

Run different optimizers on these problems

```
levels = {}

for index, ((f, f_prime, hessian), optimizer) in enumerate(
    (
        (mk_quad(0.7), gradient_descent),
        (mk_quad(0.7), gradient_descent_adaptative),
        (mk_quad(0.02), gradient_descent),
        (mk_quad(0.02), gradient_descent_adaptative),
        (mk_gauss(0.02), gradient_descent_adaptative),
        (
            (rosenbrock, rosenbrock_prime, rosenbrock_hessian),
            gradient_descent_adaptative,
        ),
        (mk_gauss(0.02), conjugate_gradient),
        ((rosenbrock, rosenbrock_prime, rosenbrock_hessian), conjugate_gradient),
        (mk_quad(0.02), newton_cg),
        (mk_gauss(0.02), newton_cg),
        ((rosenbrock, rosenbrock_prime, rosenbrock_hessian), newton_cg),
        (mk_quad(0.02), bfgs),
        (mk_gauss(0.02), bfgs),
        ((rosenbrock, rosenbrock_prime, rosenbrock_hessian), bfgs),
        (mk_quad(0.02), powell),
        (mk_gauss(0.02), powell),
        ((rosenbrock, rosenbrock_prime, rosenbrock_hessian), powell),
        (mk_gauss(0.02), nelder_mead),
        ((rosenbrock, rosenbrock_prime, rosenbrock_hessian), nelder_mead),
    )
):
    # Compute a gradient-descent
    x_i, y_i = 1.6, 1.1
    counting_f_prime = CountingFunction(f_prime)
    counting_hessian = CountingFunction(hessian)
    logging_f = LoggingFunction(f, counter=counting_f_prime.counter)
    all_x_i, all_y_i, all_f_i = optimizer(
        np.array([x_i, y_i]), logging_f, counting_f_prime, hessian=counting_hessian
    )

    # Plot the contour plot
    if not max(all_y_i) < y_max:
        x_min *= 1.2
        x_max *= 1.2
```

(continues on next page)

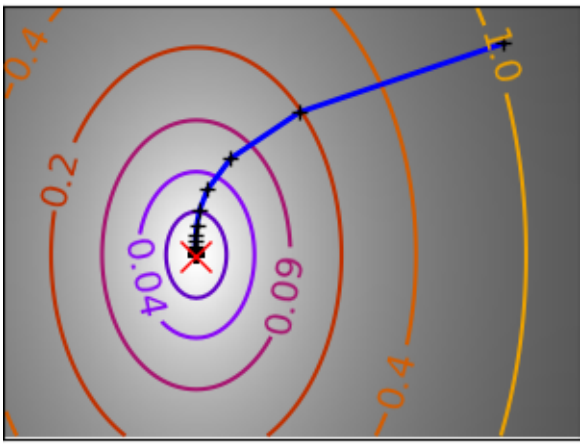
(continued from previous page)

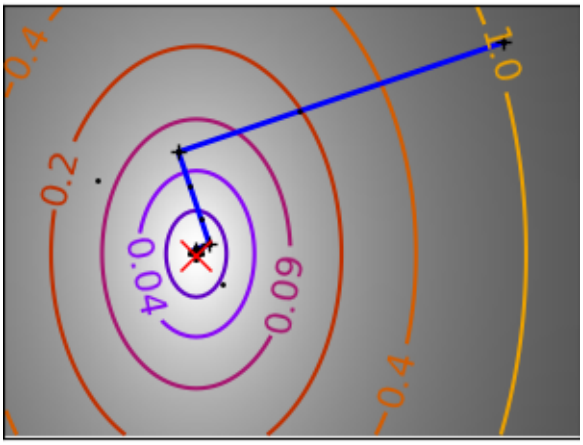
```
        y_min *= 1.2
        y_max *= 1.2
    x, y = np.mgrid[x_min:x_max:100j, y_min:y_max:100j]
    x = x.T
    y = y.T

    plt.figure(index, figsize=(3, 2.5))
    plt.clf()
    plt.axes([0, 0, 1, 1])

    X = np.concatenate((x[np.newaxis, ...], y[np.newaxis, ...]), axis=0)
    z = np.apply_along_axis(f, 0, X)
    log_z = np.log(z + 0.01)
    plt.imshow(
        log_z,
        extent=[x_min, x_max, y_min, y_max],
        cmap=plt.cm.gray_r,
        origin="lower",
        vmax=log_z.min() + 1.5 * np.ptp(log_z),
    )
    contours = plt.contour(
        log_z,
        levels=levels.get(f),
        extent=[x_min, x_max, y_min, y_max],
        cmap=plt.cm.gnuplot,
        origin="lower",
    )
    levels[f] = contours.levels
    plt.clabel(contours, inline=1, fmt=super_fmt, fontsize=14)

    plt.plot(all_x_i, all_y_i, "b-", linewidth=2)
    plt.plot(all_x_i, all_y_i, "k+")

    plt.plot(logging_f.all_x_i, logging_f.all_y_i, "k.", markersize=2)

    plt.plot([0], [0], "rx", markersize=12)

    plt.xticks(())
    plt.yticks(())
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.draw()

    plt.figure(index + 100, figsize=(4, 3))
    plt.clf()
    plt.semilogy(np.maximum(np.abs(all_f_i), 1e-30), linewidth=2, label="# iterations
↪")
    plt.ylabel("Error on f(x)")
    plt.semilogy(
        logging_f.counts,
        np.maximum(np.abs(logging_f.all_f_i), 1e-30),
        linewidth=2,
        color="g",
        label="# function calls",
    )
    plt.legend(
```

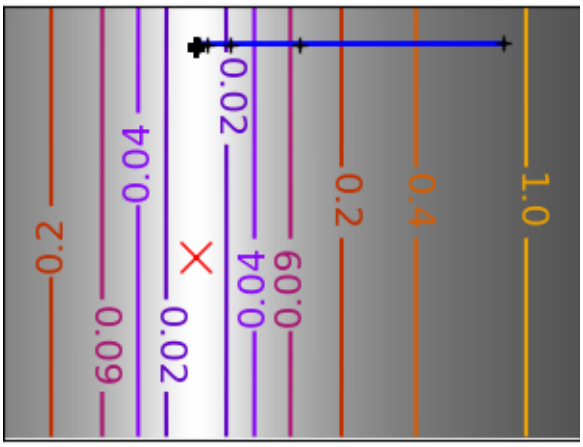
(continues on next page)

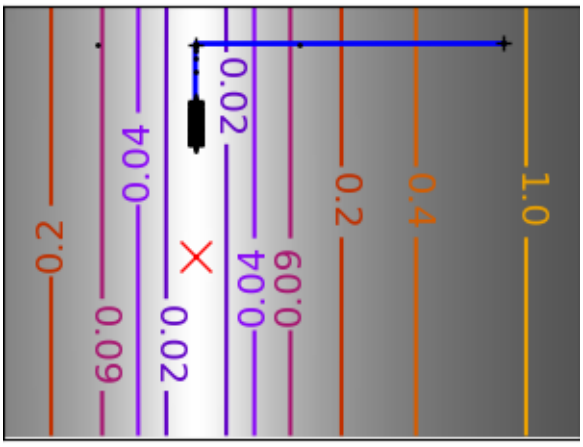
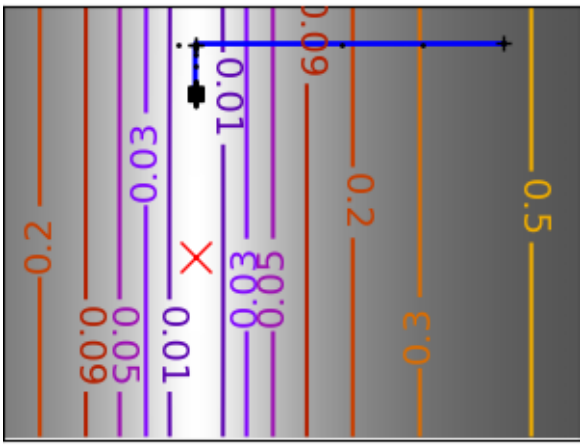
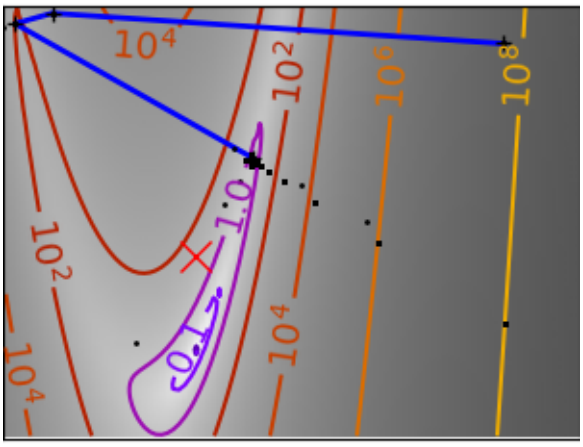
(continued from previous page)

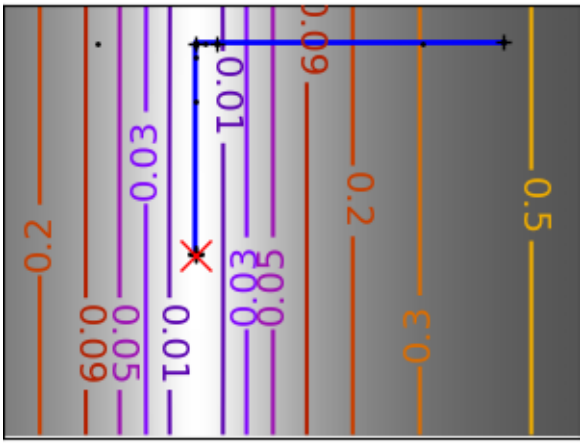
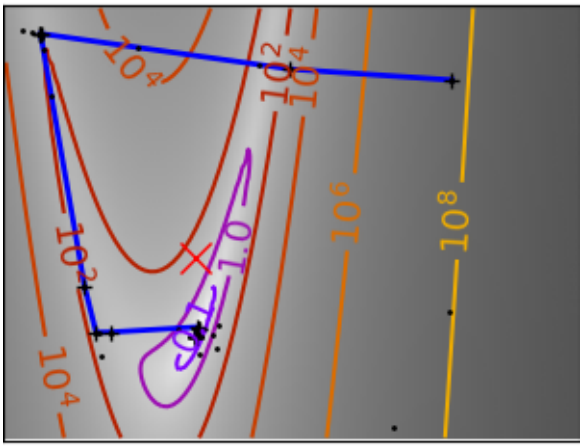
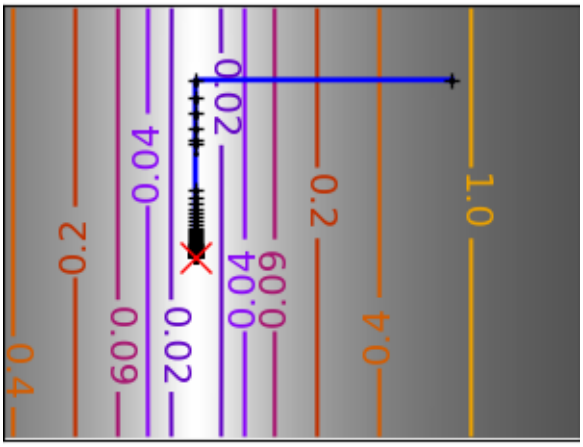
```
        loc="upper right",
        frameon=True,
        prop={"size": 11},
        borderaxespad=0,
        handlelength=1.5,
        handletextpad=0.5,
    )
    plt.tight_layout()
    plt.draw()
```

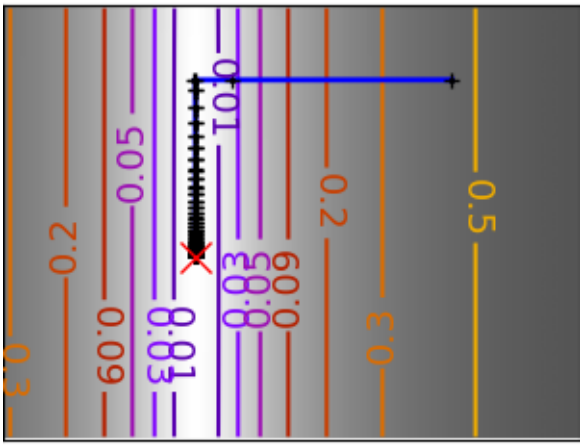
- 

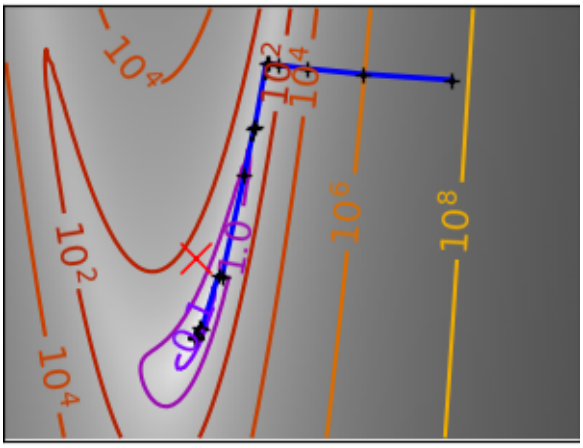
- 

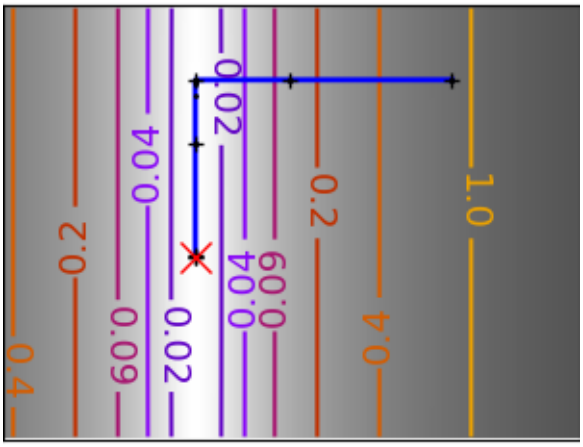
- 

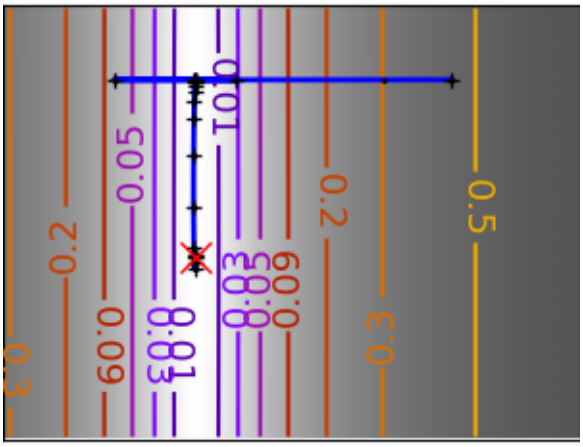
- 
0.02
0.04
0.2
0.4
1.0
0.2
0.09
0.02
0.04
0.09
- 
0.01
0.03
0.09
0.2
0.5
0.2
0.09
0.05
0.01
0.03
0.3
- 
10^4
10^2
10^0
10^8
10^2
10^0
10^4
10^4

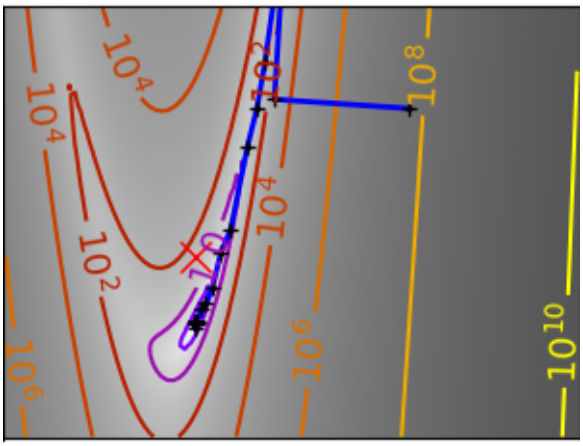
- 
0.09
0.01
0.03
0.2
0.5
2.0
0.09
0.05
0.01
0.03
0.3
- 
10^4
10^2
10^4
10^6
10^8
10^2
1.0
10^4
- 
0.02
0.04
0.2
1.0
2.0
0.04
0.09
0.09
0.02
0.4
0.4

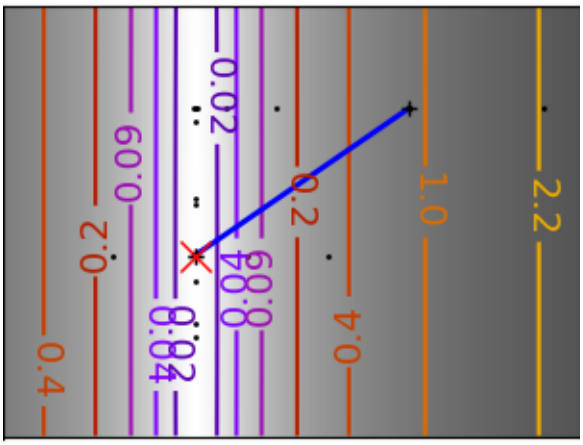
- 
0.3
0.2
0.09
0.05
0.03
0.01
0.01
0.03
0.05
0.09
0.2
0.3
0.5

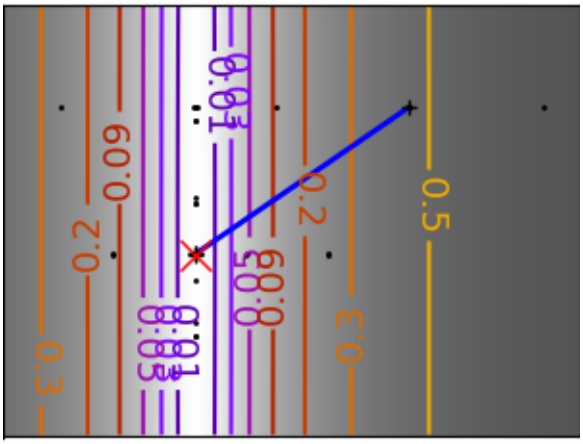
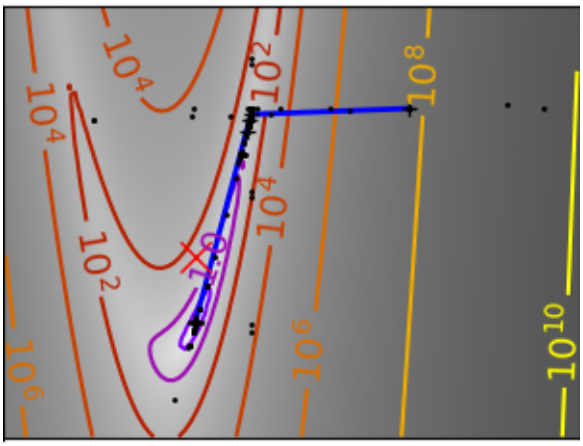
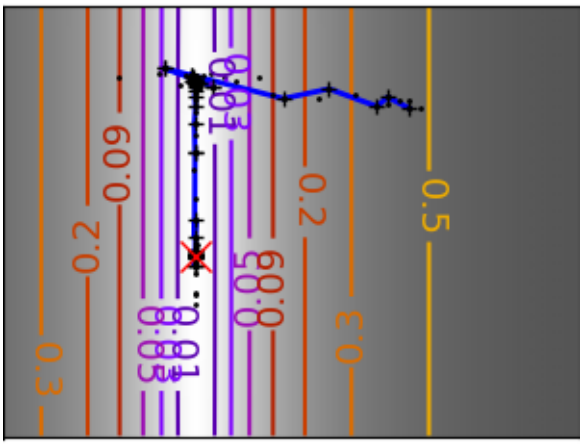
- 
10^4
10^2
10^4
10^2
10^4
10^6
10^8
1.0
0.1

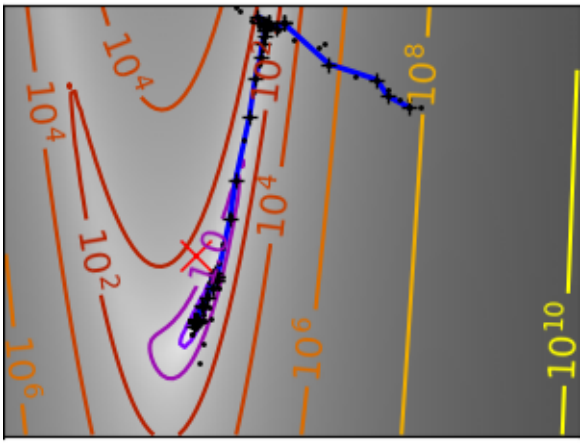
- 
0.4
0.2
0.09
0.04
0.02
0.02
0.04
0.09
0.2
0.4
1.0

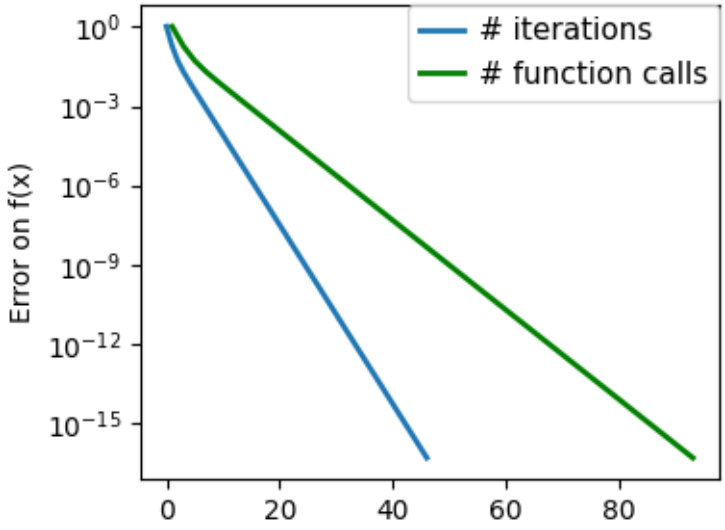
- 
0.3
0.2
0.09
0.05
0.01
0.03
0.01
0.03
0.05
0.09
0.2
0.3
0.5

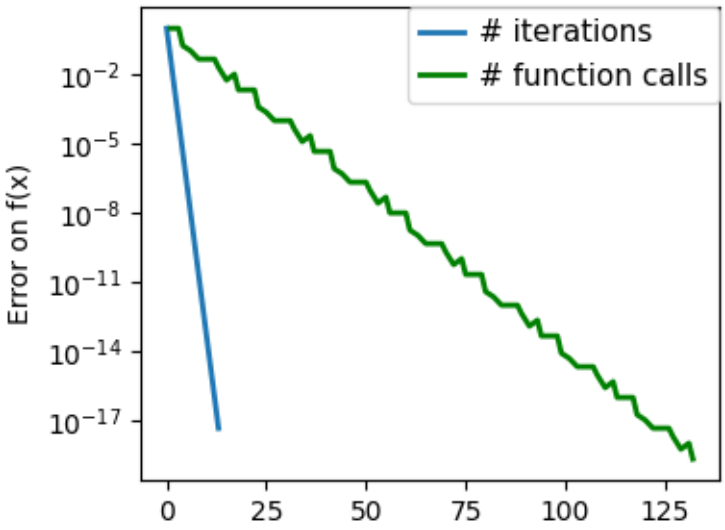
- 
10^6
10^4
10^4
10^2
10^2
10^4
10^6
10^8
10^{10}

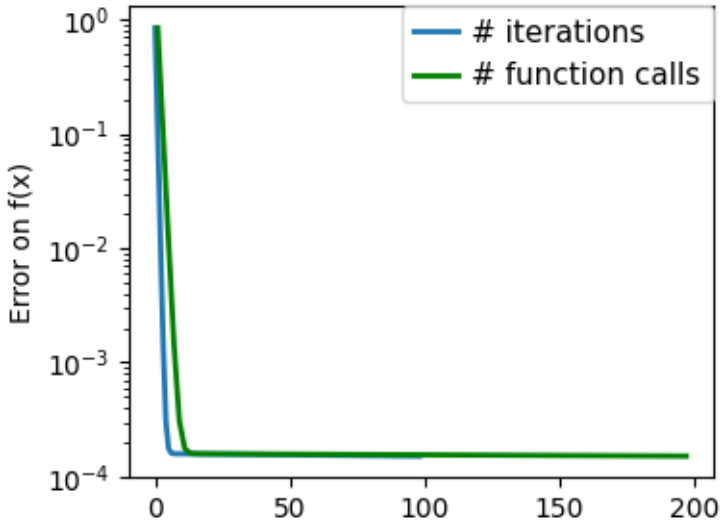
- 
0.4
0.2
0.09
0.02
0.04
0.02
0.04
0.09
0.2
0.4
1.0
2.2

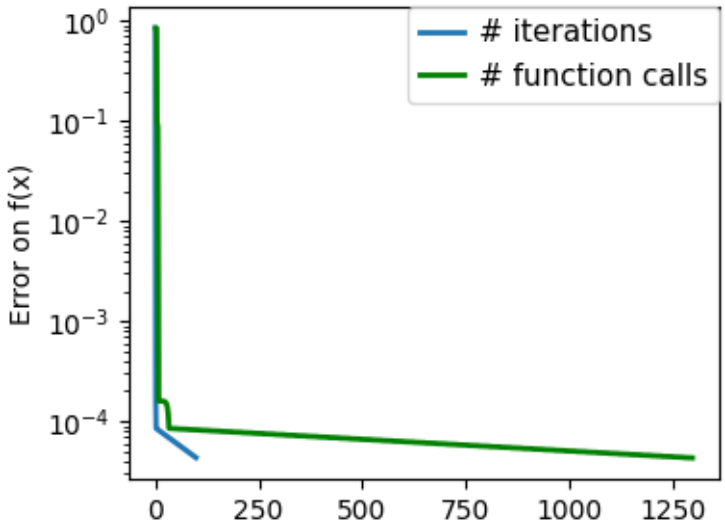
- 
0.5
0.3
0.2
0.1
0.09
0.08
0.05
0.03
0.01
0.005
0.003
- 
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
10^{-2}
10^{-4}
- 
0.5
0.3
0.2
0.1
0.09
0.08
0.05
0.03
0.01
0.005
0.003

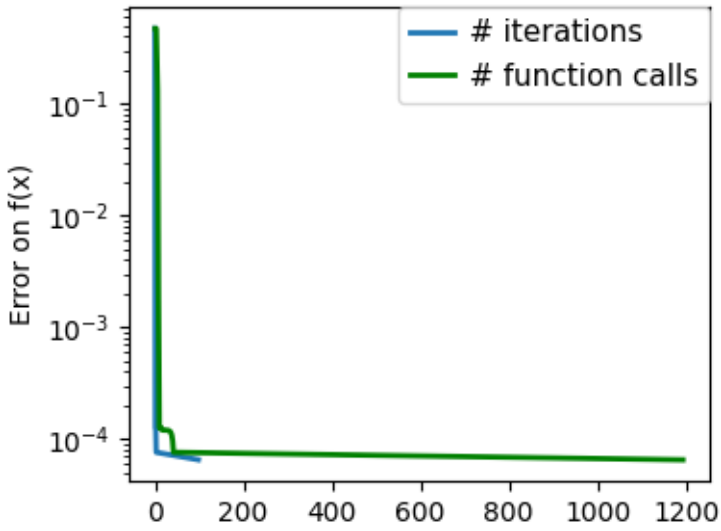
- 
10^4
10^4
10^2
10^6
10^6
10^4
10^8
10^{10}

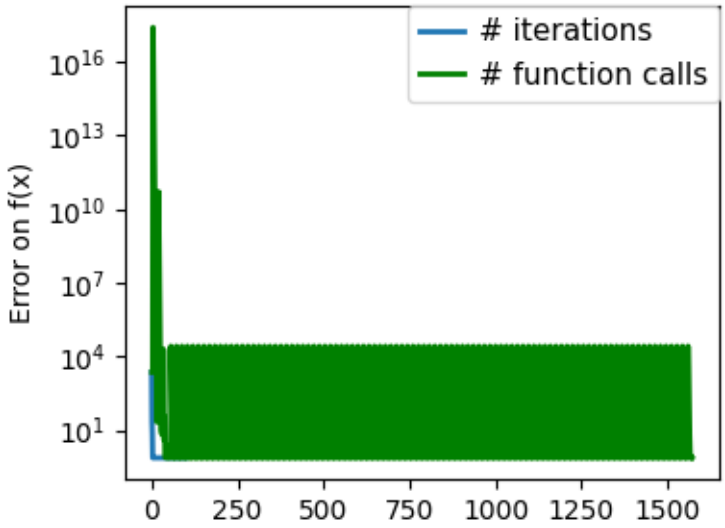
- 
iterations
function calls
Error on f(x)
10^0
10^{-3}
10^{-6}
10^{-9}
10^{-12}
10^{-15}
0
20
40
60
80

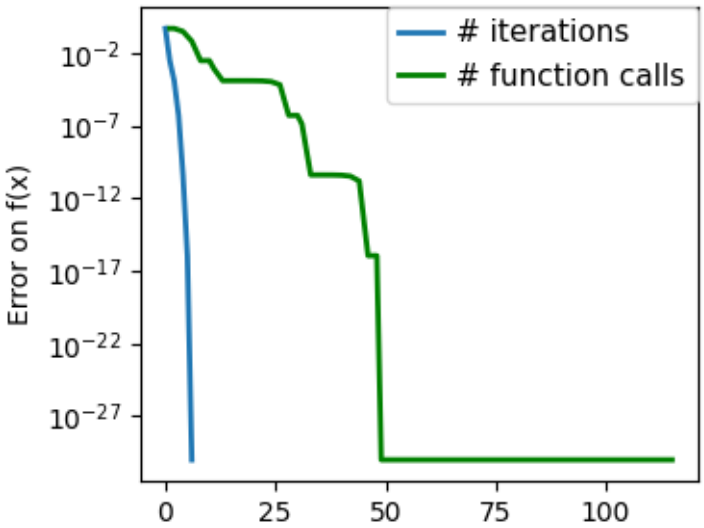
- 
iterations
function calls
Error on f(x)
10^{-2}
10^{-5}
10^{-8}
10^{-11}
10^{-14}
10^{-17}
0
25
50
75
100
125

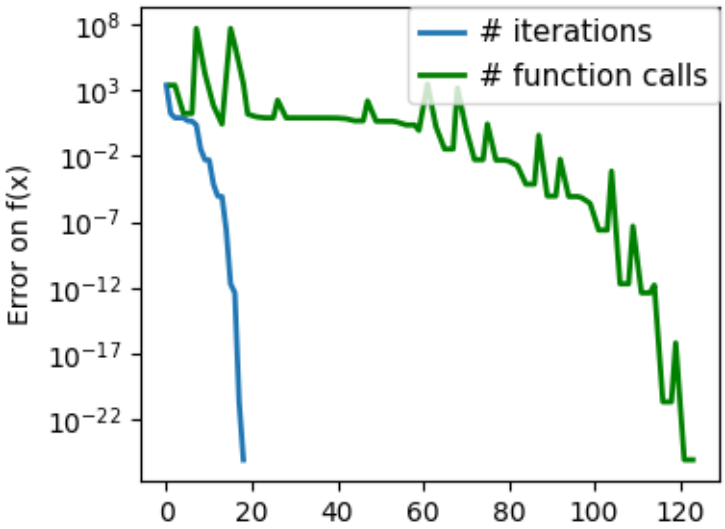

Error on f(x)
iterations
function calls
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
50
100
150
200

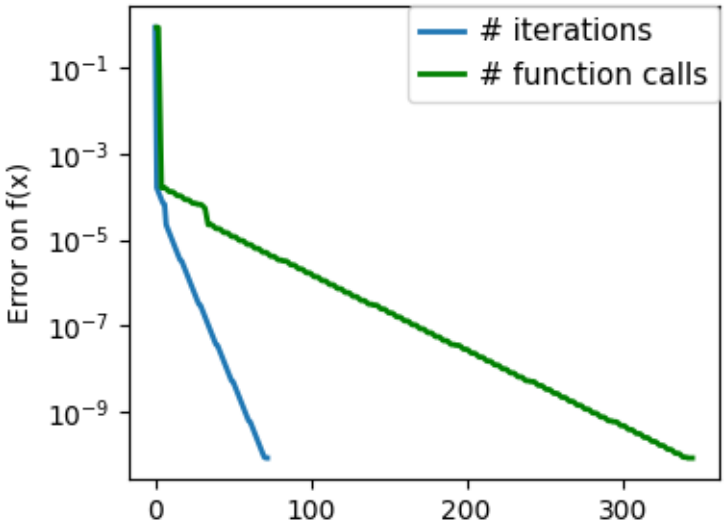

Error on f(x)
iterations
function calls
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
250
500
750
1000
1250

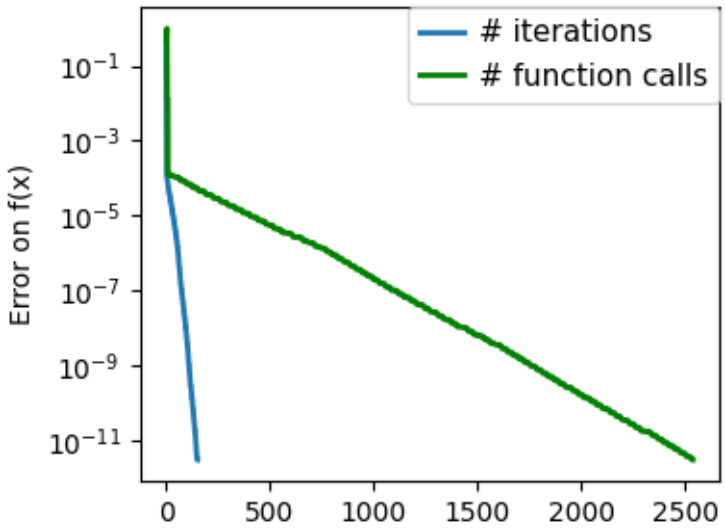

Error on f(x)
iterations
function calls
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
200
400
600
800
1000
1200

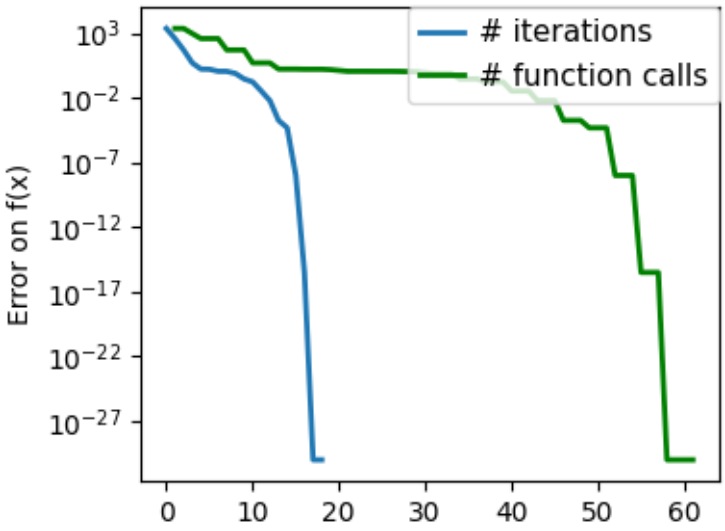
- 
iterations
function calls
Error on f(x)
10^{16}
10^{13}
10^{10}
10^{7}
10^{4}
10^{1}
0
250
500
750
1000
1250
1500

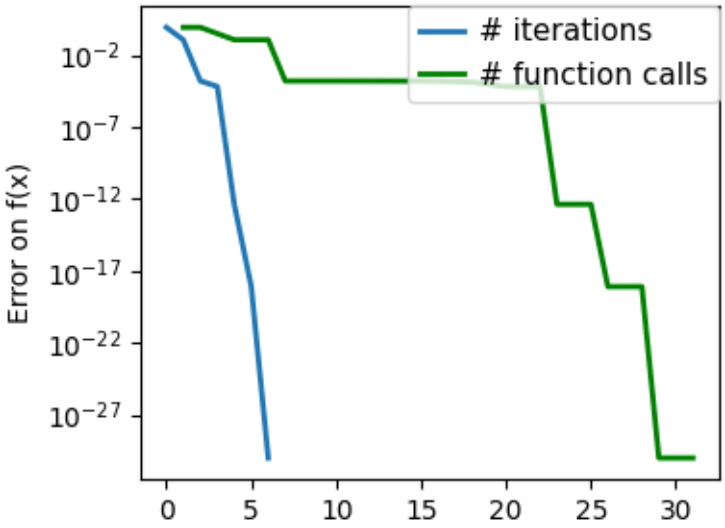
- 
iterations
function calls
Error on f(x)
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
10^{-27}
0
25
50
75
100

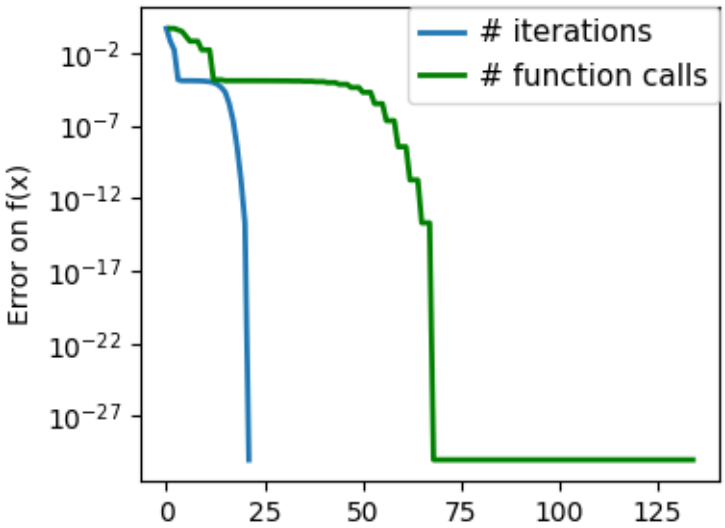
- 
iterations
function calls
Error on f(x)
10^{8}
10^{3}
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
0
20
40
60
80
100
120

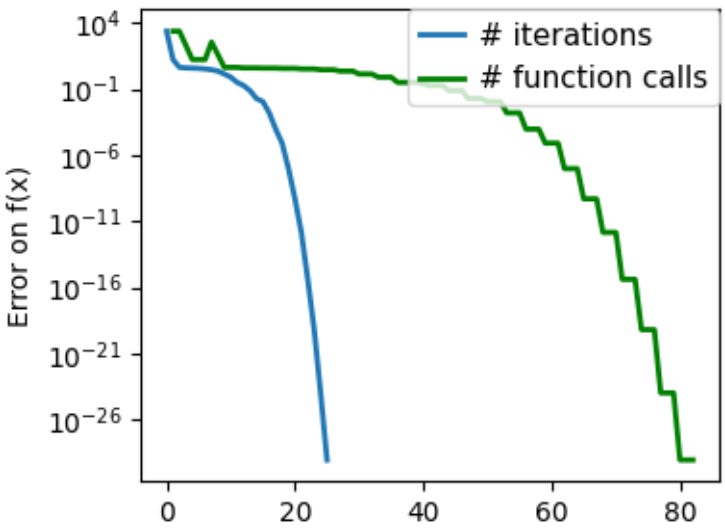
- 
Error on f(x)
iterations
function calls
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
0
100
200
300

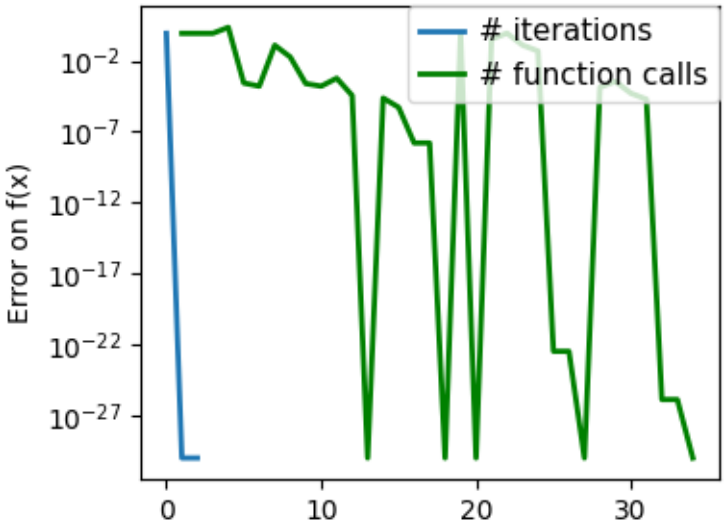
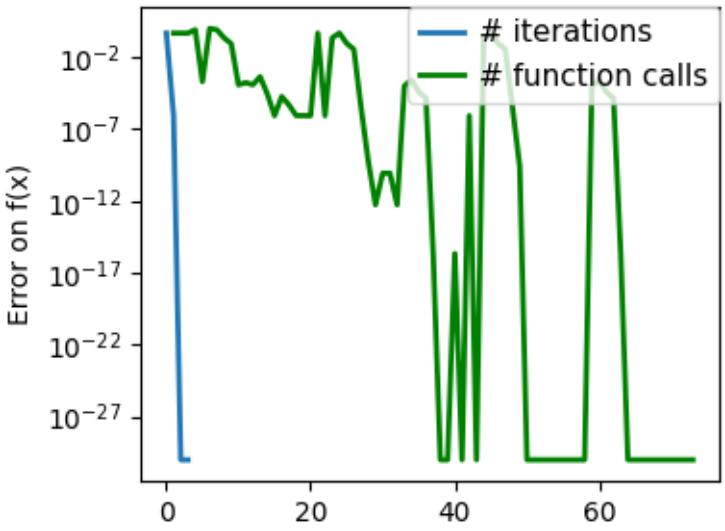
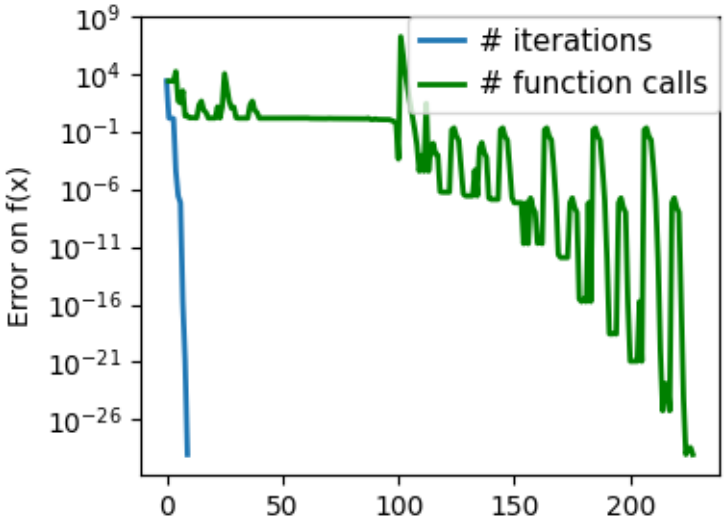
- 
Error on f(x)
iterations
function calls
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
0
500
1000
1500
2000
2500

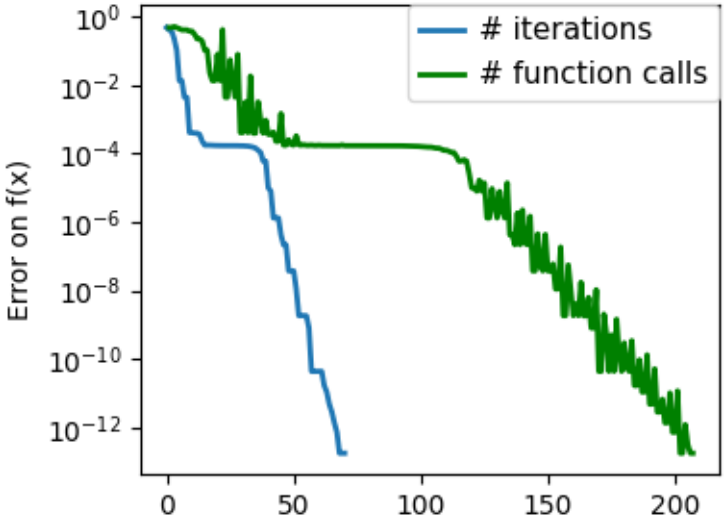
- 
Error on f(x)
iterations
function calls
10^{3}
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
10^{-27}
0
10
20
30
40
50
60

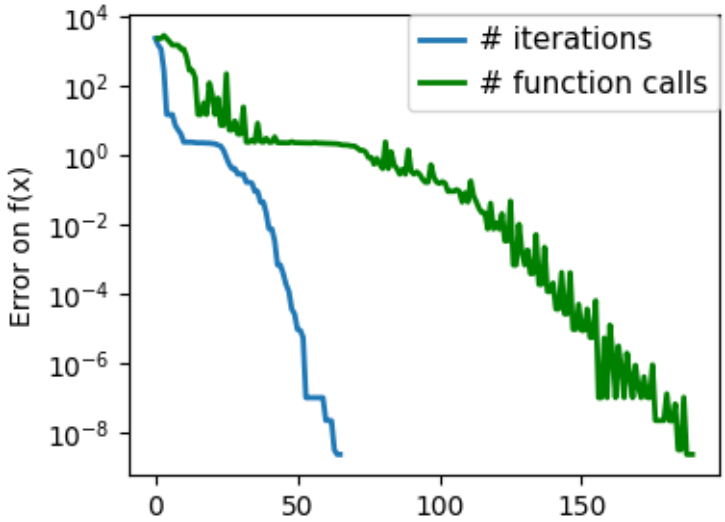
- 
Error on f(x)
iterations
function calls
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
10^{-27}
0
5
10
15
20
25
30

- 
Error on f(x)
iterations
function calls
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
10^{-27}
0
25
50
75
100
125

- 
Error on f(x)
iterations
function calls
10^{4}
10^{-1}
10^{-6}
10^{-11}
10^{-16}
10^{-21}
10^{-26}
0
20
40
60
80

- 
Error on f(x)
iterations
function calls
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
10^{-27}
0
10
20
30
- 
Error on f(x)
iterations
function calls
10^{-2}
10^{-7}
10^{-12}
10^{-17}
10^{-22}
10^{-27}
0
20
40
60
- 
Error on f(x)
iterations
function calls
10^{9}
10^{4}
10^{-1}
10^{-6}
10^{-11}
10^{-16}
10^{-21}
10^{-26}
0
50
100
150
200

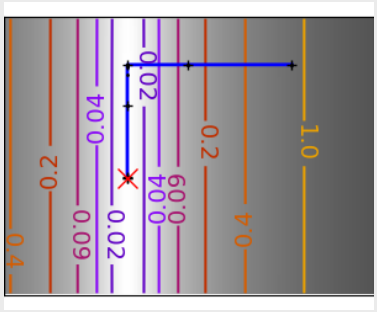


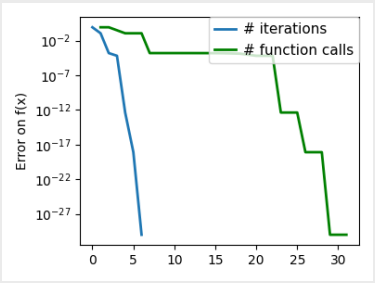



```
/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages/scipy/optimize/_
↪linesearch.py:312: LineSearchWarning: The line search algorithm did not converge
  alpha_star, phi_star, old_fval, derphi_star = scalar_search_wolfe2(
/home/runner/work/scientific-python-lectures/scientific-python-lectures/advanced/
↪mathematical_optimization/examples/plot_gradient_descent.py:70: LineSearchWarning:␣
↪The line search algorithm did not converge
  step = sp.optimize.line_search(
/home/runner/work/scientific-python-lectures/scientific-python-lectures/advanced/
↪mathematical_optimization/examples/plot_gradient_descent.py:234: RuntimeWarning:␣
↪More than 20 figures have been opened. Figures created through the pyplot interface␣
↪(`matplotlib.pyplot.figure`) are retained until explicitly closed and may consume␣
↪too much memory. (To control this warning, see the rcParam `figure.max_open_
↪warning`). Consider using `matplotlib.pyplot.close()`.
  plt.figure(index, figsize=(3, 2.5))
/home/runner/work/scientific-python-lectures/scientific-python-lectures/advanced/
↪mathematical_optimization/examples/plot_gradient_descent.py:179: OptimizeWarning:␣
↪Unknown solver options: ftol
  sp.optimize.minimize(
```

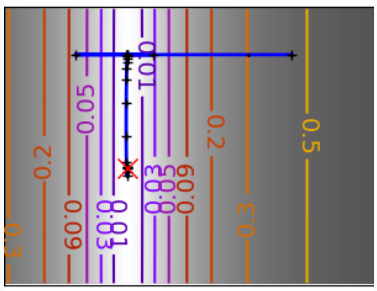
Total running time of the script: (0 minutes 6.794 seconds)

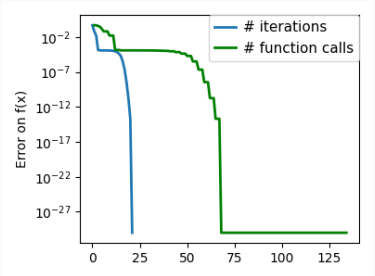
An ill-conditioned quadratic function:
On a exactly quadratic function, BFGS is not as fast as Newton's method, but still very fast.



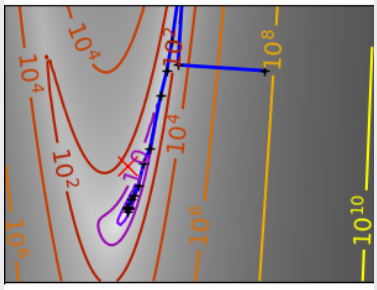


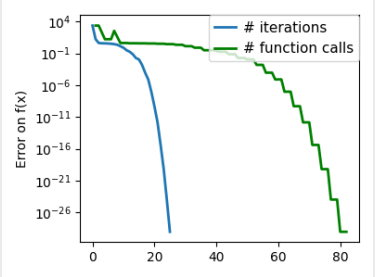
An ill-conditioned non-quadratic function:
Here BFGS does better than Newton, as its empirical estimate of the curvature is better than that given by the Hessian.





An ill-conditioned very non-quadratic function:






```
>>> def f(x):    # The rosenbrock function
...     return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2
>>> def jacobian(x):
...     return np.array((-2*.5*(1 - x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] -␣
↪x[0]**2)))
>>> sp.optimize.minimize(f, [2, -1], method="BFGS", jac=jacobian)
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 2.630637192365927e-16
       x: [ 1.000e+00  1.000e+00]
     nit: 8
     jac: [ 6.709e-08 -3.222e-08]
```

(continues on next page)

(continued from previous page)

```
hess_inv: [[ 9.999e-01  2.000e+00]
           [ 2.000e+00  4.499e+00]]
    nfev: 10
    njev: 10
```

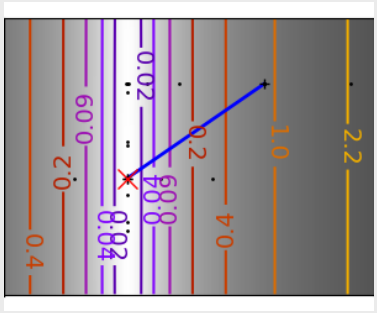
L-BFGS: Limited-memory BFGS Sits between BFGS and conjugate gradient: in very high dimensions (> 250) the Hessian matrix is too costly to compute and invert. L-BFGS keeps a low-rank version. In addition, box bounds are also supported by L-BFGS-B:

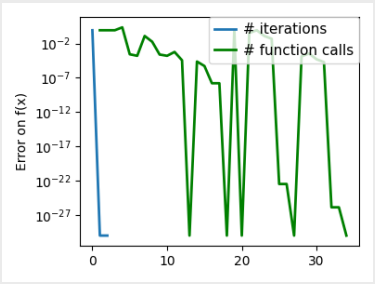
```
>>> def f(x):    # The rosenbrock function
...     return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2
>>> def jacobian(x):
...     return np.array((-2*.5*(1 - x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] -␣
↪x[0]**2)))
>>> sp.optimize.minimize(f, [2, 2], method="L-BFGS-B", jac=jacobian)
  message: CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL
  success: True
   status: 0
      fun: 1.4417677473...e-15
        x: [ 1.000e+00  1.000e+00]
      nit: 16
      jac: [ 1.023e-07 -2.593e-08]
     nfev: 17
     njev: 17
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```

13.4.12 Gradient-less methods

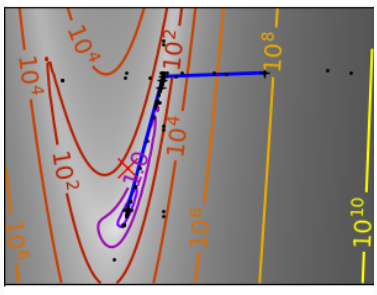
A shooting method: the Powell algorithm

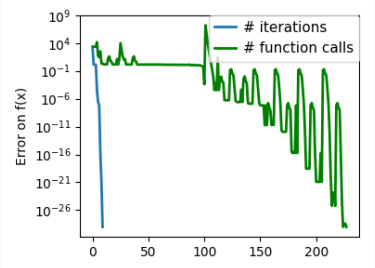
Almost a gradient approach





An ill-conditioned quadratic function: Powell's method isn't too sensitive to local ill-conditionning in low dimensions





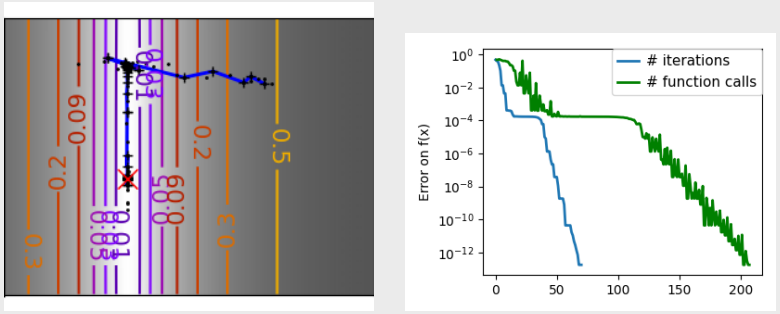
An ill-conditioned very non-quadratic function:

Simplex method: the Nelder-Mead

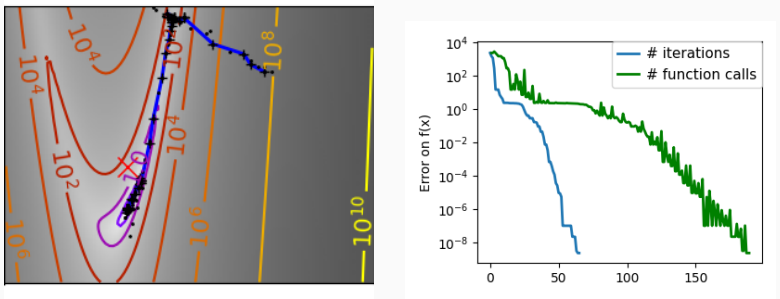
The Nelder-Mead algorithms is a generalization of dichotomy approaches to high-dimensional spaces. The algorithm works by refining a simplex, the generalization of intervals and triangles to high-dimensional spaces, to bracket the minimum.

Strong points: it is robust to noise, as it does not rely on computing gradients. Thus it can work on functions that are not locally smooth such as experimental data points, as long as they display a large-scale bell-shape behavior. However it is slower than gradient-based methods on smooth, non-noisy functions.

An ill-conditioned non-quadratic function:



An ill-conditioned very non-quadratic function:



Using the Nelder-Mead solver in `scipy.optimize.minimize()`:

```
>>> def f(x):    # The rosenbrock function
...     return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2
>>> sp.optimize.minimize(f, [2, -1], method="Nelder-Mead")
       message: Optimization terminated successfully.
       success: True
        status: 0
           fun: 1.11527915993744e-10
             x: [ 1.000e+00  1.000e+00]
           nit: 58
          nfev: 111
 final_simplex: (array([[ 1.000e+00,  1.000e+00],
                        [ 1.000e+00,  1.000e+00],
                        [ 1.000e+00,  1.000e+00]]), array([ 1.115e-10,  1.537e-10,  4.
↪988e-10]))
```

13.4.13 Global optimizers

If your problem does not admit a unique local minimum (which can be hard to test unless the function is convex), and you do not have prior information to initialize the optimization close to the solution, you may need a global optimizer.

Brute force: a grid search

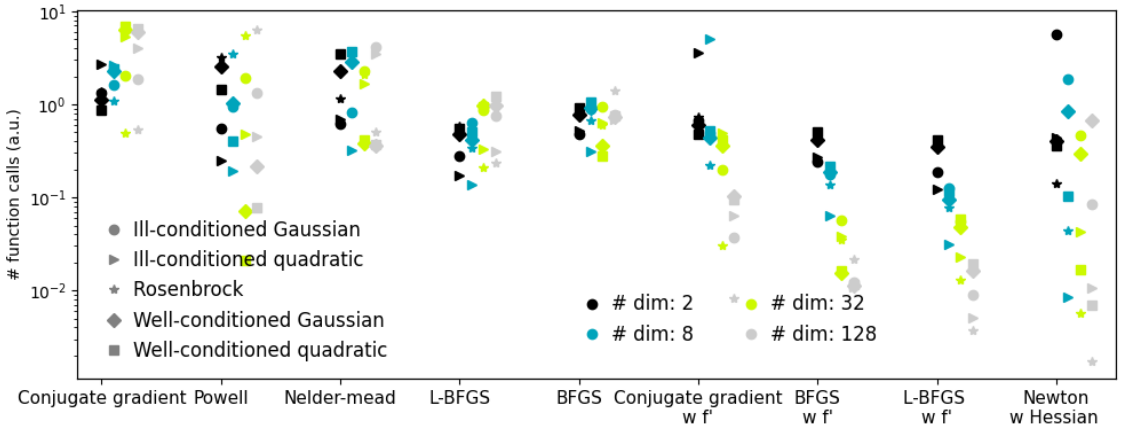
`scipy.optimize.brute()` evaluates the function on a given grid of parameters and returns the parameters corresponding to the minimum value. The parameters are specified with ranges given to `numpy.mgrid`. By default, 20 steps are taken in each direction:

```
>>> def f(x):    # The rosenbrock function
...     return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2
>>> sp.optimize.brute(f, ((-1, 2), (-1, 2)))
array([1.0000...,  1.0000...])
```

13.5 Practical guide to optimization with SciPy

13.5.1 Choosing a method

All methods are exposed as the `method` argument of `scipy.optimize.minimize()`.



Without knowledge of the gradient

- In general, prefer **BFGS** or **L-BFGS**, even if you have to approximate numerically gradients. These are also the default if you omit the parameter `method` - depending if the problem has constraints or bounds
- On well-conditioned problems, **Powell** and **Nelder-Mead**, both gradient-free methods, work well in high dimension, but they collapse for ill-conditioned problems.

With knowledge of the gradient

- **BFGS** or **L-BFGS**.
- Computational overhead of BFGS is larger than that L-BFGS, itself larger than that of conjugate gradient. On the other side, BFGS usually needs less function evaluations than CG. Thus conjugate gradient method is better than BFGS at optimizing computationally cheap functions.

With the Hessian

- If you can compute the Hessian, prefer the Newton method (**Newton-CG** or **TCG**).

If you have noisy measurements

- Use **Nelder-Mead** or **Powell**.

13.5.2 Making your optimizer faster

- Choose the right method (see above), do compute analytically the gradient and Hessian, if you can.
- Use preconditionning when possible.
- Choose your initialization points wisely. For instance, if you are running many similar optimizations, warm-restart one with the results of another.
- Relax the tolerance if you don't need precision using the parameter `tol`.

13.5.3 Computing gradients

Computing gradients, and even more Hessians, is very tedious but worth the effort. Symbolic computation with *Sympy* may come in handy.

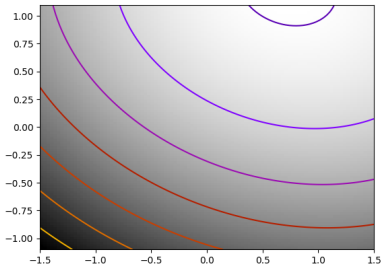
Warning

A *very* common source of optimization not converging well is human error in the computation of the gradient. You can use `scipy.optimize.check_grad()` to check that your gradient is correct. It returns the norm of the different between the gradient given, and a gradient computed numerically:

```
>>> sp.optimize.check_grad(f, jacobian, [2, -1])
np.float64(2.384185791015625e-07)
```

See also `scipy.optimize.approx_fprime()` to find your errors.

13.5.4 Synthetic exercises



Exercise: A simple (?) quadratic function

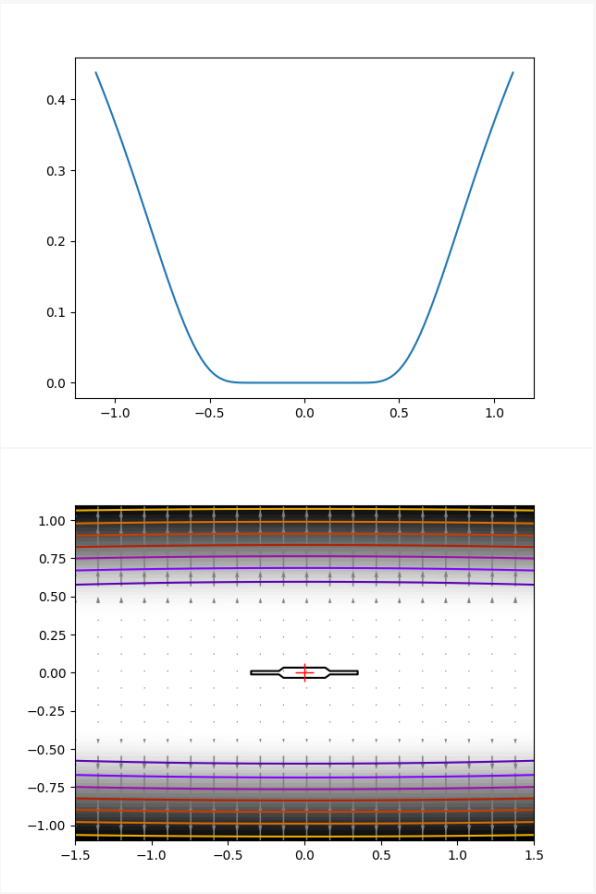
Optimize the following function, using K[0] as a starting point:

```
rng = np.random.default_rng(27446968)
K = rng.normal(size=(100, 100))

def f(x):
    return np.sum((K @ (x - 1))**2) + np.sum(x**2)**2
```

Time your approach. Find the fastest approach. Why is BFGS not working well?

Exercise: A locally flat minimum

Consider the function *exp(-1/(.1*x**2 + y**2)*. This function admits a minimum in (0, 0). Starting from an initialization at (1, 1), try to get within 1e-8 of this minimum point.



13.6 Special case: non-linear least-squares

13.6.1 Minimizing the norm of a vector function

Least square problems, minimizing the norm of a vector function, have a specific structure that can be used in the Levenberg–Marquardt algorithm implemented in `scipy.optimize.leastsq()`.

Lets try to minimize the norm of the following vectorial function:

```
>>> def f(x):
...     return np.arctan(x) - np.arctan(np.linspace(0, 1, len(x)))

>>> x0 = np.zeros(10)
>>> sp.optimize.leastsq(f, x0)
(array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ]), ...)
```

This took 67 function evaluations (check it with 'full_output=True'). What if we compute the norm ourselves and use a good generic optimizer (BFGS):

```
>>> def g(x):
...     return np.sum(f(x)**2)
>>> result = sp.optimize.minimize(g, x0, method="BFGS")
>>> result.fun
np.float64(2.6940...e-11)
```

BFGS needs more function calls, and gives a less precise result.

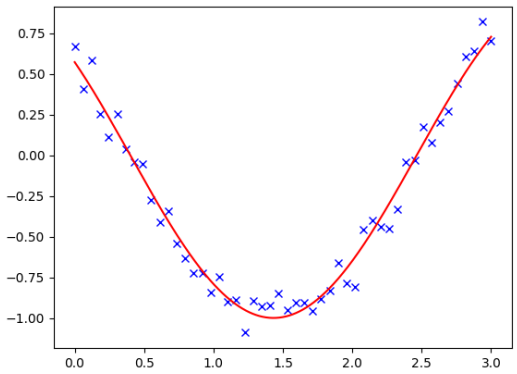
Note

leastsq is interesting compared to BFGS only if the dimensionality of the output vector is large, and larger than the number of parameters to optimize.

Warning

If the function is linear, this is a linear-algebra problem, and should be solved with `scipy.linalg.lstsq()`.

13.6.2 Curve fitting



Least square problems occur often when fitting a non-linear to data. While it is possible to construct our optimization problem ourselves, SciPy provides a helper function for this purpose: `scipy.optimize.curve_fit()`:

```
>>> def f(t, omega, phi):
...     return np.cos(omega * t + phi)

>>> x = np.linspace(0, 3, 50)
>>> rng = np.random.default_rng(27446968)
>>> y = f(x, 1.5, 1) + .1*rng.normal(size=50)

>>> sp.optimize.curve_fit(f, x, y)
(array([1.4812..., 0.9999...]), array([[ 0.0003..., -0.0004...],
       [-0.0004...,  0.0010...]]))
```

Exercise

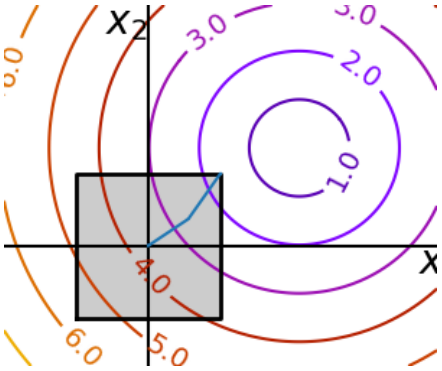
Do the same with omega = 3. What is the difficulty?

13.7 Optimization with constraints

13.7.1 Box bounds

Box bounds correspond to limiting each of the individual parameters of the optimization. Note that some problems that are not originally written as box bounds can be rewritten as such via change of variables. Both `scipy.optimize.minimize_scalar()` and `scipy.optimize.minimize()` support bound constraints with the parameter `bounds`:

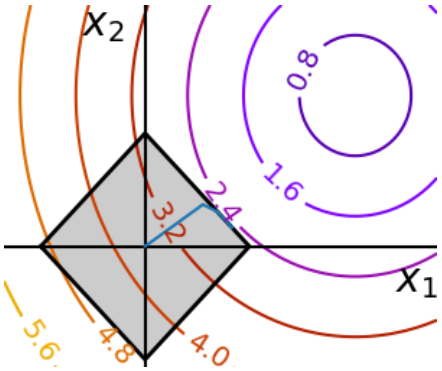
```
>>> def f(x):
...     return np.sqrt((x[0] - 3)**2 + (x[1] - 2)**2)
>>> sp.optimize.minimize(f, np.array([0, 0]), bounds=((-1.5, 1.5), (-1.5, 1.5)))
  message: CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL
  success: True
   status: 0
      fun: 1.5811388300841898
        x: [ 1.500e+00  1.500e+00]
      nit: 2
      jac: [-9.487e-01 -3.162e-01]
     nfev: 9
     njev: 3
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```



13.7.2 General constraints

Equality and inequality constraints specified as functions: $f(x) = 0$ and $g(x) < 0$.

- `scipy.optimize.fmin_slsqp()` Sequential least square programming: equality and inequality constraints:




```
>>> def f(x):
...     return np.sqrt((x[0] - 3)**2 + (x[1] - 2)**2)

>>> def constraint(x):
...     return np.atleast_1d(1.5 - np.sum(np.abs(x)))

>>> x0 = np.array([0, 0])
>>> sp.optimize.minimize(f, x0, constraints={"fun": constraint, "type": "ineq"})
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 2.47487373504...
```

(continues on next page)

(continued from previous page)

```
      x: [ 1.250e+00  2.500e-01]
    nit: 5
    jac: [-7.071e-01 -7.071e-01]
   nfev: 15
   njev: 5
```

Warning

The above problem is known as the Lasso problem in statistics, and there exist very efficient solvers for it (for instance in scikit-learn). In general do not use generic solvers when specific ones exist.

Lagrange multipliers

If you are ready to do a bit of math, many constrained optimization problems can be converted to non-constrained optimization problems using a mathematical trick known as Lagrange multipliers.

13.8 Full code examples

13.9 Examples for the mathematical optimization chapter

See also

Other Software

SciPy tries to include the best well-established, general-use, and permissively-licensed optimization algorithms available. However, even better options for a given task may be available in other libraries; please also see IPOPT and PyGMO.

CHAPTER 14

Interfacing with C

Author: *Valentin Haenel*

This chapter contains an *introduction* to the many different routes for making your native code (primarily `C/C++`) available from Python, a process commonly referred to *wrapping*. The goal of this chapter is to give you a flavour of what technologies exist and what their respective merits and shortcomings are, so that you can select the appropriate one for your specific needs. In any case, once you do start wrapping, you almost certainly will want to consult the respective documentation for your selected technique.

Chapters contents

- *Introduction*
- *Python-C-Api*
- *Ctypes*
- *SWIG*
- *Cython*
- *Summary*
- *Further Reading and References*
- *Exercises*

14.1 Introduction

This chapter covers the following techniques:

- Python-C-Api
- Ctypes
- SWIG (Simplified Wrapper and Interface Generator)
- Cython

These four techniques are perhaps the most well known ones, of which Cython is probably the most advanced one and the one you should consider using first. The others are also important, if you want to understand the wrapping problem from different angles. Having said that, there are other alternatives out there, but having understood the basics of the ones above, you will be in a position to evaluate the technique of your choice to see if it fits your needs.

The following criteria may be useful when evaluating a technology:

- Are additional libraries required?
- Is the code autogenerated?
- Does it need to be compiled?
- Is there good support for interacting with NumPy arrays?
- Does it support C++?

Before you set out, you should consider your use case. When interfacing with native code, there are usually two use-cases that come up:

- Existing code in C/C++ that needs to be leveraged, either because it already exists, or because it is faster.
- Python code too slow, push inner loops to native code

Each technology is demonstrated by wrapping the `cos` function from `math.h`. While this is a mostly a trivial example, it should serve us well to demonstrate the basics of the wrapping solution. Since each technique also includes some form of NumPy support, this is also demonstrated using an example where the cosine is computed on some kind of array.

Last but not least, two small warnings:

- All of these techniques may crash (segmentation fault) the Python interpreter, which is (usually) due to bugs in the C code.
- All the examples have been done on Linux, they *should* be possible on other operating systems.
- You will need a C compiler for most of the examples.

14.2 Python-C-Api

The Python-C-API is the backbone of the standard Python interpreter (a.k.a *CPython*). Using this API it is possible to write Python extension module in C and C++. Obviously, these extension modules can, by virtue of language compatibility, call any function written in C or C++.

When using the Python-C-API, one usually writes much boilerplate code, first to parse the arguments that were given to a function, and later to construct the return type.

Advantages

- Requires no additional libraries
- Lots of low-level control
- Entirely usable from C++

Disadvantages

- May require a substantial amount of effort
- Much overhead in the code
- Must be compiled
- High maintenance cost
- No forward compatibility across Python versions as C-API changes
- Reference count bugs are easy to create and very hard to track down.

Note

The Python-C-Api example here serves mainly for didactic reasons. Many of the other techniques actually depend on this, so it is good to have a high-level understanding of how it works. In 99% of the use-cases you will be better off, using an alternative technique.

Note

Since reference counting bugs are easy to create and hard to track down, anyone really needing to use the Python C-API should read the section about objects, types and reference counts from the official python documentation. Additionally, there is a tool by the name of cpychecker which can help discover common errors with reference counting.

14.2.1 Example

The following C-extension module, make the `cos` function from the standard math library available to Python:

```
/*  Example of wrapping cos function from math.h with the Python-C-API. */

#include <Python.h>
#include <math.h>

/*  wrapped cosine function */
static PyObject* cos_func(PyObject* self, PyObject* args)
{
    double value;
    double answer;

    /*  parse the input, from python float to c double */
    if (!PyArg_ParseTuple(args, "d", &value))
        return NULL;
    /* if the above function returns -1, an appropriate Python exception will
     * have been set, and the function simply returns NULL
     */

    /* call cos from libm */
    answer = cos(value);

    /*  construct the output from cos, from c double to python float */
    return Py_BuildValue("f", answer);
}

/*  define functions in module */
static PyMethodDef CosMethods[] =
{
```

(continues on next page)

(continued from previous page)

```
    {"cos_func", cos_func, METH_VARARGS, "evaluate the cosine"},
    {NULL, NULL, 0, NULL}
};

#if PY_MAJOR_VERSION >= 3
/* module initialization */
/* Python version 3*/
static struct PyModuleDef cModPyDem =
{
    PyModuleDef_HEAD_INIT,
    "cos_module", "Some documentation",
    -1,
    CosMethods
};

PyMODINIT_FUNC
PyInit_cos_module(void)
{
    return PyModule_Create(&cModPyDem);
}

#else

/* module initialization */
/* Python version 2 */
PyMODINIT_FUNC
initcos_module(void)
{
    (void) Py_InitModule("cos_module", CosMethods);
}

#endif
```

As you can see, there is much boilerplate, both to «massage» the arguments and return types into place and for the module initialisation. Although some of this is amortised, as the extension grows, the boilerplate required for each function(s) remains.

The standard python build system, `setuptools`, supports compiling C-extensions via a `setup.py` file:

```
from setuptools import setup, Extension


# define the extension module
cos_module = Extension("cos_module", sources=["cos_module.c"])

# run the setup
setup(ext_modules=[cos_module])
```

The setup file is called as follows:

```
$ cd advanced/interfacing_with_c/python_c_api

$ ls
cos_module.c  setup.py

$ python setup.py build_ext --inplace
running build_ext
```

(continues on next page)

(continued from previous page)

```
building 'cos_module' extension
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/include/python2.7 -c cos_module.c -o build/
↪temp.linux-x86_64-2.7/cos_module.o
gcc -pthread -shared build/temp.linux-x86_64-2.7/cos_module.o -L/home/esc/anaconda/
↪lib -lpython2.7 -o /home/esc/git-working/scientific-python-lectures/advanced/
↪interfacing_with_c/python_c_api/cos_module.so

$ ls
build/  cos_module.c  cos_module.so  setup.py
```

- `build_ext` is to build extension modules
- `--inplace` will output the compiled extension module into the current directory

The file `cos_module.so` contains the compiled extension, which we can now load in the IPython interpreter:

Note

In Python 3, the filename for compiled modules includes metadata on the Python interpreter (see PEP 3149) and is thus longer. The import statement is not affected by this.

```
In [1]: import cos_module

In [2]: cos_module?
Type:       module
String Form:<module 'cos_module' from 'cos_module.so'>
File:       /home/esc/git-working/scientific-python-lectures/advanced/interfacing_
↪with_c/python_c_api/cos_module.so
Docstring:  <no docstring>

In [3]: dir(cos_module)
Out[3]: ['__doc__', '__file__', '__name__', '__package__', 'cos_func']

In [4]: cos_module.cos_func(1.0)
Out[4]: 0.5403023058681398

In [5]: cos_module.cos_func(0.0)
Out[5]: 1.0

In [6]: cos_module.cos_func(3.14159265359)
Out[6]: -1.0
```

Now let's see how robust this is:

```
In [7]: cos_module.cos_func('foo')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-11bee483665d> in <module>()
----> 1 cos_module.cos_func('foo')
TypeError: a float is required
```

14.2.2 NumPy Support

Analog to the Python-C-API, NumPy, which is itself implemented as a C-extension, comes with the NumPy-C-API. This API can be used to create and manipulate NumPy arrays from C, when writing a custom C-extension. See also: *Advanced NumPy*.

> **Note**
>
> If you do ever need to use the NumPy C-API refer to the documentation about Arrays and Iterators.

The following example shows how to pass NumPy arrays as arguments to functions and how to iterate over NumPy arrays using the (old) NumPy-C-API. It simply takes an array as argument applies the cosine function from the `math.h` and returns a resulting new array.

```
/*  Example of wrapping the cos function from math.h using the NumPy-C-API. */

#include <Python.h>
#include <numpy/arrayobject.h>
#include <math.h>


/*  wrapped cosine function */
static PyObject* cos_func_np(PyObject* self, PyObject* args)
{
    PyArrayObject *arrays[2];  /* holds input and output array */
    PyObject *ret;
    NpyIter *iter;
    npy_uint32 op_flags[2];
    npy_uint32 iterator_flags;
    PyArray_Descr *op_dtypes[2];

    NpyIter_IterNextFunc *iternext;

    /*  parse single NumPy array argument */
    if (!PyArg_ParseTuple(args, "O!", &PyArray_Type, &arrays[0])) {
        return NULL;
    }

    arrays[1] = NULL;  /* The result will be allocated by the iterator */

    /* Set up and create the iterator */
    iterator_flags = (NPY_ITER_ZEROSIZE_OK |
                      /*
                       * Enable buffering in case the input is not behaved
                       * (native byte order or not aligned),
                       * disabling may speed up some cases when it is known to
                       * be unnecessary.
                       */
                      NPY_ITER_BUFFERED |
                      /* Manually handle innermost iteration for speed: */
                      NPY_ITER_EXTERNAL_LOOP |
                      NPY_ITER_GROWINNER);

    op_flags[0] = (NPY_ITER_READONLY |
                   /*
                    * Required that the arrays are well behaved, since the cos
                    * call below requires this.
```

(continues on next page)

(continued from previous page)

```
                 */
                NPY_ITER_NBO |
                NPY_ITER_ALIGNED);

    /* Ask the iterator to allocate an array to write the output to */
    op_flags[1] = NPY_ITER_WRITEONLY | NPY_ITER_ALLOCATE;

    /*
     * Ensure the iteration has the correct type, could be checked
     * specifically here.
     */
    op_dtypes[0] = PyArray_DescrFromType(NPY_DOUBLE);
    op_dtypes[1] = op_dtypes[0];

    /* Create the NumPy iterator object: */
    iter = NpyIter_MultiNew(2, arrays, iterator_flags,
                            /* Use input order for output and iteration */
                            NPY_KEEPORDER,
                            /* Allow only byte-swapping of input */
                            NPY_EQUIV_CASTING, op_flags, op_dtypes);
    Py_DECREF(op_dtypes[0]);  /* The second one is identical. */

    if (iter == NULL)
        return NULL;

    iternext = NpyIter_GetIterNext(iter, NULL);
    if (iternext == NULL) {
        NpyIter_Deallocate(iter);
        return NULL;
    }

    /* Fetch the output array which was allocated by the iterator: */
    ret = (PyObject *)NpyIter_GetOperandArray(iter)[1];
    Py_INCREF(ret);

    if (NpyIter_GetIterSize(iter) == 0) {
        /*
         * If there are no elements, the loop cannot be iterated.
         * This check is necessary with NPY_ITER_ZEROSIZE_OK.
         */
        NpyIter_Deallocate(iter);
        return ret;
    }

    /* The location of the data pointer which the iterator may update */
    char **dataptr = NpyIter_GetDataPtrArray(iter);
    /* The location of the stride which the iterator may update */
    npy_intp *strideptr = NpyIter_GetInnerStrideArray(iter);
    /* The location of the inner loop size which the iterator may update */
    npy_intp *innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);

    /* iterate over the arrays */
    do {
        npy_intp stride = strideptr[0];
        npy_intp count = *innersizeptr;
        /* out is always contiguous, so use double */
```

(continues on next page)

(continued from previous page)

```
        double *out = (double *)dataptr[1];
        char *in = dataptr[0];

        /* The output is allocated and guaranteed contiguous (out++ works): */
        assert(strideptr[1] == sizeof(double));

        /*
         * For optimization it can make sense to add a check for
         * stride == sizeof(double) to allow the compiler to optimize for that.
         */
        while (count--) {
            *out = cos(*(double *)in);
            out++;
            in += stride;
        }
    } while (iternext(iter));

    /* Clean up and return the result */
    NpyIter_Deallocate(iter);
    return ret;
}


/*  define functions in module */
static PyMethodDef CosMethods[] =
{
     {"cos_func_np", cos_func_np, METH_VARARGS,
         "evaluate the cosine on a NumPy array"},
     {NULL, NULL, 0, NULL}
};


#if PY_MAJOR_VERSION >= 3
/* module initialization */
/* Python version 3*/
static struct PyModuleDef cModPyDem = {
    PyModuleDef_HEAD_INIT,
    "cos_module", "Some documentation",
    -1,
    CosMethods
};
PyMODINIT_FUNC PyInit_cos_module_np(void) {
    PyObject *module;
    module = PyModule_Create(&cModPyDem);
    if(module==NULL) return NULL;
    /* IMPORTANT: this must be called */
    import_array();
    if (PyErr_Occurred()) return NULL;
    return module;
}

#else
/* module initialization */
/* Python version 2 */
PyMODINIT_FUNC initcos_module_np(void) {
    PyObject *module;
```

(continues on next page)

(continued from previous page)

```
    module = Py_InitModule("cos_module_np", CosMethods);
    if(module==NULL) return;
    /* IMPORTANT: this must be called */
    import_array();
    return;
}

#endif
```

To compile this we can use `setuptools` again. However we need to be sure to include the NumPy headers by using `numpy.get_include()`.

```
from setuptools import setup, Extension
import numpy


# define the extension module
cos_module_np = Extension(
    "cos_module_np", sources=["cos_module_np.c"], include_dirs=[numpy.get_include()]
)

# run the setup
setup(ext_modules=[cos_module_np])
```

To convince ourselves if this does actually works, we run the following test script:

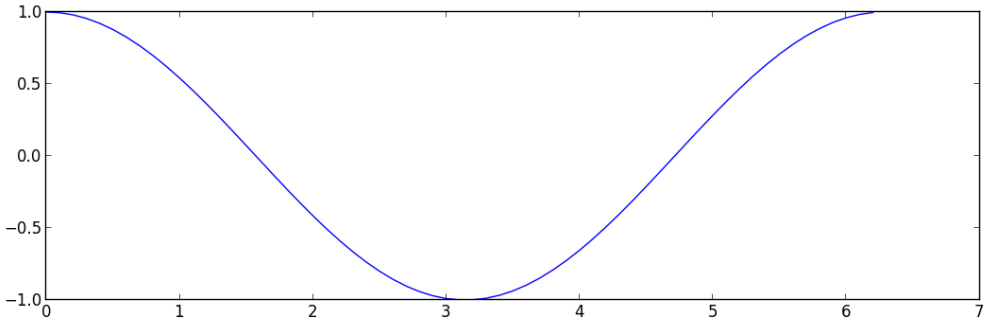
```
import cos_module_np
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 2 * np.pi, 0.1)
y = cos_module_np.cos_func_np(x)
plt.plot(x, y)
plt.show()


# Below are more specific tests for less common usage
# ---------------------------------------------------

# The function is OK with `x` not having any elements:
x_empty = np.array([], dtype=np.float64)
y_empty = cos_module_np.cos_func_np(x_empty)
assert np.array_equal(y_empty, np.array([], dtype=np.float64))

# The function can handle arbitrary dimensions and non-contiguous data.
# `x_2d` contains the same values, but has a different shape.
# Note: `x_2d.flags` shows it is not contiguous and `x2.ravel() == x`
x_2d = x.repeat(2)[::2].reshape(-1, 3)
y_2d = cos_module_np.cos_func_np(x_2d)
# When reshaped back, the same result is given:
assert np.array_equal(y_2d.ravel(), y)

# The function handles incorrect byte-order fine:
x_not_native_byteorder = x.astype(x.dtype.newbyteorder())
y_not_native_byteorder = cos_module_np.cos_func_np(x_not_native_byteorder)
assert np.array_equal(y_not_native_byteorder, y)
```

(continues on next page)

(continued from previous page)

```
# The function fails if the data type is incorrect:
x_incorrect_dtype = x.astype(np.float32)
try:
    cos_module_np.cos_func_np(x_incorrect_dtype)
    assert 0, "This cannot be reached."
except TypeError:
    # A TypeError will be raised, this can be changed by changing the
    # casting rule.
    pass
```

And this should result in the following figure:



14.3 Ctypes

Ctypes is a *foreign function library* for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

Advantages

- Part of the Python standard library
- Does not need to be compiled
- Wrapping code entirely in Python

Disadvantages

- Requires code to be wrapped to be available as a shared library (roughly speaking `*.dll` in Windows `*.so` in Linux and `*.dylib` in Mac OSX.)
- No good support for C++

14.3.1 Example

As advertised, the wrapper code is in pure Python.

```
"""Example of wrapping cos function from math.h using ctypes."""

import ctypes

# find and load the library

# OSX or linux
from ctypes.util import find_library

libm_name = find_library("m")
```

(continues on next page)

(continued from previous page)

```
assert libm_name is not None, "Cannot find libm (math) on this system :/ That's bad."

libm = ctypes.cdll.LoadLibrary(libm_name)

# Windows
# from ctypes import windll
# libm = cdll.msvcrt


# set the argument type
libm.cos.argtypes = [ctypes.c_double]
# set the return type
libm.cos.restype = ctypes.c_double


def cos_func(arg):
    """Wrapper for cos from math.h"""
    return libm.cos(arg)
```

- Finding and loading the library may vary depending on your operating system, check the documentation for details
- This may be somewhat deceptive, since the math library exists in compiled form on the system already. If you were to wrap a in-house library, you would have to compile it first, which may or may not require some additional effort.

We may now use this, as before:

```
In [8]: import cos_module

In [9]: cos_module?
Type:       module
String Form:<module 'cos_module' from 'cos_module.py'>
File:       /home/esc/git-working/scientific-python-lectures/advanced/interfacing_
↪with_c/ctypes/cos_module.py
Docstring:  <no docstring>

In [10]: dir(cos_module)
Out[10]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 'cos_func',
 'ctypes',
 'find_library',
 'libm']

In [11]: cos_module.cos_func(1.0)
Out[11]: 0.5403023058681398

In [12]: cos_module.cos_func(0.0)
Out[12]: 1.0

In [13]: cos_module.cos_func(3.14159265359)
Out[13]: -1.0
```

As with the previous example, this code is somewhat robust, although the error message is not quite as helpful, since it does not tell us what the type should be.

```
In [14]: cos_module.cos_func('foo')
---------------------------------------------------------------------------
ArgumentError                             Traceback (most recent call last)
<ipython-input-7-11bee483665d> in <module>()
----> 1 cos_module.cos_func('foo')
/home/esc/git-working/scientific-python-lectures/advanced/interfacing_with_c/ctypes/
↪cos_module.py in cos_func(arg)
     12 def cos_func(arg):
     13     ''' Wrapper for cos from math.h '''
---> 14     return libm.cos(arg)
ArgumentError: argument 1: <type 'exceptions.TypeError'>: wrong type
```

14.3.2 NumPy Support

NumPy contains some support for interfacing with ctypes. In particular there is support for exporting certain attributes of a NumPy array as ctypes data-types and there are functions to convert from C arrays to NumPy arrays and back.

For more information, consult the corresponding section in the NumPy Cookbook and the API documentation for numpy.ndarray.ctypes and numpy.ctypeslib.

For the following example, let's consider a C function in a library that takes an input and an output array, computes the cosine of the input array and stores the result in the output array.

The library consists of the following header file (although this is not strictly needed for this example, we list it for completeness):

```
void cos_doubles(double * in_array, double * out_array, int size);
```

The function implementation resides in the following C source file:

```
#include <math.h>

/*  Compute the cosine of each element in in_array, storing the result in
 *  out_array. */
void cos_doubles(double * in_array, double * out_array, int size){
    int i;
    for(i=0;i<size;i++){
        out_array[i] = cos(in_array[i]);
    }
}
```

And since the library is pure C, we can't use `setuptools` to compile it, but must use a combination of `make` and `gcc`:

```
m.PHONY : clean

libcos_doubles.so : cos_doubles.o
        gcc -shared -Wl,-soname,libcos_doubles.so -o libcos_doubles.so cos_doubles.o

cos_doubles.o : cos_doubles.c
        gcc -c -fPIC cos_doubles.c -o cos_doubles.o

clean :
        -rm -vf libcos_doubles.so cos_doubles.o cos_doubles.pyc
```

We can then compile this (on Linux) into the shared library `libcos_doubles.so`:

```
$ ls
cos_doubles.c  cos_doubles.h  cos_doubles.py  makefile  test_cos_doubles.py
$ make
gcc -c -fPIC cos_doubles.c -o cos_doubles.o
gcc -shared -Wl,-soname,libcos_doubles.so -o libcos_doubles.so cos_doubles.o
$ ls
cos_doubles.c  cos_doubles.o   libcos_doubles.so*  test_cos_doubles.py
cos_doubles.h  cos_doubles.py  makefile
```

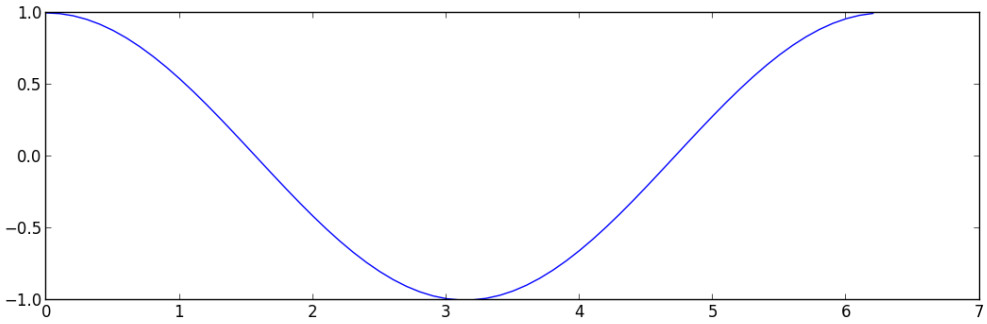
Now we can proceed to wrap this library via ctypes with direct support for (certain kinds of) NumPy arrays:

```
"""Example of wrapping a C library function that accepts a C double array as
input using the numpy.ctypeslib."""

import numpy as np
import numpy.ctypeslib as npct
from ctypes import c_int

# input type for the cos_doubles function
# must be a double array, with single dimension that is contiguous
array_1d_double = npct.ndpointer(dtype=np.double, ndim=1, flags="CONTIGUOUS")

# load the library, using NumPy mechanisms
libcd = npct.load_library("libcos_doubles", ".")

# setup the return types and argument types
libcd.cos_doubles.restype = None
libcd.cos_doubles.argtypes = [array_1d_double, array_1d_double, c_int]


def cos_doubles_func(in_array, out_array):
    return libcd.cos_doubles(in_array, out_array, len(in_array))
```

- Note the inherent limitation of contiguous single dimensional NumPy arrays, since the C functions requires this kind of buffer.
- Also note that the output array must be preallocated, for example with `numpy.zeros()` and the function will write into it's buffer.
- Although the original signature of the `cos_doubles` function is `ARRAY, ARRAY, int` the final `cos_doubles_func` takes only two NumPy arrays as arguments.

And, as before, we convince ourselves that it worked:

```
import numpy as np
import matplotlib.pyplot as plt
import cos_doubles

x = np.arange(0, 2 * np.pi, 0.1)
y = np.empty_like(x)

cos_doubles.cos_doubles_func(x, y)
plt.plot(x, y)
plt.show()
```



14.4 SWIG

SWIG, the Simplified Wrapper Interface Generator, is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages, including Python. The important thing with SWIG is, that it can autogenerate the wrapper code for you. While this is an advantage in terms of development time, it can also be a burden. The generated file tend to be quite large and may not be too human readable and the multiple levels of indirection which are a result of the wrapping process, may be a bit tricky to understand.

> **Note**
>
> The autogenerated C code uses the Python-C-Api.

Advantages

- Can automatically wrap entire libraries given the headers
- Works nicely with C++

Disadvantages

- Autogenerates enormous files
- Hard to debug if something goes wrong
- Steep learning curve

14.4.1 Example

Let's imagine that our `cos` function lives in a `cos_module` which has been written in `c` and consists of the source file `cos_module.c`:

```
#include <math.h>

double cos_func(double arg){
    return cos(arg);
}
```

and the header file `cos_module.h`:

```
double cos_func(double arg);
```

And our goal is to expose the `cos_func` to Python. To achieve this with SWIG, we must write an *interface file* which contains the instructions for SWIG.

```
/*  Example of wrapping cos function from math.h using SWIG. */
```

(continues on next page)

(continued from previous page)

```
%module cos_module
%{
    /* the resulting C file should be built as a python extension */
    #define SWIG_FILE_WITH_INIT
    /*  Includes the header in the wrapper code */
    #include "cos_module.h"
%}
/*  Parse the header file to generate wrappers */
%include "cos_module.h"
```

As you can see, not too much code is needed here. For this simple example it is enough to simply include the header file in the interface file, to expose the function to Python. However, SWIG does allow for more fine grained inclusion/exclusion of functions found in header files, check the documentation for details.

Generating the compiled wrappers is a two stage process:

1. Run the `swig` executable on the interface file to generate the files `cos_module_wrap.c`, which is the source file for the autogenerated Python C-extension and `cos_module.py`, which is the autogenerated pure python module.
2. Compile the `cos_module_wrap.c` into the `_cos_module.so`. Luckily, `setuptools` knows how to handle SWIG interface files, so that our `setup.py` is simply:

```
from setuptools import setup, Extension


setup(ext_modules=[Extension("_cos_module", sources=["cos_module.c", "cos_module.i
↪"])])
```

```
$ cd advanced/interfacing_with_c/swig

$ ls
cos_module.c  cos_module.h  cos_module.i  setup.py

$ python setup.py build_ext --inplace
running build_ext
building '_cos_module' extension
swigging cos_module.i to cos_module_wrap.c
swig -python -o cos_module_wrap.c cos_module.i
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/include/python2.7 -c cos_module.c -o build/
↪temp.linux-x86_64-2.7/cos_module.o
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/include/python2.7 -c cos_module_wrap.c -o␣
↪build/temp.linux-x86_64-2.7/cos_module_wrap.o
gcc -pthread -shared build/temp.linux-x86_64-2.7/cos_module.o build/temp.linux-x86_64-
↪2.7/cos_module_wrap.o -L/home/esc/anaconda/lib -lpython2.7 -o /home/esc/git-working/
↪scientific-python-lectures/advanced/interfacing_with_c/swig/_cos_module.so

$ ls
build/  cos_module.c  cos_module.h  cos_module.i  cos_module.py  _cos_module.so*  cos_
↪module_wrap.c  setup.py
```

We can now load and execute the `cos_module` as we have done in the previous examples:

```
In [15]: import cos_module

In [16]: cos_module?
Type:       module
String Form:<module 'cos_module' from 'cos_module.py'>
File:       /home/esc/git-working/scientific-python-lectures/advanced/interfacing_
↪with_c/swig/cos_module.py
Docstring:  <no docstring>

In [17]: dir(cos_module)
Out[17]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '_cos_module',
 '_newclass',
 '_object',
 '_swig_getattr',
 '_swig_property',
 '_swig_repr',
 '_swig_setattr',
 '_swig_setattr_nondynamic',
 'cos_func']

In [18]: cos_module.cos_func(1.0)
Out[18]: 0.5403023058681398

In [19]: cos_module.cos_func(0.0)
Out[19]: 1.0

In [20]: cos_module.cos_func(3.14159265359)
Out[20]: -1.0
```

Again we test for robustness, and we see that we get a better error message (although, strictly speaking in Python there is no `double` type):

```
In [21]: cos_module.cos_func('foo')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-11bee483665d> in <module>()
----> 1 cos_module.cos_func('foo')
TypeError: in method 'cos_func', argument 1 of type 'double'
```

14.4.2 NumPy Support

NumPy provides support for SWIG with the `numpy.i` file. This interface file defines various so-called *typemaps* which support conversion between NumPy arrays and C-Arrays. In the following example we will take a quick look at how such typemaps work in practice.

We have the same `cos_doubles` function as in the ctypes example:

```
void cos_doubles(double * in_array, double * out_array, int size);
```

```
#include <math.h>
```

(continues on next page)

(continued from previous page)

```
/*  Compute the cosine of each element in in_array, storing the result in
 *  out_array. */
void cos_doubles(double * in_array, double * out_array, int size){
    int i;
    for(i=0;i<size;i++){
        out_array[i] = cos(in_array[i]);
    }
}
```

This is wrapped as `cos_doubles_func` using the following SWIG interface file:

```
/*  Example of wrapping a C function that takes a C double array as input using
 *  NumPy typemaps for SWIG. */

%module cos_doubles
%{
    /* the resulting C file should be built as a python extension */
    #define SWIG_FILE_WITH_INIT
    /*  Includes the header in the wrapper code */
    #include "cos_doubles.h"
%}

/*  include the NumPy typemaps */
%include "numpy.i"
/*  need this for correct module initialization */
%init %{
    import_array();
%}

/*  typemaps for the two arrays, the second will be modified in-place */
%apply (double* IN_ARRAY1, int DIM1) {(double * in_array, int size_in)}
%apply (double* INPLACE_ARRAY1, int DIM1) {(double * out_array, int size_out)}

/*  Wrapper for cos_doubles that massages the types */
%inline %{
    /*  takes as input two NumPy arrays */
    void cos_doubles_func(double * in_array, int size_in, double * out_array, int␣
↪size_out) {
        /*  calls the original function, providing only the size of the first */
        cos_doubles(in_array, out_array, size_in);
    }
%}
```

- To use the NumPy typemaps, we need include the `numpy.i` file.
- Observe the call to `import_array()` which we encountered already in the NumPy-C-API example.
- Since the type maps only support the signature `ARRAY, SIZE` we need to wrap the `cos_doubles` as `cos_doubles_func` which takes two arrays including sizes as input.
- As opposed to the simple SWIG example, we don't include the `cos_doubles.h` header, There is nothing there that we wish to expose to Python since we expose the functionality through `cos_doubles_func`.

And, as before we can use `setuptools` to wrap this:

```
from setuptools import setup, Extension
import numpy
```

(continues on next page)

(continued from previous page)

```
setup(
    ext_modules=[
        Extension(
            "_cos_doubles",
            sources=["cos_doubles.c", "cos_doubles.i"],
            include_dirs=[numpy.get_include()],
        )
    ]
)
```

As previously, we need to use `include_dirs` to specify the location.

```
$ ls
cos_doubles.c  cos_doubles.h  cos_doubles.i  numpy.i  setup.py  test_cos_doubles.py
$ python setup.py build_ext -i
running build_ext
building '_cos_doubles' extension
swigging cos_doubles.i to cos_doubles_wrap.c
swig -python -o cos_doubles_wrap.c cos_doubles.i
cos_doubles.i:24: Warning(490): Fragment 'NumPy_Backward_Compatibility' not found.
cos_doubles.i:24: Warning(490): Fragment 'NumPy_Backward_Compatibility' not found.
cos_doubles.i:24: Warning(490): Fragment 'NumPy_Backward_Compatibility' not found.
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include -I/home/esc/anaconda/include/python2.7 -c cos_doubles.c -o build/temp.linux-
↪x86_64-2.7/cos_doubles.o
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include -I/home/esc/anaconda/include/python2.7 -c cos_doubles_wrap.c -o build/temp.
↪linux-x86_64-2.7/cos_doubles_wrap.o
In file included from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include/numpy/ndarraytypes.h:1722,
                 from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include/numpy/ndarrayobject.h:17,
                 from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include/numpy/arrayobject.h:15,
                 from cos_doubles_wrap.c:2706:
/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/npy_
↪deprecated_api.h:11:2: warning: #warning "Using deprecated NumPy API, disable it by
↪#defining NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION"
gcc -pthread -shared build/temp.linux-x86_64-2.7/cos_doubles.o build/temp.linux-x86_
↪64-2.7/cos_doubles_wrap.o -L/home/esc/anaconda/lib -lpython2.7 -o /home/esc/git-
↪working/scientific-python-lectures/advanced/interfacing_with_c/swig_numpy/_cos_
↪doubles.so
$ ls
build/         cos_doubles.h  cos_doubles.py    cos_doubles_wrap.c  setup.py
cos_doubles.c  cos_doubles.i  _cos_doubles.so*  numpy.i             test_cos_doubles.
↪py
```

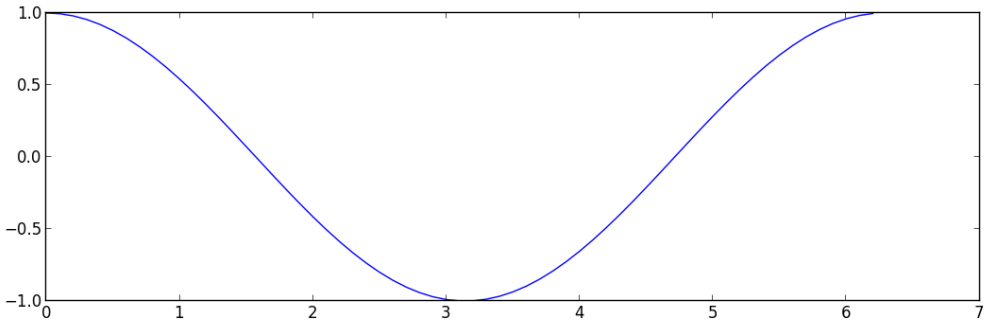
And, as before, we convince ourselves that it worked:

```
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import cos_doubles

x = np.arange(0, 2 * np.pi, 0.1)
y = np.empty_like(x)

cos_doubles.cos_doubles_func(x, y)
plt.plot(x, y)
plt.show()
```



14.5 Cython

Cython is both a Python-like language for writing C-extensions and an advanced compiler for this language. The Cython *language* is a superset of Python, which comes with additional constructs that allow you call C functions and annotate variables and class attributes with c types. In this sense one could also call it a *Python with types*.

In addition to the basic use case of wrapping native code, Cython supports an additional use-case, namely interactive optimization. Basically, one starts out with a pure-Python script and incrementally adds Cython types to the bottleneck code to optimize only those code paths that really matter.

In this sense it is quite similar to SWIG, since the code can be autogenerated but in a sense it also quite similar to ctypes since the wrapping code can (almost) be written in Python.

While others solutions that autogenerate code can be quite difficult to debug (for example SWIG) Cython comes with an extension to the GNU debugger that helps debug Python, Cython and C code.

Note

The autogenerated C code uses the Python-C-Api.

Advantages

- Python like language for writing C-extensions
- Autogenerated code
- Supports incremental optimization
- Includes a GNU debugger extension
- Support for C++ (Since version 0.13)

Disadvantages

- Must be compiled

- Requires an additional library (but only at build time, at this problem can be overcome by shipping the generated C files)

14.5.1 Example

The main Cython code for our `cos_module` is contained in the file `cos_module.pyx`:

```
""" Example of wrapping cos function from math.h using Cython. """

cdef extern from "math.h":
    double cos(double arg)

def cos_func(arg):
    return cos(arg)
```

Note the additional keywords such as `cdef` and `extern`. Also the `cos_func` is then pure Python.

Again we can use the standard `setuptools` module, but this time we need some additional pieces from `Cython.Build`:

```
from setuptools import setup, Extension
from Cython.Build import cythonize


extensions = [Extension("cos_module", sources=["cos_module.pyx"])]

setup(ext_modules=cythonize(extensions))
```

Compiling this:

```
$ cd advanced/interfacing_with_c/cython
$ ls
cos_module.pyx  setup.py
$ python setup.py build_ext --inplace
running build_ext
cythoning cos_module.pyx to cos_module.c
building 'cos_module' extension
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/include/python2.7 -c cos_module.c -o build/
↪temp.linux-x86_64-2.7/cos_module.o
gcc -pthread -shared build/temp.linux-x86_64-2.7/cos_module.o -L/home/esc/anaconda/
↪lib -lpython2.7 -o /home/esc/git-working/scientific-python-lectures/advanced/
↪interfacing_with_c/cython/cos_module.so
$ ls
build/  cos_module.c  cos_module.pyx  cos_module.so*  setup.py
```

And running it:

```
In [22]: import cos_module

In [23]: cos_module?
Type:       module
String Form:<module 'cos_module' from 'cos_module.so'>
File:       /home/esc/git-working/scientific-python-lectures/advanced/interfacing_
↪with_c/cython/cos_module.so
Docstring:  <no docstring>
```

(continues on next page)

(continued from previous page)

```
In [24]: dir(cos_module)
Out[24]:
['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__test__',
 'cos_func']

In [25]: cos_module.cos_func(1.0)
Out[25]: 0.5403023058681398

In [26]: cos_module.cos_func(0.0)
Out[26]: 1.0

In [27]: cos_module.cos_func(3.14159265359)
Out[27]: -1.0
```

And, testing a little for robustness, we can see that we get good error messages:

```
In [28]: cos_module.cos_func('foo')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-11bee483665d> in <module>()
----> 1 cos_module.cos_func('foo')
/home/esc/git-working/scientific-python-lectures/advanced/interfacing_with_c/cython/
↪cos_module.so in cos_module.cos_func (cos_module.c:506)()
TypeError: a float is required
```

Additionally, it is worth noting that `Cython` ships with complete declarations for the C math library, which simplifies the code above to become:

```
""" Simpler example of wrapping cos function from math.h using Cython. """

from libc.math cimport cos

def cos_func(arg):
    return cos(arg)
```

In this case the `cimport` statement is used to import the `cos` function.

14.5.2 NumPy Support

Cython has support for NumPy via the `numpy.pyx` file which allows you to add the NumPy array type to your Cython code. I.e. like specifying that variable `i` is of type `int`, you can specify that variable `a` is of type `numpy.ndarray` with a given `dtype`. Also, certain optimizations such as bounds checking are supported. Look at the corresponding section in the Cython documentation. In case you want to pass NumPy arrays as C arrays to your Cython wrapped C functions, there is a section about this in the Cython documentation.

In the following example, we will show how to wrap the familiar `cos_doubles` function using Cython.

```
void cos_doubles(double * in_array, double * out_array, int size);
```

```
#include <math.h>
```

(continues on next page)

(continued from previous page)

```
/*  Compute the cosine of each element in in_array, storing the result in
 *  out_array. */
void cos_doubles(double * in_array, double * out_array, int size){
    int i;
    for(i=0;i<size;i++){
        out_array[i] = cos(in_array[i]);
    }
}
```

This is wrapped as `cos_doubles_func` using the following Cython code:

```
""" Example of wrapping a C function that takes C double arrays as input using
    the NumPy declarations from Cython """

# cimport the Cython declarations for NumPy
cimport numpy as np

# if you want to use the NumPy-C-API from Cython
# (not strictly necessary for this example, but good practice)
np.import_array()

# cdefine the signature of our c function
cdef extern from "cos_doubles.h":
    void cos_doubles (double * in_array, double * out_array, int size)

# create the wrapper code, with NumPy type annotations
def cos_doubles_func(np.ndarray[double, ndim=1, mode="c"] in_array not None,
                     np.ndarray[double, ndim=1, mode="c"] out_array not None):
    cos_doubles(<double*> np.PyArray_DATA(in_array),
                <double*> np.PyArray_DATA(out_array),
                in_array.shape[0])
```

And can be compiled using `setuptools`:

```
from setuptools import setup, Extension
from Cython.Build import cythonize
import numpy


extensions = [
    Extension(
        "cos_doubles",
        sources=["_cos_doubles.pyx", "cos_doubles.c"],
        include_dirs=[numpy.get_include()],
    )
]

setup(ext_modules=cythonize(extensions))
```

- As with the previous compiled NumPy examples, we need the `include_dirs` option.

```
$ ls
cos_doubles.c  cos_doubles.h  _cos_doubles.pyx  setup.py  test_cos_doubles.py
$ python setup.py build_ext -i
running build_ext
cythoning _cos_doubles.pyx to _cos_doubles.c
building 'cos_doubles' extension
```

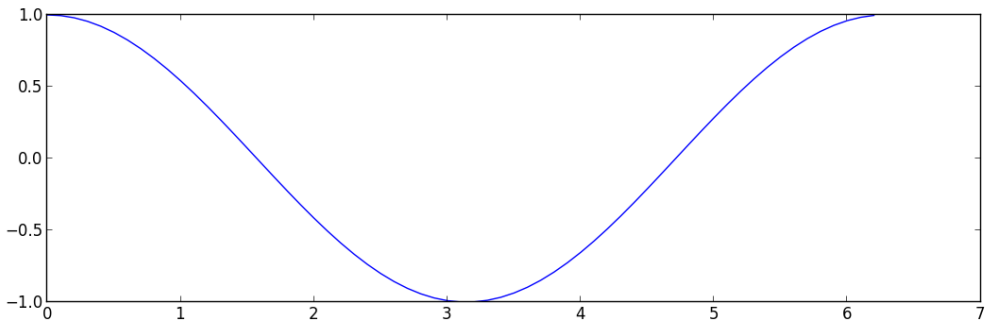
(continues on next page)

(continued from previous page)

```
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include -I/home/esc/anaconda/include/python2.7 -c _cos_doubles.c -o build/temp.
↪linux-x86_64-2.7/_cos_doubles.o
In file included from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include/numpy/ndarraytypes.h:1722,
                 from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include/numpy/ndarrayobject.h:17,
                 from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include/numpy/arrayobject.h:15,
                 from _cos_doubles.c:253:
/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/npy_
↪deprecated_api.h:11:2: warning: #warning "Using deprecated NumPy API, disable it by
↪#defining NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION"
/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/__ufunc_api.
↪h:236: warning: '_import_umath' defined but not used
gcc -pthread -fno-strict-aliasing -g -O2 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↪prototypes -fPIC -I/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/
↪include -I/home/esc/anaconda/include/python2.7 -c cos_doubles.c -o build/temp.linux-
↪x86_64-2.7/cos_doubles.o
gcc -pthread -shared build/temp.linux-x86_64-2.7/_cos_doubles.o build/temp.linux-x86_
↪64-2.7/cos_doubles.o -L/home/esc/anaconda/lib -lpython2.7 -o /home/esc/git-working/
↪scientific-python-lectures/advanced/interfacing_with_c/cython_numpy/cos_doubles.so
$ ls
build/  _cos_doubles.c  cos_doubles.c  cos_doubles.h  _cos_doubles.pyx  cos_doubles.
↪so*  setup.py  test_cos_doubles.py
```

And, as before, we convince ourselves that it worked:

```
import numpy as np
import matplotlib.pyplot as plt
import cos_doubles

x = np.arange(0, 2 * np.pi, 0.1)
y = np.empty_like(x)

cos_doubles.cos_doubles_func(x, y)
plt.plot(x, y)
plt.show()
```



14.6 Summary

In this section four different techniques for interfacing with native code have been presented. The table below roughly summarizes some of the aspects of the techniques.

x	Part of CPython	Compiled	Autogenerated	NumPy Support
Python-C-API	`True`	`True`	`False`	`True`
Ctypes	`True`	`False`	`False`	`True`
Swig	`False`	`True`	`True`	`True`
Cython	`False`	`True`	`True`	`True`

Of all three presented techniques, Cython is the most modern and advanced. In particular, the ability to optimize code incrementally by adding types to your Python code is unique.

14.7 Further Reading and References

- Gaël Varoquaux's blog post about avoiding data copies provides some insight on how to handle memory management cleverly. If you ever run into issues with large datasets, this is a reference to come back to for some inspiration.

14.8 Exercises

Since this is a brand new section, the exercises are considered more as pointers as to what to look at next, so pick the ones that you find more interesting. If you have good ideas for exercises, please let us know!

1. Download the source code for each example and compile and run them on your machine.
2. Make trivial changes to each example and convince yourself that this works. (E.g. change `cos` for `sin`.)
3. Most of the examples, especially the ones involving NumPy may still be fragile and respond badly to input errors. Look for ways to crash the examples, figure what the problem is and devise a potential solution. Here are some ideas:
    1. Numerical overflow.
    2. Input and output arrays that have different lengths.
    3. Multidimensional array.
    4. Empty array
    5. Arrays with non-`double` types
4. Use the `%timeit` IPython magic to measure the execution time of the various solutions

14.8.1 Python-C-API

1. Modify the NumPy example such that the function takes two input arguments, where the second is the preallocated output array, making it similar to the other NumPy examples.
2. Modify the example such that the function only takes a single input array and modifies this in place.
3. Try to fix the example to use the new NumPy iterator protocol. If you manage to obtain a working solution, please submit a pull-request on github.
4. You may have noticed, that the NumPy-C-API example is the only NumPy example that does not wrap `cos_doubles` but instead applies the `cos` function directly to the elements of the NumPy array. Does this have any advantages over the other techniques.

5. Can you wrap `cos_doubles` using only the NumPy-C-API. You may need to ensure that the arrays have the correct type, are one dimensional and contiguous in memory.

14.8.2 Ctypes

1. Modify the NumPy example such that `cos_doubles_func` handles the preallocation for you, thus making it more like the NumPy-C-API example.

14.8.3 SWIG

1. Look at the code that SWIG autogenerates, how much of it do you understand?
2. Modify the NumPy example such that `cos_doubles_func` handles the preallocation for you, thus making it more like the NumPy-C-API example.
3. Modify the `cos_doubles` C function so that it returns an allocated array. Can you wrap this using SWIG typemaps? If not, why not? Is there a workaround for this specific situation? (Hint: you know the size of the output array, so it may be possible to construct a NumPy array from the returned `double *`.)

14.8.4 Cython

1. Look at the code that Cython autogenerates. Take a closer look at some of the comments that Cython inserts. What do you see?
2. Look at the section Working with NumPy from the Cython documentation to learn how to incrementally optimize a pure python script that uses NumPy.
3. Modify the NumPy example such that `cos_doubles_func` handles the preallocation for you, thus making it more like the NumPy-C-API example.

Part III

Packages and applications

This part of the *Scientific Python Lectures* is dedicated to various scientific packages useful for extended needs.

CHAPTER 15

Statistics in Python

Author: *Gaël Varoquaux*

Requirements

- Standard scientific Python environment (NumPy, SciPy, matplotlib)
- Pandas
- Statsmodels
- Seaborn

To install Python and these dependencies, we recommend that you download Anaconda Python or, preferably, use the package manager if you are under Ubuntu or other linux.

See also

- **Bayesian statistics in Python**: This chapter does not cover tools for Bayesian statistics. Of particular interest for Bayesian modelling is PyMC, which implements a probabilistic programming language in Python.
- **Read a statistics book**: The Think stats book is available as free PDF or in print and is a great introduction to statistics.

Tip

Why Python for statistics?

R is a language dedicated to statistics. Python is a general-purpose language with statistics modules. R has more statistical analysis features than Python, and specialized syntaxes. However, when it comes to building complex analysis pipelines that mix statistics with e.g. image analysis, text mining, or control of a physical experiment, the richness of Python is an invaluable asset.

Contents

- *Data representation and interaction*
  - *Data as a table*
  - *The pandas data-frame*
- *Hypothesis testing: comparing two groups*
  - *Student's t-test: the simplest statistical test*
  - *Paired tests: repeated measurements on the same individuals*
- *Linear models, multiple factors, and analysis of variance*
  - *"formulas" to specify statistical models in Python*
  - *Multiple Regression: including multiple factors*
  - *Post-hoc hypothesis testing: analysis of variance (ANOVA)*
- *More visualization: seaborn for statistical exploration*
  - *Pairplot: scatter matrices*
  - *lmplot: plotting a univariate regression*
- *Testing for interactions*
- *Full code for the figures*
- *Solutions to this chapter's exercises*

Tip

In this document, the Python inputs are represented with the sign ">>>".

Disclaimer: Gender questions

Some of the examples of this tutorial are chosen around gender questions. The reason is that on such questions controlling the truth of a claim actually matters to many people.

15.1 Data representation and interaction

15.1.1 Data as a table

The setting that we consider for statistical analysis is that of multiple *observations* or *samples* described by a set of different *attributes* or *features*. The data can than be seen as a 2D table, or matrix, with

columns giving the different attributes of the data, and rows the observations. For instance, the data contained in `examples/brain_size.csv`:

```
"";"Gender";"FSIQ";"VIQ";"PIQ";"Weight";"Height";"MRI_Count"
"1";"Female";133;132;124;"118";"64.5";816932
"2";"Male";140;150;124;".";"72.5";1001121
"3";"Male";139;123;150;"143";"73.3";1038437
"4";"Male";133;129;128;"172";"68.8";965353
"5";"Female";137;132;134;"147";"65.0";951545
```

15.1.2 The pandas data-frame

Tip

We will store and manipulate this data in a `pandas.DataFrame`, from the pandas module. It is the Python equivalent of the spreadsheet table. It is different from a 2D `numpy` array as it has named columns, can contain a mixture of different data types by column, and has elaborate selection and pivotal mechanisms.

Creating dataframes: reading data files or converting arrays

Separator

It is a CSV file, but the separator is ";"

Reading from a CSV file: Using the above CSV file that gives observations of brain size and weight and IQ (Willerman et al. 1991), the data are a mixture of numerical and categorical values:

```
>>> import pandas
>>> data = pandas.read_csv('examples/brain_size.csv', sep=';', na_values=".")
>>> data
   Unnamed: 0  Gender  FSIQ  VIQ  PIQ  Weight  Height  MRI_Count
0           1  Female   133  132  124   118.0    64.5     816932
1           2    Male   140  150  124     NaN    72.5    1001121
2           3    Male   139  123  150   143.0    73.3    1038437
3           4    Male   133  129  128   172.0    68.8     965353
4           5  Female   137  132  134   147.0    65.0     951545
...
```

Warning

Missing values

The weight of the second individual is missing in the CSV file. If we don't specify the missing value (NA = not available) marker, we will not be able to do statistical analysis.

Creating from arrays: A `pandas.DataFrame` can also be seen as a dictionary of 1D 'series', eg arrays or lists. If we have 3 `numpy` arrays:

```
>>> import numpy as np
>>> t = np.linspace(-6, 6, 20)
>>> sin_t = np.sin(t)
>>> cos_t = np.cos(t)
```

We can expose them as a pandas.DataFrame:

```
>>> pandas.DataFrame({'t': t, 'sin': sin_t, 'cos': cos_t})
          t       sin       cos
0  -6.000000  0.279415  0.960170
1  -5.368421  0.792419  0.609977
2  -4.736842  0.999701  0.024451
3  -4.105263  0.821291 -0.570509
4  -3.473684  0.326021 -0.945363
5  -2.842105 -0.295030 -0.955488
6  -2.210526 -0.802257 -0.596979
7  -1.578947 -0.999967 -0.008151
8  -0.947368 -0.811882  0.583822
...
```

Other inputs: pandas can input data from SQL, excel files, or other formats. See the pandas documentation.

Manipulating data

data is a pandas.DataFrame, that resembles R's dataframe:

```
>>> data.shape    # 40 rows and 8 columns
(40, 8)

>>> data.columns  # It has columns
Index(['Unnamed: 0', 'Gender', 'FSIQ', 'VIQ', 'PIQ', 'Weight', 'Height',
       'MRI_Count'],
      dtype='object')

>>> print(data['Gender'])  # Columns can be addressed by name
0     Female
1       Male
2       Male
3       Male
4     Female
...

>>> # Simpler selector
>>> data[data['Gender'] == 'Female']['VIQ'].mean()
np.float64(109.45)
```

Note

For a quick view on a large dataframe, use its *describe* method: `pandas.DataFrame.describe()`.

groupby: splitting a dataframe on values of categorical variables:

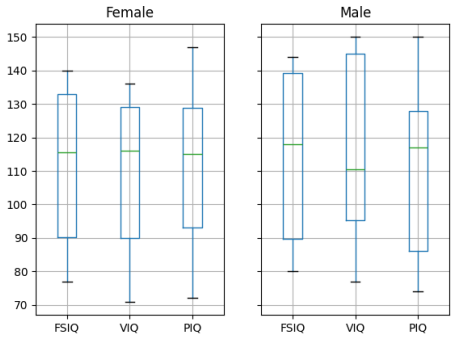
```
>>> groupby_gender = data.groupby('Gender')
>>> for gender, value in groupby_gender['VIQ']:
...     print((gender, value.mean()))
('Female', np.float64(109.45))
('Male', np.float64(115.25))
```

groupby_gender is a powerful object that exposes many operations on the resulting group of dataframes:

```
>>> groupby_gender.mean()
        Unnamed: 0   FSIQ     VIQ     PIQ      Weight     Height  MRI_Count
Gender
Female       19.65  111.9  109.45  110.45  137.200000  65.765000   862654.6
Male         21.35  115.0  115.25  111.60  166.444444  71.431579   954855.4
```

Tip

Use tab-completion on *groupby_gender* to find more. Other common grouping functions are median, count (useful for checking to see the amount of missing values in different subsets) or sum. Groupby evaluation is lazy, no work is done until an aggregation function is applied.



Exercise

- What is the mean value for VIQ for the full population?
- How many males/females were included in this study?

  Hint use 'tab completion' to find out the methods that can be called, instead of 'mean' in the above example.

- What is the average value of MRI counts expressed in log units, for males and females?

Note

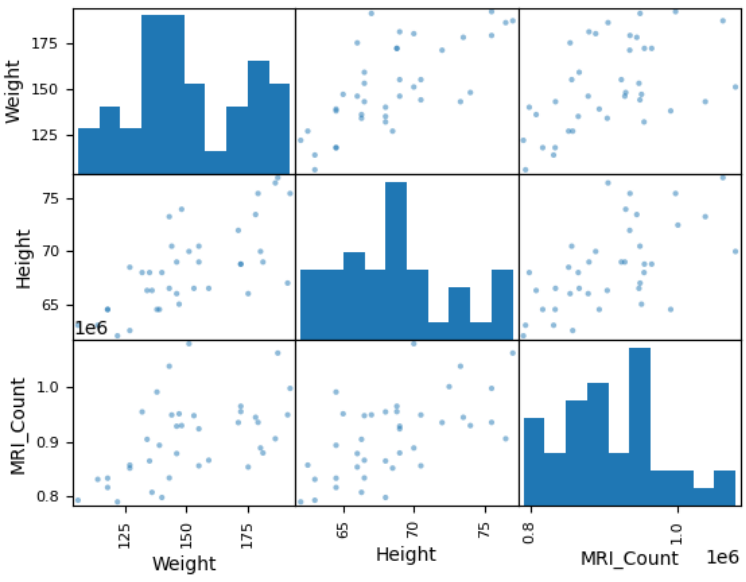
groupby_gender.boxplot is used for the plots above (see this example).

Plotting data

Pandas comes with some plotting tools (`pandas.plotting`, using matplotlib behind the scene) to display statistics of the data in dataframes:

Scatter matrices:

```
>>> from pandas import plotting
>>> plotting.scatter_matrix(data[['Weight', 'Height', 'MRI_Count']])
array([[<Axes: xlabel='Weight', ylabel='Weight'>,
        <Axes: xlabel='Height', ylabel='Weight'>,
        <Axes: xlabel='MRI_Count', ylabel='Weight'>],
       [<Axes: xlabel='Weight', ylabel='Height'>,
        <Axes: xlabel='Height', ylabel='Height'>,
        <Axes: xlabel='MRI_Count', ylabel='Height'>],
       [<Axes: xlabel='Weight', ylabel='MRI_Count'>,
        <Axes: xlabel='Height', ylabel='MRI_Count'>,
        <Axes: xlabel='MRI_Count', ylabel='MRI_Count'>]], dtype=object)
```




```
>>> plotting.scatter_matrix(data[['PIQ', 'VIQ', 'FSIQ']])
array([[<Axes: xlabel='PIQ', ylabel='PIQ'>,
        <Axes: xlabel='VIQ', ylabel='PIQ'>,
        <Axes: xlabel='FSIQ', ylabel='PIQ'>],
       [<Axes: xlabel='PIQ', ylabel='VIQ'>,
        <Axes: xlabel='VIQ', ylabel='VIQ'>,
```

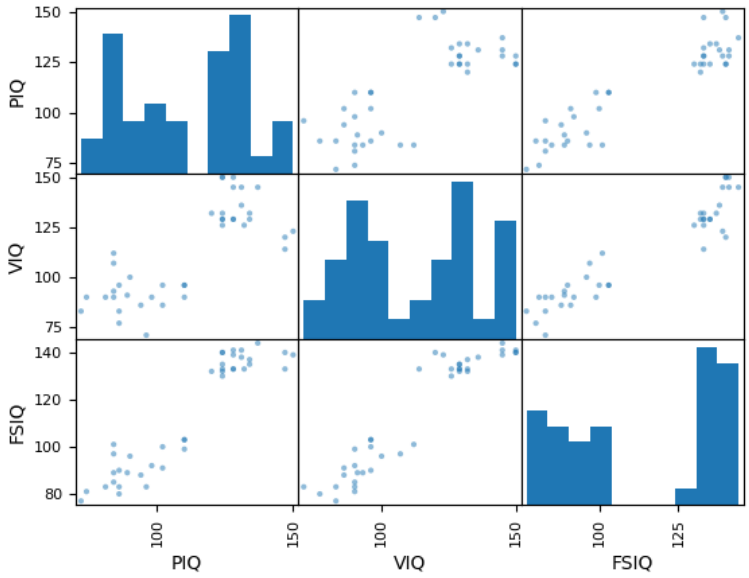
(continues on next page)

(continued from previous page)

```
        <Axes: xlabel='FSIQ', ylabel='VIQ'>],
       [<Axes: xlabel='PIQ', ylabel='FSIQ'>,
        <Axes: xlabel='VIQ', ylabel='FSIQ'>,
        <Axes: xlabel='FSIQ', ylabel='FSIQ'>]], dtype=object)
```

Two populations

The IQ metrics are bimodal, as if there are 2 sub-populations.



Exercise

Plot the scatter matrix for males only, and for females only. Do you think that the 2 sub-populations correspond to gender?

15.2 Hypothesis testing: comparing two groups

For simple statistical tests, we will use the `scipy.stats` sub-module of SciPy:

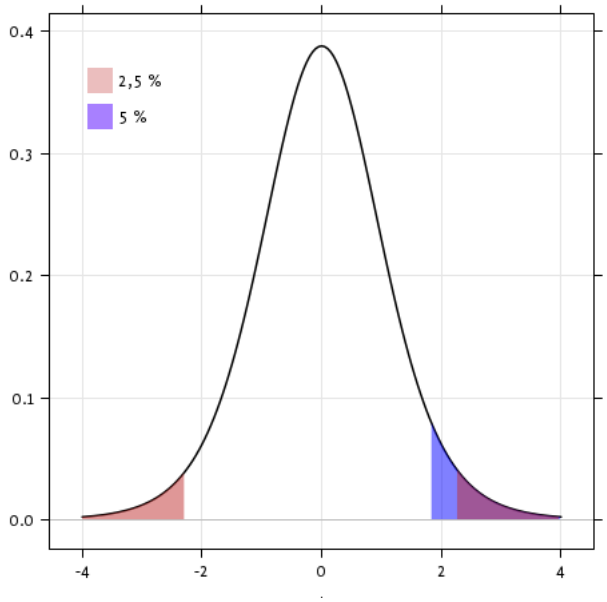
```
>>> import scipy as sp
```

See also

SciPy is a vast library. For a quick summary to the whole library, see the *scipy* chapter.

15.2.1 Student's t-test: the simplest statistical test

One-sample tests: testing the value of a population mean



`scipy.stats.ttest_1samp()` tests the null hypothesis that the mean of the population underlying the data is equal to a given value. It returns the T statistic, and the p-value (see the function's help):

```
>>> sp.stats.ttest_1samp(data['VIQ'], 0)
TtestResult(statistic=np.float64(30.088099970...), pvalue=np.float64(1.32891964...e-
↪28), df=np.int64(39))
```

The p-value of 10^-28 indicates that such an extreme value of the statistic is unlikely to be observed under the null hypothesis. This may be taken as evidence that the null hypothesis is false and that the population mean IQ (VIQ measure) is not 0.

Technically, the p-value of the t-test is derived under the assumption that the means of samples drawn from the population are normally distributed. This condition is exactly satisfied when the population itself is normally distributed; however, due to the central limit theorem, the condition is nearly true for reasonably large samples drawn from populations that follow a variety of non-normal distributions.

Nonetheless, if we are concerned that violation of the normality assumptions will affect the conclusions of the test, we can use a Wilcoxon signed-rank test, which relaxes this assumption at the expense of test power:

```
>>> sp.stats.wilcoxon(data['VIQ'])
WilcoxonResult(statistic=np.float64(0.0), pvalue=np.float64(3.4881726...e-08))
```

Two-sample t-test: testing for difference across populations

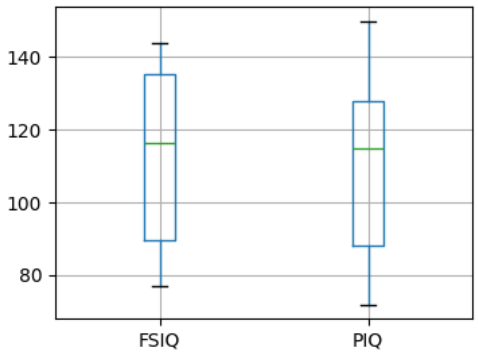
We have seen above that the mean VIQ in the male and female samples were different. To test whether this difference is significant (and suggests that there is a difference in population means), we perform a two-sample t-test using `scipy.stats.ttest_ind()`:

```
>>> female_viq = data[data['Gender'] == 'Female']['VIQ']
>>> male_viq = data[data['Gender'] == 'Male']['VIQ']
>>> sp.stats.ttest_ind(female_viq, male_viq)
TtestResult(statistic=np.float64(-0.77261617232...), pvalue=np.float64(0.
↪4445287677858...), df=np.float64(38.0))
```

The corresponding non-parametric test is the Mann–Whitney U test, `scipy.stats.mannwhitneyu()`.

```
>>> sp.stats.mannwhitneyu(female_viq, male_viq)
MannwhitneyuResult(statistic=np.float64(164.5), pvalue=np.float64(0.34228868687...))
```

15.2.2 Paired tests: repeated measurements on the same individuals

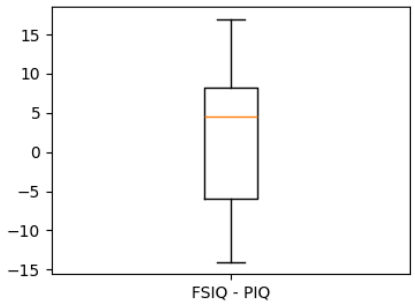


PIQ, VIQ, and FSIQ give three measures of IQ. Let us test whether FISQ and PIQ are significantly different. We can use an "independent sample" test:

```
>>> sp.stats.ttest_ind(data['FSIQ'], data['PIQ'])
TtestResult(statistic=np.float64(0.46563759638...), pvalue=np.float64(0.64277250...),␣
↪df=np.float64(78.0))
```

The problem with this approach is that it ignores an important relationship between observations: FSIQ and PIQ are measured on the same individuals. Thus, the variance due to inter-subject variability is confounding, reducing the power of the test. This variability can be removed using a "paired test" or "repeated measures test":

```
>>> sp.stats.ttest_rel(data['FSIQ'], data['PIQ'])
TtestResult(statistic=np.float64(1.784201940...), pvalue=np.float64(0.082172638183...
↪), df=np.int64(39))
```



This is equivalent to a one-sample test on the differences between paired observations:

```
>>> sp.stats.ttest_1samp(data['FSIQ'] - data['PIQ'], 0)
TtestResult(statistic=np.float64(1.784201940...), pvalue=np.float64(0.082172638...),␣
↪df=np.int64(39))
```

Accordingly, we can perform a nonparametric version of the test with `wilcoxon`.

```
>>> sp.stats.wilcoxon(data['FSIQ'], data['PIQ'], method="approx")
WilcoxonResult(statistic=np.float64(274.5), pvalue=np.float64(0.106594927135...))
```

Exercise

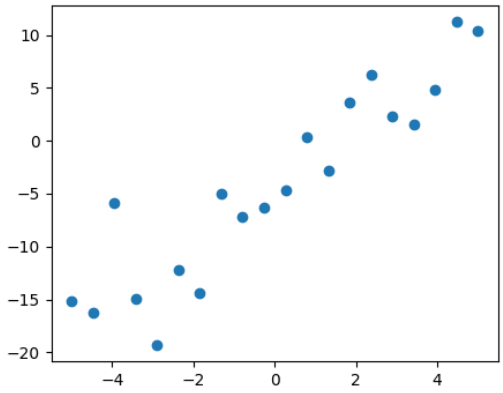
- Test the difference between weights in males and females.
- Use non parametric statistics to test the difference between VIQ in males and females.

Conclusion: we find that the data does not support the hypothesis that males and females have different VIQ.

15.3 Linear models, multiple factors, and analysis of variance

15.3.1 "formulas" to specify statistical models in Python

A simple linear regression



Given two set of observations, x and y, we want to test the hypothesis that y is a linear function of x. In other terms:

$$y = x * coef + intercept + e$$

where e is observation noise. We will use the statsmodels module to:

1. Fit a linear model. We will use the simplest strategy, ordinary least squares (OLS).
2. Test that *coef* is non zero.

First, we generate simulated data according to the model:

```
>>> import numpy as np
>>> x = np.linspace(-5, 5, 20)
>>> rng = np.random.default_rng(27446968)
>>> # normal distributed noise
>>> y = -5 + 3*x + 4 * rng.normal(size=x.shape)
>>> # Create a data frame containing all the relevant variables
>>> data = pandas.DataFrame({'x': x, 'y': y})
```

"formulas" for statistics in Python

See the statsmodels documentation

Then we specify an OLS model and fit it:

```
>>> from statsmodels.formula.api import ols
>>> model = ols("y ~ x", data).fit()
```

We can inspect the various statistics derived from the fit:

```
>>> print(model.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.901
Model:                            OLS   Adj. R-squared:                  0.896
Method:                 Least Squares   F-statistic:                     164.5
Date:                              ...   Prob (F-statistic):           1.72e-10
Time:                              ...   Log-Likelihood:                -51.758
No. Observations:                  20   AIC:                             107.5
Df Residuals:                      18   BIC:                             109.5
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -4.2948      0.759     -5.661      0.000      -5.889      -2.701
x              3.2060      0.250     12.825      0.000       2.681       3.731
==============================================================================
Omnibus:                        1.218   Durbin-Watson:                   1.796
Prob(Omnibus):                  0.544   Jarque-Bera (JB):                0.999
Skew:                           0.503   Prob(JB):                        0.607
Kurtosis:                       2.568   Cond. No.                         3.03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Terminology:

Statsmodels uses a statistical terminology: the y variable in statsmodels is called 'endogenous' while the x variable is called exogenous. This is discussed in more detail here.

To simplify, y (endogenous) is the value you are trying to predict, while x (exogenous) represents the features you are using to make the prediction.

Exercise

Retrieve the estimated parameters from the model above. **Hint**: use tab-completion to find the relevant attribute.

Categorical variables: comparing groups or multiple categories

Let us go back the data on brain size:

```
>>> data = pandas.read_csv('examples/brain_size.csv', sep=';', na_values=".")
```

We can write a comparison between IQ of male and female using a linear model:

```
>>> model = ols("VIQ ~ Gender + 1", data).fit()
>>> print(model.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                    VIQ   R-squared:                       0.015
Model:                            OLS   Adj. R-squared:                 -0.010
Method:                 Least Squares   F-statistic:                    0.5969
Date:                               ...   Prob (F-statistic):              0.445
Time:                               ...   Log-Likelihood:                -182.42
No. Observations:                  40   AIC:                             368.8
Df Residuals:                      38   BIC:                             372.2
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      109.4500      5.308     20.619      0.000      98.704     120.196
Gender[T.Male]   5.8000      7.507      0.773      0.445      -9.397      20.997
==============================================================================
Omnibus:                       26.188   Durbin-Watson:                   1.709
Prob(Omnibus):                  0.000   Jarque-Bera (JB):                3.703
Skew:                           0.010   Prob(JB):                        0.157
Kurtosis:                       1.510   Cond. No.                         2.62
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Tips on specifying model

Forcing categorical: the 'Gender' is automatically detected as a categorical variable, and thus each of its different values are treated as different entities.

An integer column can be forced to be treated as categorical using:

```
>>> model = ols('VIQ ~ C(Gender)', data).fit()
```

Intercept: We can remove the intercept using *- 1* in the formula, or force the use of an intercept using *+ 1*.

Tip

By default, statsmodels treats a categorical variable with K possible values as K-1 'dummy' boolean variables (the last level being absorbed into the intercept term). This is almost always a good default choice - however, it is possible to specify different encodings for categorical variables (https://www.statsmodels.org/devel/contrasts.html).

Link to t-tests between different FSIQ and PIQ

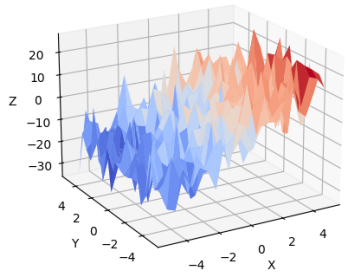
To compare different types of IQ, we need to create a "long-form" table, listing IQs, where the type of IQ is indicated by a categorical variable:

```
>>> data_fisq = pandas.DataFrame({'iq': data['FSIQ'], 'type': 'fsiq'})
>>> data_piq = pandas.DataFrame({'iq': data['PIQ'], 'type': 'piq'})
>>> data_long = pandas.concat((data_fisq, data_piq))
>>> print(data_long)
     iq  type
0   133  fsiq
1   140  fsiq
2   139  fsiq
3   133  fsiq
4   137  fsiq
...  ...   ...
35  128   piq
36  124   piq
37   94   piq
38   74   piq
39   89   piq

[80 rows x 2 columns]

>>> model = ols("iq ~ type", data_long).fit()
>>> print(model.summary())
                            OLS Regression Results
...
==========================...
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------...
Intercept     113.4500      3.683     30.807      0.000       106.119   120.781
type[T.piq]    -2.4250      5.208     -0.466      0.643       -12.793     7.943
...
```

We can see that we retrieve the same values for t-test and corresponding p-values for the effect of the type of iq than the previous t-test:

```
>>> sp.stats.ttest_ind(data['FSIQ'], data['PIQ'])
TtestResult(statistic=np.float64(0.46563759638...), pvalue=np.float64(0.64277250...
↪), df=np.float64(78.0))
```

15.3.2 Multiple Regression: including multiple factors



Consider a linear model explaining a variable z (the dependent variable) with 2 variables x and y:

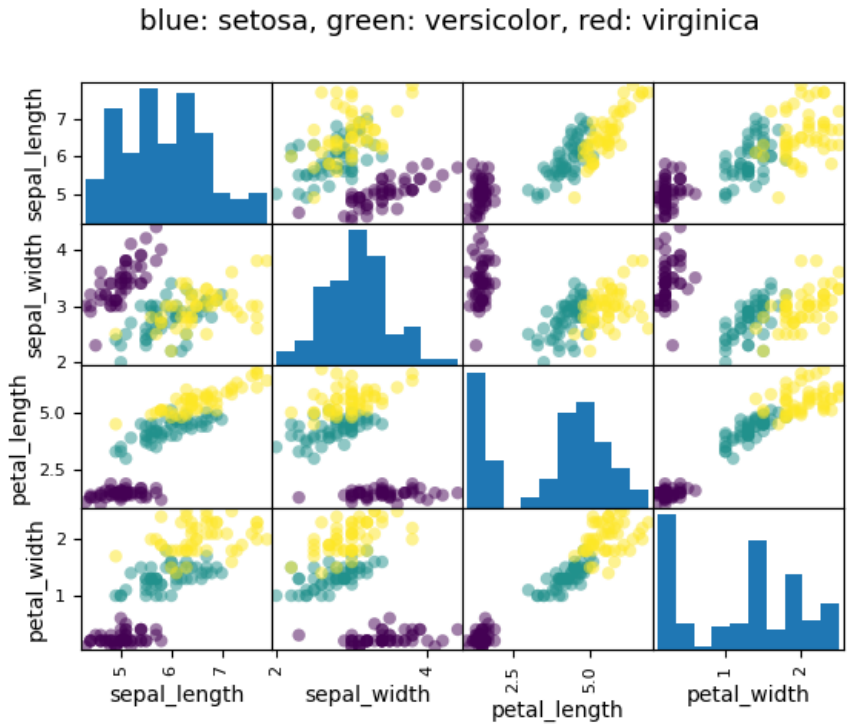
$$z = x\,c_1 + y\,c_2 + i + e$$

Such a model can be seen in 3D as fitting a plane to a cloud of (x, y, z) points.

Example: the iris data (`examples/iris.csv`)

Tip

Sepal and petal size tend to be related: bigger flowers are bigger! But is there in addition a systematic effect of species?




```
>>> data = pandas.read_csv('examples/iris.csv')
>>> model = ols('sepal_width ~ name + petal_length', data).fit()
>>> print(model.summary())
                            OLS Regression Results
==============================...
Dep. Variable:            sepal_width   R-squared:                       0.478
Model:                            OLS   Adj. R-squared:                  0.468
Method:                 Least Squares   F-statistic:                     44.63
Date:                               ...   Prob (F-statistic):           1.58e-20
Time:                               ...   Log-Likelihood:                -38.185
No. Observations:                 150   AIC:                             84.37
Df Residuals:                     146   BIC:                             96.41
Df Model:                           3
Covariance Type:            nonrobust
==============================...
                          coef    std err          t      P>|t|  [0.025      0.975]
------------------------------------------...
Intercept               2.9813      0.099     29.989      0.000      2.785      3.178
name[T.versicolor]     -1.4821      0.181     -8.190      0.000     -1.840     -1.124
name[T.virginica]      -1.6635      0.256     -6.502      0.000     -2.169     -1.158
petal_length            0.2983      0.061      4.920      0.000      0.178      0.418
==============================...
Omnibus:                        2.868   Durbin-Watson:                   1.753
Prob(Omnibus):                  0.238   Jarque-Bera (JB):                2.885
Skew:                          -0.082   Prob(JB):                        0.236
Kurtosis:                       3.659   Cond. No.                         54.0
==============================...

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

15.3.3 Post-hoc hypothesis testing: analysis of variance (ANOVA)

In the above iris example, we wish to test if the petal length is different between versicolor and virginica, after removing the effect of sepal width. This can be formulated as testing the difference between the coefficient associated to versicolor and virginica in the linear model estimated above (it is an Analysis of Variance, ANOVA). For this, we write a **vector of 'contrast'** on the parameters estimated: we want to test `"name[T.versicolor] - name[T.virginica]"`, with an F-test:

```
>>> print(model.f_test([0, 1, -1, 0]))
<F test: F=3.24533535..., p=0.07369..., df_denom=146, df_num=1>
```

Is this difference significant?

Exercise

Going back to the brain size + IQ data, test if the VIQ of male and female are different after removing the effect of brain size, height and weight.

15.4 More visualization: seaborn for statistical exploration

Seaborn combines simple statistical fits with plotting on pandas dataframes.

Let us consider a data giving wages and many other personal information on 500 individuals (Berndt, ER. The Practice of Econometrics. 1991. NY: Addison-Wesley).

Tip

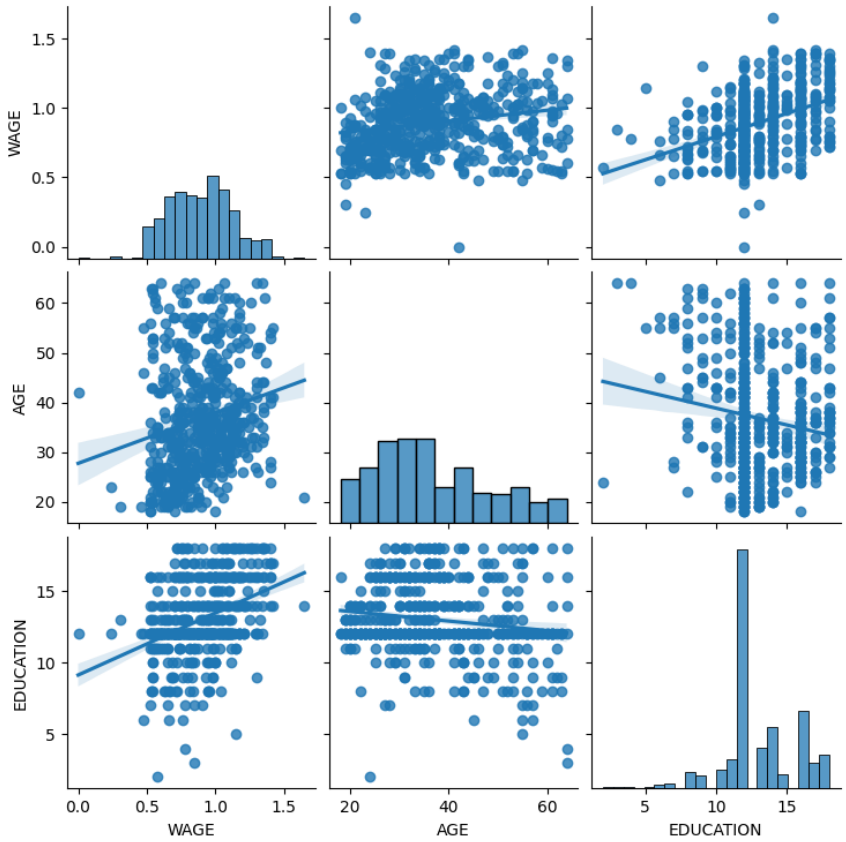
The full code loading and plotting of the wages data is found in corresponding example.

```
>>> print(data)
     EDUCATION  SOUTH  SEX  EXPERIENCE  UNION      WAGE  AGE  RACE  \
0            8      0    1          21      0  0.707570   35     2
1            9      0    1          42      0  0.694605   57     3
2           12      0    0           1      0  0.824126   19     3
3           12      0    0           4      0  0.602060   22     3
...
```

15.4.1 Pairplot: scatter matrices

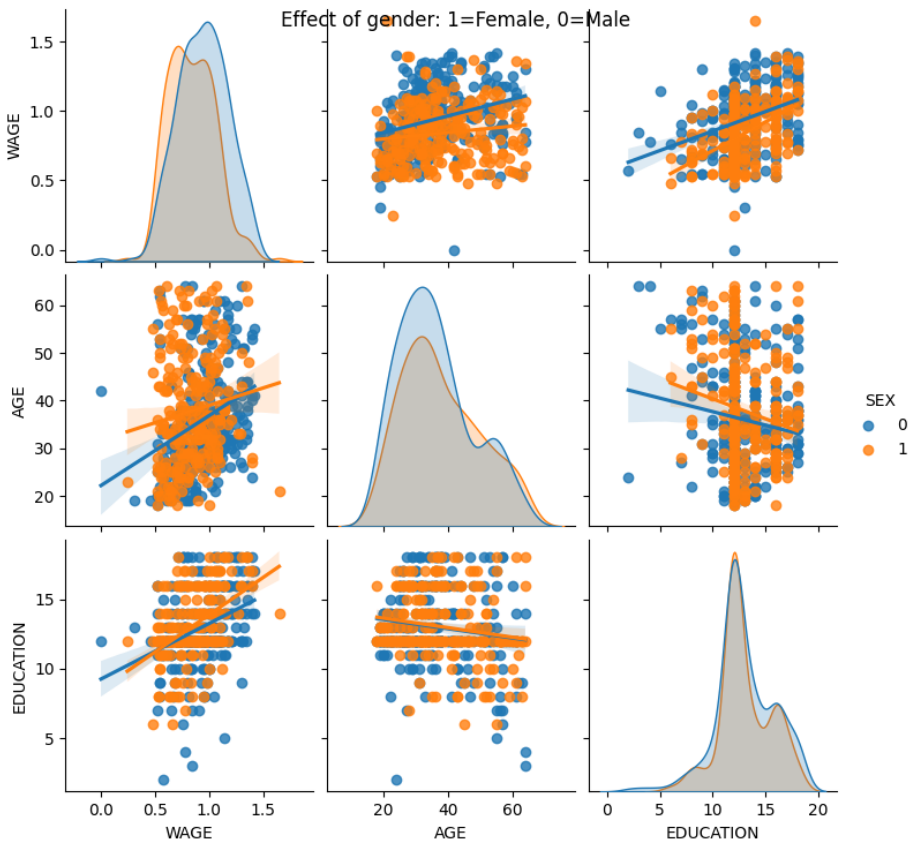
We can easily have an intuition on the interactions between continuous variables using `seaborn.pairplot()` to display a scatter matrix:

```
>>> import seaborn
>>> seaborn.pairplot(data, vars=['WAGE', 'AGE', 'EDUCATION'],
...                  kind='reg')
```



Categorical variables can be plotted as the hue:

```
>>> seaborn.pairplot(data, vars=['WAGE', 'AGE', 'EDUCATION'],
...                  kind='reg', hue='SEX')
```



Look and feel and matplotlib settings

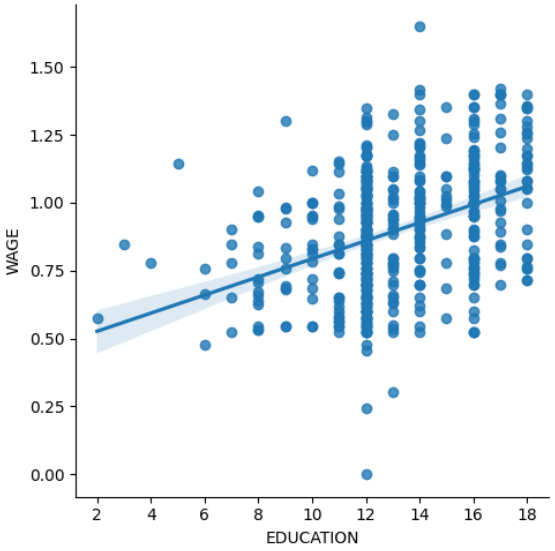
Seaborn changes the default of matplotlib figures to achieve a more "modern", "excel-like" look. It does that upon import. You can reset the default using:

```
>>> import matplotlib.pyplot as plt
>>> plt.rcdefaults()
```

Tip

To switch back to seaborn settings, or understand better styling in seaborn, see the relevant section of the seaborn documentation.

15.4.2 lmplot: plotting a univariate regression



A regression capturing the relation between one variable and another, eg wage, and education, can be plotted using `seaborn.lmplot()`:

```
>>> seaborn.lmplot(y='WAGE', x='EDUCATION', data=data)
```

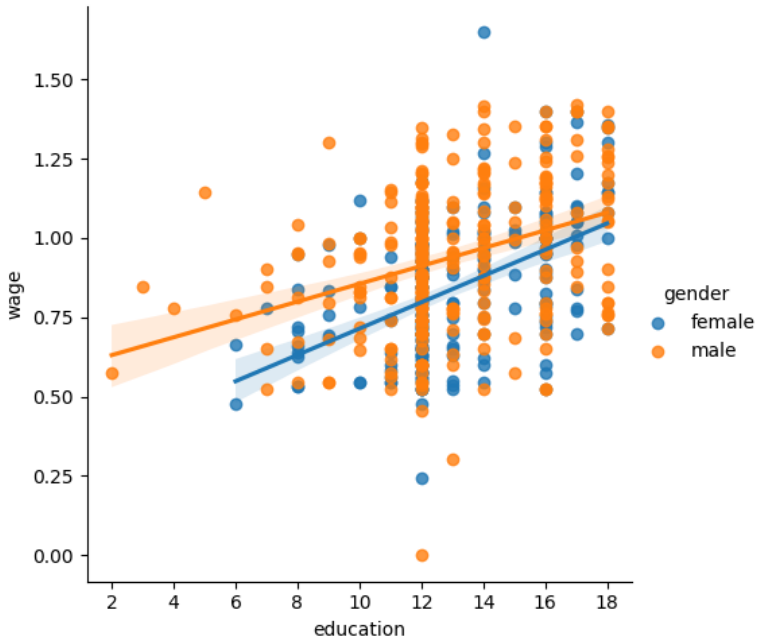
Robust regression

Tip

Given that, in the above plot, there seems to be a couple of data points that are outside of the main cloud to the right, they might be outliers, not representative of the population, but driving the regression.

To compute a regression that is less sensitive to outliers, one must use a robust model. This is done in seaborn using `robust=True` in the plotting functions, or in statsmodels by replacing the use of the OLS by a "Robust Linear Model", `statsmodels.formula.api.rlm()`.

15.5 Testing for interactions



Do wages increase more with education for males than females?

Tip

The plot above is made of two different fits. We need to formulate a single model that tests for a variance of slope across the two populations. This is done via an "interaction".

```
>>> result = sm.ols(formula='wage ~ education + gender + education * gender',
...                 data=data).fit()
>>> print(result.summary())
...
                            coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept                   0.2998      0.072      4.173      0.000       0.159       0.441
gender[T.male]              0.2750      0.093      2.972      0.003       0.093       0.457
education                   0.0415      0.005      7.647      0.000       0.031       0.052
education:gender[T.male]   -0.0134      0.007     -1.919      0.056      -0.027       0.000
============================...
...
```

Can we conclude that education benefits males more than females?

Take home messages

- Hypothesis testing and p-values give you the **significance** of an effect / difference.
- **Formulas** (with categorical variables) enable you to express rich links in your data.
- **Visualizing** your data and fitting simple models give insight into the data.

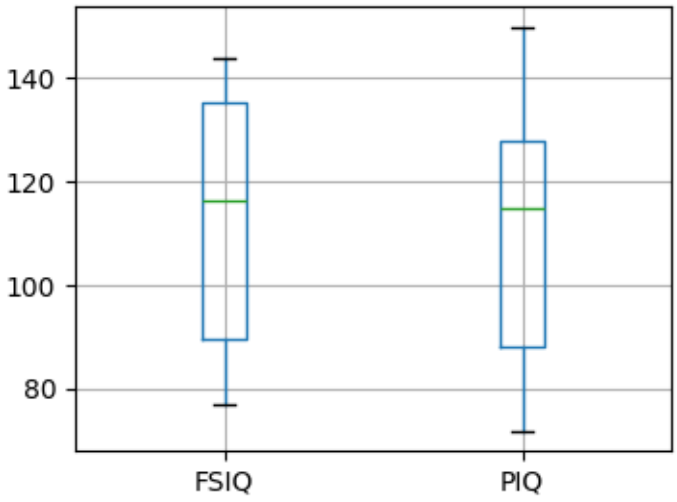
- **Conditionning** (adding factors that can explain all or part of the variation) is an important modeling aspect that changes the interpretation.

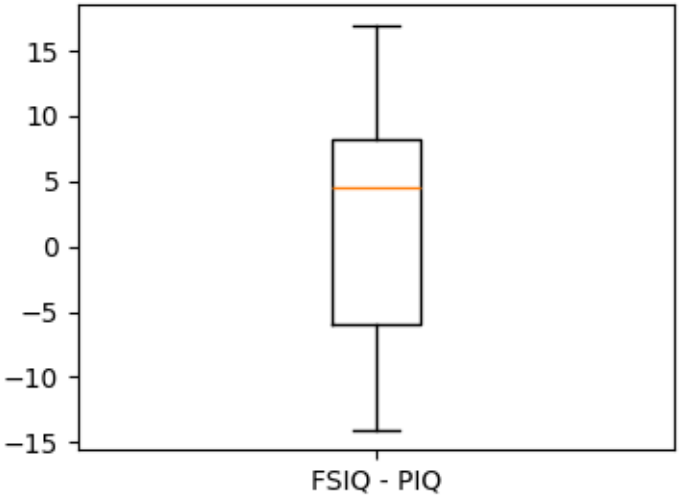
15.6 Full code for the figures

Code examples for the statistics chapter.

15.6.1 Boxplots and paired differences

Plot boxplots for FSIQ, PIQ, and the paired difference between the two: while the spread (error bars) for FSIQ and PIQ are very large, there is a systematic (common) effect due to the subjects. This effect is cancelled out in the difference and the spread of the difference ("paired" by subject) is much smaller than the spread of the individual measures.

- 

- 

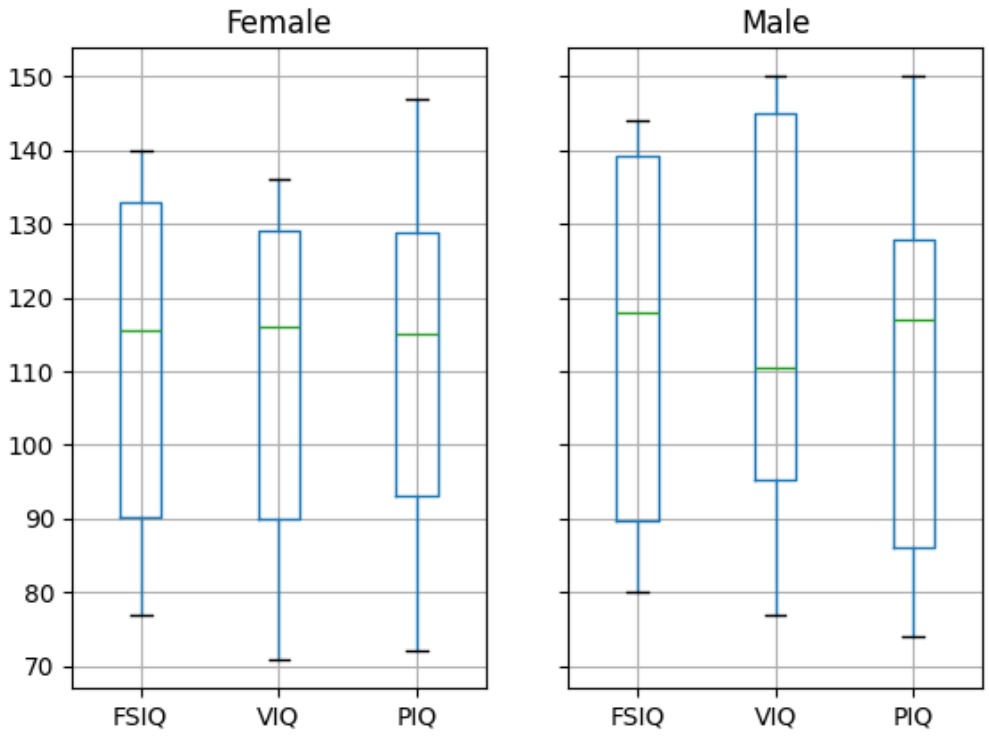
```
import pandas

import matplotlib.pyplot as plt

data = pandas.read_csv("brain_size.csv", sep=";", na_values=".")

# Box plot of FSIQ and PIQ (different measures od IQ)
plt.figure(figsize=(4, 3))
data.boxplot(column=["FSIQ", "PIQ"])

# Boxplot of the difference
plt.figure(figsize=(4, 3))
plt.boxplot(data["FSIQ"] - data["PIQ"])
plt.xticks((1,), ("FSIQ - PIQ",))

plt.show()
```

Total running time of the script: (0 minutes 0.072 seconds)

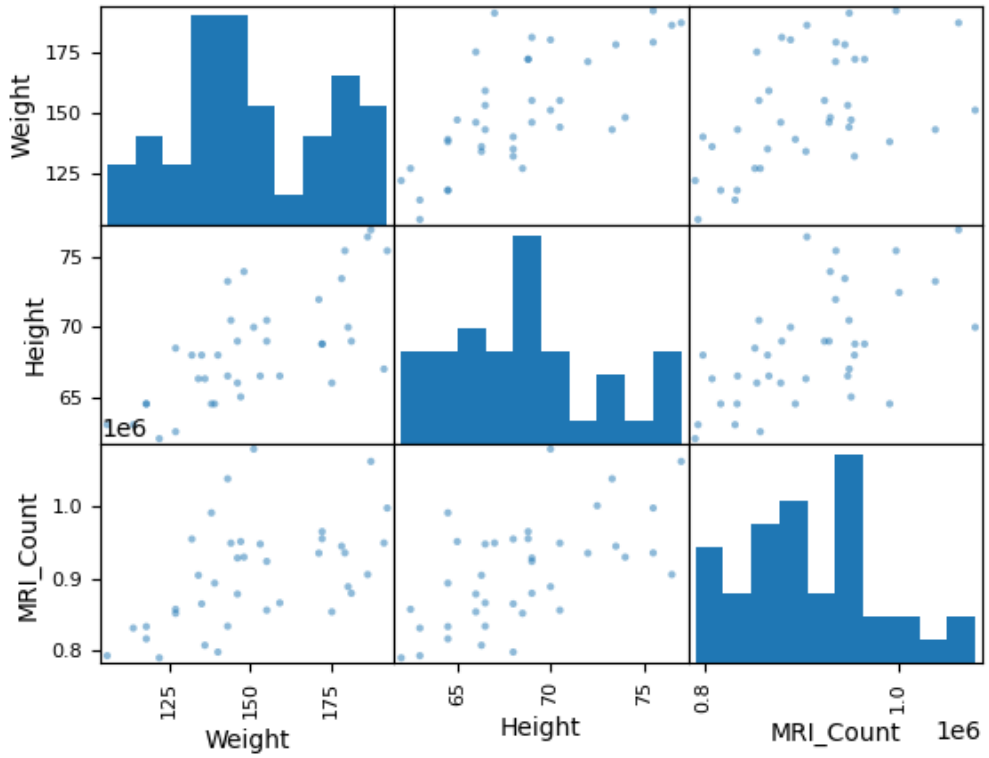
15.6.2 Plotting simple quantities of a pandas dataframe

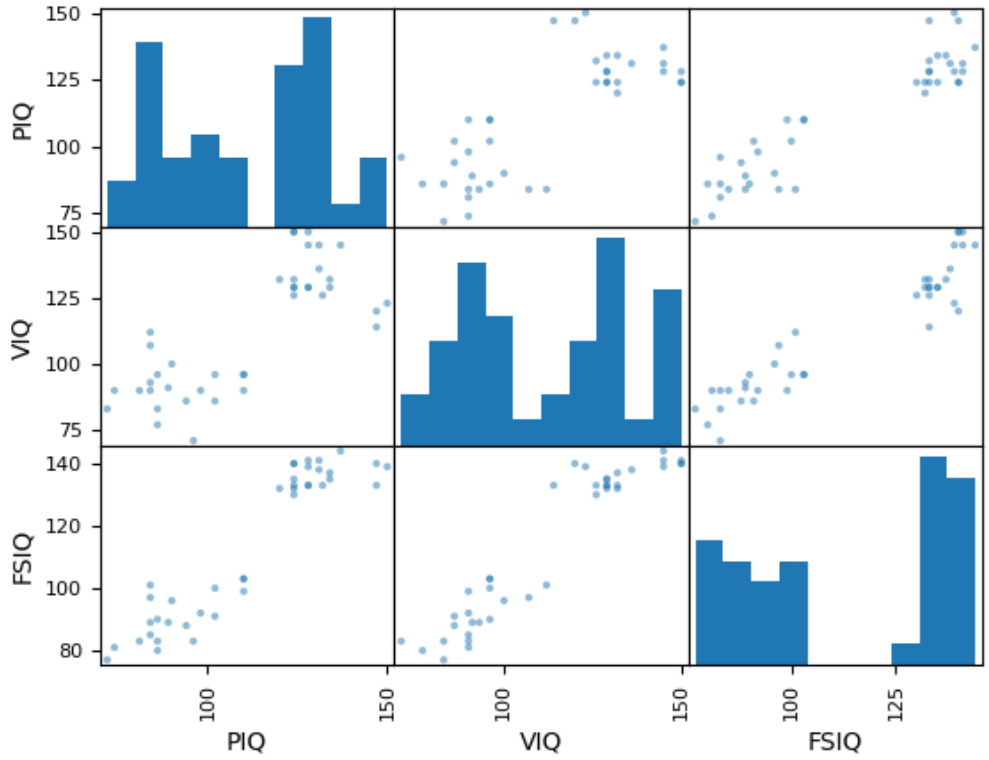
This example loads from a CSV file data with mixed numerical and categorical entries, and plots a few quantities, separately for females and males, thanks to the pandas integrated plotting tool (that uses matplotlib behind the scene).

See http://pandas.pydata.org/pandas-docs/stable/visualization.html



-

- 

- 


```
import pandas
```

(continues on next page)

(continued from previous page)

```
data = pandas.read_csv("brain_size.csv", sep=";", na_values=".")

# Box plots of different columns for each gender
groupby_gender = data.groupby("Gender")
groupby_gender.boxplot(column=["FSIQ", "VIQ", "PIQ"])

from pandas import plotting

# Scatter matrices for different columns
plotting.scatter_matrix(data[["Weight", "Height", "MRI_Count"]])
plotting.scatter_matrix(data[["PIQ", "VIQ", "FSIQ"]])

import matplotlib.pyplot as plt

plt.show()
```

Total running time of the script: (0 minutes 0.539 seconds)

15.6.3 Analysis of Iris petal and sepal sizes

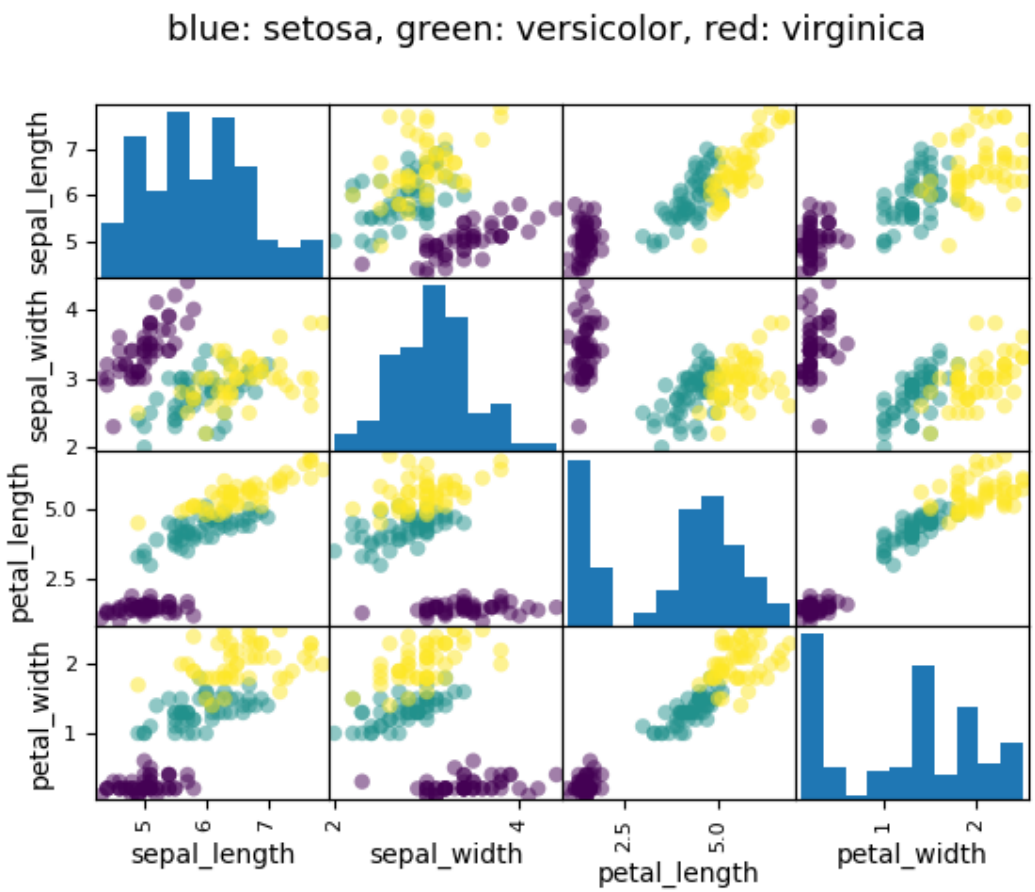
Illustrate an analysis on a real dataset:

- Visualizing the data to formulate intuitions
- Fitting of a linear model
- Hypothesis test of the effect of a categorical variable in the presence of a continuous confound

```
import matplotlib.pyplot as plt

import pandas
from pandas import plotting

from statsmodels.formula.api import ols

# Load the data
data = pandas.read_csv("iris.csv")
```

Plot a scatter matrix

```
# Express the names as categories
categories = pandas.Categorical(data["name"])

# The parameter 'c' is passed to plt.scatter and will control the color
plotting.scatter_matrix(data, c=categories.codes, marker="o")

fig = plt.gcf()
fig.suptitle("blue: setosa, green: versicolor, red: virginica", size=13)
```




```
Text(0.5, 0.98, 'blue: setosa, green: versicolor, red: virginica')
```

Statistical analysis

```
# Let us try to explain the sepal length as a function of the petal
# width and the category of iris

model = ols("sepal_width ~ name + petal_length", data).fit()
print(model.summary())

# Now formulate a "contrast", to test if the offset for versicolor and
# virginica are identical

print("Testing the difference between effect of versicolor and virginica")
print(model.f_test([0, 1, -1, 0]))
plt.show()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:            sepal_width   R-squared:                       0.478
Model:                            OLS   Adj. R-squared:                  0.468
Method:                 Least Squares   F-statistic:                     44.63
Date:                Fri, 13 Jun 2025   Prob (F-statistic):           1.58e-20
Time:                        14:07:02   Log-Likelihood:                -38.185
No. Observations:                 150   AIC:                             84.37
Df Residuals:                     146   BIC:                             96.41
Df Model:                           3
Covariance Type:            nonrobust
```

(continues on next page)

(continued from previous page)

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      2.9813      0.099     29.989      0.000       2.785       3.178
name[T.versicolor]    -1.4821      0.181     -8.190      0.000      -1.840      -1.124
name[T.virginica]     -1.6635      0.256     -6.502      0.000      -2.169      -1.158
petal_length      0.2983      0.061      4.920      0.000       0.178       0.418
==============================================================================
Omnibus:                        2.868   Durbin-Watson:                   1.753
Prob(Omnibus):                  0.238   Jarque-Bera (JB):                2.885
Skew:                          -0.082   Prob(JB):                        0.236
Kurtosis:                       3.659   Cond. No.                         54.0
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
Testing the difference between effect of versicolor and virginica
<F test: F=3.245335346574177, p=0.0736905878170114, df_denom=146, df_num=1>
```

Total running time of the script: (0 minutes 0.429 seconds)

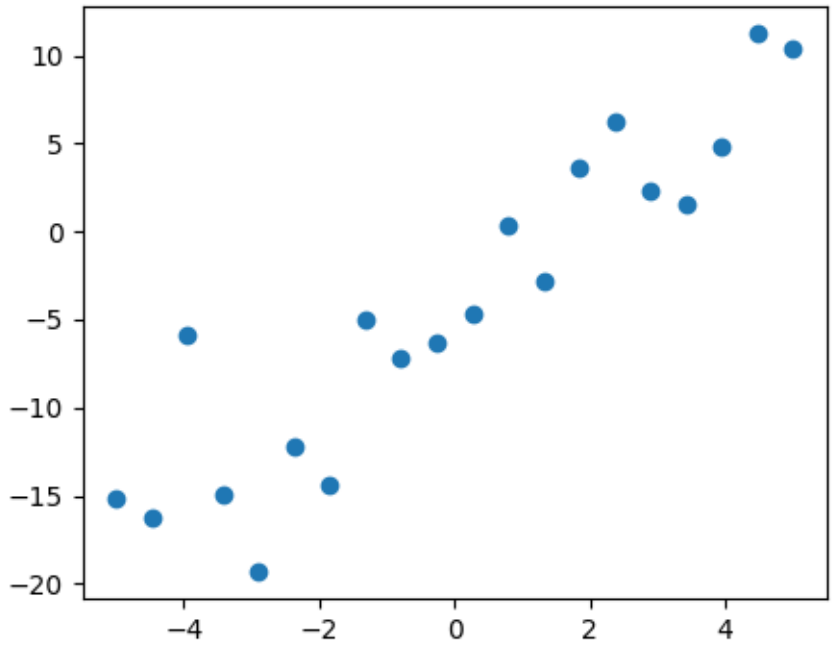
15.6.4 Simple Regression

Fit a simple linear regression using 'statsmodels', compute corresponding p-values.

```
# Original author: Thomas Haslwanter

import numpy as np
import matplotlib.pyplot as plt
import pandas

# For statistics. Requires statsmodels 5.0 or more
from statsmodels.formula.api import ols

# Analysis of Variance (ANOVA) on linear models
from statsmodels.stats.anova import anova_lm
```

Generate and show the data

```
x = np.linspace(-5, 5, 20)

# To get reproducible values, provide a seed value
rng = np.random.default_rng(27446968)

y = -5 + 3 * x + 4 * np.random.normal(size=x.shape)

# Plot the data
plt.figure(figsize=(5, 4))
plt.plot(x, y, "o")
```




```
[<matplotlib.lines.Line2D object at 0x7f3b1be9a240>]
```

Multilinear regression model, calculating fit, P-values, confidence intervals etc.

```
# Convert the data into a Pandas DataFrame to use the formulas framework
# in statsmodels
data = pandas.DataFrame({"x": x, "y": y})

# Fit the model
model = ols("y ~ x", data).fit()

# Print the summary
print(model.summary())

# Perform analysis of variance on fitted linear model
anova_results = anova_lm(model)

print("\nANOVA results")
print(anova_results)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.845
Model:                            OLS   Adj. R-squared:                  0.836
Method:                 Least Squares   F-statistic:                     97.76
Date:                Fri, 13 Jun 2025   Prob (F-statistic):           1.06e-08
Time:                        14:07:02   Log-Likelihood:                -53.560
No. Observations:                  20   AIC:                             111.1
Df Residuals:                      18   BIC:                             113.1
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
```

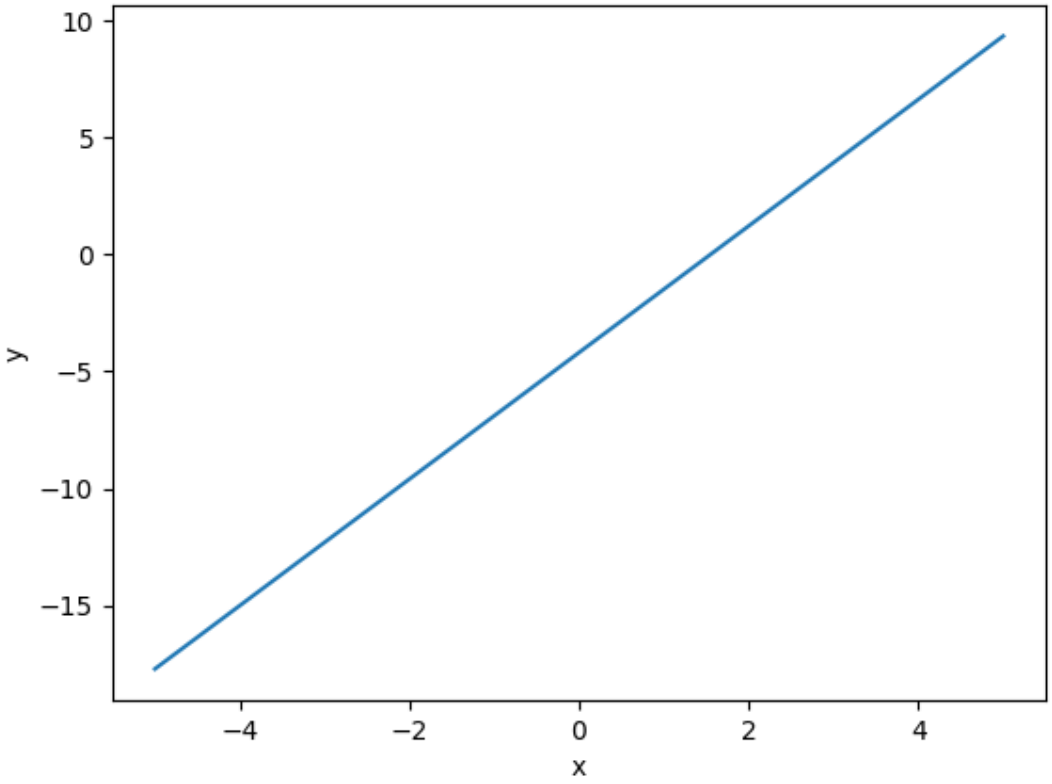
(continues on next page)

(continued from previous page)

```
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -4.1877      0.830     -5.044      0.000      -5.932      -2.444
x              2.7046      0.274      9.887      0.000       2.130       3.279
==============================================================================
Omnibus:                        1.871   Durbin-Watson:                   1.930
Prob(Omnibus):                  0.392   Jarque-Bera (JB):                0.597
Skew:                           0.337   Prob(JB):                        0.742
Kurtosis:                       3.512   Cond. No.                         3.03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.

ANOVA results
            df       sum_sq      mean_sq          F        PR(>F)
x          1.0  1347.476043  1347.476043  97.760281  1.062847e-08
Residual  18.0   248.102486    13.783471        NaN           NaN
```

Plot the fitted model

```
# Retrieve the parameter estimates
offset, coef = model._results.params
plt.plot(x, x * coef + offset)
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



Total running time of the script: (0 minutes 0.105 seconds)

15.6.5 Multiple Regression

Calculate using 'statsmodels' just the best fit, or all the corresponding statistical parameters.

Also shows how to make 3d plots.

```
# Original author: Thomas Haslwanter

import numpy as np
import matplotlib.pyplot as plt
import pandas

# For 3d plots. This import is necessary to have 3D plotting below
from mpl_toolkits.mplot3d import Axes3D

# For statistics. Requires statsmodels 5.0 or more
from statsmodels.formula.api import ols

# Analysis of Variance (ANOVA) on linear models
from statsmodels.stats.anova import anova_lm
```

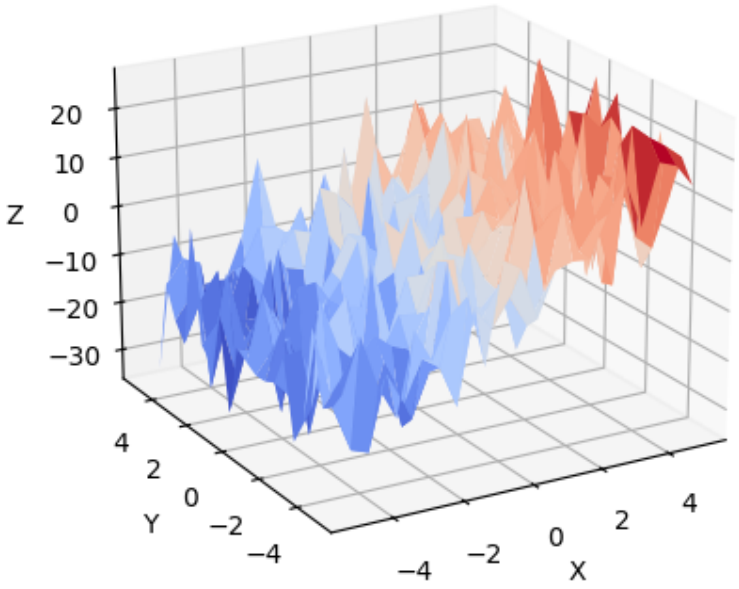
Generate and show the data

```
x = np.linspace(-5, 5, 21)
# We generate a 2D grid
X, Y = np.meshgrid(x, x)
```

(continues on next page)

(continued from previous page)

```
# To get reproducible values, provide a seed value
rng = np.random.default_rng(27446968)

# Z is the elevation of this 2D grid
Z = -5 + 3 * X - 0.5 * Y + 8 * np.random.normal(size=X.shape)

# Plot the data
ax: Axes3D = plt.figure().add_subplot(projection="3d")
surf = ax.plot_surface(X, Y, Z, cmap="coolwarm", rstride=1, cstride=1)
ax.view_init(20, -120)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
```




```
Text(-0.10764513121260137, 0.009865032686848034, 'Z')
```

Multilinear regression model, calculating fit, P-values, confidence intervals etc.

```
# Convert the data into a Pandas DataFrame to use the formulas framework
# in statsmodels

# First we need to flatten the data: it's 2D layout is not relevant.
X = X.flatten()
Y = Y.flatten()
Z = Z.flatten()
```

(continues on next page)

(continued from previous page)

```
data = pandas.DataFrame({"x": X, "y": Y, "z": Z})

# Fit the model
model = ols("z ~ x + y", data).fit()

# Print the summary
print(model.summary())

print("\nRetrieving manually the parameter estimates:")
print(model._results.params)
# should be array([-4.99754526,  3.00250049, -0.50514907])

# Perform analysis of variance on fitted linear model
anova_results = anova_lm(model)

print("\nANOVA results")
print(anova_results)

plt.show()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      z   R-squared:                       0.579
Model:                            OLS   Adj. R-squared:                  0.577
Method:                 Least Squares   F-statistic:                     300.7
Date:                Fri, 13 Jun 2025   Prob (F-statistic):           6.43e-83
Time:                        14:07:02   Log-Likelihood:                -1552.0
No. Observations:                 441   AIC:                             3110.
Df Residuals:                     438   BIC:                             3122.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -4.4332      0.390    -11.358      0.000      -5.200      -3.666
x              3.0861      0.129     23.940      0.000       2.833       3.340
y             -0.6856      0.129     -5.318      0.000      -0.939      -0.432
==============================================================================
Omnibus:                        0.560   Durbin-Watson:                   1.967
Prob(Omnibus):                  0.756   Jarque-Bera (JB):                0.651
Skew:                          -0.077   Prob(JB):                        0.722
Kurtosis:                       2.893   Cond. No.                         3.03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.

Retrieving manually the parameter estimates:
[-4.43322435  3.08614608 -0.68556194]

ANOVA results
             df        sum_sq       mean_sq           F        PR(>F)
x           1.0  38501.973182  38501.973182  573.111646  1.365553e-81
y           1.0   1899.955512   1899.955512   28.281320  1.676135e-07
Residual  438.0  29425.094352     67.180581         NaN           NaN
```

Total running time of the script: (0 minutes 0.112 seconds)

15.6.6 Test for an education/gender interaction in wages

Wages depend mostly on education. Here we investigate how this dependence is related to gender: not only does gender create an offset in wages, it also seems that wages increase more with education for males than females.

Does our data support this last hypothesis? We will test this using statsmodels' formulas (http://statsmodels.sourceforge.net/stable/example_formulas.html).

Load and massage the data

```
import pandas

import urllib.request
import os

if not os.path.exists("wages.txt"):
    # Download the file if it is not present
    url = "http://lib.stat.cmu.edu/datasets/CPS_85_Wages"
    with urllib.request.urlopen(url) as r, open("wages.txt", "wb") as f:
        f.write(r.read())

# EDUCATION: Number of years of education
# SEX: 1=Female, 0=Male
# WAGE: Wage (dollars per hour)
data = pandas.read_csv(
    "wages.txt",
    skiprows=27,
    skipfooter=6,
    sep=None,
    header=None,
    names=["education", "gender", "wage"],
    usecols=[0, 2, 5],
)

# Convert genders to strings (this is particularly useful so that the
# statsmodels formulas detects that gender is a categorical variable)
import numpy as np

data["gender"] = np.choose(data.gender, ["male", "female"])

# Log-transform the wages, because they typically are increased with
# multiplicative factors
data["wage"] = np.log10(data["wage"])
```

```
/home/runner/work/scientific-python-lectures/scientific-python-lectures/packages/
↪statistics/examples/plot_wage_education_gender.py:32: ParserWarning: Falling back␣
↪to the 'python' engine because the 'c' engine does not support skipfooter; you can␣
↪avoid this warning by specifying engine='python'.
  data = pandas.read_csv(
```

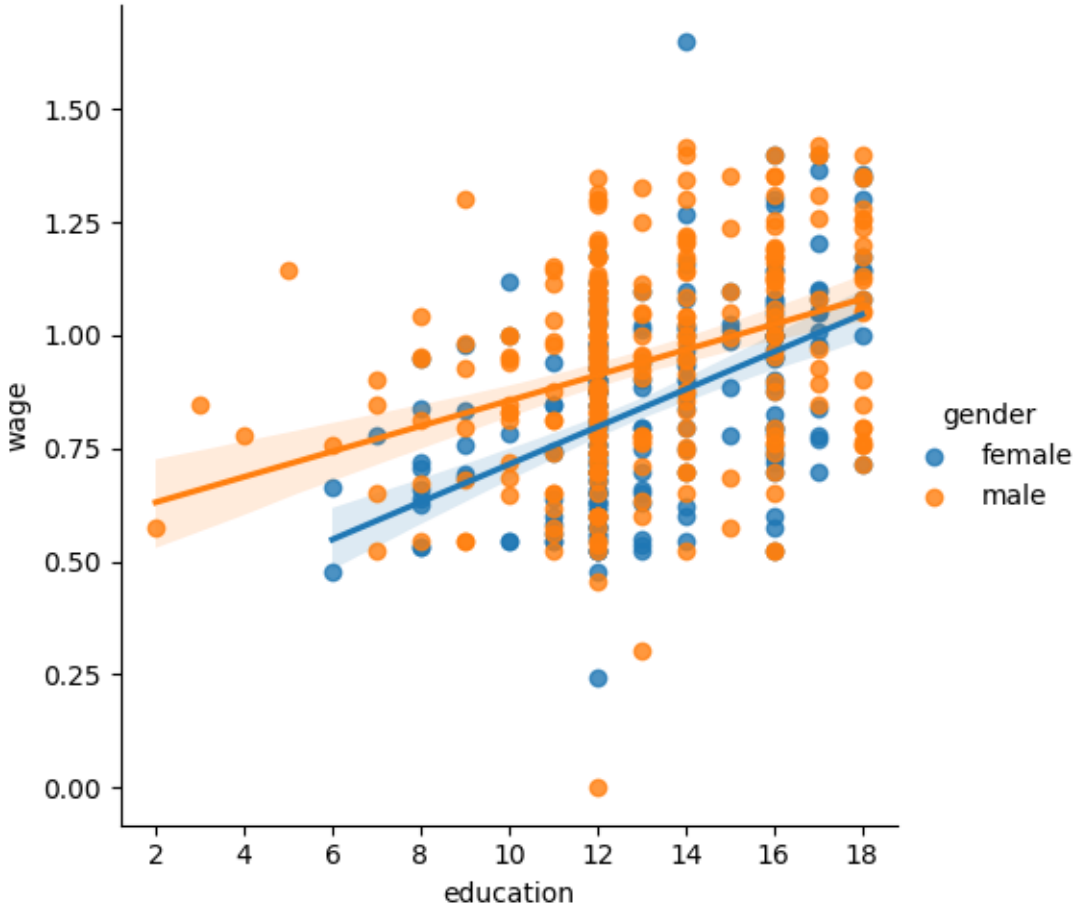
simple plotting

```
import seaborn
```

(continues on next page)

(continued from previous page)

```
# Plot 2 linear fits for male and female.
seaborn.lmplot(y="wage", x="education", hue="gender", data=data)
```




```
<seaborn.axisgrid.FacetGrid object at 0x7f3b1c2fa990>
```

statistical analysis

```
import statsmodels.formula.api as sm

# Note that this model is not the plot displayed above: it is one
# joined model for male and female, not separate models for male and
# female. The reason is that a single model enables statistical testing
result = sm.ols(formula="wage ~ education + gender", data=data).fit()
print(result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   wage   R-squared:                       0.193
Model:                            OLS   Adj. R-squared:                  0.190
Method:                 Least Squares   F-statistic:                     63.42
Date:                Fri, 13 Jun 2025   Prob (F-statistic):           2.01e-25
Time:                        14:07:03   Log-Likelihood:                 86.654
No. Observations:                 534   AIC:                            -167.3
Df Residuals:                     531   BIC:                            -154.5
Df Model:                           2
```

(continues on next page)

(continued from previous page)

```
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.4053      0.046      8.732      0.000       0.314       0.496
gender[T.male]     0.1008      0.018      5.625      0.000       0.066       0.136
education      0.0334      0.003      9.768      0.000       0.027       0.040
==============================================================================
Omnibus:                        4.675   Durbin-Watson:                   1.792
Prob(Omnibus):                  0.097   Jarque-Bera (JB):                4.876
Skew:                          -0.147   Prob(JB):                       0.0873
Kurtosis:                       3.365   Cond. No.                         69.7
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

The plots above highlight that there is not only a different offset in wage but also a different slope

We need to model this using an interaction

```
result = sm.ols(
    formula="wage ~ education + gender + education * gender", data=data
).fit()
print(result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   wage   R-squared:                       0.198
Model:                            OLS   Adj. R-squared:                  0.194
Method:                 Least Squares   F-statistic:                     43.72
Date:                Fri, 13 Jun 2025   Prob (F-statistic):           2.94e-25
Time:                        14:07:03   Log-Likelihood:                 88.503
No. Observations:                 534   AIC:                            -169.0
Df Residuals:                     530   BIC:                            -151.9
Df Model:                           3
Covariance Type:            nonrobust
================================================================================================
                                   coef    std err          t      P>|t|      [0.025      ␣
↪0.975]
------------------------------------------------------------------------------------------------
↪------
Intercept                      0.2998      0.072      4.173      0.000       0.159      ␣
↪ 0.441
gender[T.male]                 0.2750      0.093      2.972      0.003       0.093      ␣
↪ 0.457
education                      0.0415      0.005      7.647      0.000       0.031      ␣
↪ 0.052
education:gender[T.male]      -0.0134      0.007     -1.919      0.056      -0.027      ␣
↪ 0.000
==============================================================================
Omnibus:                        4.838   Durbin-Watson:                   1.825
Prob(Omnibus):                  0.089   Jarque-Bera (JB):                5.000
Skew:                          -0.156   Prob(JB):                       0.0821
Kurtosis:                       3.356   Cond. No.                         194.
```

(continues on next page)

(continued from previous page)

```
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Looking at the p-value of the interaction of gender and education, the data does not support the hypothesis that education benefits males more than female (p-value > 0.05).

```
import matplotlib.pyplot as plt

plt.show()
```

Total running time of the script: (0 minutes 0.523 seconds)

15.6.7 Visualizing factors influencing wages

This example uses seaborn to quickly plot various factors relating wages, experience, and education.

Seaborn (https://seaborn.pydata.org) is a library that combines visualization and statistical fits to show trends in data.

Note that importing seaborn changes the matplotlib style to have an "excel-like" feeling. This changes affect other matplotlib figures. To restore defaults once this example is run, we would need to call plt.rcdefaults().

```
# Standard library imports
import os

import matplotlib.pyplot as plt
```

Load the data

```
import pandas
import requests

if not os.path.exists("wages.txt"):
    # Download the file if it is not present
    r = requests.get("http://lib.stat.cmu.edu/datasets/CPS_85_Wages")
    with open("wages.txt", "wb") as f:
        f.write(r.content)

# Give names to the columns
names = [
    "EDUCATION: Number of years of education",
    "SOUTH: 1=Person lives in South, 0=Person lives elsewhere",
    "SEX: 1=Female, 0=Male",
    "EXPERIENCE: Number of years of work experience",
    "UNION: 1=Union member, 0=Not union member",
    "WAGE: Wage (dollars per hour)",
    "AGE: years",
    "RACE: 1=Other, 2=Hispanic, 3=White",
    "OCCUPATION: 1=Management, 2=Sales, 3=Clerical, 4=Service, 5=Professional, 6=Other
↪",
    "SECTOR: 0=Other, 1=Manufacturing, 2=Construction",
    "MARR: 0=Unmarried,  1=Married",
```

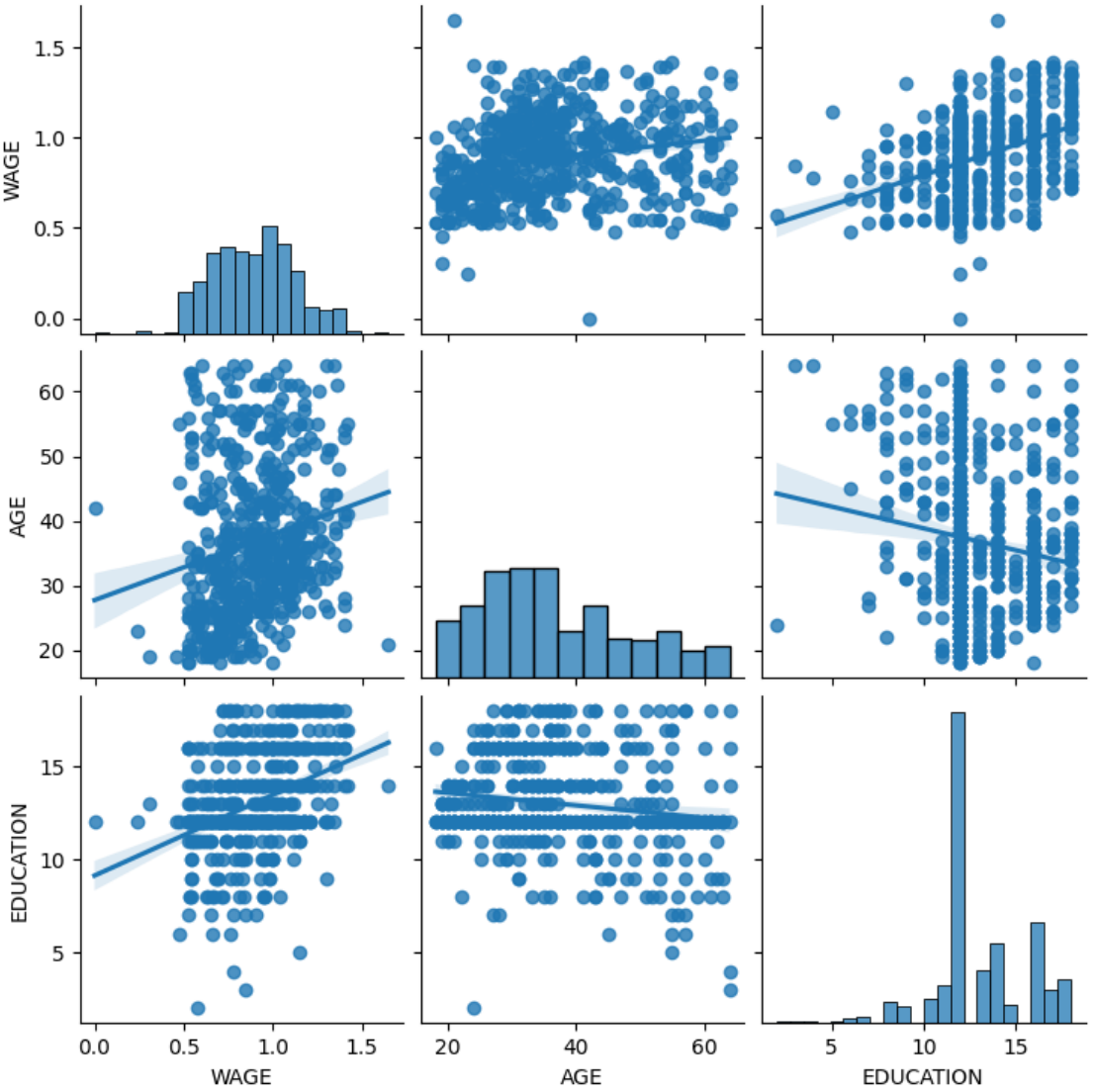
(continues on next page)

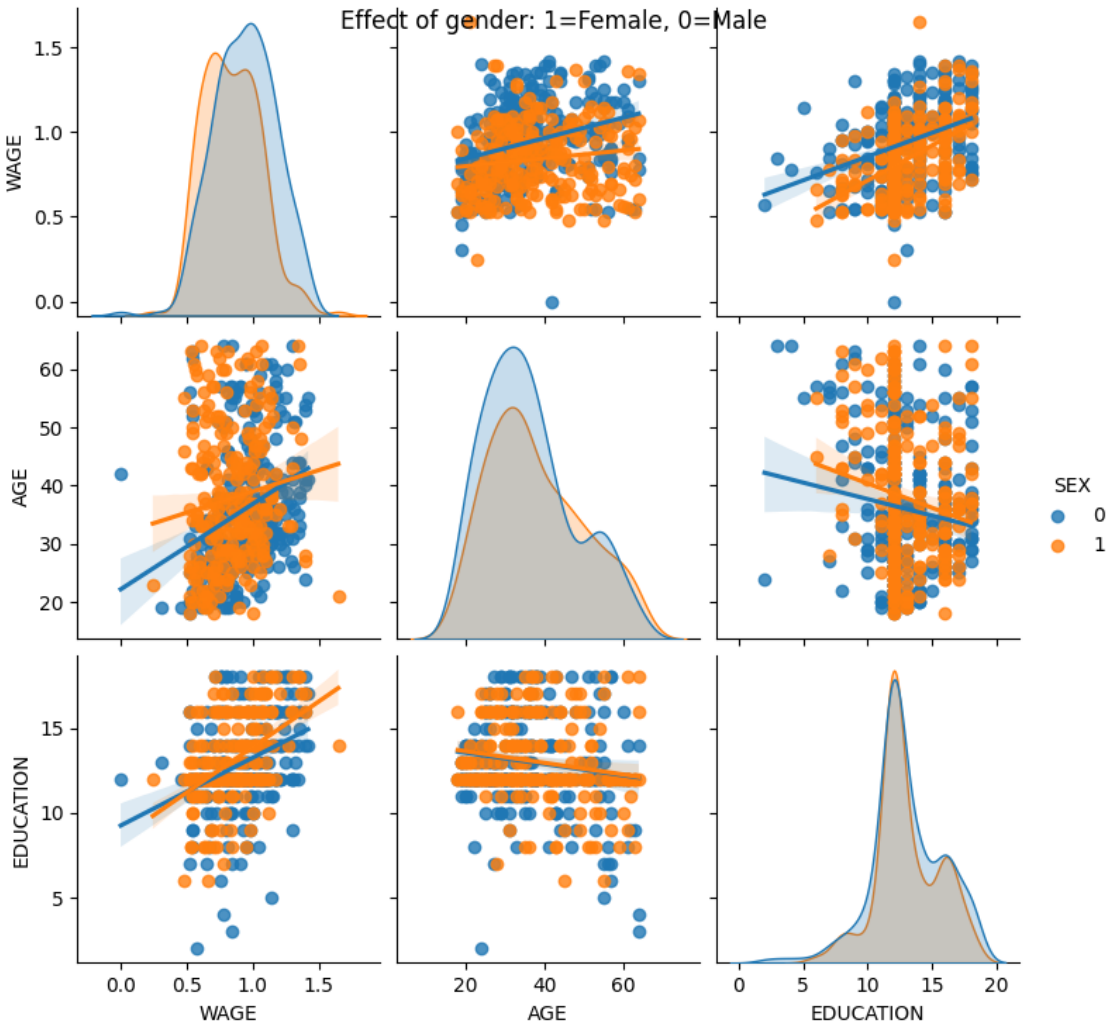
(continued from previous page)

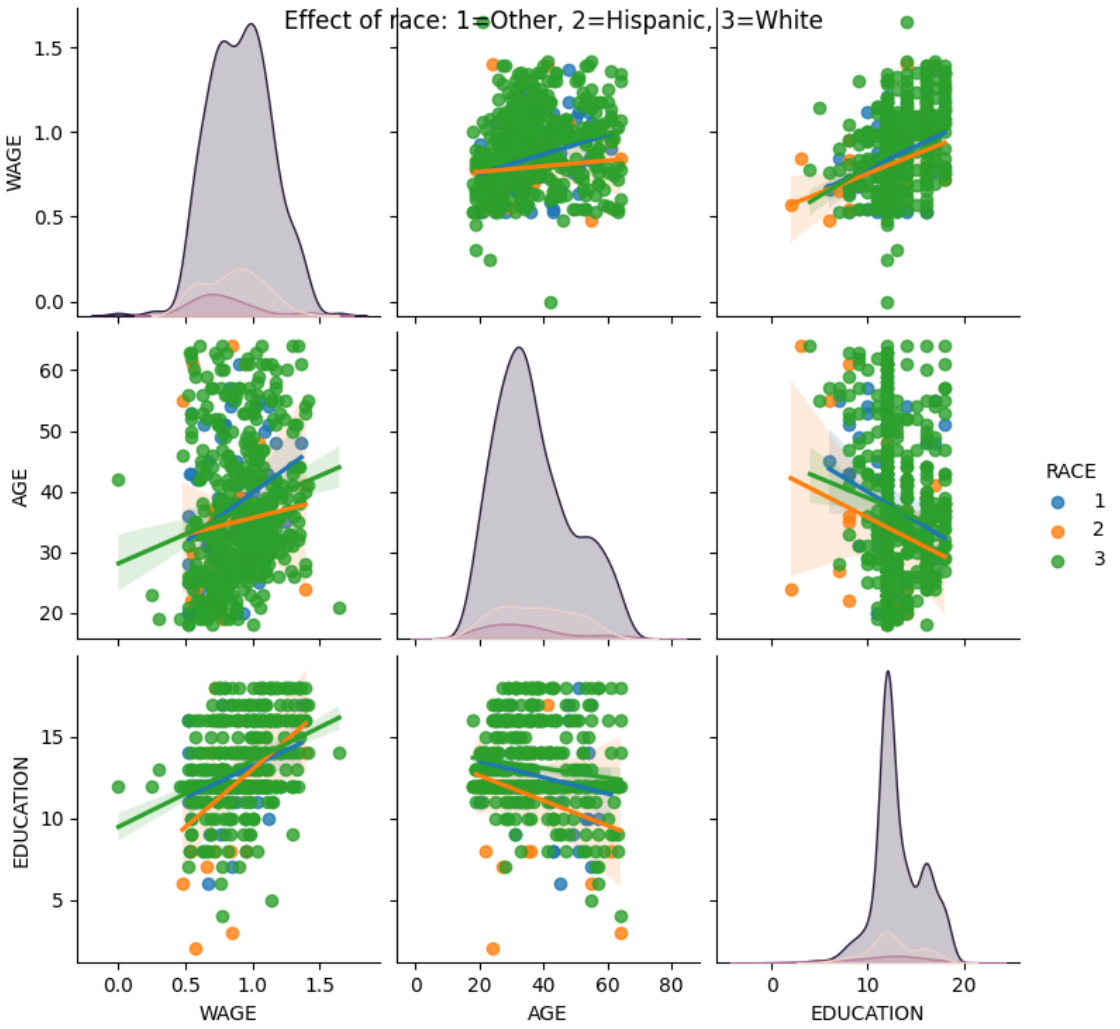
```
]

short_names = [n.split(":")[0] for n in names]

data = pandas.read_csv(
    "wages.txt", skiprows=27, skipfooter=6, sep=None, header=None, engine="python"
)
data.columns = pandas.Index(short_names)

# Log-transform the wages, because they typically are increased with
# multiplicative factors
import numpy as np

data["WAGE"] = np.log10(data["WAGE"])
```

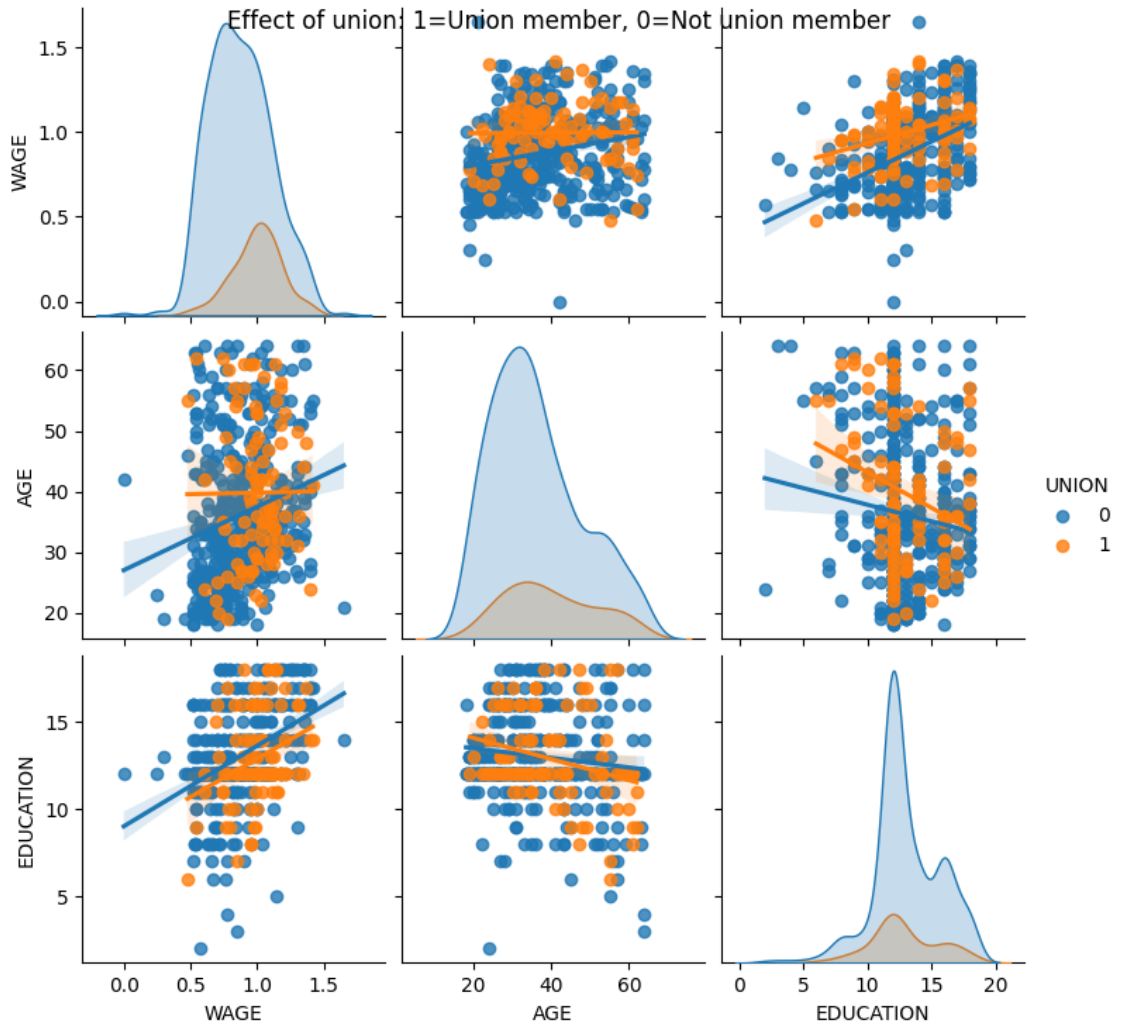
Plot scatter matrices highlighting different aspects

```
import seaborn

seaborn.pairplot(data, vars=["WAGE", "AGE", "EDUCATION"], kind="reg")

seaborn.pairplot(data, vars=["WAGE", "AGE", "EDUCATION"], kind="reg", hue="SEX")
plt.suptitle("Effect of gender: 1=Female, 0=Male")

seaborn.pairplot(data, vars=["WAGE", "AGE", "EDUCATION"], kind="reg", hue="RACE")
plt.suptitle("Effect of race: 1=Other, 2=Hispanic, 3=White")

seaborn.pairplot(data, vars=["WAGE", "AGE", "EDUCATION"], kind="reg", hue="UNION")
plt.suptitle("Effect of union: 1=Union member, 0=Not union member")
```


WAGE
AGE
EDUCATION
0.0
0.5
1.0
1.5
20
30
40
50
60
5
10
15


Effect of gender: 1=Female, 0=Male
WAGE
AGE
EDUCATION
SEX
0
1

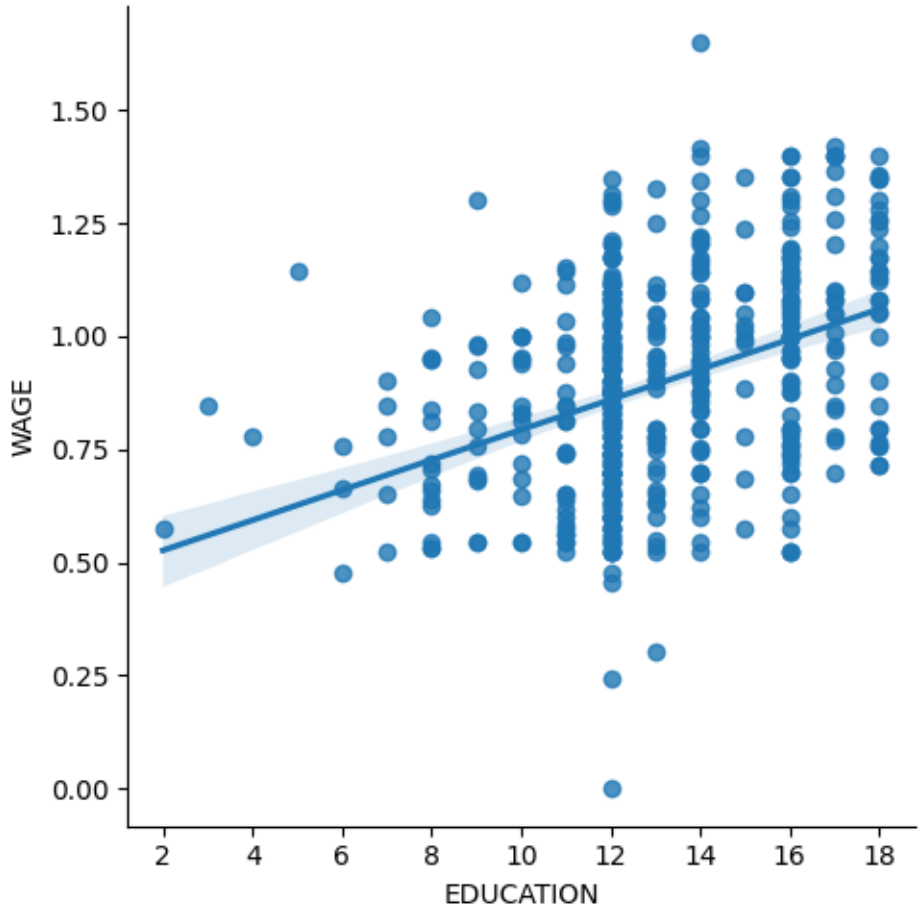

Effect of race: 1=Other, 2=Hispanic, 3=White
WAGE
AGE
EDUCATION
RACE
1
2
3
0.0
0.5
1.0
1.5
20
30
40
50
60
5
10
15
0
20
40
60
80
0
10
20




```
Text(0.5, 0.98, 'Effect of union: 1=Union member, 0=Not union member')
```

Plot a simple regression

```
seaborn.lmplot(y="WAGE", x="EDUCATION", data=data)

plt.show()
```



Total running time of the script: (0 minutes 8.469 seconds)

15.6.8 Air fares before and after 9/11

This is a business-intelligence (BI) like application.

What is interesting here is that we may want to study fares as a function of the year, paired accordingly to the trips, or forgetting the year, only as a function of the trip endpoints.

Using statsmodels' linear models, we find that both with an OLS (ordinary least square) and a robust fit, the intercept and the slope are significantly non-zero: the air fares have decreased between 2000 and 2001, and their dependence on distance travelled has also decreased

```
# Standard library imports
import os
```

Load the data

```
import pandas
import requests

if not os.path.exists("airfares.txt"):
    # Download the file if it is not present
    r = requests.get(
        "https://users.stat.ufl.edu/~winner/data/airq4.dat",
        verify=False,  # Wouldn't normally do this, but this site's certificate
        # is not yet distributed
```

(continues on next page)

(continued from previous page)

```
    )
    with open("airfares.txt", "wb") as f:
        f.write(r.content)

# As a separator, ' +' is a regular expression that means 'one of more
# space'
data = pandas.read_csv(
    "airfares.txt",
    delim_whitespace=True,
    header=0,
    names=[
        "city1",
        "city2",
        "pop1",
        "pop2",
        "dist",
        "fare_2000",
        "nb_passengers_2000",
        "fare_2001",
        "nb_passengers_2001",
    ],
)

# we log-transform the number of passengers
import numpy as np

data["nb_passengers_2000"] = np.log10(data["nb_passengers_2000"])
data["nb_passengers_2001"] = np.log10(data["nb_passengers_2001"])
```

```
/home/runner/work/scientific-python-lectures/scientific-python-lectures/packages/
↪statistics/examples/plot_airfare.py:38: FutureWarning: The 'delim_whitespace'␣
↪keyword in pd.read_csv is deprecated and will be removed in a future version. Use␣
↪``sep='\s+'`` instead
  data = pandas.read_csv(
```

Make a dataframe with the year as an attribute, instead of separate columns

```
# This involves a small danse in which we separate the dataframes in 2,
# one for year 2000, and one for 2001, before concatenating again.

# Make an index of each flight
data_flat = data.reset_index()

data_2000 = data_flat[
    ["city1", "city2", "pop1", "pop2", "dist", "fare_2000", "nb_passengers_2000"]
]
# Rename the columns
data_2000.columns = pandas.Index(
    ["city1", "city2", "pop1", "pop2", "dist", "fare", "nb_passengers"]
)
# Add a column with the year
data_2000.insert(0, "year", 2000)

data_2001 = data_flat[
    ["city1", "city2", "pop1", "pop2", "dist", "fare_2001", "nb_passengers_2001"]
]
```

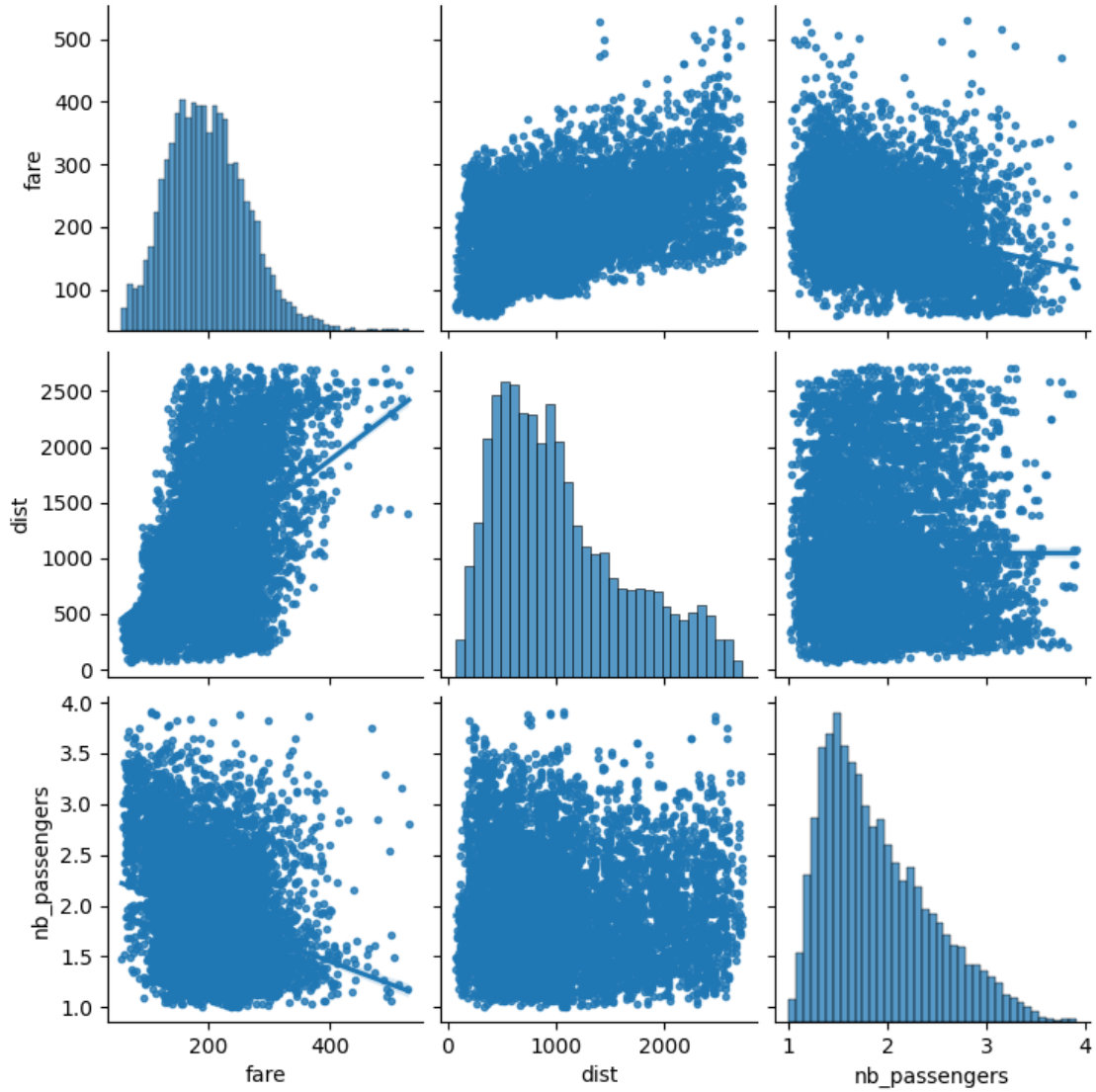
(continues on next page)

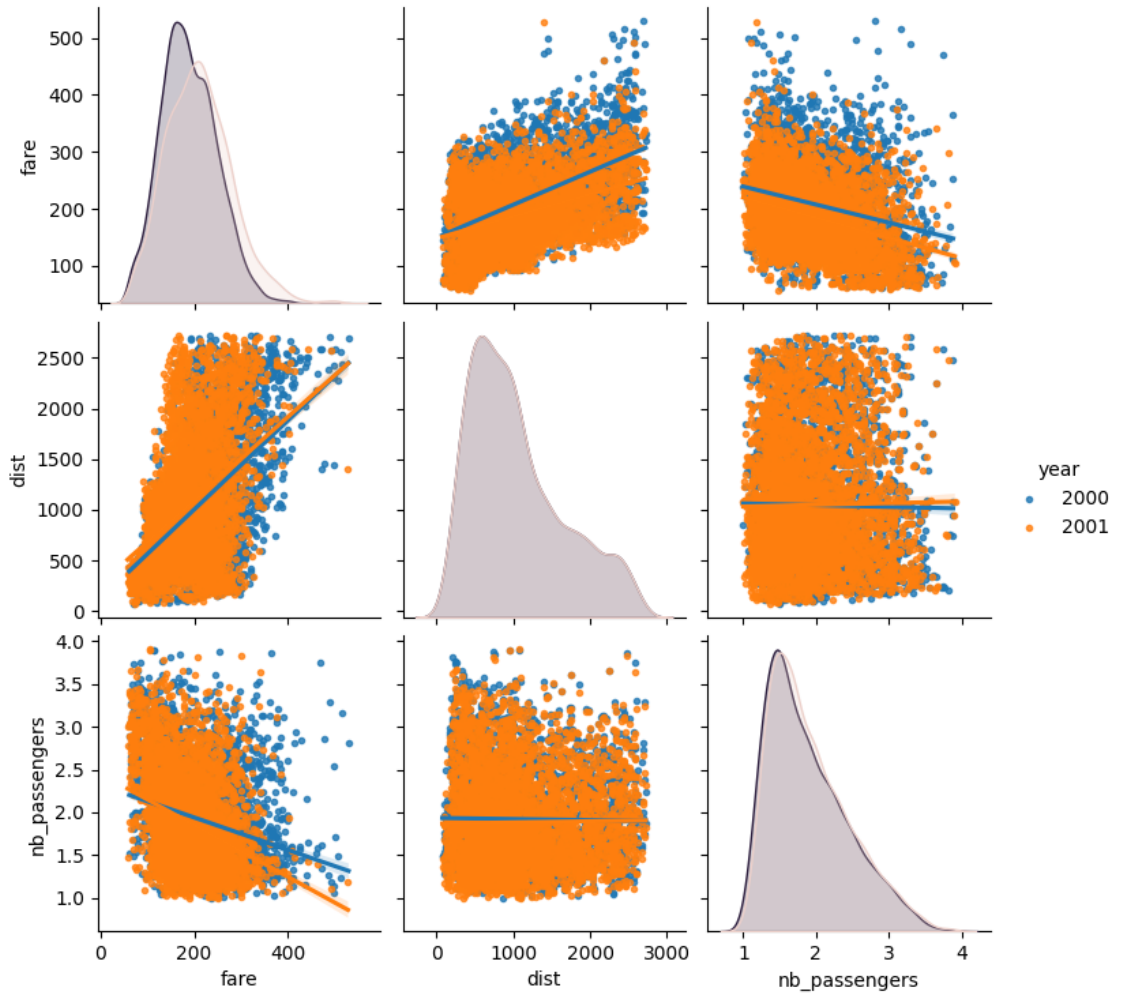
(continued from previous page)

```
# Rename the columns
data_2001.columns = pandas.Index(
    ["city1", "city2", "pop1", "pop2", "dist", "fare", "nb_passengers"]
)
# Add a column with the year
data_2001.insert(0, "year", 2001)

data_flat = pandas.concat([data_2000, data_2001])
```

Plot scatter matrices highlighting different aspects

```
import seaborn

seaborn.pairplot(
    data_flat, vars=["fare", "dist", "nb_passengers"], kind="reg", markers="."
)

# A second plot, to show the effect of the year (ie the 9/11 effect)
seaborn.pairplot(
    data_flat,
    vars=["fare", "dist", "nb_passengers"],
    kind="reg",
    hue="year",
    markers=".",
)
```

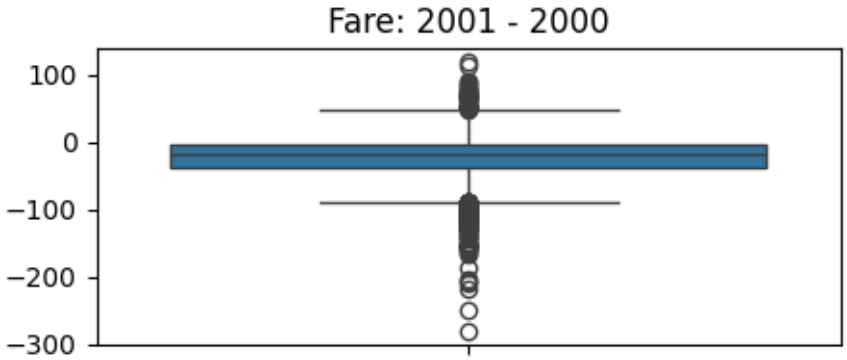
- 
fare
dist
nb_passengers
fare
dist
nb_passengers

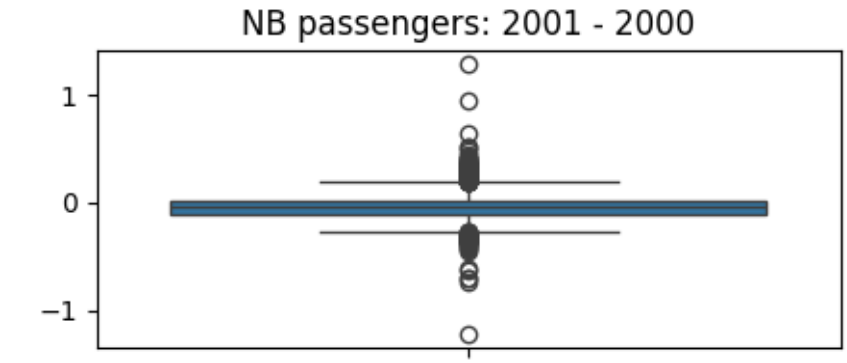



```
<seaborn.axisgrid.PairGrid object at 0x7f3b1bfa07d0>
```

Plot the difference in fare

```
import matplotlib.pyplot as plt

plt.figure(figsize=(5, 2))
seaborn.boxplot(data.fare_2001 - data.fare_2000)
plt.title("Fare: 2001 - 2000")
plt.subplots_adjust()

plt.figure(figsize=(5, 2))
seaborn.boxplot(data.nb_passengers_2001 - data.nb_passengers_2000)
plt.title("NB passengers: 2001 - 2000")
plt.subplots_adjust()
```



- 

Statistical testing: dependence of fare on distance and number of passengers

```
import statsmodels.formula.api as sm

result = sm.ols(formula="fare ~ 1 + dist + nb_passengers", data=data_flat).fit()
print(result.summary())

# Using a robust fit
result = sm.rlm(formula="fare ~ 1 + dist + nb_passengers", data=data_flat).fit()
print(result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   fare   R-squared:                       0.275
Model:                            OLS   Adj. R-squared:                  0.275
Method:                 Least Squares   F-statistic:                     1585.
Date:                Fri, 13 Jun 2025   Prob (F-statistic):               0.00
Time:                        14:07:19   Log-Likelihood:                -45532.
No. Observations:                8352   AIC:                         9.107e+04
Df Residuals:                    8349   BIC:                         9.109e+04
Df Model:                           2
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept      211.2428      2.466     85.669      0.000     206.409     216.076
dist             0.0484      0.001     48.149      0.000       0.046       0.050
nb_passengers  -32.8925      1.127    -29.191      0.000     -35.101     -30.684
==============================================================================
Omnibus:                      604.051   Durbin-Watson:                   1.446
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              740.733
Skew:                           0.710   Prob(JB):                    1.42e-161
```

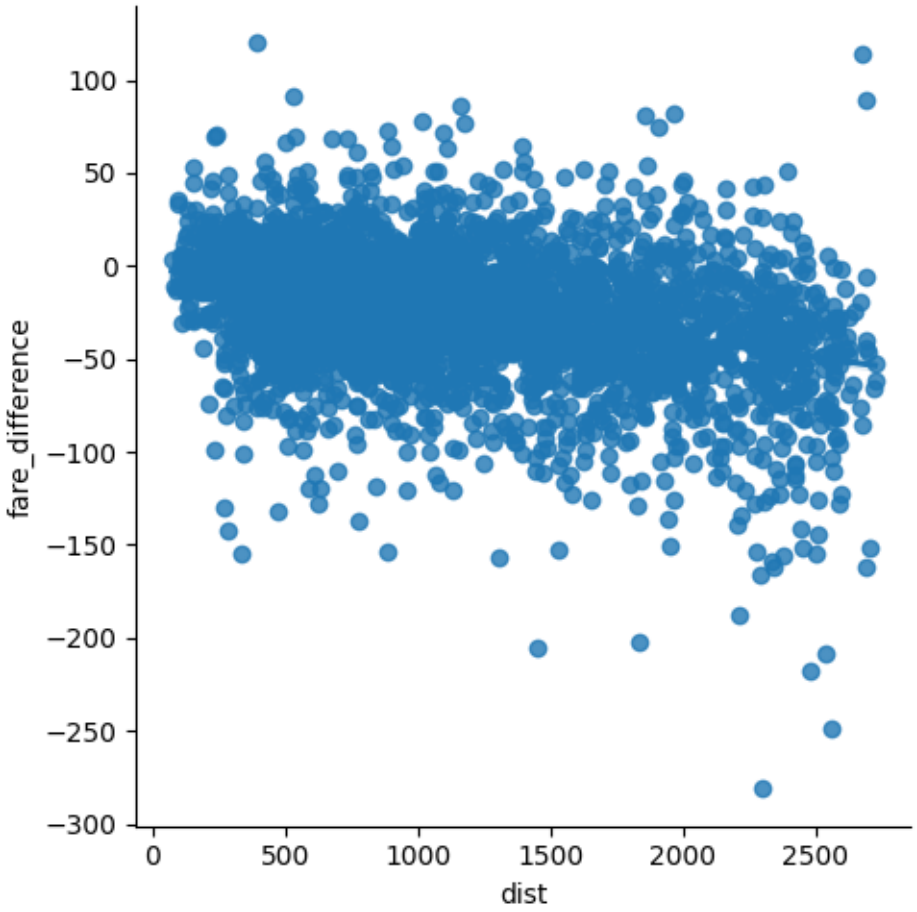
(continues on next page)

(continued from previous page)

```
Kurtosis:                       3.338   Cond. No.                         5.23e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 5.23e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
                    Robust linear Model Regression Results
==============================================================================
Dep. Variable:                   fare   No. Observations:                 8352
Model:                            RLM   Df Residuals:                     8349
Method:                          IRLS   Df Model:                            2
Norm:                          HuberT
Scale Est.:                       mad
Cov Type:                          H1
Date:                Fri, 13 Jun 2025
Time:                        14:07:19
No. Iterations:                    12
==============================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      215.0848      2.448     87.856      0.000     210.287     219.883
dist             0.0460      0.001     46.166      0.000       0.044       0.048
nb_passengers  -35.2686      1.119    -31.526      0.000     -37.461     -33.076
==============================================================================

If the model instance has been used for another fit with different fit parameters,␣
↪then the fit options might not be the correct ones anymore .
```

Statistical testing: regression of fare on distance: 2001/2000 difference

```
result = sm.ols(formula="fare_2001 - fare_2000 ~ 1 + dist", data=data).fit()
print(result.summary())

# Plot the corresponding regression
data["fare_difference"] = data["fare_2001"] - data["fare_2000"]
seaborn.lmplot(x="dist", y="fare_difference", data=data)

plt.show()
```




```
                            OLS Regression Results
==============================================================================
Dep. Variable:              fare_2001   R-squared:                       0.159
Model:                            OLS   Adj. R-squared:                  0.159
Method:                 Least Squares   F-statistic:                     791.7
Date:                Fri, 13 Jun 2025   Prob (F-statistic):          1.20e-159
Time:                        14:07:19   Log-Likelihood:                -22640.
No. Observations:                4176   AIC:                         4.528e+04
Df Residuals:                    4174   BIC:                         4.530e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    148.0279      1.673     88.480      0.000     144.748     151.308
dist           0.0388      0.001     28.136      0.000       0.036       0.041
==============================================================================
Omnibus:                      136.558   Durbin-Watson:                   1.544
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              149.624
Skew:                           0.462   Prob(JB):                     3.23e-33
Kurtosis:                       2.920   Cond. No.                     2.40e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

(continues on next page)

(continued from previous page)

```
[2] The condition number is large, 2.4e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Total running time of the script: (0 minutes 7.068 seconds)

15.7 Solutions to this chapter's exercises

15.7.1 Solutions to this chapter's exercises

Relating Gender and IQ

Going back to the brain size + IQ data, test if the VIQ of male and female are different after removing the effect of brain size, height and weight.

Notice that here 'Gender' is a categorical value. As it is a non-float data type, statsmodels is able to automatically infer this.

```
import pandas
from statsmodels.formula.api import ols

data = pandas.read_csv("../brain_size.csv", sep=";", na_values=".")

model = ols("VIQ ~ Gender + MRI_Count + Height", data).fit()
print(model.summary())

# Here, we don't need to define a contrast, as we are testing a single
# coefficient of our model, and not a combination of coefficients.
# However, defining a contrast, which would then be a 'unit contrast',
# will give us the same results
print(model.f_test([0, 1, 0, 0]))
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    VIQ   R-squared:                       0.246
Model:                            OLS   Adj. R-squared:                  0.181
Method:                 Least Squares   F-statistic:                     3.809
Date:                Fri, 13 Jun 2025   Prob (F-statistic):             0.0184
Time:                        14:07:19   Log-Likelihood:                -172.34
No. Observations:                  39   AIC:                             352.7
Df Residuals:                      35   BIC:                             359.3
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      166.6258     88.824      1.876      0.069     -13.696     346.948
Gender[T.Male]   8.8524     10.710      0.827      0.414     -12.890      30.595
MRI_Count        0.0002   6.46e-05      2.615      0.013    3.78e-05       0.000
Height          -3.0837      1.276     -2.417      0.021      -5.674      -0.494
==============================================================================
Omnibus:                        7.373   Durbin-Watson:                   2.109
Prob(Omnibus):                  0.025   Jarque-Bera (JB):                2.252
Skew:                           0.005   Prob(JB):                        0.324
Kurtosis:                       1.823   Cond. No.                     2.40e+07
==============================================================================
```

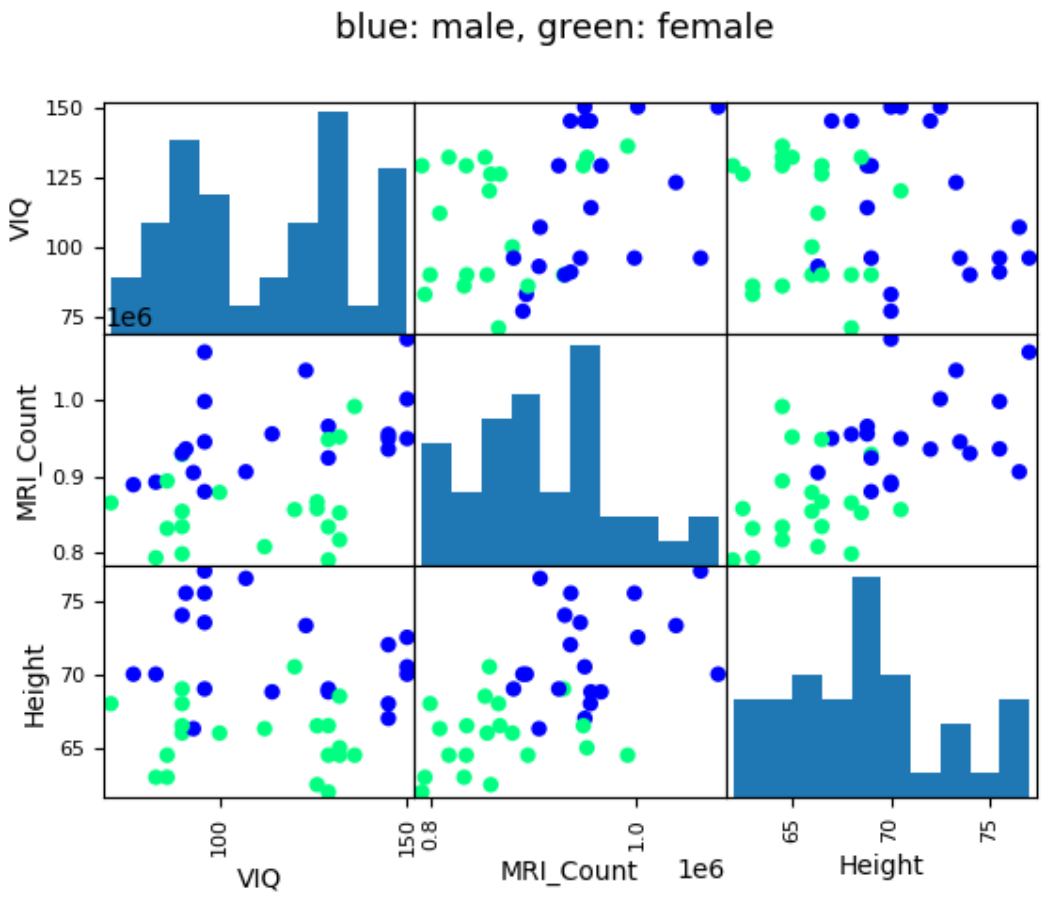
(continues on next page)

(continued from previous page)

```
Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 2.4e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
<F test: F=0.683196084584229, p=0.4140878441244694, df_denom=35, df_num=1>
```

Here we plot a scatter matrix to get intuitions on our results. This goes beyond what was asked in the exercise

```
# This plotting is useful to get an intuitions on the relationships between
# our different variables

from pandas import plotting
import matplotlib.pyplot as plt

# Fill in the missing values for Height for plotting
data["Height"] = data["Height"].ffill()

# The parameter 'c' is passed to plt.scatter and will control the color
# The same holds for parameters 'marker', 'alpha' and 'cmap', that
# control respectively the type of marker used, their transparency and
# the colormap
plotting.scatter_matrix(
    data[["VIQ", "MRI_Count", "Height"]],
    c=(data["Gender"] == "Female"),
    marker="o",
    alpha=1,
    cmap="winter",
)

fig = plt.gcf()
fig.suptitle("blue: male, green: female", size=13)

plt.show()
```



Total running time of the script: (0 minutes 0.239 seconds)

CHAPTER 16

Sympy : Symbolic Mathematics in Python

Author: *Fabian Pedregosa*

Objectives

1. Evaluate expressions with arbitrary precision.
2. Perform algebraic manipulations on symbolic expressions.
3. **Perform basic calculus tasks (limits, differentiation and** integration) with symbolic expressions.
4. Solve polynomial and transcendental equations.
5. Solve some differential equations.

What is SymPy? SymPy is a Python library for symbolic mathematics. It aims to be an alternative to systems such as Mathematica or Maple while keeping the code as simple as possible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.

Sympy documentation and packages for installation can be found on https://www.sympy.org/

Chapters contents

- *First Steps with SymPy*
  - *Using SymPy as a calculator*
  - *Symbols*

- *Algebraic manipulations*
  - *Expand*
  - *Simplify*
- *Calculus*
  - *Limits*
  - *Differentiation*
  - *Series expansion*
  - *Integration*
- *Equation solving*
- *Linear Algebra*
  - *Matrices*
  - *Differential Equations*

16.1 First Steps with SymPy

16.1.1 Using SymPy as a calculator

SymPy defines three numerical types: `Real`, `Rational` and `Integer`.

The Rational class represents a rational number as a pair of two Integers: the numerator and the denominator, so `Rational(1, 2)` represents 1/2, `Rational(5, 2)` 5/2 and so on:

```
>>> import sympy as sym
>>> a = sym.Rational(1, 2)

>>> a
1/2

>>> a*2
1
```

SymPy uses mpmath in the background, which makes it possible to perform computations using arbitrary-precision arithmetic. That way, some special constants, like *e*, *pi*, *oo* (Infinity), are treated as symbols and can be evaluated with arbitrary precision:

```
>>> sym.pi**2
pi**2

>>> sym.pi.evalf()
3.14159265358979

>>> (sym.pi + sym.exp(1)).evalf()
5.85987448204884
```

as you see, `evalf` evaluates the expression to a floating-point number.

There is also a class representing mathematical infinity, called `oo`:

```
>>> sym.oo > 99999
True
```

(continues on next page)

(continued from previous page)

```
>>> sym.oo + 1
oo
```

Exercises

1. Calculate $\sqrt{2}$ with 100 decimals.
2. Calculate $1/2 + 1/3$ in rational arithmetic.

16.1.2 Symbols

In contrast to other Computer Algebra Systems, in SymPy you have to declare symbolic variables explicitly:

```
>>> x = sym.Symbol('x')
>>> y = sym.Symbol('y')
```

Then you can manipulate them:

```
>>> x + y + x - y
2*x

>>> (x + y) ** 2
(x + y)**2
```

Symbols can now be manipulated using some of python operators: `+`, `-`, `*`, `**` (arithmetic), `&`, `|`, `~`, `>>`, `<<` (boolean).

Printing

Sympy allows for control of the display of the output. From here we use the following setting for printing:

```
>>> sym.init_printing(use_unicode=False, wrap_line=True)
```

16.2 Algebraic manipulations

SymPy is capable of performing powerful algebraic manipulations. We'll take a look into some of the most frequently used: expand and simplify.

16.2.1 Expand

Use this to expand an algebraic expression. It will try to denest powers and multiplications:

```
>>> sym.expand((x + y) ** 3)
 3      2          2    3
x  + 3*x *y + 3*x*y  + y
>>> 3 * x * y ** 2 + 3 * y * x ** 2 + x ** 3 + y ** 3
 3      2          2    3
x  + 3*x *y + 3*x*y  + y
```

Further options can be given in form on keywords:

```
>>> sym.expand(x + y, complex=True)
re(x) + re(y) + I*im(x) + I*im(y)
>>> sym.I * sym.im(x) + sym.I * sym.im(y) + sym.re(x) + sym.re(y)
re(x) + re(y) + I*im(x) + I*im(y)

>>> sym.expand(sym.cos(x + y), trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)
>>> sym.cos(x) * sym.cos(y) - sym.sin(x) * sym.sin(y)
-sin(x)*sin(y) + cos(x)*cos(y)
```

16.2.2 Simplify

Use simplify if you would like to transform an expression into a simpler form:

```
>>> sym.simplify((x + x * y) / x)
y + 1
```

Simplification is a somewhat vague term, and more precises alternatives to simplify exists: `powsimp` (simplification of exponents), `trigsimp` (for trigonometric expressions) , `logcombine`, `radsimp`, together.

Exercises

1. Calculate the expanded form of $(x + y)^6$.
2. Simplify the trigonometric expression $\sin(x)/\cos(x)$

16.3 Calculus

16.3.1 Limits

Limits are easy to use in SymPy, they follow the syntax `limit(function, variable, point)`, so to compute the limit of $f(x)$ as $x \to 0$, you would issue `limit(f, x, 0)`:

```
>>> sym.limit(sym.sin(x) / x, x, 0)
1
```

you can also calculate the limit at infinity:

```
>>> sym.limit(x, x, sym.oo)
oo

>>> sym.limit(1 / x, x, sym.oo)
0

>>> sym.limit(x ** x, x, 0)
1
```

16.3.2 Differentiation

You can differentiate any SymPy expression using `diff(func, var)`. Examples:

```
>>> sym.diff(sym.sin(x), x)
cos(x)
>>> sym.diff(sym.sin(2 * x), x)
2*cos(2*x)
```

(continues on next page)

(continued from previous page)

```
>>> sym.diff(sym.tan(x), x)
   2
tan (x) + 1
```

You can check that it is correct by:

```
>>> sym.limit((sym.tan(x + y) - sym.tan(x)) / y, y, 0)
   1
-------
   2
cos (x)
```

Which is equivalent since

$$\sec(x) = \frac{1}{\cos(x)} and \sec^2(x) = \tan^2(x) + 1.$$

You can check this as well:

```
>>> sym.trigsimp(sym.diff(sym.tan(x), x))
   1
-------
   2
cos (x)
```

Higher derivatives can be calculated using the `diff(func, var, n)` method:

```
>>> sym.diff(sym.sin(2 * x), x, 1)
2*cos(2*x)

>>> sym.diff(sym.sin(2 * x), x, 2)
-4*sin(2*x)

>>> sym.diff(sym.sin(2 * x), x, 3)
-8*cos(2*x)
```

16.3.3 Series expansion

SymPy also knows how to compute the Taylor series of an expression at a point. Use `series(expr, var)`:

```
>>> sym.series(sym.cos(x), x)
     2    4
    x    x     / 6\
1 - -- + -- + O\x /
    2    24
>>> sym.series(1/sym.cos(x), x)
     2      4
    x    5*x     / 6\
1 + -- + ---- + O\x /
    2     24
```

Exercises

1. Calculate $\lim_{x \to 0} \sin(x)/x$
2. Calculate the derivative of $log(x)$ for x.

16.3.4 Integration

SymPy has support for indefinite and definite integration of transcendental elementary and special functions via `integrate()` facility, which uses the powerful extended Risch-Norman algorithm and some heuristics and pattern matching. You can integrate elementary functions:

```
>>> sym.integrate(6 * x ** 5, x)
 6
x
>>> sym.integrate(sym.sin(x), x)
-cos(x)
>>> sym.integrate(sym.log(x), x)
x*log(x) - x
>>> sym.integrate(2 * x + sym.sinh(x), x)
 2
x  + cosh(x)
```

Also special functions are handled easily:

```
>>> sym.integrate(sym.exp(-x ** 2) * sym.erf(x), x)
  ____    2
\/ pi *erf (x)
--------------
      4
```

It is possible to compute definite integral:

```
>>> sym.integrate(x**3, (x, -1, 1))
0
>>> sym.integrate(sym.sin(x), (x, 0, sym.pi / 2))
1
>>> sym.integrate(sym.cos(x), (x, -sym.pi / 2, sym.pi / 2))
2
```

Also improper integrals are supported as well:

```
>>> sym.integrate(sym.exp(-x), (x, 0, sym.oo))
1
>>> sym.integrate(sym.exp(-x ** 2), (x, -sym.oo, sym.oo))
  ____
\/ pi
```

16.4 Equation solving

SymPy is able to solve algebraic equations, in one and several variables using `solveset()`:

```
>>> sym.solveset(x ** 4 - 1, x)
{-1, 1, -I, I}
```

As you can see it takes as first argument an expression that is supposed to be equaled to 0. It also has (limited) support for transcendental equations:

```
>>> sym.solveset(sym.exp(x) + 1, x)
{I*(2*n*pi + pi) | n in Integers}
```

Systems of linear equations

Sympy is able to solve a large part of polynomial equations, and is also capable of solving multiple equations with respect to multiple variables giving a tuple as second argument. To do this you use the `solve()` command:

```
>>> solution = sym.solve((x + 5 * y - 2, -3 * x + 6 * y - 15), (x, y))
>>> solution[x], solution[y]
(-3, 1)
```

Another alternative in the case of polynomial equations is *factor*. *factor* returns the polynomial factorized into irreducible terms, and is capable of computing the factorization over various domains:

```
>>> f = x ** 4 - 3 * x ** 2 + 1
>>> sym.factor(f)
/ 2        \ / 2        \
\x  - x - 1/*\x  + x - 1/

>>> sym.factor(f, modulus=5)
       2        2
(x - 2) *(x + 2)
```

SymPy is also able to solve boolean equations, that is, to decide if a certain boolean expression is satisfiable or not. For this, we use the function satisfiable:

```
>>> sym.satisfiable(x & y)
{x: True, y: True}
```

This tells us that `(x & y)` is True whenever `x` and `y` are both True. If an expression cannot be true, i.e. no values of its arguments can make the expression True, it will return False:

```
>>> sym.satisfiable(x & ~x)
False
```

Exercises

1. Solve the system of equations $x + y = 2$, $2 \cdot x + y = 0$
2. Are there boolean values `x`, `y` that make `(~x | y) & (~y | x)` true?

16.5 Linear Algebra

16.5.1 Matrices

Matrices are created as instances from the Matrix class:

```
>>> sym.Matrix([[1, 0], [0, 1]])
[1  0]
[    ]
[0  1]
```

unlike a NumPy array, you can also put Symbols in it:

```
>>> x, y = sym.symbols('x, y')
>>> A = sym.Matrix([[1, x], [y, 1]])
>>> A
[1  x]
[    ]
[y  1]
```

(continues on next page)

(continued from previous page)

```
>>> A**2
[x*y + 1    2*x  ]
[                ]
[  2*y    x*y + 1]
```

16.5.2 Differential Equations

SymPy is capable of solving (some) Ordinary Differential. To solve differential equations, use dsolve. First, create an undefined function by passing cls=Function to the symbols function:

```
>>> f, g = sym.symbols('f g', cls=sym.Function)
```

f and g are now undefined functions. We can call f(x), and it will represent an unknown function:

```
>>> f(x)
f(x)

>>> f(x).diff(x, x) + f(x)
         2
        d
f(x) + ---(f(x))
         2
       dx

>>> sym.dsolve(f(x).diff(x, x) + f(x), f(x))
f(x) = C1*sin(x) + C2*cos(x)
```

Keyword arguments can be given to this function in order to help if find the best possible resolution system. For example, if you know that it is a separable equations, you can use keyword `hint='separable'` to force dsolve to resolve it as a separable equation:

```
>>> sym.dsolve(sym.sin(x) * sym.cos(f(x)) + sym.cos(x) * sym.sin(f(x)) * f(x).diff(x),
↪ f(x), hint='separable')
               /  C1  \                    /  C1  \
 [f(x) = - acos|------| + 2*pi, f(x) = acos|------|]
               \cos(x)/                    \cos(x)/
```

Exercises

1. Solve the Bernoulli differential equation

$$x\frac{df(x)}{x} + f(x) - f(x)^2 = 0$$

2. Solve the same equation using `hint='Bernoulli'`. What do you observe ?

CHAPTER 17

scikit-image: image processing

Author: *Emmanuelle Gouillart*

scikit-image is a Python package dedicated to image processing, using NumPy arrays as image objects. This chapter describes how to use `scikit-image` for various image processing tasks, and how it relates to other scientific Python modules such as NumPy and SciPy.

See also

For basic image manipulation, such as image cropping or simple filtering, a large number of simple operations can be realized with NumPy and SciPy only. See *Image manipulation and processing using NumPy and SciPy*.

Note that you should be familiar with the content of the previous chapter before reading the current one, as basic operations such as masking and labeling are a prerequisite.

Chapters contents

- *Introduction and concepts*
  - *`scikit-image` and the scientific Python ecosystem*
  - *What is included in scikit-image*
- *Importing*
- *Example data*
- *Input/output, data types and colorspaces*
  - *Data types*

  - *Colorspaces*
- *Image preprocessing / enhancement*
  - *Local filters*
  - *Non-local filters*
  - *Mathematical morphology*
- *Image segmentation*
  - *Binary segmentation: foreground + background*
  - *Marker based methods*
- *Measuring regions' properties*
- *Data visualization and interaction*
- *Feature extraction for computer vision*
- *Full code examples*
- *Examples for the scikit-image chapter*

17.1 Introduction and concepts

Images are NumPy's arrays `np.ndarray`

image
: `np.ndarray`

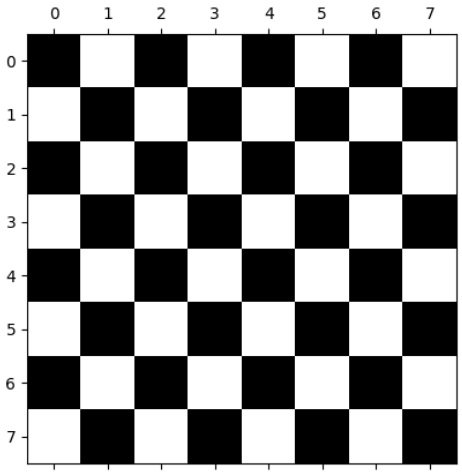
pixels
: array values: `a[2, 3]`

channels
: array dimensions

image encoding
: `dtype` (`np.uint8`, `np.uint16`, `np.float`)

filters
: functions (`numpy`, `skimage`, `scipy`)

```
>>> import numpy as np
>>> check = np.zeros((8, 8))
>>> check[::2, 1::2] = 1
>>> check[1::2, ::2] = 1
>>> import matplotlib.pyplot as plt
>>> plt.imshow(check, cmap='gray', interpolation='nearest')
<matplotlib.image.AxesImage object at ...>
```



17.1.1 `scikit-image` and the scientific Python ecosystem

`scikit-image` is packaged in both `pip` and `conda`-based Python installations, as well as in most Linux distributions. Other Python packages for image processing & visualization that operate on NumPy arrays include:

`scipy.ndimage`
: For N-dimensional arrays. Basic filtering, mathematical morphology, regions properties

Mahotas
: With a focus on high-speed implementations.

Napari
: A fast, interactive, multi-dimensional image viewer built in Qt.

Some powerful C++ image processing libraries also have Python bindings:

OpenCV
: A highly optimized computer vision library with a focus on real-time applications.

ITK
: The Insight ToolKit, especially useful for registration and working with 3D images.

To varying degrees, these tend to be less Pythonic and NumPy-friendly.

17.1.2 What is included in scikit-image

- Website: https://scikit-image.org/
- Gallery of examples: https://scikit-image.org/docs/stable/auto_examples/

The library contains predominantly image processing algorithms, but also utility functions to ease data handling and processing. It contains the following submodules:

`color`
: Color space conversion.

`data`
: Test images and example data.

`draw`
: Drawing primitives (lines, text, etc.) that operate on NumPy arrays.

`exposure`
: Image intensity adjustment, e.g., histogram equalization, etc.

`feature`
: Feature detection and extraction, e.g., texture analysis corners, etc.

`filters`
: Sharpening, edge finding, rank filters, thresholding, etc.

`graph`
: Graph-theoretic operations, e.g., shortest paths.

`io`
: Reading, saving, and displaying images and video.

`measure`
: Measurement of image properties, e.g., region properties and contours.

`metrics`
: Metrics corresponding to images, e.g. distance metrics, similarity, etc.

`morphology`
: Morphological operations, e.g., opening or skeletonization.

`restoration`
: Restoration algorithms, e.g., deconvolution algorithms, denoising, etc.

`segmentation`
: Partitioning an image into multiple regions.

`transform`
: Geometric and other transforms, e.g., rotation or the Radon transform.

`util`
: Generic utilities.

17.2 Importing

We import `scikit-image` using the convention:

```
>>> import skimage as ski
```

Most functionality lives in subpackages, e.g.:

```
>>> image = ski.data.cat()
```

You can list all submodules with:

```
>>> for m in dir(ski): print(m)
__version__
color
data
draw
exposure
feature
filters
future
graph
io
measure
metrics
morphology
registration
restoration
segmentation
```

(continues on next page)

(continued from previous page)

```
transform
util
```

Most `scikit-image` functions take NumPy `ndarrays` as arguments

```
>>> camera = ski.data.camera()
>>> camera.dtype
dtype('uint8')
>>> camera.shape
(512, 512)
>>> filtered_camera = ski.filters.gaussian(camera, sigma=1)
>>> type(filtered_camera)
<class 'numpy.ndarray'>
```

17.3 Example data

To start off, we need example images to work with. The library ships with a few of these:

`skimage.data`

```
>>> image = ski.data.cat()
>>> image.shape
(300, 451, 3)
```

17.4 Input/output, data types and colorspaces

I/O: `skimage.io`

Save an image to disk: `skimage.io.imsave()`

```
>>> ski.io.imsave("cat.png", image)
```

Reading from files: `skimage.io.imread()`

```
>>> cat = ski.io.imread("cat.png")
```

This works with many data formats supported by the ImageIO library.

Loading also works with URLs:

```
>>> logo = ski.io.imread('https://scikit-image.org/_static/img/logo.png')
```

17.4.1 Data types

Image ndarrays can be represented either by integers (signed or unsigned) or floats.

Careful with overflows with integer data types

```
>>> camera = ski.data.camera()
>>> camera.dtype
dtype('uint8')
>>> camera_multiply = 3 * camera
```

Different integer sizes are possible: 8-, 16- or 32-bytes, signed or unsigned.

Warning

An important (if questionable) **skimage convention**: float images are supposed to lie in [-1, 1] (in order to have comparable contrast for all float images)

```
>>> camera_float = ski.util.img_as_float(camera)
>>> camera.max(), camera_float.max()
(np.uint8(255), np.float64(1.0))
```

Some image processing routines need to work with float arrays, and may hence output an array with a different type and the data range from the input array

```
>>> camera_sobel = ski.filters.sobel(camera)
>>> camera_sobel.max()
np.float64(0.644...)
```

Utility functions are provided in `skimage` to convert both the dtype and the data range, following skimage's conventions: `util.img_as_float`, `util.img_as_ubyte`, etc.

See the user guide for more details.

17.4.2 Colorspaces

Color images are of shape (N, M, 3) or (N, M, 4) (when an alpha channel encodes transparency)

```
>>> face = sp.datasets.face()
>>> face.shape
(768, 1024, 3)
```

Routines converting between different colorspaces (RGB, HSV, LAB etc.) are available in `skimage.color` : `color.rgb2hsv`, `color.lab2rgb`, etc. Check the docstring for the expected dtype (and data range) of input images.

3D images

Most functions of `skimage` can take 3D images as input arguments. Check the docstring to know if a function can be used on 3D images (for example MRI or CT images).

Exercise

Open a color image on your disk as a NumPy array.

Find a skimage function computing the histogram of an image and plot the histogram of each color channel

Convert the image to grayscale and plot its histogram.

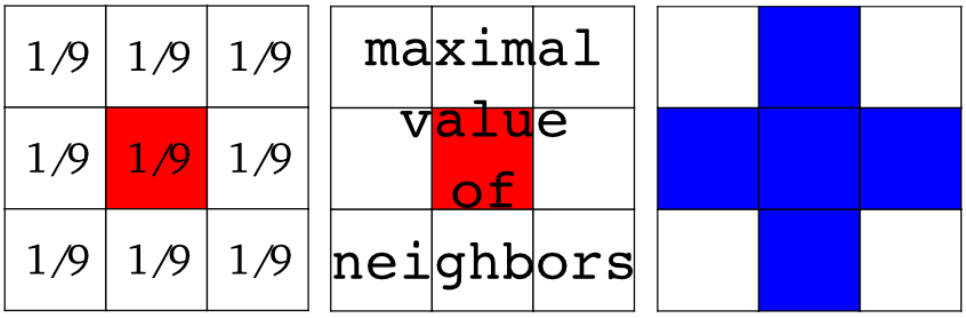
17.5 Image preprocessing / enhancement

Goals: denoising, feature (edges) extraction, ...

17.5.1 Local filters

Local filters replace the value of pixels by a function of the values of neighboring pixels. The function can be linear or non-linear.

Neighbourhood: square (choose size), disk, or more complicated *structuring element*.

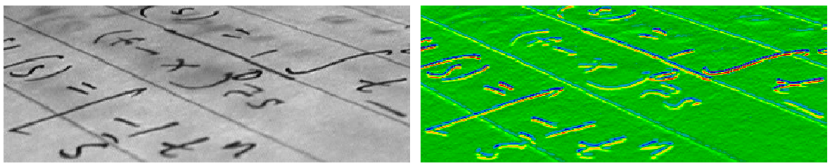


Example : horizontal Sobel filter

```
>>> text = ski.data.text()
>>> hsobel_text = ski.filters.sobel_h(text)
```

Uses the following linear kernel for computing horizontal gradients:

```
1   2   1
0   0   0
-1  -2  -1
```



17.5.2 Non-local filters

Non-local filters use a large region of the image (or all the image) to transform the value of one pixel:

```
>>> camera = ski.data.camera()
>>> camera_equalized = ski.exposure.equalize_hist(camera)
```

Enhances contrast in large almost uniform regions.

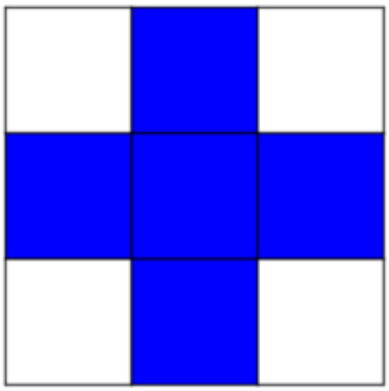
17.5.3 Mathematical morphology

See wikipedia for an introduction on mathematical morphology.

Probe an image with a simple shape (a **structuring element**), and modify this image according to how the shape locally fits or misses the image.

Default structuring element: 4-connectivity of a pixel

```
>>> # Import structuring elements to make them more easily accessible
>>> from skimage.morphology import disk, diamond

>>> diamond(1)
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]], dtype=uint8)
```



Erosion = minimum filter. Replace the value of a pixel by the minimal value covered by the structuring element.:

```
>>> a = np.zeros((7,7), dtype=np.uint8)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> ski.morphology.binary_erosion(a, diamond(1)).astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> #Erosion removes objects smaller than the structure
>>> ski.morphology.binary_erosion(a, diamond(2)).astype(np.uint8)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

Dilation: maximum filter:

```
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.,  0.]])
>>> ski.morphology.binary_dilation(a, diamond(1)).astype(np.uint8)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

Opening: erosion + dilation:

```
>>> a = np.zeros((5,5), dtype=int)
>>> a[1:4, 1:4] = 1; a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> ski.morphology.binary_opening(a, diamond(1)).astype(np.uint8)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

Opening removes small objects and smoothes corners.

Grayscale mathematical morphology

Mathematical morphology operations are also available for (non-binary) grayscale images (int or float type). Erosion and dilation correspond to minimum (resp. maximum) filters.

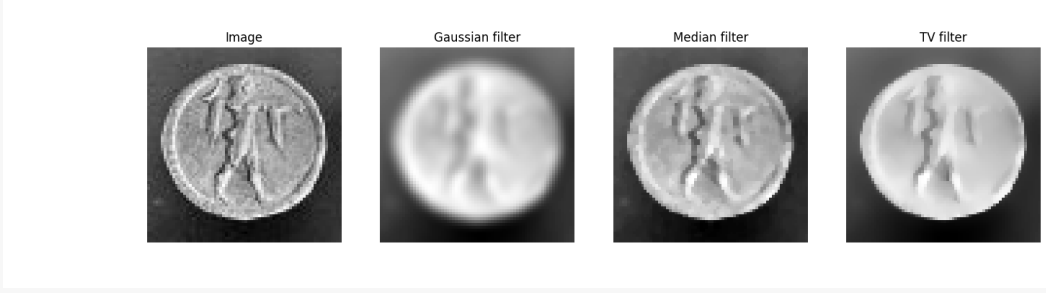
Higher-level mathematical morphology are available: tophat, skeletonization, etc.

See also

Basic mathematical morphology is also implemented in `scipy.ndimage.morphology`. The `scipy.ndimage` implementation works on arbitrary-dimensional arrays.

Example of filters comparison: image denoising

```
>>> coins = ski.data.coins()
>>> coins_zoom = coins[10:80, 300:370]
>>> median_coins = ski.filters.median(
...     coins_zoom, disk(1)
... )
>>> tv_coins = ski.restoration.denoise_tv_chambolle(
...     coins_zoom, weight=0.1
... )
>>> gaussian_coins = ski.filters.gaussian(coins, sigma=2)
```



17.6 Image segmentation

Image segmentation is the attribution of different labels to different regions of the image, for example in order to extract the pixels of an object of interest.

17.6.1 Binary segmentation: foreground + background

Histogram-based method: Otsu thresholding

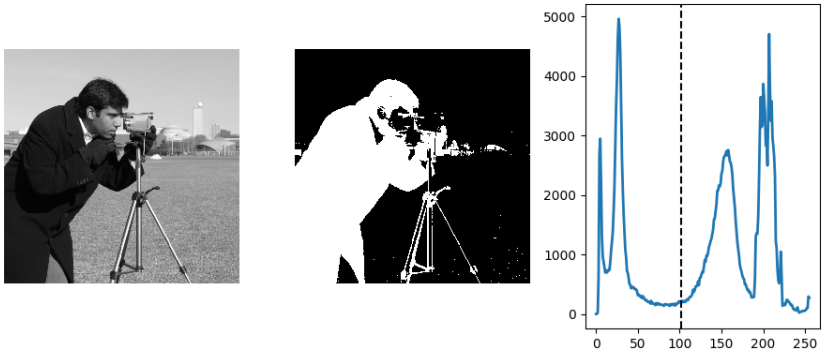
Tip

The Otsu method is a simple heuristic to find a threshold to separate the foreground from the background.

Earlier scikit-image versions

`skimage.filters` is called `skimage.filter` in earlier versions of scikit-image

```
camera = ski.data.camera()
val = ski.filters.threshold_otsu(camera)
mask = camera < val
```



Labeling connected components of a discrete image

Tip

Once you have separated foreground objects, it is use to separate them from each other. For this, we can assign a different integer labels to each one.

Synthetic data:

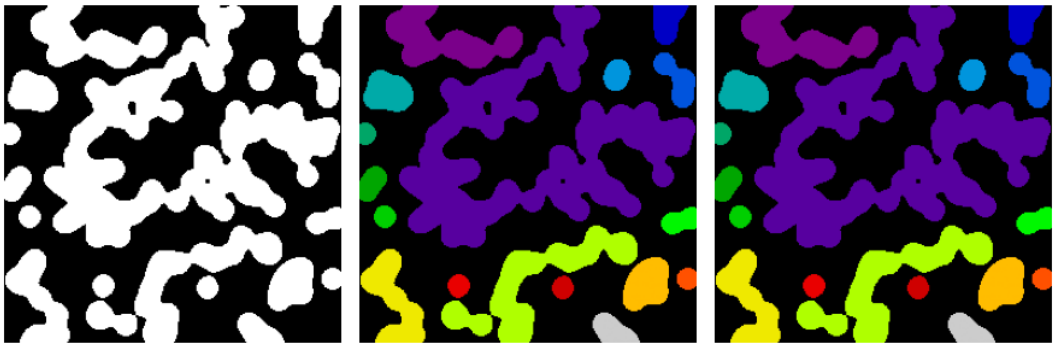
```
>>> n = 20
>>> l = 256
>>> im = np.zeros((l, l))
>>> rng = np.random.default_rng()
>>> points = l * rng.random((2, n ** 2))
>>> im[(points[0]).astype(int), (points[1]).astype(int)] = 1
>>> im = ski.filters.gaussian(im, sigma=l / (4. * n))
>>> blobs = im > im.mean()
```

Label all connected components:

```
>>> all_labels = ski.measure.label(blobs)
```

Label only foreground connected components:

```
>>> blobs_labels = ski.measure.label(blobs, background=0)
```



See also

`scipy.ndimage.find_objects()` is useful to return slices on object in an image.

17.6.2 Marker based methods

If you have markers inside a set of regions, you can use these to segment the regions.

Watershed segmentation

The Watershed (`skimage.segmentation.watershed()`) is a region-growing approach that fills "basins" in the image

```
>>> # Generate an initial image with two overlapping circles
>>> x, y = np.indices((80, 80))
>>> x1, y1, x2, y2 = 28, 28, 44, 52
>>> r1, r2 = 16, 20
>>> mask_circle1 = (x - x1) ** 2 + (y - y1) ** 2 < r1 ** 2
>>> mask_circle2 = (x - x2) ** 2 + (y - y2) ** 2 < r2 ** 2
>>> image = np.logical_or(mask_circle1, mask_circle2)
>>> # Now we want to separate the two objects in image
>>> # Generate the markers as local maxima of the distance
>>> # to the background
>>> import scipy as sp
>>> distance = sp.ndimage.distance_transform_edt(image)
>>> peak_idx = ski.feature.peak_local_max(
...     distance, footprint=np.ones((3, 3)), labels=image
... )
>>> peak_mask = np.zeros_like(distance, dtype=bool)
>>> peak_mask[tuple(peak_idx.T)] = True
>>> markers = ski.morphology.label(peak_mask)
>>> labels_ws = ski.segmentation.watershed(
...     -distance, markers, mask=image
... )
```

Random walker segmentation

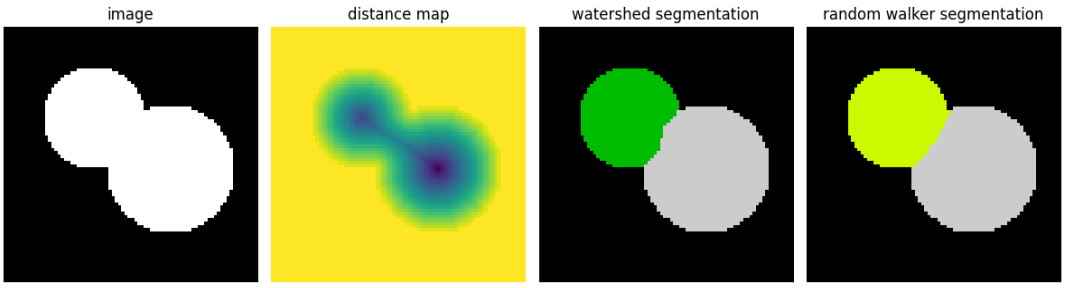
The random walker algorithm (`skimage.segmentation.random_walker()`) is similar to the Watershed, but with a more "probabilistic" approach. It is based on the idea of the diffusion of labels in the image:

```
>>> # Transform markers image so that 0-valued pixels are to
>>> # be labelled, and -1-valued pixels represent background
```

(continues on next page)

(continued from previous page)

```
>>> markers[~image] = -1
>>> labels_rw = ski.segmentation.random_walker(image, markers)
```



Postprocessing label images

`skimage` provides several utility functions that can be used on label images (ie images where different discrete values identify different regions). Functions names are often self-explaining: `skimage.segmentation.clear_border()`, `skimage.segmentation.relabel_from_one()`, `skimage.morphology.remove_small_objects()`, etc.

Exercise

- Load the `coins` image from the `data` submodule.
- Separate the coins from the background by testing several segmentation methods: Otsu thresholding, adaptive thresholding, and watershed or random walker segmentation.
- If necessary, use a postprocessing function to improve the coins / background segmentation.

17.7 Measuring regions' properties

Example: compute the size and perimeter of the two segmented regions:

```
>>> properties = ski.measure.regionprops(labels_rw)
>>> [float(prop.area) for prop in properties]
[770.0, 1168.0]
>>> [prop.perimeter for prop in properties]
[np.float64(100.91...), np.float64(126.81...)]
```

See also

for some properties, functions are available as well in `scipy.ndimage.measurements` with a different API (a list is returned).

Exercise (continued)

- Use the binary image of the coins and background from the previous exercise.
- Compute an image of labels for the different coins.
- Compute the size and eccentricity of all coins.

17.8 Data visualization and interaction

Meaningful visualizations are useful when testing a given processing pipeline.

Some image processing operations:

```
>>> coins = ski.data.coins()
>>> mask = coins > ski.filters.threshold_otsu(coins)
>>> clean_border = ski.segmentation.clear_border(mask)
```

Visualize binary result:

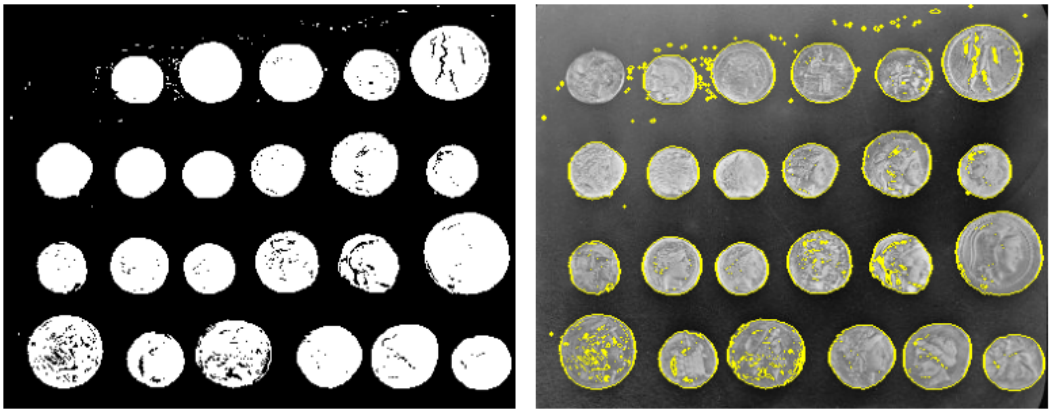
```
>>> plt.figure()
<Figure size ... with 0 Axes>
>>> plt.imshow(clean_border, cmap='gray')
<matplotlib.image.AxesImage object at 0x...>
```

Visualize contour

```
>>> plt.figure()
<Figure size ... with 0 Axes>
>>> plt.imshow(coins, cmap='gray')
<matplotlib.image.AxesImage object at 0x...>
>>> plt.contour(clean_border, [0.5])
<matplotlib.contour.QuadContourSet ...>
```

Use `skimage` dedicated utility function:

```
>>> coins_edges = ski.segmentation.mark_boundaries(
...     coins, clean_border.astype(int)
... )
```



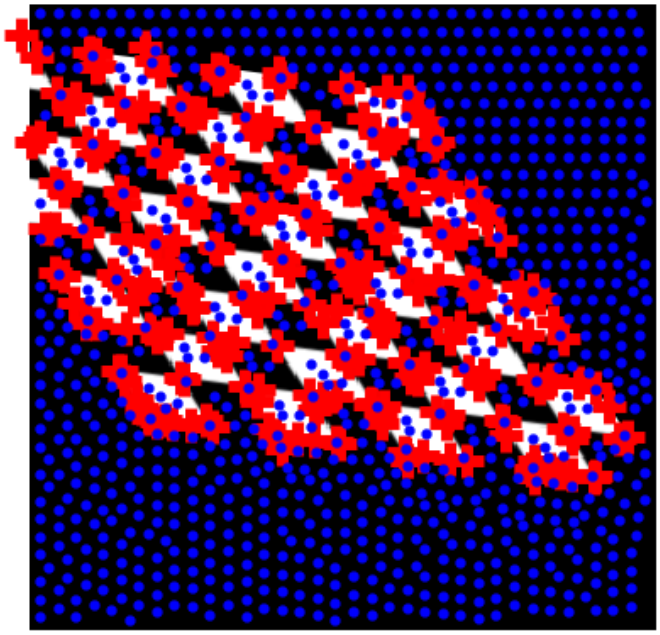
17.9 Feature extraction for computer vision

Geometric or textural descriptor can be extracted from images in order to

- classify parts of the image (e.g. sky vs. buildings)
- match parts of different images (e.g. for object detection)
- and many other applications of Computer Vision

Example: detecting corners using Harris detector

```
tform = ski.transform.AffineTransform(
    scale=(1.3, 1.1), rotation=1, shear=0.7,
    translation=(210, 50)
)
image = ski.transform.warp(
    data.checkerboard(), tform.inverse, output_shape=(350, 350)
)

coords = ski.feature.corner_peaks(
    ski.feature.corner_harris(image), min_distance=5
)
coords_subpix = ski.feature.corner_subpix(
    image, coords, window_size=13
)
```



(this example is taken from the plot_corner example in scikit-image)

Points of interest such as corners can then be used to match objects in different images, as described in the plot_matching example of scikit-image.

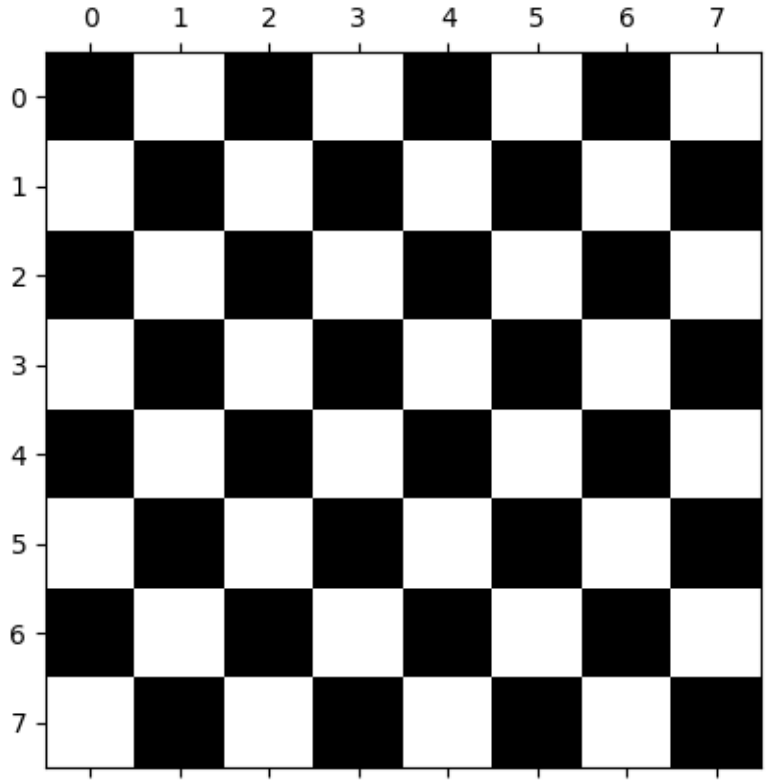
17.10 Full code examples

17.11 Examples for the scikit-image chapter

17.11.1 Creating an image

How to create an image with basic NumPy commands : `np.zeros`, slicing...

This examples show how to create a simple checkerboard.




```
import numpy as np
import matplotlib.pyplot as plt

check = np.zeros((8, 8))
check[::2, 1::2] = 1
check[1::2, ::2] = 1
plt.matshow(check, cmap="gray")
plt.show()
```

Total running time of the script: (0 minutes 0.048 seconds)

17.11.2 Displaying a simple image

Load and display an image

```
import matplotlib.pyplot as plt
from skimage import data

camera = data.camera()


plt.figure(figsize=(4, 4))
plt.imshow(camera, cmap="gray", interpolation="nearest")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.085 seconds)

17.11.3 Integers can overflow

An illustration of overflow problem arising when working with integers

```
import matplotlib.pyplot as plt
from skimage import data

camera = data.camera()
camera_multiply = 3 * camera

plt.figure(figsize=(8, 4))
plt.subplot(121)
plt.imshow(camera, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(122)
plt.imshow(camera_multiply, cmap="gray", interpolation="nearest")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.130 seconds)

17.11.4 Equalizing the histogram of an image

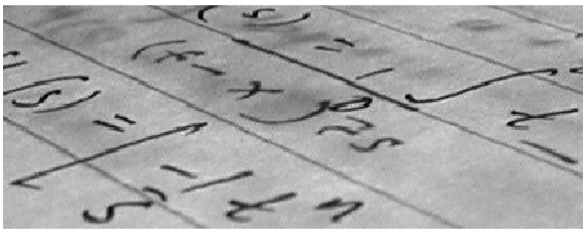
Histogram equalizing makes images have a uniform histogram.

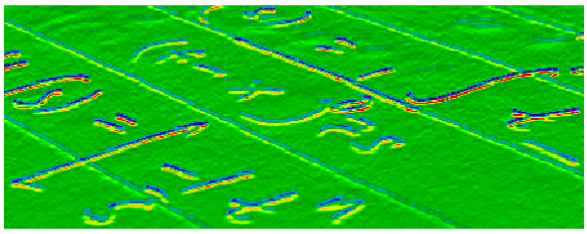
```
from skimage import data, exposure
import matplotlib.pyplot as plt

camera = data.camera()
camera_equalized = exposure.equalize_hist(camera)

plt.figure(figsize=(7, 3))

plt.subplot(121)
plt.imshow(camera, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(122)
plt.imshow(camera_equalized, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.100 seconds)

17.11.5 Computing horizontal gradients with the Sobel filter

This example illustrates the use of the horizontal Sobel filter, to compute horizontal gradients.





```
from skimage import data
from skimage import filters
import matplotlib.pyplot as plt

text = data.text()
hsobel_text = filters.sobel_h(text)
```

(continues on next page)

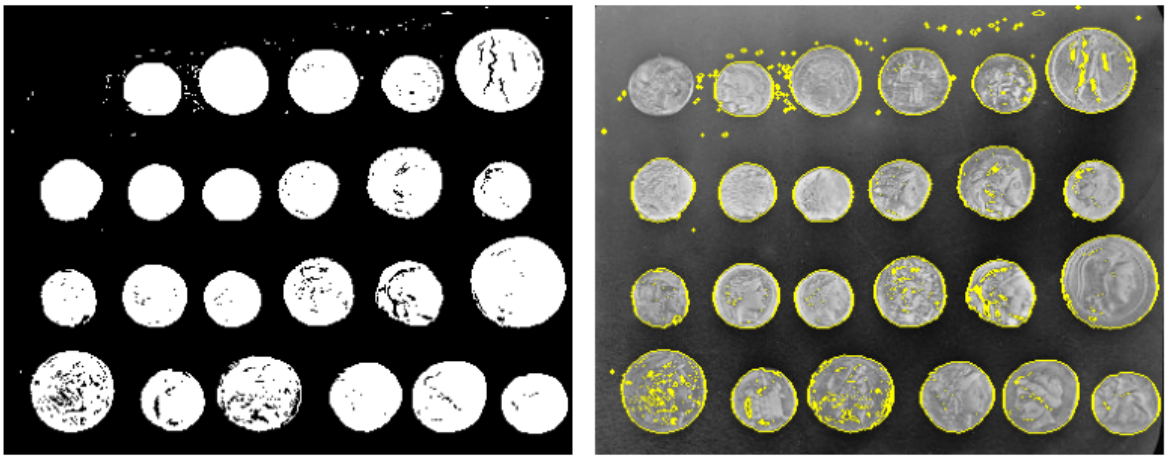
(continued from previous page)

```
plt.figure(figsize=(12, 3))

plt.subplot(121)
plt.imshow(text, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(122)
plt.imshow(hsobel_text, cmap="nipy_spectral", interpolation="nearest")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.100 seconds)

17.11.6 Segmentation contours

Visualize segmentation contours on original grayscale image.



```
from skimage import data, segmentation
from skimage import filters
import matplotlib.pyplot as plt
import numpy as np

coins = data.coins()
mask = coins > filters.threshold_otsu(coins)
clean_border = segmentation.clear_border(mask).astype(int)

coins_edges = segmentation.mark_boundaries(coins, clean_border)

plt.figure(figsize=(8, 3.5))
plt.subplot(121)
plt.imshow(clean_border, cmap="gray")
plt.axis("off")
plt.subplot(122)
plt.imshow(coins_edges)
plt.axis("off")
```

(continues on next page)

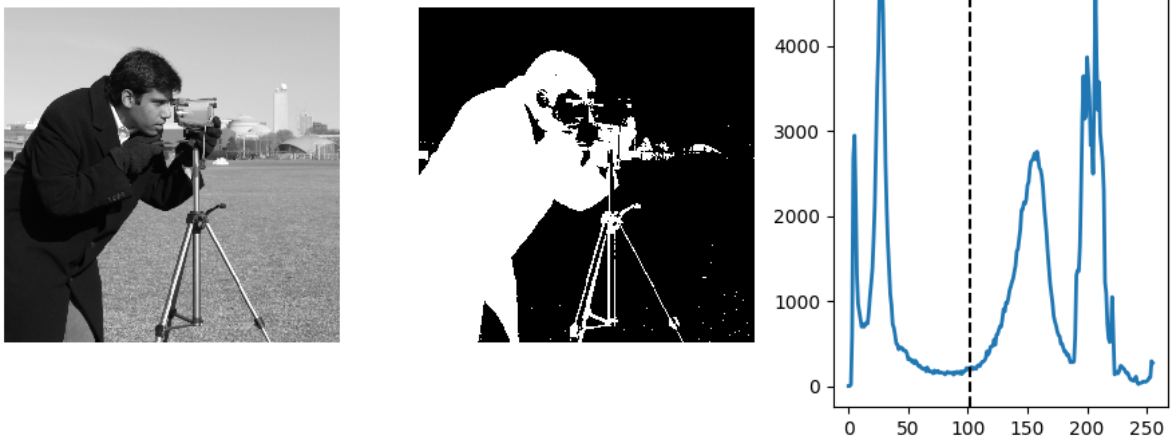
(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.096 seconds)

17.11.7 Otsu thresholding

This example illustrates automatic Otsu thresholding.

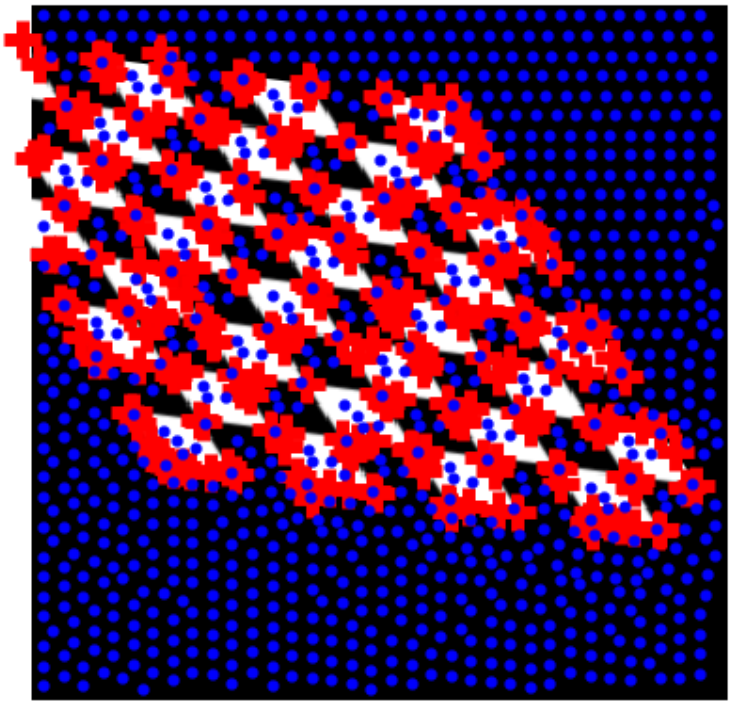



```
import matplotlib.pyplot as plt
from skimage import data
from skimage import filters
from skimage import exposure

camera = data.camera()
val = filters.threshold_otsu(camera)

hist, bins_center = exposure.histogram(camera)

plt.figure(figsize=(9, 4))
plt.subplot(131)
plt.imshow(camera, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(132)
plt.imshow(camera < val, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.subplot(133)
plt.plot(bins_center, hist, lw=2)
plt.axvline(val, color="k", ls="--")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.131 seconds)

17.11.8 Affine transform

Warping and affine transforms of images.

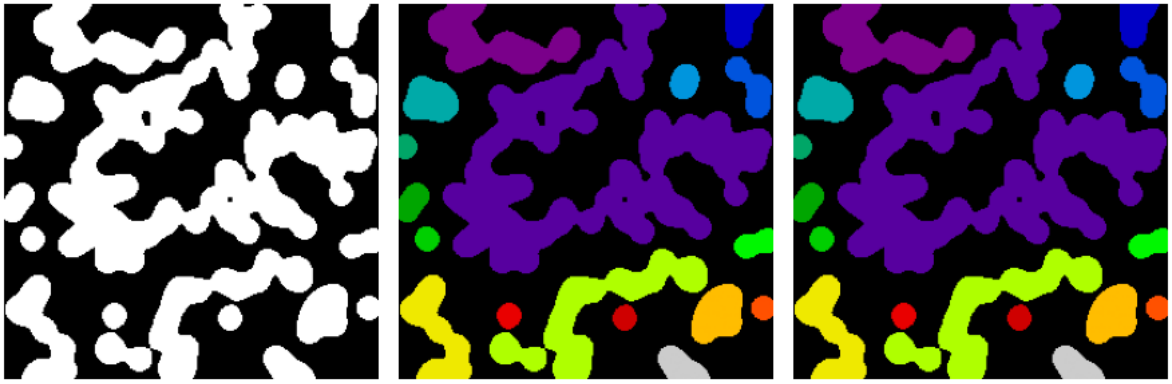


```
import matplotlib.pyplot as plt

from skimage import data
from skimage.feature import corner_harris, corner_subpix, corner_peaks
from skimage.transform import warp, AffineTransform


tform = AffineTransform(scale=(1.3, 1.1), rotation=1, shear=0.7, translation=(210,␣
↪50))
image = warp(data.checkerboard(), tform.inverse, output_shape=(350, 350))

coords = corner_peaks(corner_harris(image), min_distance=5)
coords_subpix = corner_subpix(image, coords, window_size=13)

plt.gray()
plt.imshow(image, interpolation="nearest")
plt.plot(coords_subpix[:, 1], coords_subpix[:, 0], "+r", markersize=15, mew=5)
plt.plot(coords[:, 1], coords[:, 0], ".b", markersize=7)
plt.axis("off")
plt.show()
```

Total running time of the script: (0 minutes 1.461 seconds)

17.11.9 Labelling connected components of an image

This example shows how to label connected components of a binary image, using the dedicated skimage.measure.label function.

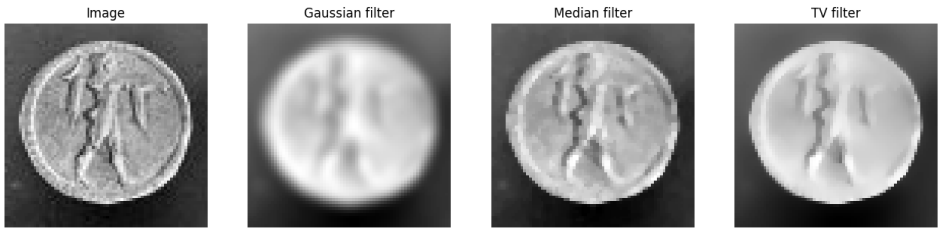


```
from skimage import measure
from skimage import filters
import matplotlib.pyplot as plt
import numpy as np

n = 12
l = 256
rng = np.random.default_rng(27446968)
im = np.zeros((l, l))
points = l * rng.random((2, n**2))
im[(points[0]).astype(int), (points[1]).astype(int)] = 1
im = filters.gaussian(im, sigma=l / (4.0 * n))
blobs = im > 0.7 * im.mean()

all_labels = measure.label(blobs)
blobs_labels = measure.label(blobs, background=0)

plt.figure(figsize=(9, 3.5))
plt.subplot(131)
plt.imshow(blobs, cmap="gray")
plt.axis("off")
plt.subplot(132)
plt.imshow(all_labels, cmap="nipy_spectral")
plt.axis("off")
plt.subplot(133)
plt.imshow(blobs_labels, cmap="nipy_spectral")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.075 seconds)

17.11.10 Various denoising filters

This example compares several denoising filters available in scikit-image: a Gaussian filter, a median filter, and total variation denoising.



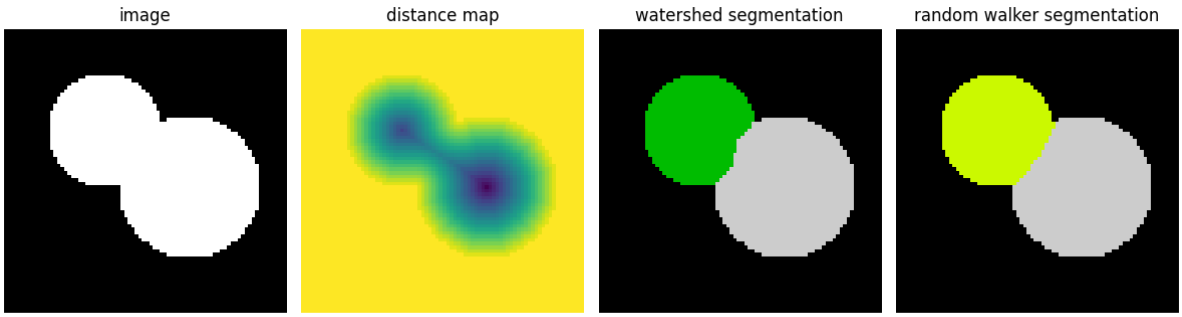

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data
from skimage import filters
from skimage import restoration

coins = data.coins()
gaussian_filter_coins = filters.gaussian(coins, sigma=2)
med_filter_coins = filters.median(coins, np.ones((3, 3)))
tv_filter_coins = restoration.denoise_tv_chambolle(coins, weight=0.1)

plt.figure(figsize=(16, 4))
plt.subplot(141)
plt.imshow(coins[10:80, 300:370], cmap="gray", interpolation="nearest")
plt.axis("off")
plt.title("Image")
plt.subplot(142)
plt.imshow(gaussian_filter_coins[10:80, 300:370], cmap="gray", interpolation="nearest
↪")
plt.axis("off")
plt.title("Gaussian filter")
plt.subplot(143)
plt.imshow(med_filter_coins[10:80, 300:370], cmap="gray", interpolation="nearest")
plt.axis("off")
plt.title("Median filter")
plt.subplot(144)
plt.imshow(tv_filter_coins[10:80, 300:370], cmap="gray", interpolation="nearest")
plt.axis("off")
plt.title("TV filter")
plt.show()
```

Total running time of the script: (0 minutes 0.159 seconds)

17.11.11 Watershed and random walker for segmentation

This example compares two segmentation methods in order to separate two connected disks: the watershed algorithm, and the random walker algorithm.

Both segmentation methods require seeds, that are pixels belonging unambigusouly to a reagion. Here, local maxima of the distance map to the background are used as seeds.




```
import numpy as np
from skimage.segmentation import watershed
from skimage.feature import peak_local_max
from skimage import measure
from skimage.segmentation import random_walker
import matplotlib.pyplot as plt
import scipy as sp

# Generate an initial image with two overlapping circles
x, y = np.indices((80, 80))
x1, y1, x2, y2 = 28, 28, 44, 52
r1, r2 = 16, 20
mask_circle1 = (x - x1) ** 2 + (y - y1) ** 2 < r1**2
mask_circle2 = (x - x2) ** 2 + (y - y2) ** 2 < r2**2
image = np.logical_or(mask_circle1, mask_circle2)
# Now we want to separate the two objects in image
# Generate the markers as local maxima of the distance
# to the background
distance = sp.ndimage.distance_transform_edt(image)
peak_idx = peak_local_max(distance, footprint=np.ones((3, 3)), labels=image)
peak_mask = np.zeros_like(distance, dtype=bool)
peak_mask[tuple(peak_idx.T)] = True
markers = measure.label(peak_mask)
labels_ws = watershed(-distance, markers, mask=image)

markers[~image] = -1
labels_rw = random_walker(image, markers)

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
plt.imshow(image, cmap="gray", interpolation="nearest")
plt.axis("off")
plt.title("image")
plt.subplot(142)
plt.imshow(-distance, interpolation="nearest")
plt.axis("off")
plt.title("distance map")
plt.subplot(143)
plt.imshow(labels_ws, cmap="nipy_spectral", interpolation="nearest")
plt.axis("off")
plt.title("watershed segmentation")
plt.subplot(144)
plt.imshow(labels_rw, cmap="nipy_spectral", interpolation="nearest")
plt.axis("off")
plt.title("random walker segmentation")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.163 seconds)

CHAPTER 18

scikit-learn: machine learning in Python

Authors: *Gael Varoquaux*

Prerequisites

- *numpy*
- *scipy*
- *matplotlib (optional)*
- *ipython (the enhancements come handy)*

Acknowledgements

This chapter is adapted from a tutorial given by Gaël Varoquaux, Jake Vanderplas, Olivier Grisel.

See also

Data science in Python

- The *Statistics in Python* chapter may also be of interest for readers looking into machine learning.

- The documentation of scikit-learn is very complete and didactic.

Chapters contents

- *Introduction: problem settings*
- *Basic principles of machine learning with scikit-learn*
- *Supervised Learning: Classification of Handwritten Digits*
- *Supervised Learning: Regression of Housing Data*
- *Measuring prediction performance*
- *Unsupervised Learning: Dimensionality Reduction and Visualization*
- *Parameter selection, Validation, and Testing*
- *Examples for the scikit-learn chapter*

18.1 Introduction: problem settings

18.1.1 What is machine learning?

Tip

Machine Learning is about building programs with **tunable parameters** that are adjusted automatically so as to improve their behavior by **adapting to previously seen data.**

Machine Learning can be considered a subfield of **Artificial Intelligence** since those algorithms can be seen as building blocks to make computers learn to behave more intelligently by somehow **generalizing** rather that just storing and retrieving data items like a database system would do.

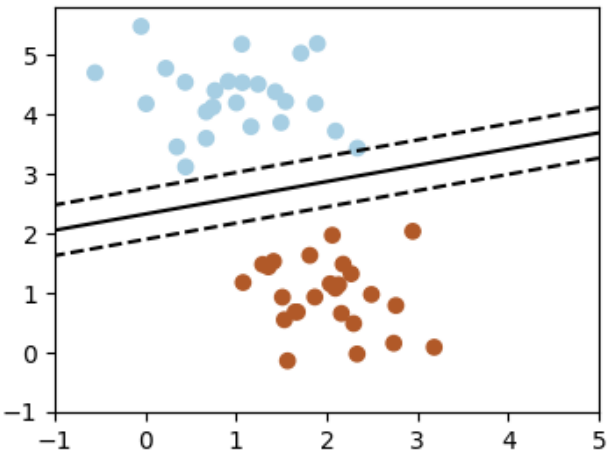


Fig. 1: A classification problem

We'll take a look at two very simple machine learning tasks here. The first is a **classification** task: the figure shows a collection of two-dimensional data, colored according to two different class labels. A classification algorithm may be used to draw a dividing boundary between the two clusters of points:

By drawing this separating line, we have learned a model which can **generalize** to new data: if you were to drop another point onto the plane which is unlabeled, this algorithm could now **predict** whether it's a blue or a red point.

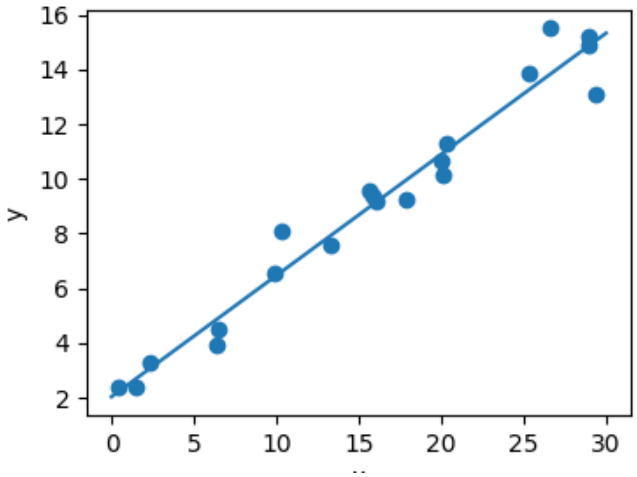


Fig. 2: A regression problem

The next simple task we'll look at is a **regression** task: a simple best-fit line to a set of data.

Again, this is an example of fitting a model to data, but our focus here is that the model can make generalizations about new data. The model has been **learned** from the training data, and can be used to predict the result of test data: here, we might be given an x-value, and the model would allow us to predict the y value.

18.1.2 Data in scikit-learn

The data matrix

Machine learning algorithms implemented in scikit-learn expect data to be stored in a **two-dimensional array or matrix**. The arrays can be either `numpy` arrays, or in some cases `scipy.sparse` matrices. The size of the array is expected to be `[n_samples, n_features]`

- **n_samples:** The number of samples: each sample is an item to process (e.g. classify). A sample can be a document, a picture, a sound, a video, an astronomical object, a row in database or CSV file, or whatever you can describe with a fixed set of quantitative traits.
- **n_features:** The number of features or distinct traits that can be used to describe each item in a quantitative manner. Features are generally real-valued, but may be boolean or discrete-valued in some cases.

Tip

The number of features must be fixed in advance. However it can be very high dimensional (e.g. millions of features) with most of them being zeros for a given sample. This is a case where `scipy.sparse` matrices can be useful, in that they are much more memory-efficient than NumPy arrays.

A Simple Example: the Iris Dataset

The application problem

As an example of a simple dataset, let us a look at the iris data stored by scikit-learn. Suppose we want to recognize species of irises. The data consists of measurements of three different species of irises:

Setosa Iris	Versicolor Iris	Virginica Iris

Quick Question:

If we want to design an algorithm to recognize iris species, what might the data be?

Remember: we need a 2D array of size `[n_samples x n_features]`.

- What would the `n_samples` refer to?
- What might the `n_features` refer to?

Remember that there must be a **fixed** number of features for each sample, and feature number `i` must be a similar kind of quantity for each sample.

Loading the Iris Data with Scikit-learn

Scikit-learn has a very straightforward set of data on these iris species. The data consist of the following:

- Features in the Iris dataset:
  - sepal length (cm)
  - sepal width (cm)
  - petal length (cm)
  - petal width (cm)
- Target classes to predict:
  - Setosa
  - Versicolour
  - Virginica

`scikit-learn` embeds a copy of the iris CSV file along with a function to load it into NumPy arrays:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
```

> **Note**
>
> **Import sklearn** Note that scikit-learn is imported as `sklearn`

The features of each sample flower are stored in the `data` attribute of the dataset:

```
>>> print(iris.data.shape)
(150, 4)
>>> n_samples, n_features = iris.data.shape
>>> print(n_samples)
150
>>> print(n_features)
4
>>> print(iris.data[0])
[5.1  3.5  1.4  0.2]
```

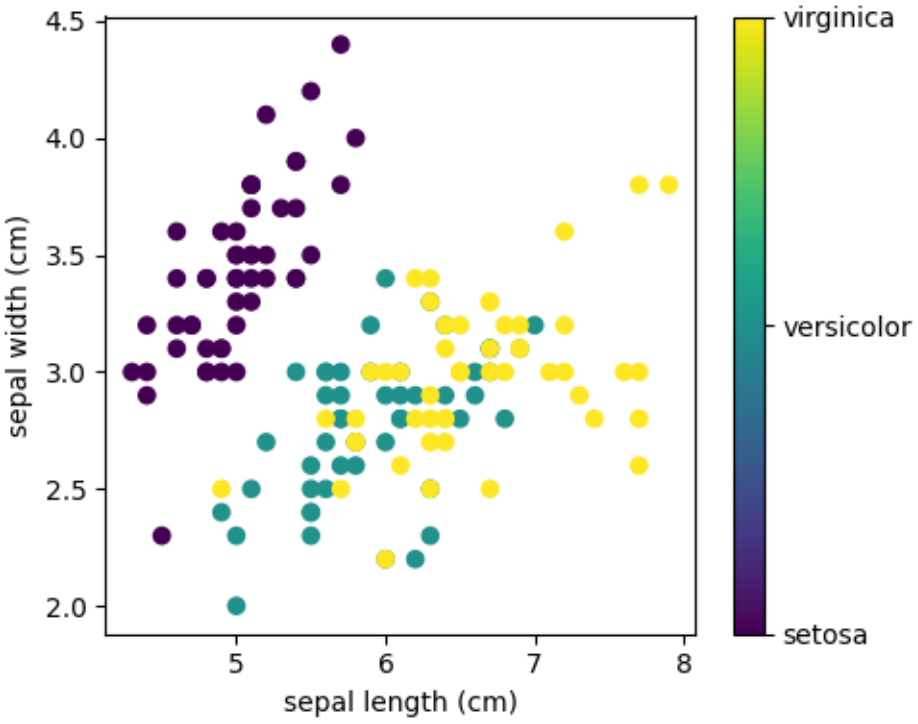
The information about the class of each sample is stored in the `target` attribute of the dataset:

```
>>> print(iris.target.shape)
(150,)
>>> print(iris.target)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

The names of the classes are stored in the last attribute, namely `target_names`:

```
>>> print(iris.target_names)
['setosa' 'versicolor' 'virginica']
```

This data is four-dimensional, but we can visualize two of the dimensions at a time using a scatter plot:



Exercise:

Can you choose 2 features to find a plot where it is easier to separate the different classes of irises?

Hint: click on the figure above to see the code that generates it, and modify this code.

18.2 Basic principles of machine learning with scikit-learn

18.2.1 Introducing the scikit-learn estimator object

Every algorithm is exposed in scikit-learn via an ''Estimator" object. For instance a linear regression is: `sklearn.linear_model.LinearRegression`

```
>>> from sklearn.linear_model import LinearRegression
```

Estimator parameters: All the parameters of an estimator can be set when it is instantiated:

```
>>> model = LinearRegression(n_jobs=1)
>>> print(model)
LinearRegression(n_jobs=1)
```

Fitting on data

Let's create some simple data with *numpy*:

```
>>> import numpy as np
>>> x = np.array([0, 1, 2])
>>> y = np.array([0, 1, 2])

>>> X = x[:, np.newaxis] # The input data for sklearn is 2D: (samples == 3 x features
↪== 1)
>>> X
array([[0],
       [1],
       [2]])

>>> model.fit(X, y)
LinearRegression(n_jobs=1)
```

Estimated parameters: When data is fitted with an estimator, parameters are estimated from the data at hand. All the estimated parameters are attributes of the estimator object ending by an underscore:

```
>>> model.coef_
array([1.])
```

18.2.2 Supervised Learning: Classification and regression

In **Supervised Learning**, we have a dataset consisting of both features and labels. The task is to construct an estimator which is able to predict the label of an object given the set of features. A relatively simple example is predicting the species of iris given a set of measurements of its flower. This is a relatively simple task. Some more complicated examples are:

- given a multicolor image of an object through a telescope, determine whether that object is a star, a quasar, or a galaxy.
- given a photograph of a person, identify the person in the photo.

- given a list of movies a person has watched and their personal rating of the movie, recommend a list of movies they would like (So-called *recommender systems*: a famous example is the Netflix Prize).

Tip

What these tasks have in common is that there is one or more unknown quantities associated with the object which needs to be determined from other observed quantities.

Supervised learning is further broken down into two categories, **classification** and **regression**. In classification, the label is discrete, while in regression, the label is continuous. For example, in astronomy, the task of determining whether an object is a star, a galaxy, or a quasar is a classification problem: the label is from three distinct categories. On the other hand, we might wish to estimate the age of an object based on such observations: this would be a regression problem, because the label (age) is a continuous quantity.

Classification: K nearest neighbors (kNN) is one of the simplest learning strategies: given a new, unknown observation, look up in your reference database which ones have the closest features and assign the predominant class. Let's try it out on our iris classification problem:

```
from sklearn import neighbors, datasets
iris = datasets.load_iris()
X, y = iris.data, iris.target
knn = neighbors.KNeighborsClassifier(n_neighbors=1)
knn.fit(X, y)
# What kind of iris has 3cm x 5cm sepal and 4cm x 2cm petal?
print(iris.target_names[knn.predict([[3, 5, 4, 2]])])
```

Regression: The simplest possible regression setting is the linear regression one:

```
from sklearn.linear_model import LinearRegression

# x from 0 to 30
rng = np.random.default_rng()
x = 30 * rng.random((20, 1))

# y = a*x + b with noise
y = 0.5 * x + 1.0 + rng.normal(size=x.shape)

# create a linear regression model
model = LinearRegression()
model.fit(x, y)

# predict y from the data
x_new = np.linspace(0, 30, 100)
y_new = model.predict(x_new[:, np.newaxis])
```

18.2.3 A recap on Scikit-learn's estimator interface

Scikit-learn strives to have a uniform interface across all methods, and we'll see examples of these below. Given a scikit-learn *estimator* object named `model`, the following methods are available:

In all Estimators

- `model.fit()` : fit training data. For supervised learning applications, this accepts two arguments: the data `X` and the labels `y` (e.g. `model.fit(X, y)`). For unsupervised learning applications, this accepts only a single argument, the data `X` (e.g. `model.fit(X)`).

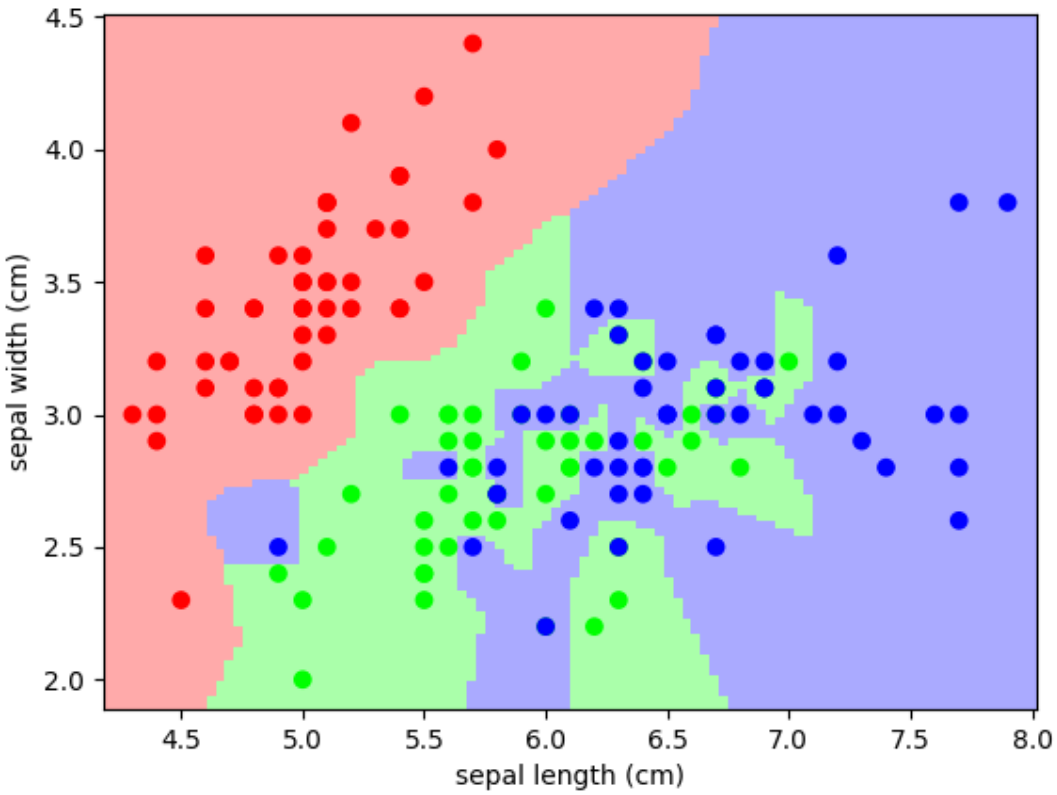


Fig. 3: A plot of the sepal space and the prediction of the KNN

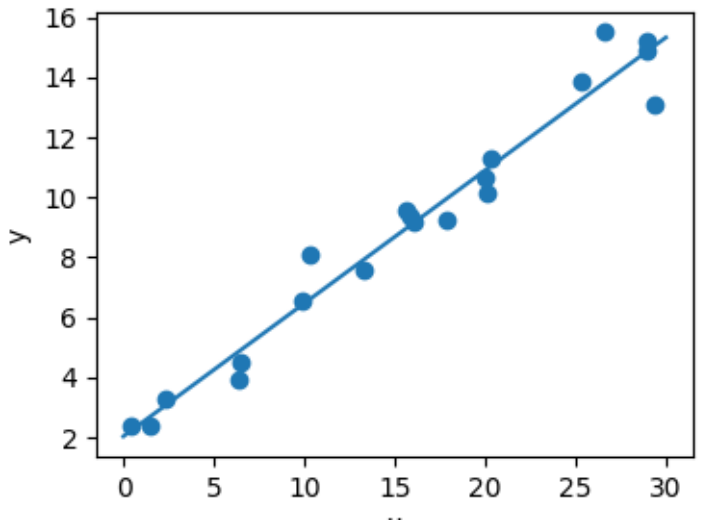


Fig. 4: A plot of a simple linear regression.

In supervised estimators

- `model.predict()` : given a trained model, predict the label of a new set of data. This method accepts one argument, the new data `X_new` (e.g. `model.predict(X_new)`), and returns the learned label for each object in the array.
- `model.predict_proba()` : For classification problems, some estimators also provide this method, which returns the probability that a new observation has each categorical label. In this case, the label with the highest probability is returned by `model.predict()`.
- `model.score()` : for classification or regression problems, most (all?) estimators implement a score method. Scores are between 0 and 1, with a larger score indicating a better fit.

In unsupervised estimators

- `model.transform()` : given an unsupervised model, transform new data into the new basis. This also accepts one argument `X_new`, and returns the new representation of the data based on the unsupervised model.
- `model.fit_transform()` : some estimators implement this method, which more efficiently performs a fit and a transform on the same input data.

18.2.4 Regularization: what it is and why it is necessary

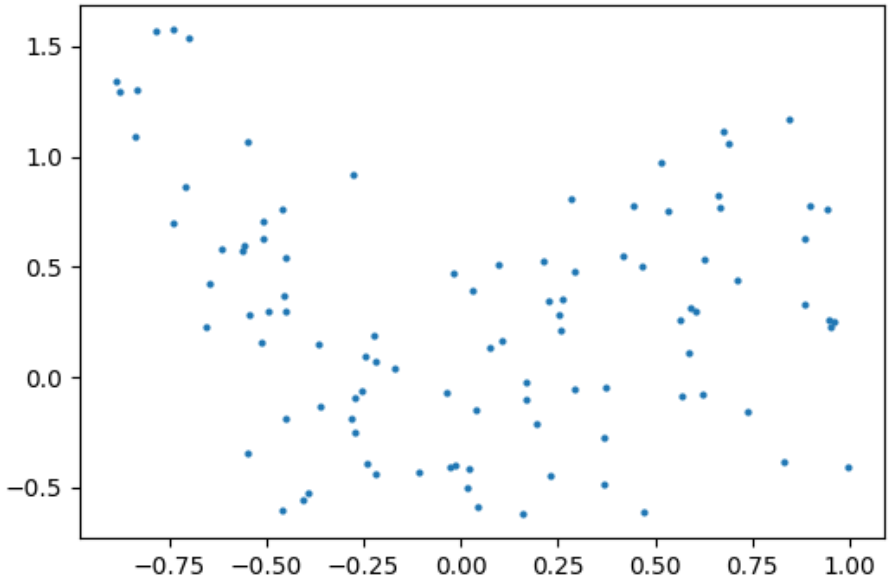
Preferring simpler models

Train errors Suppose you are using a 1-nearest neighbor estimator. How many errors do you expect on your train set?

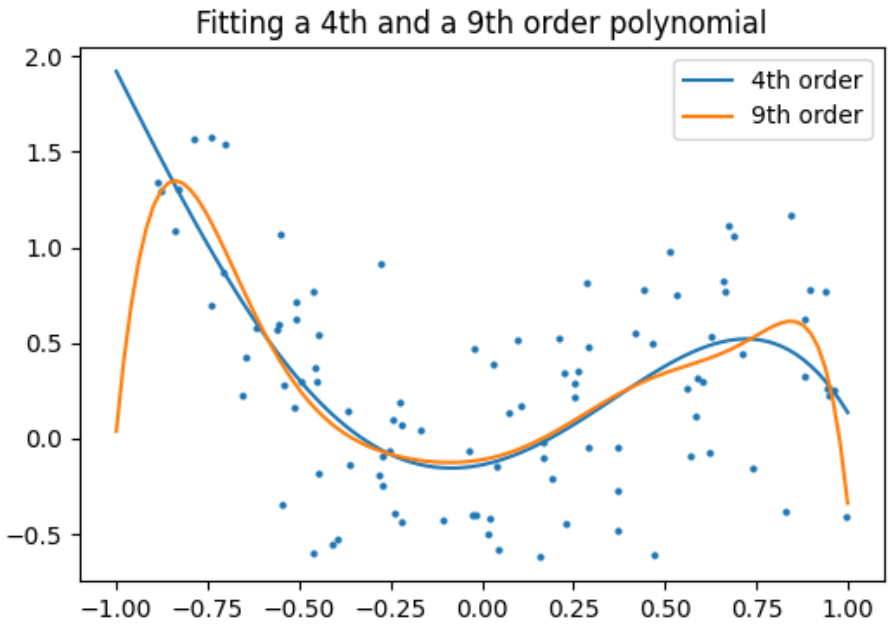
- Train set error is not a good measurement of prediction performance. You need to leave out a test set.
- In general, we should accept errors on the train set.

An example of regularization The core idea behind regularization is that we are going to prefer models that are simpler, for a certain definition of ''simpler", even if they lead to more errors on the train set.

As an example, let's generate with a 9th order polynomial, with noise:

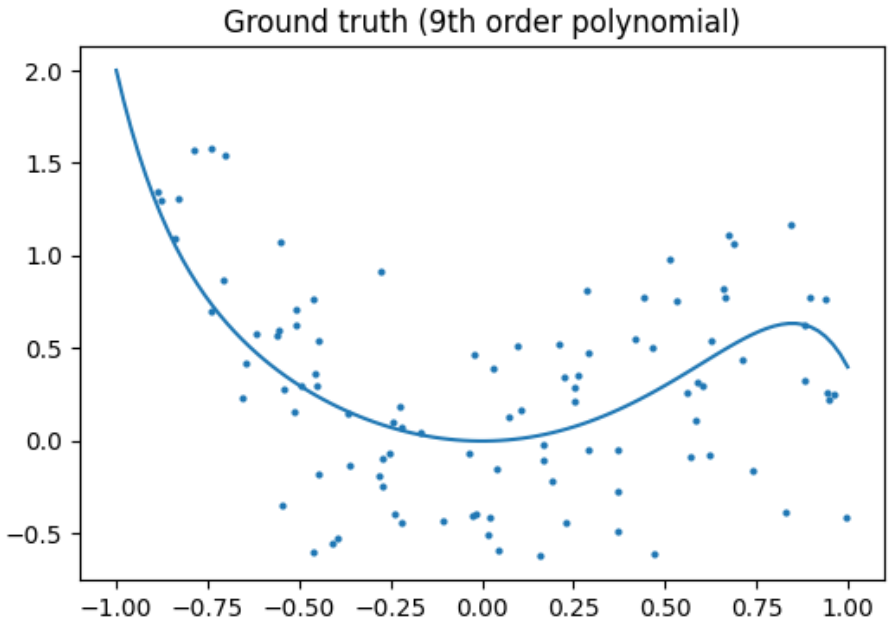


And now, let's fit a 4th order and a 9th order polynomial to the data.



With your naked eyes, which model do you prefer, the 4th order one, or the 9th order one?

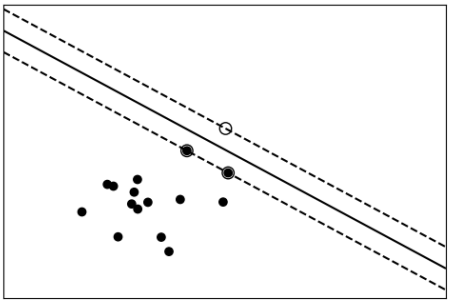
Let's look at the ground truth:



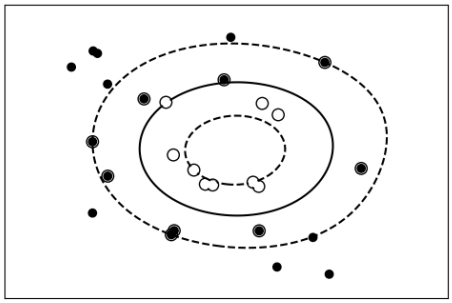
Tip

Regularization is ubiquitous in machine learning. Most scikit-learn estimators have a parameter to tune the amount of regularization. For instance, with k-NN, it is 'k', the number of nearest neighbors used to make the decision. k=1 amounts to no regularization: 0 error on the training set, whereas large k will push toward smoother decision boundaries in the feature space.

Simple versus complex models for classification



A linear separation



A non-linear separation

Tip

For classification models, the decision boundary, that separates the class expresses the complexity of the model. For instance, a linear model, that makes a decision based on a linear combination of features, is more complex than a non-linear one.

18.3 Supervised Learning: Classification of Handwritten Digits

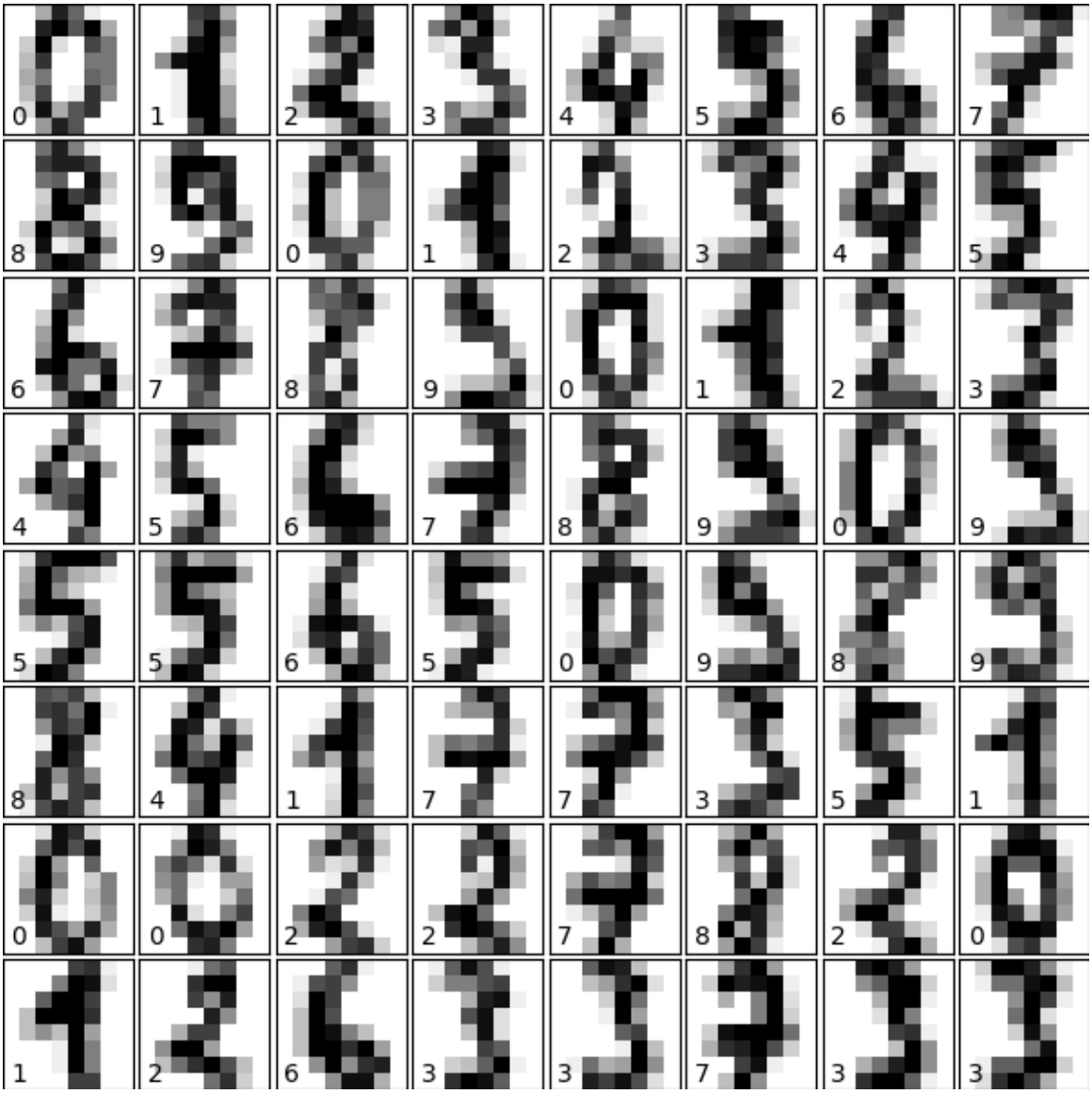
18.3.1 The nature of the data

Code and notebook

Python code and Jupyter notebook for this section are found *here*

In this section we'll apply scikit-learn to the classification of handwritten digits. This will go a bit beyond the iris classification we saw before: we'll discuss some of the metrics which can be used in evaluating the effectiveness of a classification model.

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
```



Let us visualize the data and remind us what we're looking at (click on the figure for the full code):

```
# plot the digits: each image is 8x8 pixels
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')
```

18.3.2 Visualizing the Data on its principal components

A good first-step for many problems is to visualize the data using a *Dimensionality Reduction* technique. We'll start with the most straightforward one, Principal Component Analysis (PCA).

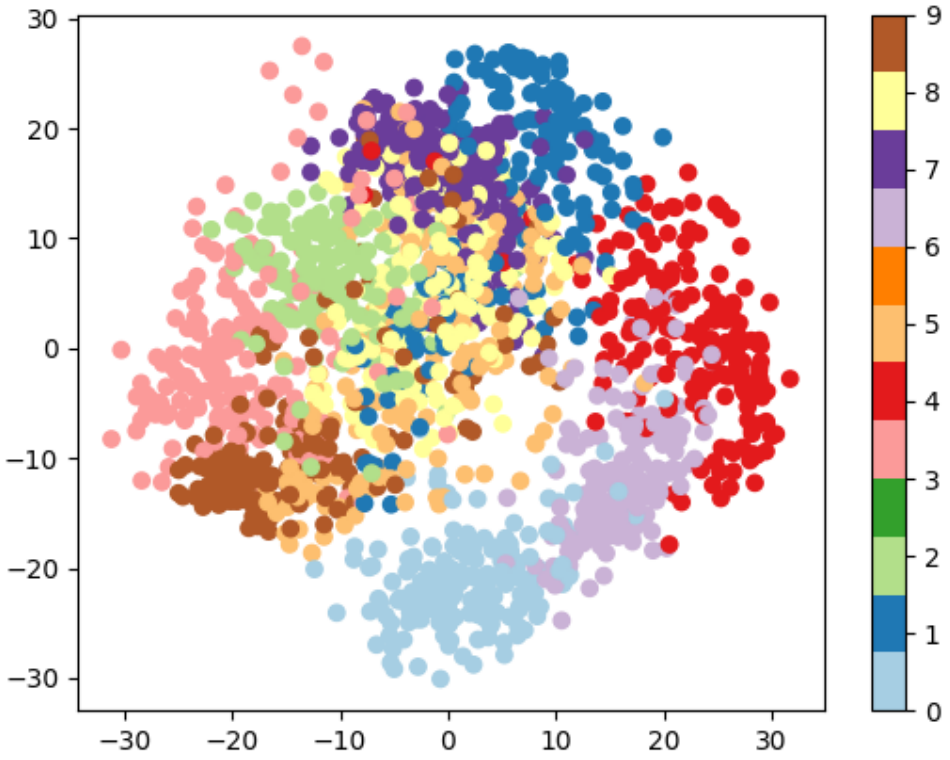
PCA seeks orthogonal linear combinations of the features which show the greatest variance, and as such, can help give you a good idea of the structure of the data set.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> proj = pca.fit_transform(digits.data)
>>> plt.scatter(proj[:, 0], proj[:, 1], c=digits.target)
<matplotlib.collections.PathCollection object at ...>
>>> plt.colorbar()
```

(continues on next page)

(continued from previous page)

```
<matplotlib.colorbar.Colorbar object at ...>
```



Question

Given these projections of the data, which numbers do you think a classifier might have trouble distinguishing?

18.3.3 Gaussian Naive Bayes Classification

For most classification problems, it's nice to have a simple, fast method to provide a quick baseline classification. If the simple and fast method is sufficient, then we don't have to waste CPU cycles on more complex models. If not, we can use the results of the simple method to give us clues about our data.

One good method to keep in mind is Gaussian Naive Bayes (`sklearn.naive_bayes.GaussianNB`).

Old scikit-learn versions

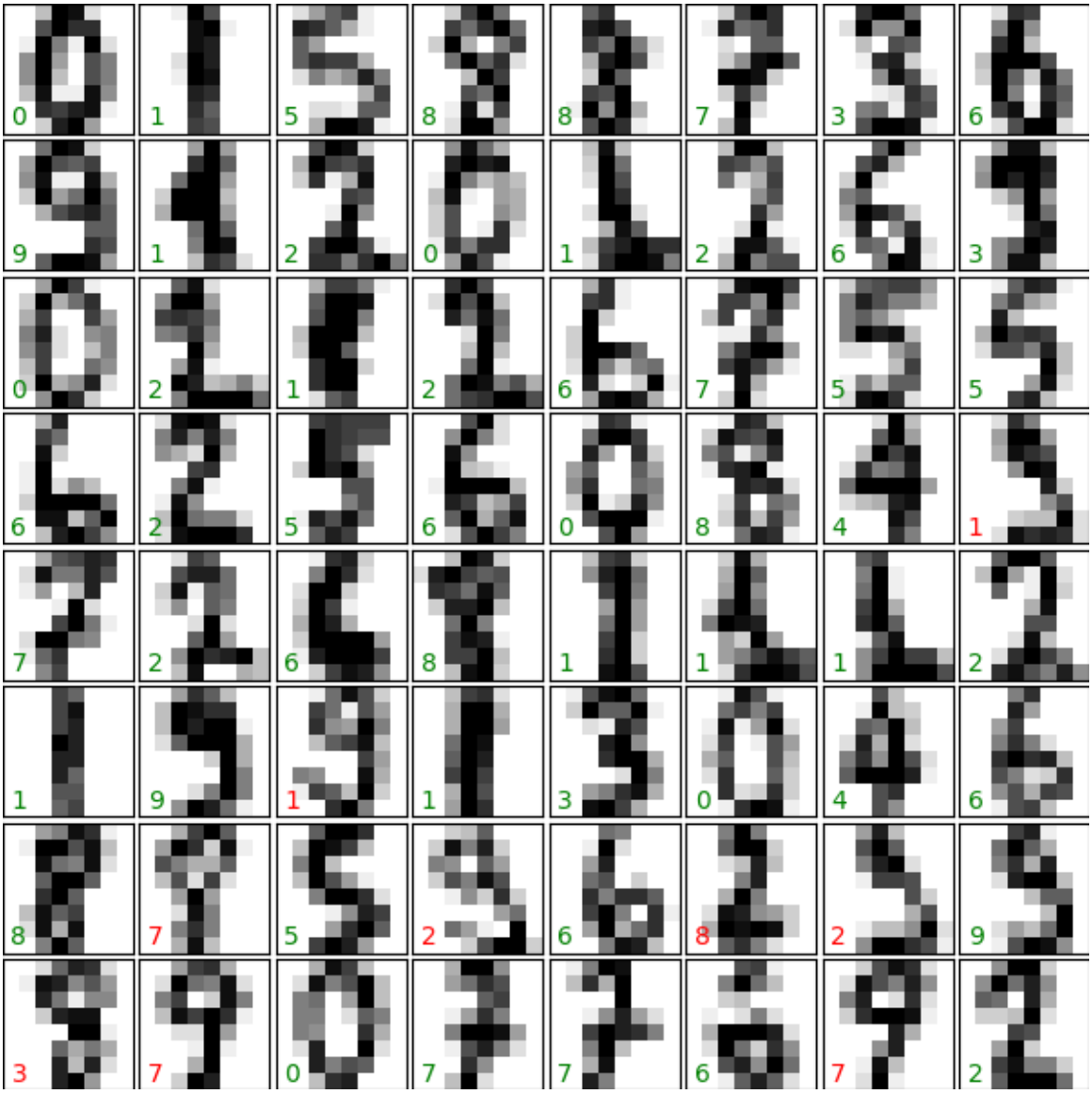
`train_test_split()` is imported from `sklearn.cross_validation`

Tip

Gaussian Naive Bayes fits a Gaussian distribution to each training label independently on each feature, and uses this to quickly give a rough classification. It is generally not sufficiently accurate for real-world data, but can perform surprisingly well, for instance on text data.

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.model_selection import train_test_split

>>> # split the data into training and validation sets
>>> X_train, X_test, y_train, y_test = train_test_split(
...    digits.data, digits.target, random_state=42)

>>> # train the model
>>> clf = GaussianNB()
>>> clf.fit(X_train, y_train)
GaussianNB()

>>> # use the model to predict the labels of the test data
>>> predicted = clf.predict(X_test)
>>> expected = y_test
>>> print(predicted)
[6 9 3 7 2 2 5 8 5 2 1 1 7 0 4 8 3 7 8 8 4 3 9 7 5 6 3 5 6 3...]
>>> print(expected)
[6 9 3 7 2 1 5 2 5 2 1 9 4 0 4 2 3 7 8 8 4 3 9 7 5 6 3 5 6 3...]
```

As above, we plot the digits with the predicted labels to get an idea of how well the classification is working.



Question

Why did we split the data into training and validation sets?

18.3.4 Quantitative Measurement of Performance

We'd like to measure the performance of our estimator without having to resort to plotting examples. A simple method might be to simply compare the number of matches:

```
>>> matches = (predicted == expected)
>>> print(matches.sum())
385
>>> print(len(matches))
450
>>> matches.sum() / float(len(matches))
np.float64(0.8555...)
```

We see that more than 80% of the 450 predictions match the input. But there are other more sophisticated metrics that can be used to judge the performance of a classifier: several are available in the `sklearn.metrics` submodule.

One of the most useful metrics is the `classification_report`, which combines several measures and prints a table with the results:

```
>>> from sklearn import metrics
>>> print(metrics.classification_report(expected, predicted))
              precision    recall  f1-score   support

           0       1.00      0.95      0.98        43
           1       0.85      0.78      0.82        37
           2       0.85      0.61      0.71        38
           3       0.97      0.83      0.89        46
           4       0.98      0.84      0.90        55
           5       0.90      0.95      0.93        59
           6       0.90      0.96      0.92        45
           7       0.71      0.98      0.82        41
           8       0.60      0.89      0.72        38
           9       0.90      0.73      0.80        48

    accuracy                           0.86       450
   macro avg       0.87      0.85      0.85       450
weighted avg       0.88      0.86      0.86       450
```

Another enlightening metric for this sort of multi-label classification is a *confusion matrix*: it helps us visualize which labels are being interchanged in the classification errors:

```
>>> print(metrics.confusion_matrix(expected, predicted))
[[41  0  0  0  0  1  0  1  0  0]
 [ 0 29  2  0  0  0  0  0  4  2]
 [ 0  2 23  0  0  0  1  0 12  0]
 [ 0  0  1 38  0  1  0  0  5  1]
 [ 0  0  0  0 46  0  2  7  0  0]
 [ 0  0  0  0  0 56  1  1  0  1]
 [ 0  0  0  0  1  1 43  0  0  0]
 [ 0  0  0  0  0  1  0 40  0  0]
 [ 0  2  0  0  0  0  0  2 34  0]
 [ 0  1  1  1  0  2  1  5  2 35]]
```

We see here that in particular, the numbers 1, 2, 3, and 9 are often being labeled 8.

18.4 Supervised Learning: Regression of Housing Data

Here we'll do a short example of a regression problem: learning a continuous value from a set of features.

18.4.1 A quick look at the data

Code and notebook

Python code and Jupyter notebook for this section are found *here*

We'll use the California house prices set, available in scikit-learn. This records measurements of 8 attributes of housing markets in California, as well as the median price. The question is: can you predict the price of a new market given its attributes?:

```
>>> from sklearn.datasets import fetch_california_housing
>>> data = fetch_california_housing(as_frame=True)
>>> print(data.data.shape)
```

(continues on next page)

(continued from previous page)

```
(20640, 8)
>>> print(data.target.shape)
(20640,)
```

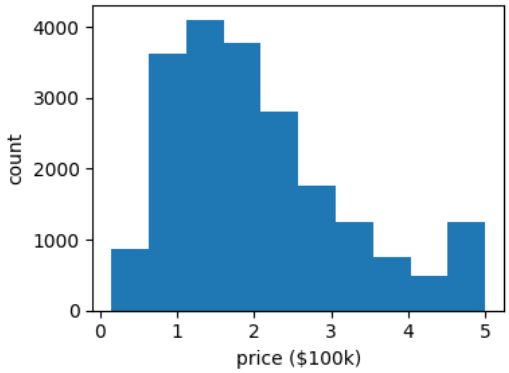
We can see that there are just over 20000 data points.

The `DESCR` variable has a long description of the dataset:

```
>>> print(data.DESCR)
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

:Number of Instances: 20640

:Number of Attributes: 8 numeric, predictive attributes and the target

:Attribute Information:
    - MedInc        median income in block group
    - HouseAge      median house age in block group
    - AveRooms      average number of rooms per household
    - AveBedrms     average number of bedrooms per household
    - Population    block group population
    - AveOccup      average number of household members
    - Latitude      block group latitude
    - Longitude     block group longitude

:Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the U.S.
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, these
columns may take surprisingly large values for block groups with few households
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. rubric:: References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
  Statistics and Probability Letters, 33 (1997) 291-297
```

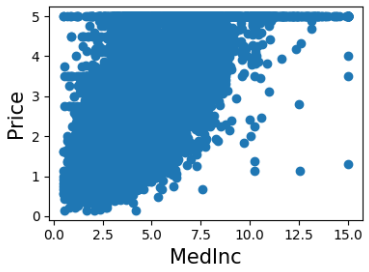
It often helps to quickly visualize pieces of the data using histograms, scatter plots, or other plot types. With matplotlib, let us show a histogram of the target values: the median price in each neighborhood:

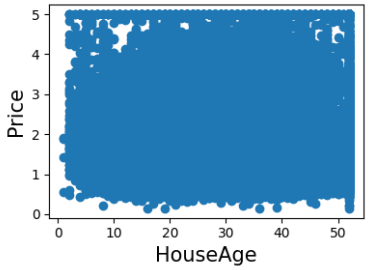
```
>>> plt.hist(data.target)
(array([...
```

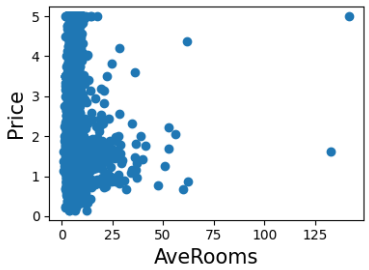


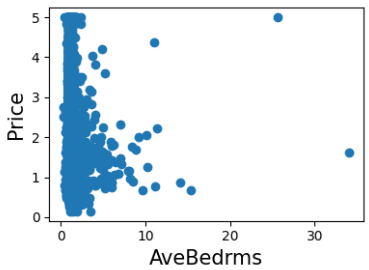
Let's have a quick look to see if some features are more relevant than others for our problem:

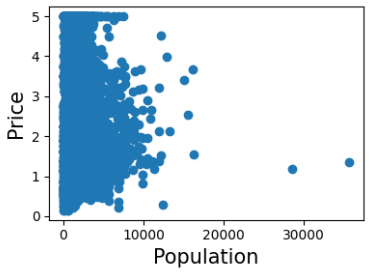
```
>>> for index, feature_name in enumerate(data.feature_names):
...     plt.figure()
...     plt.scatter(data.data[feature_name], data.target)
<Figure size...
```

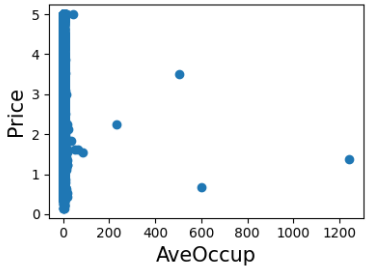


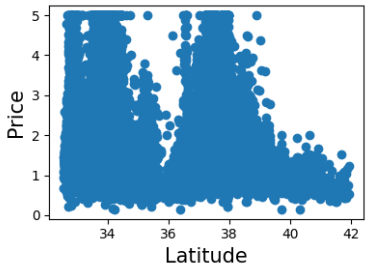


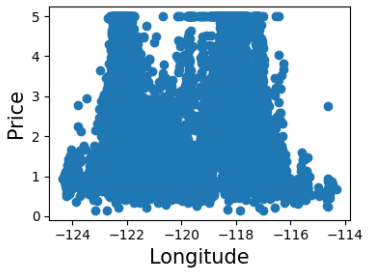












This is a manual version of a technique called **feature selection**.

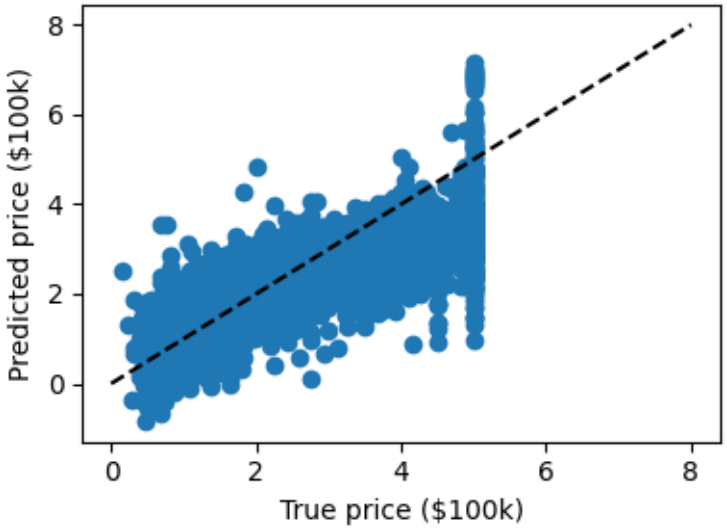
Tip

Sometimes, in Machine Learning it is useful to use feature selection to decide which features are the most useful for a particular problem. Automated methods exist which quantify this sort of exercise of choosing the most informative features.

18.4.2 Predicting Home Prices: a Simple Linear Regression

Now we'll use `scikit-learn` to perform a simple linear regression on the housing data. There are many possibilities of regressors to use. A particularly simple one is `LinearRegression`: this is basically a wrapper around an ordinary least squares calculation.

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(data.data, data.target)
>>> from sklearn.linear_model import LinearRegression
>>> clf = LinearRegression()
>>> clf.fit(X_train, y_train)
LinearRegression()
>>> predicted = clf.predict(X_test)
>>> expected = y_test
>>> print("RMS: %s" % np.sqrt(np.mean((predicted - expected) ** 2)))
RMS: 0.7...
```



We can plot the error: expected as a function of predicted:

```
>>> plt.scatter(expected, predicted)
<matplotlib.collections.PathCollection object at ...>
```

Tip

The prediction at least correlates with the true price, though there are clearly some biases. We could imagine evaluating the performance of the regressor by, say, computing the RMS residuals between the true and predicted price. There are some subtleties in this, however, which we'll cover in a later section.

Exercise: Gradient Boosting Tree Regression

There are many other types of regressors available in scikit-learn: we'll try a more powerful one here.

Use the GradientBoostingRegressor class to fit the housing data.

hint You can copy and paste some of the above code, replacing `LinearRegression` with `GradientBoostingRegressor`:

```
from sklearn.ensemble import GradientBoostingRegressor
# Instantiate the model, fit the results, and scatter in vs. out
```

Solution The solution is found in *the code of this chapter*

18.5 Measuring prediction performance

18.5.1 A quick test on the K-neighbors classifier

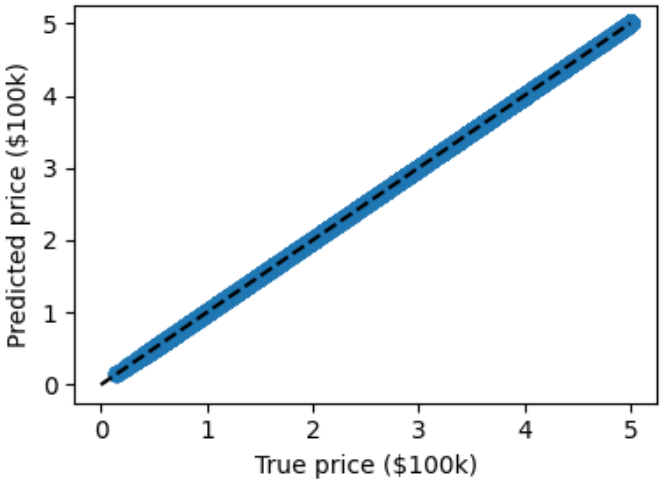
Here we'll continue to look at the digits data, but we'll switch to the K-Neighbors classifier. The K-neighbors classifier is an instance-based classifier. The K-neighbors classifier predicts the label of an unknown point based on the labels of the K nearest points in the parameter space.

```
>>> # Get the data
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> X = digits.data
>>> y = digits.target

>>> # Instantiate and train the classifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> clf = KNeighborsClassifier(n_neighbors=1)
>>> clf.fit(X, y)
KNeighborsClassifier(...)

>>> # Check the results using metrics
>>> from sklearn import metrics
>>> y_pred = clf.predict(X)

>>> print(metrics.confusion_matrix(y_pred, y))
[[178   0   0   0   0   0   0   0   0   0]
 [  0 182   0   0   0   0   0   0   0   0]
 [  0   0 177   0   0   0   0   0   0   0]
 [  0   0   0 183   0   0   0   0   0   0]
 [  0   0   0   0 181   0   0   0   0   0]
 [  0   0   0   0   0 182   0   0   0   0]
 [  0   0   0   0   0   0 181   0   0   0]
 [  0   0   0   0   0   0   0 179   0   0]
 [  0   0   0   0   0   0   0   0 174   0]
 [  0   0   0   0   0   0   0   0   0 180]]
```

Apparently, we've found a perfect classifier! But this is misleading for the reasons we saw before: the classifier essentially "memorizes" all the samples it has already seen. To really test how well this algorithm does, we need to try some samples it *hasn't* yet seen.

This problem also occurs with regression models. In the following we fit an other instance-based model named "decision tree" to the California Housing price dataset we introduced previously:

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.tree import DecisionTreeRegressor

>>> data = fetch_california_housing(as_frame=True)
>>> clf = DecisionTreeRegressor().fit(data.data, data.target)
>>> predicted = clf.predict(data.data)
>>> expected = data.target

>>> plt.scatter(expected, predicted)
<matplotlib.collections.PathCollection object at ...>
>>> plt.plot([0, 50], [0, 50], '--k')
[<matplotlib.lines.Line2D object at ...]
```



Here again the predictions are seemingly perfect as the model was able to perfectly memorize the training set.

Warning

Performance on test set

Performance on test set does not measure overfit (as described above)

18.5.2 A correct approach: Using a validation set

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data.

To avoid over-fitting, we have to define two different sets:

- a training set X_train, y_train which is used for learning the parameters of a predictive model
- a testing set X_test, y_test which is used for evaluating the fitted predictive model

In scikit-learn such a random split can be quickly computed with the `train_test_split()` function:

```
>>> from sklearn import model_selection
>>> X = digits.data
>>> y = digits.target

>>> X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
...                                         test_size=0.25, random_state=0)

>>> print("%r, %r, %r" % (X.shape, X_train.shape, X_test.shape))
(1797, 64), (1347, 64), (450, 64)
```

Now we train on the training data, and test on the testing data:

```
>>> clf = KNeighborsClassifier(n_neighbors=1).fit(X_train, y_train)
>>> y_pred = clf.predict(X_test)

>>> print(metrics.confusion_matrix(y_test, y_pred))
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 43  0  0  0  0  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
 [ 0  0 43  1  0  0  0  0  0  0]
 [ 0  0  0 45  0  0  0  0  0  0]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  0  0  0  0 47  0  0  0  1]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  0  0  0  0  0  0 48  0  0]
 [ 0  0  0  0  0  0  0  0 48  0]
 [ 0  0  0  1  0  1  0  0  0 45]]
>>> print(metrics.classification_report(y_test, y_pred))
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       1.00      1.00      1.00        43
           2       1.00      0.98      0.99        44
           3       0.96      1.00      0.98        45
           4       1.00      1.00      1.00        38
           5       0.98      0.98      0.98        48
           6       1.00      1.00      1.00        52
           7       1.00      1.00      1.00        48
           8       1.00      1.00      1.00        48
           9       0.98      0.96      0.97        47

    accuracy                           0.99       450
   macro avg       0.99      0.99      0.99       450
weighted avg       0.99      0.99      0.99       450
```

The averaged f1-score is often used as a convenient measure of the overall performance of an algorithm. It appears in the bottom row of the classification report; it can also be accessed directly:

```
>>> metrics.f1_score(y_test, y_pred, average="macro")
0.991367...
```

The over-fitting we saw previously can be quantified by computing the f1-score on the training data itself:

```
>>> metrics.f1_score(y_train, clf.predict(X_train), average="macro")
1.0
```

Note

Regression metrics In the case of regression models, we need to use different metrics, such as explained variance.

18.5.3 Model Selection via Validation

Tip

We have applied Gaussian Naives, support vectors machines, and K-nearest neighbors classifiers to the digits dataset. Now that we have these validation tools in place, we can ask quantitatively which of the three estimators works best for this dataset.

- With the default hyper-parameters for each estimator, which gives the best f1 score on the **validation set**? Recall that hyperparameters are the parameters set when you instantiate the classifier: for example, the `n_neighbors` in `clf = KNeighborsClassifier(n_neighbors=1)`

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.svm import LinearSVC

>>> X = digits.data
>>> y = digits.target
>>> X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
...                                test_size=0.25, random_state=0)

>>> for Model in [GaussianNB(), KNeighborsClassifier(), LinearSVC(dual=False)]:
...     clf = Model.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print('%s: %s' %
...           (Model.__class__.__name__, metrics.f1_score(y_test, y_pred,␣
↪average="macro")))
GaussianNB: 0.8...
KNeighborsClassifier: 0.9...
LinearSVC: 0.9...
```

- For each classifier, which value for the hyperparameters gives the best results for the digits data? For `LinearSVC`, use `loss='l2'` and `loss='l1'`. For `KNeighborsClassifier` we use `n_neighbors` between 1 and 10. Note that `GaussianNB` does not have any adjustable hyperparameters.

```
LinearSVC(loss='l1'): 0.930570687535
LinearSVC(loss='l2'): 0.933068826918
--------------------
KNeighbors(n_neighbors=1): 0.991367521884
KNeighbors(n_neighbors=2): 0.984844206884
KNeighbors(n_neighbors=3): 0.986775344954
KNeighbors(n_neighbors=4): 0.980371905382
KNeighbors(n_neighbors=5): 0.980456280495
KNeighbors(n_neighbors=6): 0.975792419414
KNeighbors(n_neighbors=7): 0.978064579214
KNeighbors(n_neighbors=8): 0.978064579214
KNeighbors(n_neighbors=9): 0.978064579214
KNeighbors(n_neighbors=10): 0.975555089773
```

Solution: *code source*

18.5.4 Cross-validation

Cross-validation consists in repeatedly splitting the data in pairs of train and test sets, called 'folds'. Scikit-learn comes with a function to automatically compute score on all these folds. Here we do `KFold` with k=5.

```
>>> clf = KNeighborsClassifier()
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(clf, X, y, cv=5)
array([0.947...,  0.955...,  0.966...,  0.980...,  0.963... ])
```

We can use different splitting strategies, such as random splitting:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> cv = ShuffleSplit(n_splits=5)
>>> cross_val_score(clf, X, y, cv=cv)
array([...])
```

Tip

There exists many different cross-validation strategies in scikit-learn. They are often useful to take in account non iid datasets.

18.5.5 Hyperparameter optimization with cross-validation

Consider regularized linear models, such as *Ridge Regression*, which uses l2 regularization, and *Lasso Regression*, which uses l1 regularization. Choosing their regularization parameter is important.

Let us set these parameters on the Diabetes dataset, a simple regression problem. The diabetes data consists of 10 physiological variables (age, sex, weight, blood pressure) measure on 442 patients, and an indication of disease progression after one year:

```
>>> from sklearn.datasets import load_diabetes
>>> data = load_diabetes()
>>> X, y = data.data, data.target
>>> print(X.shape)
(442, 10)
```

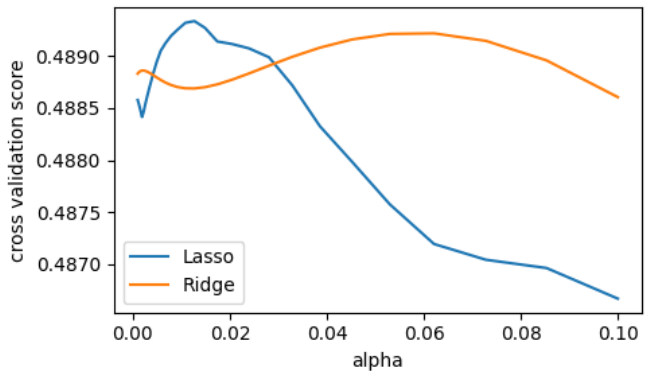
With the default hyper-parameters: we compute the cross-validation score:

```
>>> from sklearn.linear_model import Ridge, Lasso

>>> for Model in [Ridge, Lasso]:
...     model = Model()
...     print('%s: %s' % (Model.__name__, cross_val_score(model, X, y).mean()))
Ridge: 0.4...
Lasso: 0.3...
```

Basic Hyperparameter Optimization

We compute the cross-validation score as a function of alpha, the strength of the regularization for `Lasso` and `Ridge`. We choose 20 values of alpha between 0.0001 and 1:

```
>>> alphas = np.logspace(-3, -1, 30)

>>> for Model in [Lasso, Ridge]:
...     scores = [cross_val_score(Model(alpha), X, y, cv=3).mean()
...               for alpha in alphas]
...     plt.plot(alphas, scores, label=Model.__name__)
[<matplotlib.lines.Line2D object at ...
```



Question

Can we trust our results to be actually useful?

Automatically Performing Grid Search

`sklearn.grid_search.GridSearchCV` is constructed with an estimator, as well as a dictionary of parameter values to be searched. We can find the optimal parameters this way:

```
>>> from sklearn.model_selection import GridSearchCV
>>> for Model in [Ridge, Lasso]:
...     gscv = GridSearchCV(Model(), dict(alpha=alphas), cv=3).fit(X, y)
...     print('%s: %s' % (Model.__name__, gscv.best_params_))
Ridge: {'alpha': np.float64(0.06210169418915616)}
Lasso: {'alpha': np.float64(0.01268961003167922)}
```

Built-in Hyperparameter Search

For some models within scikit-learn, cross-validation can be performed more efficiently on large datasets. In this case, a cross-validated version of the particular model is included. The cross-validated versions of `Ridge` and `Lasso` are `RidgeCV` and `LassoCV`, respectively. Parameter search on these estimators can be performed as follows:

```
>>> from sklearn.linear_model import RidgeCV, LassoCV
>>> for Model in [RidgeCV, LassoCV]:
...     model = Model(alphas=alphas, cv=3).fit(X, y)
...     print('%s: %s' % (Model.__name__, model.alpha_))
RidgeCV: 0.0621016941892
LassoCV: 0.0126896100317
```

We see that the results match those returned by GridSearchCV

Nested cross-validation

How do we measure the performance of these estimators? We have used data to set the hyperparameters, so we need to test on actually new data. We can do this by running `cross_val_score()` on our CV objects. Here there are 2 cross-validation loops going on, this is called *'nested cross validation'*:

```
for Model in [RidgeCV, LassoCV]:
    scores = cross_val_score(Model(alphas=alphas, cv=3), X, y, cv=3)
    print(Model.__name__, np.mean(scores))
```

Note

Note that these results do not match the best results of our curves above, and `LassoCV` seems to under-perform `RidgeCV`. The reason is that setting the hyper-parameter is harder for Lasso, thus the estimation error on this hyper-parameter is larger.

18.6 Unsupervised Learning: Dimensionality Reduction and Visualization

Unsupervised learning is applied on X without y: data without labels. A typical use case is to find hidden structure in the data.

18.6.1 Dimensionality Reduction: PCA

Dimensionality reduction derives a set of new artificial features smaller than the original feature set. Here we'll use Principal Component Analysis (PCA), a dimensionality reduction that strives to retain most of the variance of the original data. We'll use `sklearn.decomposition.PCA` on the iris dataset:

```
>>> X = iris.data
>>> y = iris.target
```

Tip

`PCA` computes linear combinations of the original features using a truncated Singular Value Decomposition of the matrix X, to project the data onto a base of the top singular vectors.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2, whiten=True)
>>> pca.fit(X)
PCA(n_components=2, whiten=True)
```

Once fitted, `PCA` exposes the singular vectors in the `components_` attribute:

```
>>> pca.components_
array([[ 0.3..., -0.08...,  0.85...,  0.3...],
       [ 0.6...,  0.7..., -0.1..., -0.07...]])
```

Other attributes are available as well:

```
>>> pca.explained_variance_ratio_
array([0.92...,  0.053...])
```

Let us project the iris dataset along those first two dimensions::

```
>>> X_pca = pca.transform(X)
>>> X_pca.shape
(150, 2)
```

`PCA` `normalizes` and `whitens` the data, which means that the data is now centered on both components with unit variance:

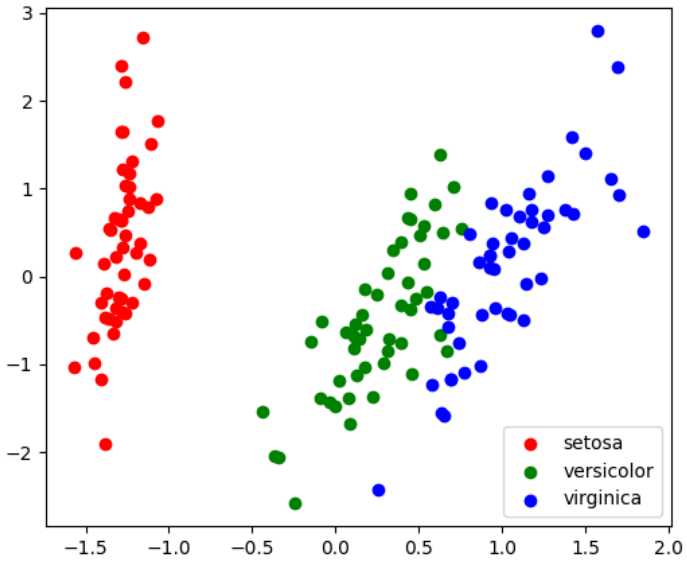
```
>>> X_pca.mean(axis=0)
array([...e-15,  ...e-15])
>>> X_pca.std(axis=0, ddof=1)
array([1.,  1.])
```

Furthermore, the samples components do no longer carry any linear correlation:

```
>>> np.corrcoef(X_pca.T)
array([[1.00000000e+00,   0.0],
       [0.0,   1.00000000e+00]])
```

With a number of retained components 2 or 3, PCA is useful to visualize the dataset:

```
>>> target_ids = range(len(iris.target_names))
>>> for i, c, label in zip(target_ids, 'rgbcmykw', iris.target_names):
...     plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1],
...                 c=c, label=label)
<matplotlib.collections.PathCollection ...
```



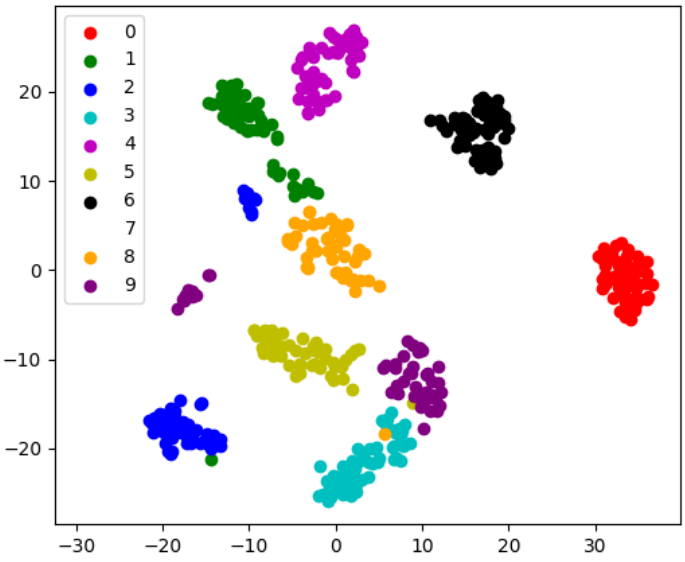
Tip

Note that this projection was determined *without* any information about the labels (represented by the colors): this is the sense in which the learning is **unsupervised**. Nevertheless, we see that the projection gives us insight into the distribution of the different flowers in parameter space: notably, *iris setosa* is much more distinct than the other two species.

18.6.2 Visualization with a non-linear embedding: tSNE

For visualization, more complex embeddings can be useful (for statistical analysis, they are harder to control). `sklearn.manifold.TSNE` is such a powerful manifold learning method. We apply it to the *digits* dataset, as the digits are vectors of dimension 8*8 = 64. Embedding them in 2D enables visualization:

```
>>> # Take the first 500 data points: it's hard to see 1500 points
>>> X = digits.data[:500]
>>> y = digits.target[:500]

>>> # Fit and transform with a TSNE
>>> from sklearn.manifold import TSNE
>>> tsne = TSNE(n_components=2, learning_rate='auto', init='random', random_state=0)
>>> X_2d = tsne.fit_transform(X)

>>> # Visualize the data
>>> plt.scatter(X_2d[:, 0], X_2d[:, 1], c=y)
<matplotlib.collections.PathCollection object at ...>
```



fit_transform

As `TSNE` cannot be applied to new data, we need to use its *fit_transform* method.

`sklearn.manifold.TSNE` separates quite well the different classes of digits even though it had no access to the class information.

Exercise: Other dimension reduction of digits

`sklearn.manifold` has many other non-linear embeddings. Try them out on the digits dataset. Could you judge their quality without knowing the labels `y`?

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> # ...
```

18.7 Parameter selection, Validation, and Testing

18.7.1 Hyperparameters, Over-fitting, and Under-fitting

See also

This section is adapted from Andrew Ng's excellent Coursera course

The issues associated with validation and cross-validation are some of the most important aspects of the practice of machine learning. Selecting the optimal model for your data is vital, and is a piece of the problem that is not often appreciated by machine learning practitioners.

The central question is: **If our estimator is underperforming, how should we move forward?**

- Use simpler or more complicated model?
- Add more features to each observed data point?
- Add more training samples?

The answer is often counter-intuitive. In particular, **Sometimes using a more complicated model will give worse results.** Also, **Sometimes adding training data will not improve your results.** The ability to determine what steps will improve your model is what separates the successful machine learning practitioners from the unsuccessful.

Bias-variance trade-off: illustration on a simple regression problem

Code and notebook

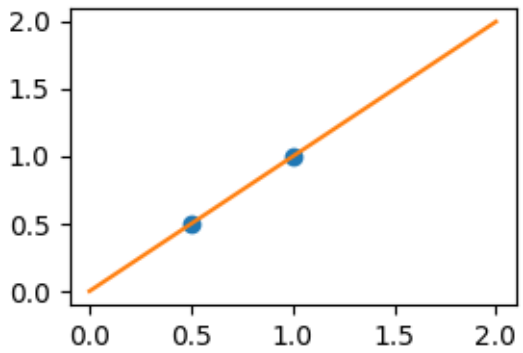
Python code and Jupyter notebook for this section are found *here*

Let us start with a simple 1D regression problem. This will help us to easily visualize the data and the model, and the results generalize easily to higher-dimensional datasets. We'll explore a simple **linear regression** problem, with `sklearn.linear_model`.

```
X = np.c_[0.5, 1].T
y = [0.5, 1]
X_test = np.c_[0, 2].T
```

Without noise, as linear regression fits the data perfectly

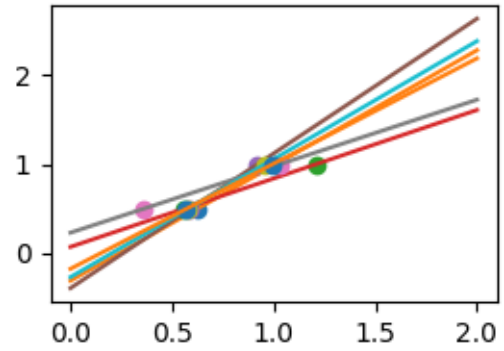
```
from sklearn import linear_model

regr = linear_model.LinearRegression()
regr.fit(X, y)
plt.plot(X, y, "o")
plt.plot(X_test, regr.predict(X_test))
```




```
[<matplotlib.lines.Line2D object at 0x7f3b1b2dd6d0>]
```

In real life situation, we have noise (e.g. measurement noise) in our data:

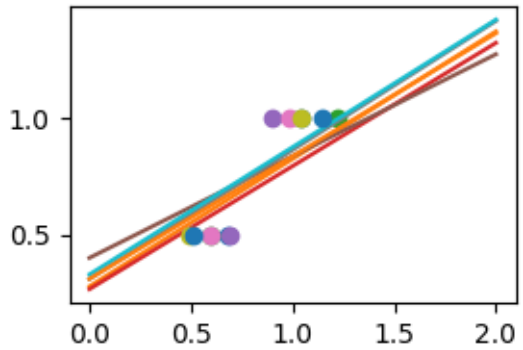
```
rng = np.random.default_rng(27446968)
for _ in range(6):
    noisy_X = X + np.random.normal(loc=0, scale=0.1, size=X.shape)
    plt.plot(noisy_X, y, "o")
    regr.fit(noisy_X, y)
    plt.plot(X_test, regr.predict(X_test))
```



As we can see, our linear model captures and amplifies the noise in the data. It displays a lot of variance.

We can use another linear estimator that uses regularization, the `Ridge` estimator. This estimator regularizes the coefficients by shrinking them to zero, under the assumption that very high correlations are often spurious. The alpha parameter controls the amount of shrinkage used.

```
regr = linear_model.Ridge(alpha=0.1)
np.random.seed(0)
for _ in range(6):
    noisy_X = X + np.random.normal(loc=0, scale=0.1, size=X.shape)
    plt.plot(noisy_X, y, "o")
    regr.fit(noisy_X, y)
    plt.plot(X_test, regr.predict(X_test))

plt.show()
```



As we can see, the estimator displays much less variance. However it systematically under-estimates the coefficient. It displays a biased behavior.

This is a typical example of **bias/variance tradeof**: non-regularized estimator are not biased, but they can display a lot of variance. Highly-regularized models have little variance, but high bias. This bias is not necessarily a bad thing: what matters is choosing the tradeoff between bias and variance that leads to the best prediction performance. For a specific dataset there is a sweet spot corresponding to the highest complexity that the data can support, depending on the amount of noise and of observations available.

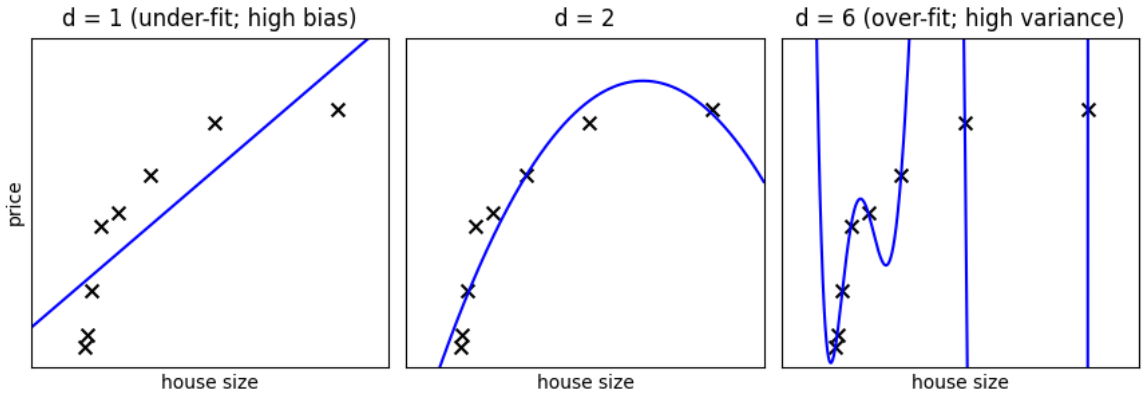
18.7.2 Visualizing the Bias/Variance Tradeoff

Tip

Given a particular dataset and a model (e.g. a polynomial), we'd like to understand whether bias (underfit) or variance limits prediction, and how to tune the *hyperparameter* (here `d`, the degree of

the polynomial) to give the best fit.

On a given data, let us fit a simple polynomial regression model with varying degrees:



Tip

In the above figure, we see fits for three different values of `d`. For `d = 1`, the data is under-fit. This means that the model is too simplistic: no straight line will ever be a good fit to this data. In this case, we say that the model suffers from high bias. The model itself is biased, and this will be reflected in the fact that the data is poorly fit. At the other extreme, for `d = 6` the data is over-fit. This means that the model has too many free parameters (6 in this case) which can be adjusted to perfectly fit the training data. If we add a new point to this plot, though, chances are it will be very far from the curve representing the degree-6 fit. In this case, we say that the model suffers from high variance. The reason for the term "high variance" is that if any of the input points are varied slightly, it could result in a very different model.

In the middle, for `d = 2`, we have found a good mid-point. It fits the data fairly well, and does not suffer from the bias and variance problems seen in the figures on either side. What we would like is a way to quantitatively identify bias and variance, and optimize the metaparameters (in this case, the polynomial degree d) in order to determine the best algorithm.

Polynomial regression with scikit-learn

A polynomial regression is built by pipelining `PolynomialFeatures` and a `LinearRegression`:

```
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import PolynomialFeatures
>>> from sklearn.linear_model import LinearRegression
>>> model = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())
```

Validation Curves

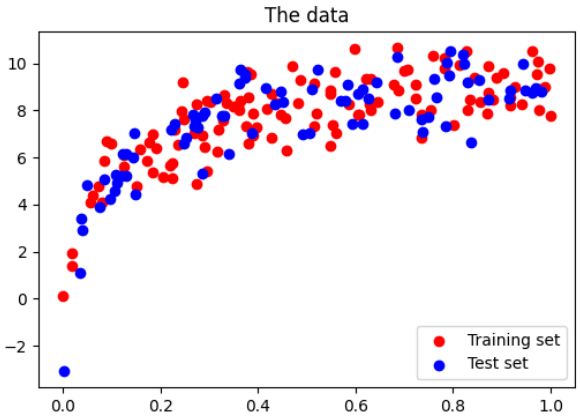
Let us create a dataset like in the example above:

```
>>> def generating_func(x, rng, err=0.5):
...     return rng.normal(10 - 1. / (x + 0.1), err)

>>> # randomly sample more data
>>> rng = np.random.default_rng(27446968)
```

(continues on next page)

(continued from previous page)

```
>>> x = rng.random(size=200)
>>> y = generating_func(x, err=1., rng=rng)
```



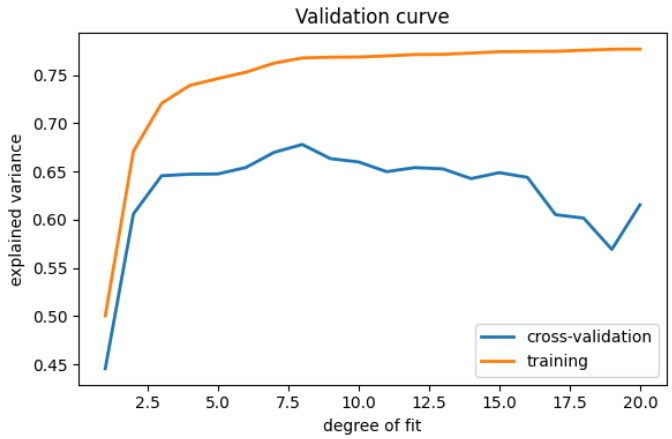
Central to quantify bias and variance of a model is to apply it on *test data*, sampled from the same distribution as the train, but that will capture independent noise:

```
>>> xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.4)
```

Validation curve A validation curve consists in varying a model parameter that controls its complexity (here the degree of the polynomial) and measures both error of the model on training data, and on test data (*eg* with cross-validation). The model parameter is then adjusted so that the test error is minimized:

We use `sklearn.model_selection.validation_curve()` to compute train and test error, and plot it:

```
>>> from sklearn.model_selection import validation_curve

>>> degrees = np.arange(1, 21)

>>> model = make_pipeline(PolynomialFeatures(), LinearRegression())

>>> # Vary the "degrees" on the pipeline step "polynomialfeatures"
>>> train_scores, validation_scores = validation_curve(
...                 model, x[:, np.newaxis], y,
...                 param_name='polynomialfeatures__degree',
...                 param_range=degrees)

>>> # Plot the mean train score and validation score across folds
>>> plt.plot(degrees, validation_scores.mean(axis=1), label='cross-validation')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(degrees, train_scores.mean(axis=1), label='training')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.legend(loc='best')
<matplotlib.legend.Legend object at ...>
```



This figure shows why validation is important. On the left side of the plot, we have very low-degree polynomial, which under-fit the data. This leads to a low explained variance for both the training set and the validation set. On the far right side of the plot, we have a very high degree polynomial, which over-fits the data. This can be seen in the fact that the training explained variance is very high, while on the validation set, it is low. Choosing `d` around 4 or 5 gets us the best tradeoff.

Tip

The astute reader will realize that something is amiss here: in the above plot, `d = 4` gives the best results. But in the previous plot, we found that `d = 6` vastly over-fits the data. What's going on here? The difference is the **number of training points** used. In the previous example, there were only eight training points. In this example, we have 100. As a general rule of thumb, the more training points used, the more complicated model can be used. But how can you determine for a given model whether more training points will be helpful? A useful diagnostic for this are learning curves.

Learning Curves

A learning curve shows the training and validation score as a function of the number of training points. Note that when we train on a subset of the training data, the training score is computed using this subset, not the full training set. This curve gives a quantitative view into how beneficial it will be to add training samples.

Questions:

- As the number of training samples are increased, what do you expect to see for the training score? For the validation score?
- Would you expect the training score to be higher or lower than the validation score? Would you ever expect this to change?

`scikit-learn` provides `sklearn.model_selection.learning_curve()`:

```
>>> from sklearn.model_selection import learning_curve
>>> train_sizes, train_scores, validation_scores = learning_curve(
...     model, x[:, np.newaxis], y, train_sizes=np.logspace(-1, 0, 20))

>>> # Plot the mean train score and validation score across folds
>>> plt.plot(train_sizes, validation_scores.mean(axis=1), label='cross-validation')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(train_sizes, train_scores.mean(axis=1), label='training')
[<matplotlib.lines.Line2D object at ...>]
```

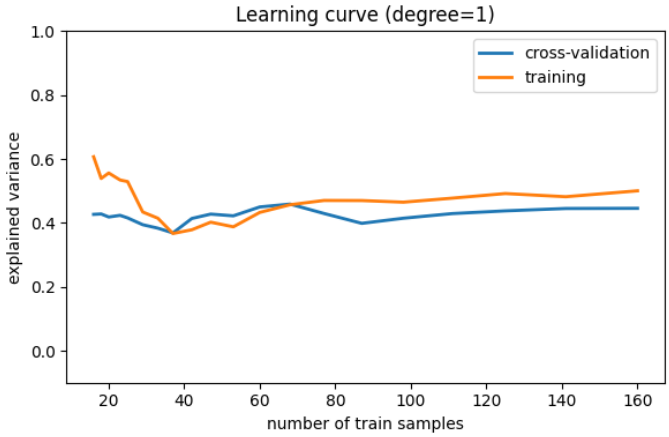


Fig. 5: For a `degree=1` model

Note that the validation score *generally increases* with a growing training set, while the training score *generally decreases* with a growing training set. As the training size increases, they will converge to a single value.

From the above discussion, we know that `d = 1` is a high-bias estimator which under-fits the data. This is indicated by the fact that both the training and validation scores are low. When confronted with this type of learning curve, we can expect that adding more training data will not help: both lines converge to a relatively low score.

When the learning curves have converged to a low score, we have a high bias model.

A high-bias model can be improved by:

- Using a more sophisticated model (i.e. in this case, increase `d`)
- Gather more features for each sample.
- Decrease regularization in a regularized model.

Increasing the number of samples, however, does not improve a high-bias model.

Now let's look at a high-variance (i.e. over-fit) model:

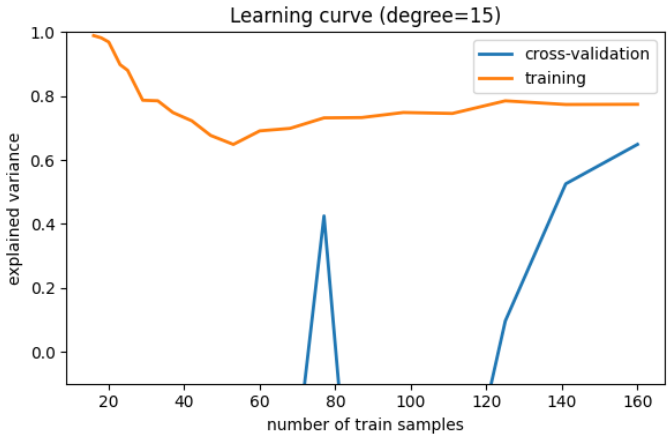


Fig. 6: For a `degree=15` model

Here we show the learning curve for `d = 15`. From the above discussion, we know that `d = 15` is a **high-variance** estimator which **over-fits** the data. This is indicated by the fact that the training score is much higher than the validation score. As we add more samples to this training set, the training score will continue to decrease, while the cross-validation error will continue to increase, until they meet in the middle.

Learning curves that have not yet converged with the full training set indicate a high-variance, over-fit model.

A high-variance model can be improved by:

- Gathering more training samples.
- Using a less-sophisticated model (i.e. in this case, make `d` smaller)
- Increasing regularization.

In particular, gathering more features for each sample will not help the results.

18.7.3 Summary on model selection

We've seen above that an under-performing algorithm can be due to two possible situations: high bias (under-fitting) and high variance (over-fitting). In order to evaluate our algorithm, we set aside a portion of our training data for cross-validation. Using the technique of learning curves, we can train on progressively larger subsets of the data, evaluating the training error and cross-validation error to determine whether our algorithm has high variance or high bias. But what do we do with this information?

High Bias

If a model shows high **bias**, the following actions might help:

- **Add more features**. In our example of predicting home prices, it may be helpful to make use of information such as the neighborhood the house is in, the year the house was built, the size of the lot, etc. Adding these features to the training and test sets can improve a high-bias estimator
- **Use a more sophisticated model**. Adding complexity to the model can help improve on bias. For a polynomial fit, this can be accomplished by increasing the degree d. Each learning technique has its own methods of adding complexity.
- **Use fewer samples**. Though this will not improve the classification, a high-bias algorithm can attain nearly the same error with a smaller training sample. For algorithms which are computationally expensive, reducing the training sample size can lead to very large improvements in speed.
- **Decrease regularization**. Regularization is a technique used to impose simplicity in some machine learning models, by adding a penalty term that depends on the characteristics of the parameters. If a model has high bias, decreasing the effect of regularization can lead to better results.

High Variance

If a model shows **high variance**, the following actions might help:

- **Use fewer features**. Using a feature selection technique may be useful, and decrease the over-fitting of the estimator.
- **Use a simpler model**. Model complexity and over-fitting go hand-in-hand.
- **Use more training samples**. Adding training samples can reduce the effect of over-fitting, and lead to improvements in a high variance estimator.
- **Increase Regularization**. Regularization is designed to prevent over-fitting. In a high-variance model, increasing regularization can lead to better results.

These choices become very important in real-world situations. For example, due to limited telescope time, astronomers must seek a balance between observing a large number of objects, and observing a large number of features for each object. Determining which is more important for a particular learning task can inform the observing strategy that the astronomer employs.

18.7.4 A last word of caution: separate validation and test set

Using validation schemes to determine hyper-parameters means that we are fitting the hyper-parameters to the particular validation set. In the same way that parameters can be over-fit to the training set, hyperparameters can be over-fit to the validation set. Because of this, the validation error tends to under-predict the classification error of new data.

For this reason, it is recommended to split the data into three sets:

- The **training set**, used to train the model (usually ~60% of the data)
- The **validation set**, used to validate the model (usually ~20% of the data)
- The **test set**, used to evaluate the expected error of the validated model (usually ~20% of the data)

Many machine learning practitioners do not separate test set and validation set. But if your goal is to gauge the error of a model on unknown data, using an independent test set is vital.

18.8 Examples for the scikit-learn chapter

18.8.1 Demo PCA in 2D

Load the iris data

```
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
y = iris.target
```

Fit a PCA

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2, whiten=True)
pca.fit(X)
```

Project the data in 2D

```
X_pca = pca.transform(X)
```

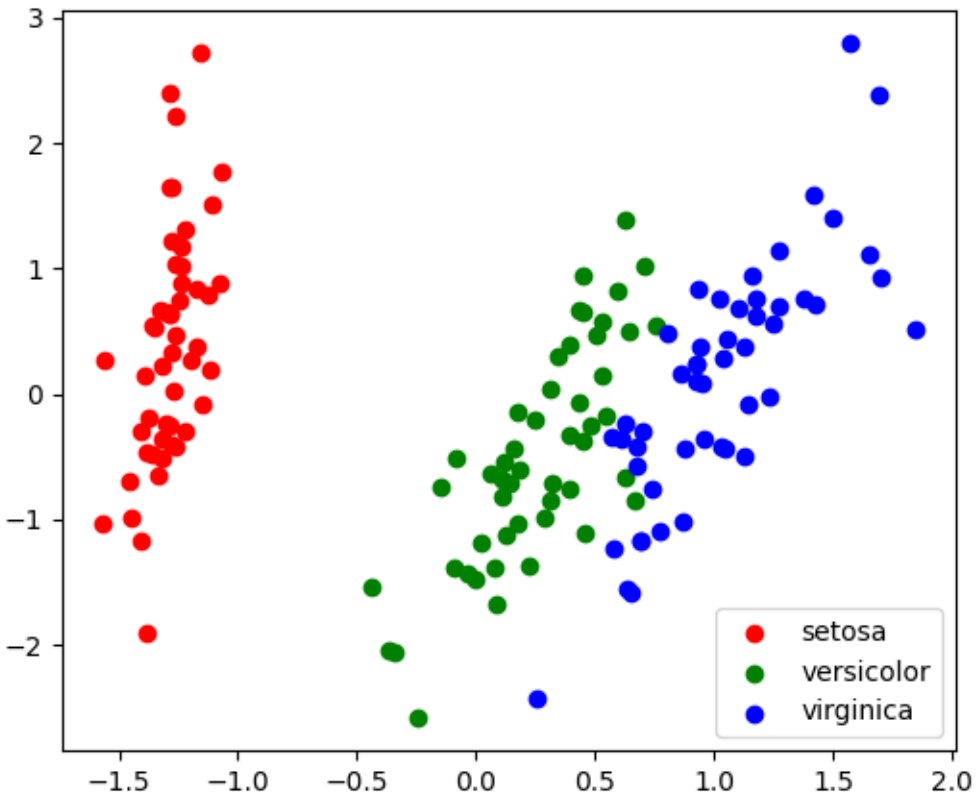
Visualize the data

```
target_ids = range(len(iris.target_names))

import matplotlib.pyplot as plt

plt.figure(figsize=(6, 5))
for i, c, label in zip(target_ids, "rgbcmykw", iris.target_names, strict=False):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], c=c, label=label)
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 0.101 seconds)

18.8.2 Measuring Decision Tree performance

Demonstrates overfit when testing on train set.

Get the data

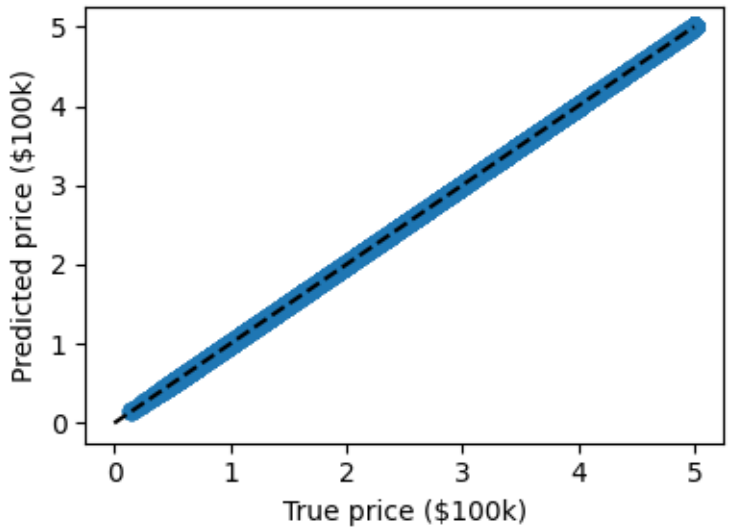
```
from sklearn.datasets import fetch_california_housing

data = fetch_california_housing(as_frame=True)
```

Train and test a model

```
from sklearn.tree import DecisionTreeRegressor

clf = DecisionTreeRegressor().fit(data.data, data.target)

predicted = clf.predict(data.data)
expected = data.target
```

Plot predicted as a function of expected

```
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 3))
plt.scatter(expected, predicted)
plt.plot([0, 5], [0, 5], "--k")
plt.axis("tight")
plt.xlabel("True price ($100k)")
plt.ylabel("Predicted price ($100k)")
plt.tight_layout()
```



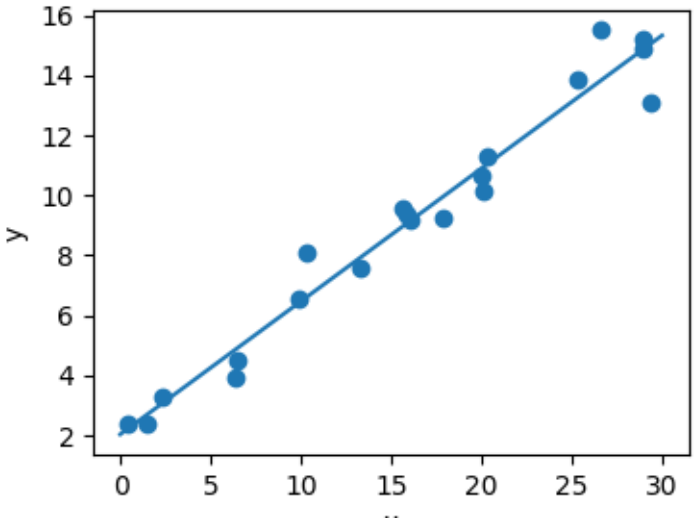
Pretty much no errors!

This is too good to be true: we are testing the model on the train data, which is not a measure of generalization.

The results are not valid

Total running time of the script: (0 minutes 1.971 seconds)

18.8.3 A simple linear regression



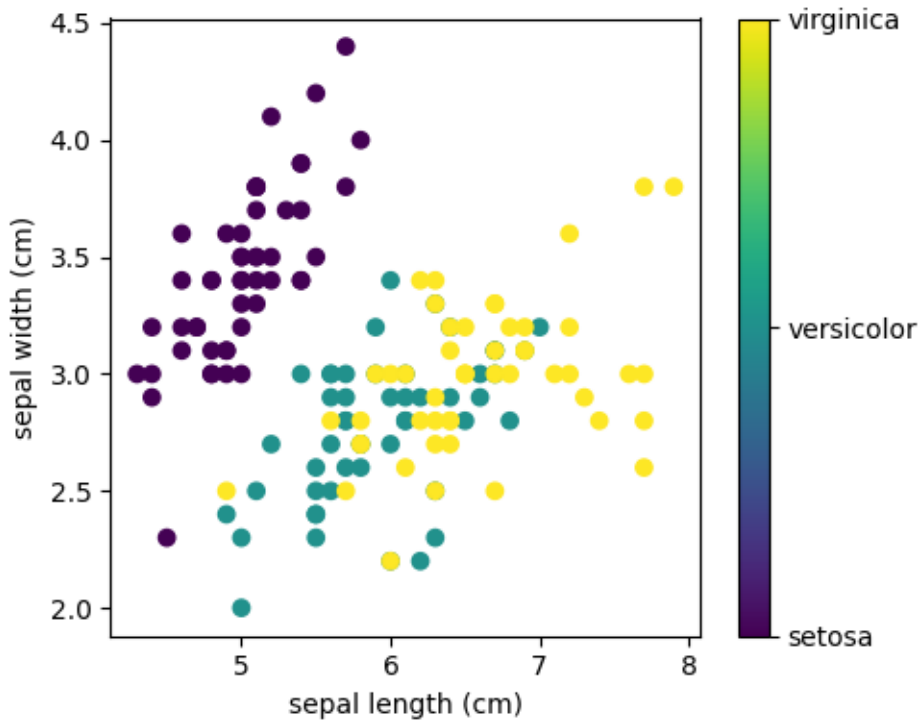
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# x from 0 to 30
rng = np.random.default_rng()
x = 30 * rng.random((20, 1))

# y = a*x + b with noise
y = 0.5 * x + 1.0 + rng.normal(size=x.shape)

# create a linear regression model
model = LinearRegression()
model.fit(x, y)

# predict y from the data
x_new = np.linspace(0, 30, 100)
y_new = model.predict(x_new[:, np.newaxis])

# plot the results
plt.figure(figsize=(4, 3))
ax = plt.axes()
ax.scatter(x, y)
ax.plot(x_new, y_new)

ax.set_xlabel("x")
ax.set_ylabel("y")

ax.axis("tight")


plt.show()
```

Total running time of the script: (0 minutes 0.048 seconds)

18.8.4 Plot 2D views of the iris dataset

Plot a simple scatter plot of 2 features of the iris dataset.

Note that more elaborate visualization of this dataset is detailed in the *Statistics in Python* chapter.




```
# Load the data
from sklearn.datasets import load_iris

iris = load_iris()

from matplotlib import ticker
import matplotlib.pyplot as plt

# The indices of the features that we are plotting
x_index = 0
y_index = 1

# this formatter will label the colorbar with the correct target names
formatter = ticker.FuncFormatter(lambda i, *args: iris.target_names[int(i)])

plt.figure(figsize=(5, 4))
plt.scatter(iris.data[:, x_index], iris.data[:, y_index], c=iris.target)
plt.colorbar(ticks=[0, 1, 2], format=formatter)
plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index])

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.090 seconds)

18.8.5 tSNE to visualize digits

Here we use `sklearn.manifold.TSNE` to visualize the digits datasets. Indeed, the digits are vectors in a 8*8 = 64 dimensional space. We want to project them in 2D for visualization. tSNE is often a good solution, as it groups and separates data points based on their local relationship.

Load the iris data

```
from sklearn import datasets

digits = datasets.load_digits()
# Take the first 500 data points: it's hard to see 1500 points
X = digits.data[:500]
y = digits.target[:500]
```

Fit and transform with a TSNE

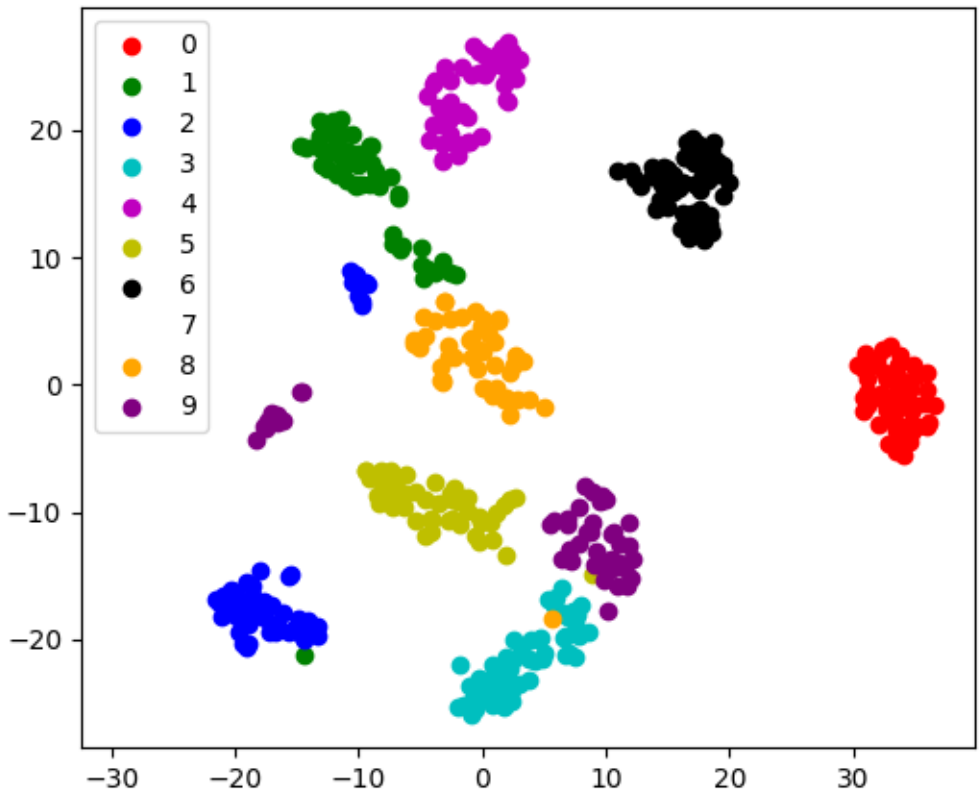
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=0)
```

Project the data in 2D

```
X_2d = tsne.fit_transform(X)
```

Visualize the data

```
target_ids = range(len(digits.target_names))

import matplotlib.pyplot as plt

plt.figure(figsize=(6, 5))
colors = "r", "g", "b", "c", "m", "y", "k", "w", "orange", "purple"
for i, c, label in zip(target_ids, colors, digits.target_names, strict=True):
    plt.scatter(X_2d[y == i, 0], X_2d[y == i, 1], c=c, label=label)
plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 1.217 seconds)

18.8.6 Use the RidgeCV and LassoCV to set the regularization parameter

Load the diabetes dataset

```
from sklearn.datasets import load_diabetes

data = load_diabetes()
X, y = data.data, data.target
print(X.shape)
```

```
(442, 10)
```

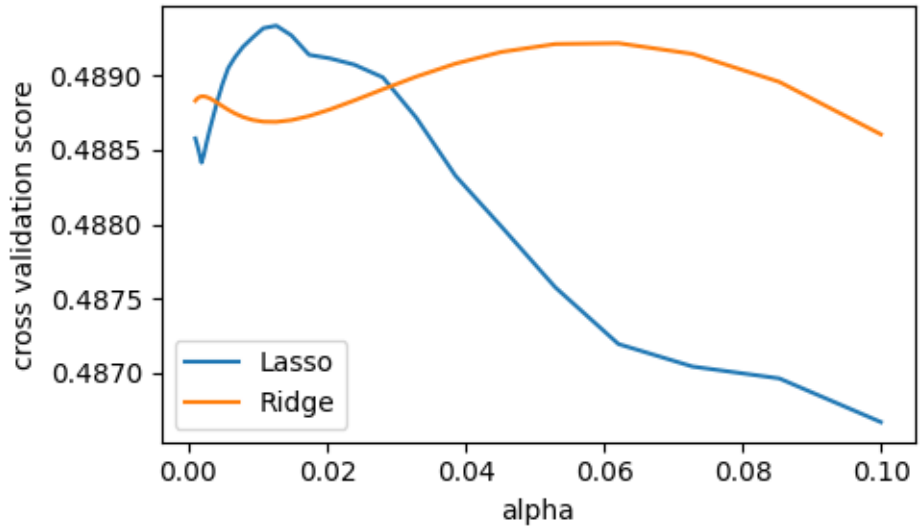
Compute the cross-validation score with the default hyper-parameters

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge, Lasso

for Model in [Ridge, Lasso]:
    model = Model()
    print(f"{Model.__name__}: {cross_val_score(model, X, y).mean()}")
```

```
Ridge: 0.410174971340889
Lasso: 0.3375593674654274
```

We compute the cross-validation score as a function of alpha, the strength of the regularization for Lasso and Ridge

```
import numpy as np
import matplotlib.pyplot as plt

alphas = np.logspace(-3, -1, 30)

plt.figure(figsize=(5, 3))

for Model in [Lasso, Ridge]:
    scores = [cross_val_score(Model(alpha), X, y, cv=3).mean() for alpha in alphas]
    plt.plot(alphas, scores, label=Model.__name__)

plt.legend(loc="lower left")
plt.xlabel("alpha")
plt.ylabel("cross validation score")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.420 seconds)

18.8.7 Plot variance and regularization in linear models

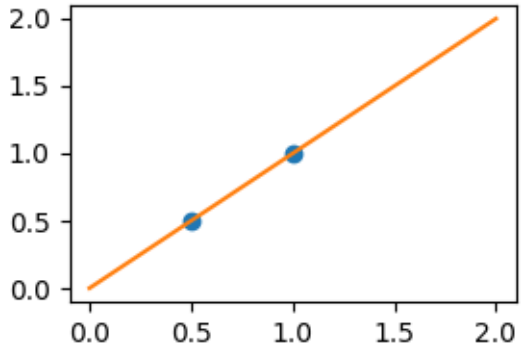
```
import numpy as np

# Smaller figures
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (3, 2)
```

We consider the situation where we have only 2 data point

```
X = np.c_[0.5, 1].T
y = [0.5, 1]
X_test = np.c_[0, 2].T
```

Without noise, as linear regression fits the data perfectly

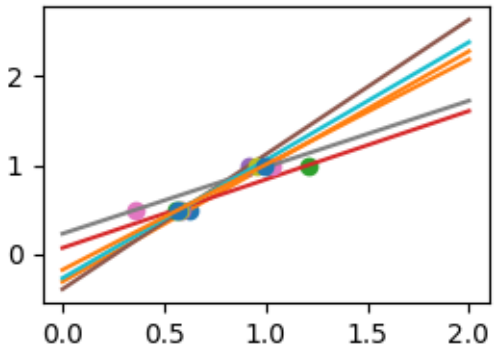
```
from sklearn import linear_model

regr = linear_model.LinearRegression()
regr.fit(X, y)
plt.plot(X, y, "o")
plt.plot(X_test, regr.predict(X_test))
```




```
[<matplotlib.lines.Line2D object at 0x7f3b1b2dd6d0>]
```

In real life situation, we have noise (e.g. measurement noise) in our data:

```
rng = np.random.default_rng(27446968)
for _ in range(6):
    noisy_X = X + np.random.normal(loc=0, scale=0.1, size=X.shape)
    plt.plot(noisy_X, y, "o")
    regr.fit(noisy_X, y)
    plt.plot(X_test, regr.predict(X_test))
```



As we can see, our linear model captures and amplifies the noise in the data. It displays a lot of variance.

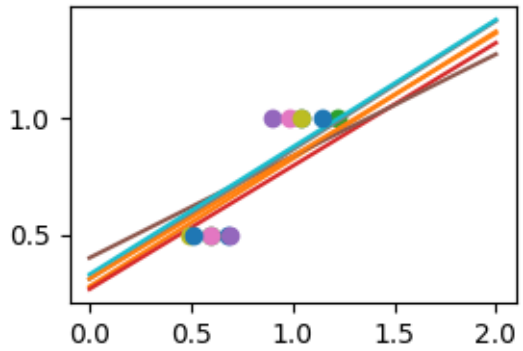
We can use another linear estimator that uses regularization, the `Ridge` estimator. This estimator regularizes the coefficients by shrinking them to zero, under the assumption that very high correlations are often spurious. The alpha parameter controls the amount of shrinkage used.

```
regr = linear_model.Ridge(alpha=0.1)
np.random.seed(0)
for _ in range(6):
    noisy_X = X + np.random.normal(loc=0, scale=0.1, size=X.shape)
    plt.plot(noisy_X, y, "o")
    regr.fit(noisy_X, y)
    plt.plot(X_test, regr.predict(X_test))
```

(continues on next page)

(continued from previous page)

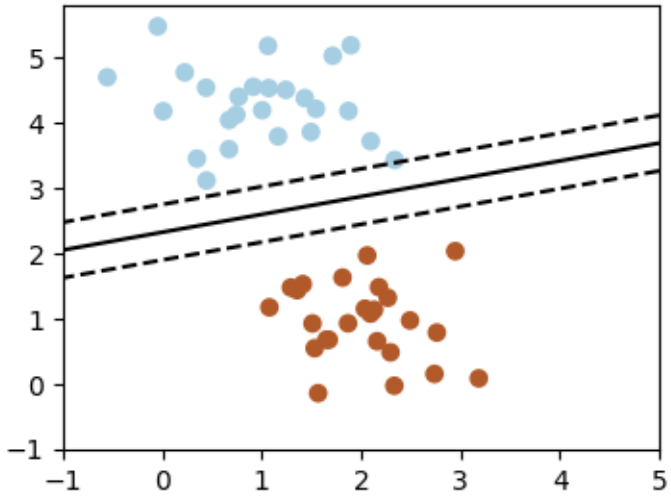
```
plt.show()
```



Total running time of the script: (0 minutes 0.108 seconds)

18.8.8 Simple picture of the formal problem of machine learning

This example generates simple synthetic data points and shows a separating hyperplane on them.




```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import make_blobs

# we create 50 separable synthetic points
X, Y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.60)

# fit the model
clf = SGDClassifier(loss="hinge", alpha=0.01, fit_intercept=True)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
xx = np.linspace(-1, 5, 10)
yy = np.linspace(-1, 5, 10)
```

(continues on next page)

(continued from previous page)

```
X1, X2 = np.meshgrid(xx, yy)
Z = np.empty(X1.shape)
for (i, j), val in np.ndenumerate(X1):
    x1 = val
    x2 = X2[i, j]
    p = clf.decision_function([[x1, x2]])
    Z[i, j] = p[0]

plt.figure(figsize=(4, 3))
ax = plt.axes()
ax.contour(
    X1, X2, Z, [-1.0, 0.0, 1.0], colors="k", linestyles=["dashed", "solid", "dashed"]
)
ax.scatter(X[:, 0], X[:, 1], c=Y, cmap="Paired")

ax.axis("tight")


plt.show()
```

Total running time of the script: (0 minutes 0.056 seconds)

18.8.9 Compare classifiers on the digits data

Compare the performance of a variety of classifiers on a test set for the digits data.

```
LinearSVC: 0.9344942114287969
GaussianNB: 0.8332741681010102
KNeighborsClassifier: 0.9804562804949924
------------------
/opt/hostedtoolcache/Python/3.12.11/x64/lib/python3.12/site-packages/sklearn/svm/_
↪base.py:1249: ConvergenceWarning: Liblinear failed to converge, increase the number␣
↪of iterations.
  warnings.warn(
LinearSVC(loss='hinge'): 0.9294570108037394
LinearSVC(loss='squared_hinge'): 0.9344942114287969
--------------------
KNeighbors(n_neighbors=1): 0.9913675218842191
KNeighbors(n_neighbors=2): 0.9848442068835102
KNeighbors(n_neighbors=3): 0.9867753449543099
KNeighbors(n_neighbors=4): 0.9803719053818863
KNeighbors(n_neighbors=5): 0.9804562804949924
KNeighbors(n_neighbors=6): 0.9757924194139573
KNeighbors(n_neighbors=7): 0.9780645792142071
KNeighbors(n_neighbors=8): 0.9780645792142071
KNeighbors(n_neighbors=9): 0.9780645792142071
KNeighbors(n_neighbors=10): 0.9755550897728812
```

```
from sklearn import model_selection, datasets, metrics
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

digits = datasets.load_digits()
X = digits.data
y = digits.target
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size=0.25, random_state=0
)

for Model in [LinearSVC, GaussianNB, KNeighborsClassifier]:
    clf = Model().fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(f"{Model.__name__}: {metrics.f1_score(y_test, y_pred, average='macro')}")

print("------------------")

# test SVC loss
for loss in ["hinge", "squared_hinge"]:
    clf = LinearSVC(loss=loss).fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(
        f"LinearSVC(loss='{loss}'): {metrics.f1_score(y_test, y_pred, average='macro
↪')}"
    )

print("--------------------")

# test the number of neighbors
for n_neighbors in range(1, 11):
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(
        f"KNeighbors(n_neighbors={n_neighbors}): {metrics.f1_score(y_test, y_pred,␣
↪average='macro')}"
    )
```

Total running time of the script: (0 minutes 0.267 seconds)

18.8.10 Plot fitting a 9th order polynomial

Fits data generated from a 9th order polynomial with model of 4th order and 9th order polynomials, to demonstrate that often simpler models are to be preferred

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

from sklearn import linear_model

# Create color maps for 3-class classification problem, as with iris
cmap_light = ListedColormap(["#FFAAAA", "#AAFFAA", "#AAAAFF"])
cmap_bold = ListedColormap(["#FF0000", "#00FF00", "#0000FF"])
```

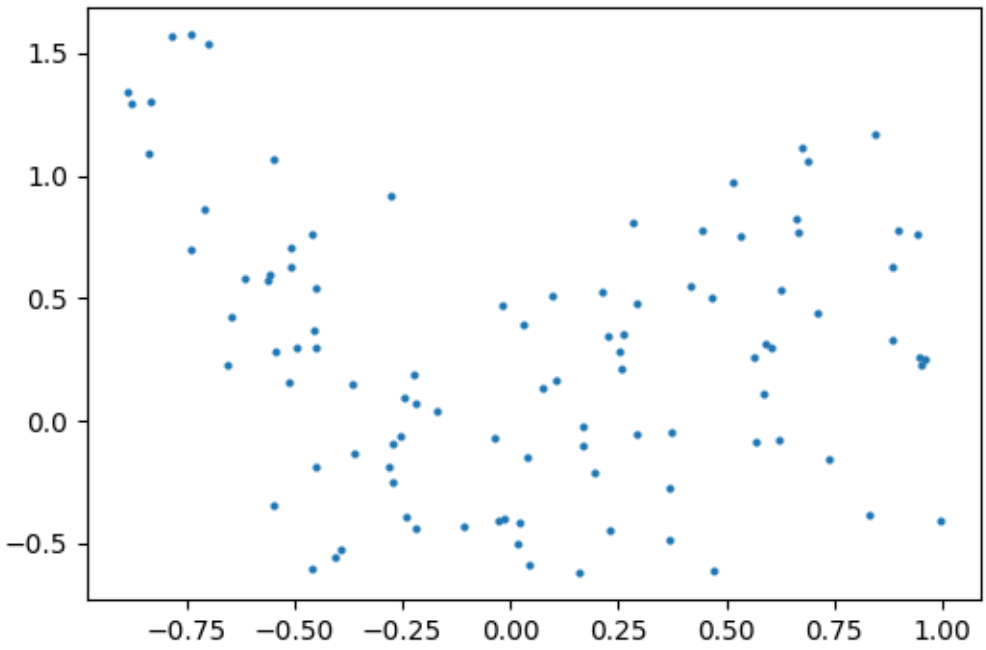
(continues on next page)

(continued from previous page)

```
rng = np.random.default_rng(27446968)
x = 2 * rng.random(100) - 1

f = lambda t: 1.2 * t**2 + 0.1 * t**3 - 0.4 * t**5 - 0.5 * t**9
y = f(x) + 0.4 * rng.normal(size=100)

x_test = np.linspace(-1, 1, 100)
```

The data

```
plt.figure(figsize=(6, 4))
plt.scatter(x, y, s=4)
```




```
<matplotlib.collections.PathCollection object at 0x7f3b1b6d33b0>
```

Fitting 4th and 9th order polynomials

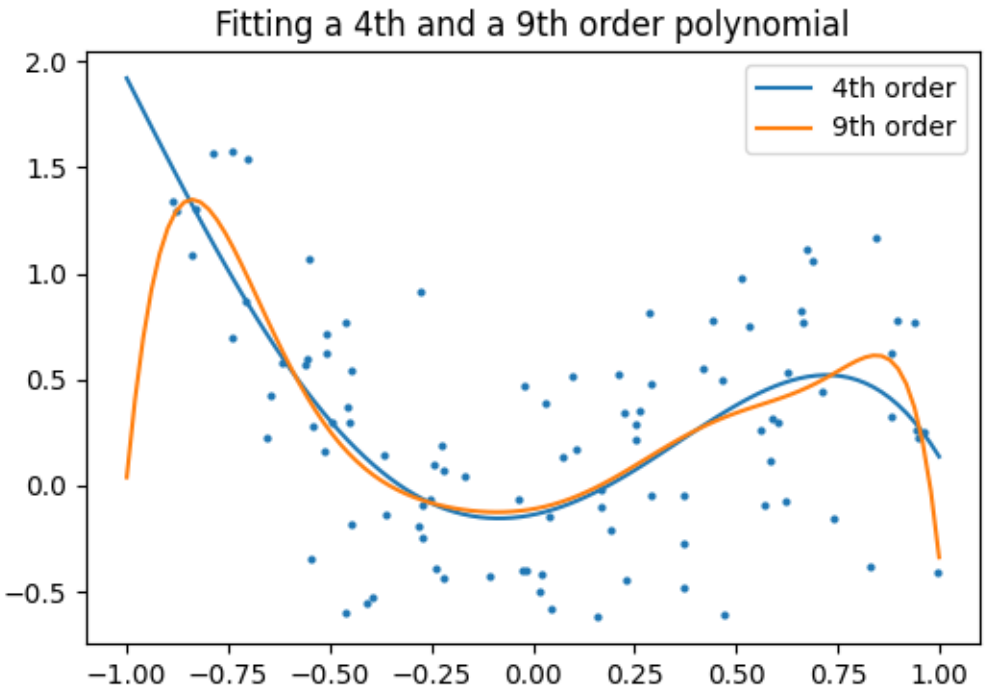
For this we need to engineer features: the n_th powers of x:

```
plt.figure(figsize=(6, 4))
plt.scatter(x, y, s=4)

X = np.array([x**i for i in range(5)]).T
X_test = np.array([x_test**i for i in range(5)]).T
regr = linear_model.LinearRegression()
regr.fit(X, y)
plt.plot(x_test, regr.predict(X_test), label="4th order")

X = np.array([x**i for i in range(10)]).T
X_test = np.array([x_test**i for i in range(10)]).T
regr = linear_model.LinearRegression()
```

(continues on next page)

(continued from previous page)

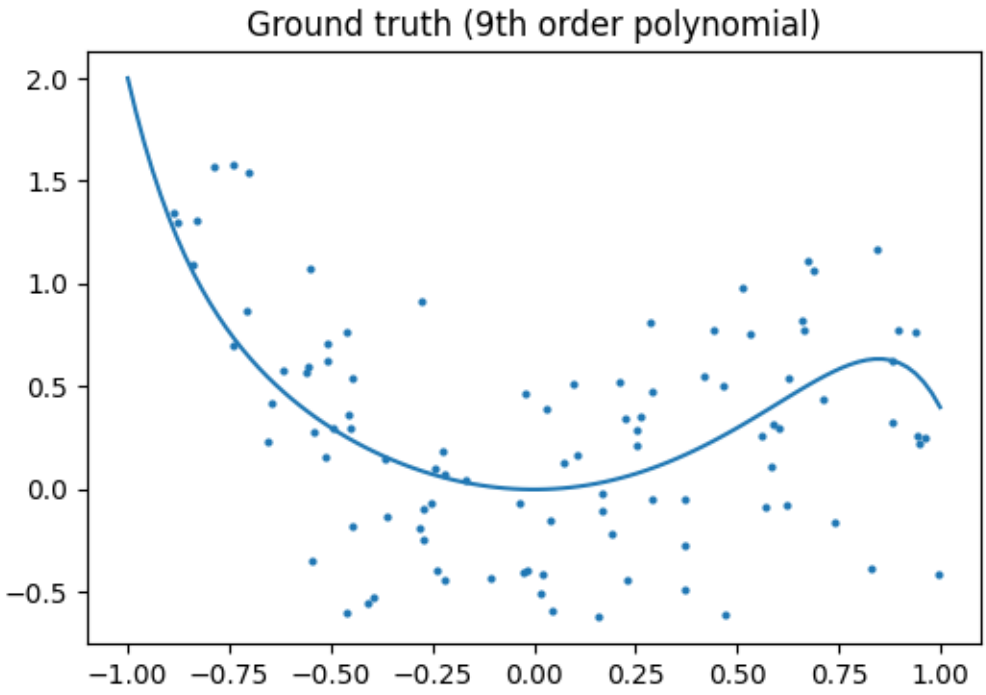
```
regr.fit(X, y)
plt.plot(x_test, regr.predict(X_test), label="9th order")

plt.legend(loc="best")
plt.axis("tight")
plt.title("Fitting a 4th and a 9th order polynomial")
```




```
Text(0.5, 1.0, 'Fitting a 4th and a 9th order polynomial')
```

Ground truth

```
plt.figure(figsize=(6, 4))
plt.scatter(x, y, s=4)
plt.plot(x_test, f(x_test), label="truth")
plt.axis("tight")
plt.title("Ground truth (9th order polynomial)")

plt.show()
```



Total running time of the script: (0 minutes 0.173 seconds)

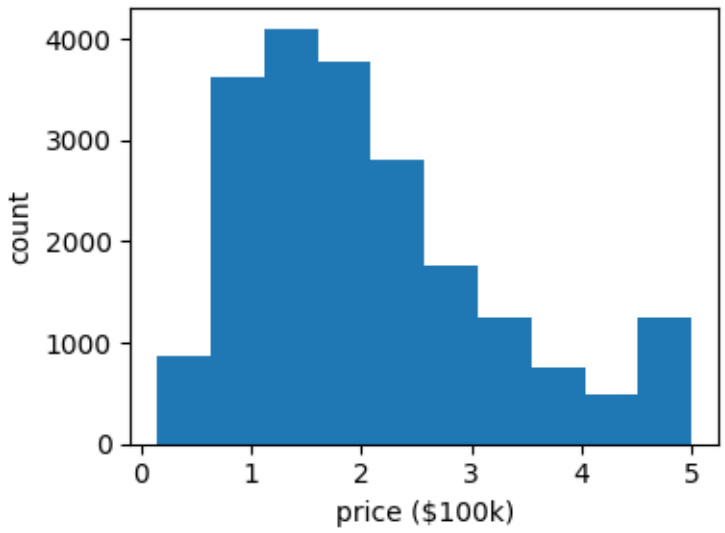
18.8.11 A simple regression analysis on the California housing data

Here we perform a simple regression analysis on the California housing data, exploring two types of regressors.

```
from sklearn.datasets import fetch_california_housing

data = fetch_california_housing(as_frame=True)
```

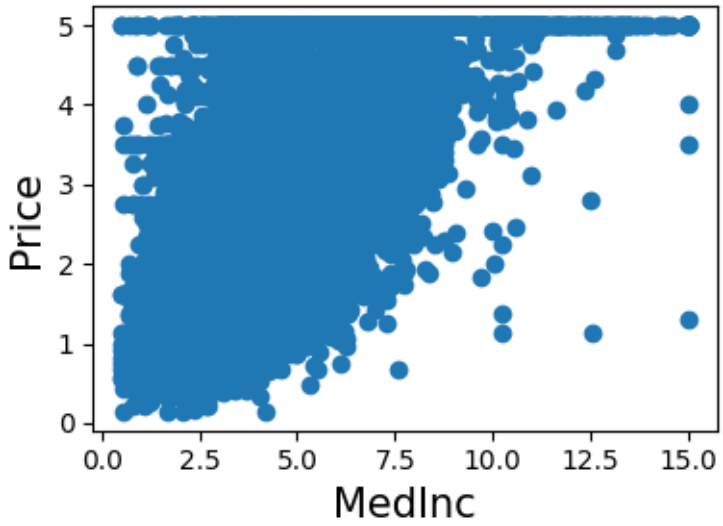
Print a histogram of the quantity to predict: price

```
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 3))
plt.hist(data.target)
plt.xlabel("price ($100k)")
plt.ylabel("count")
plt.tight_layout()
```

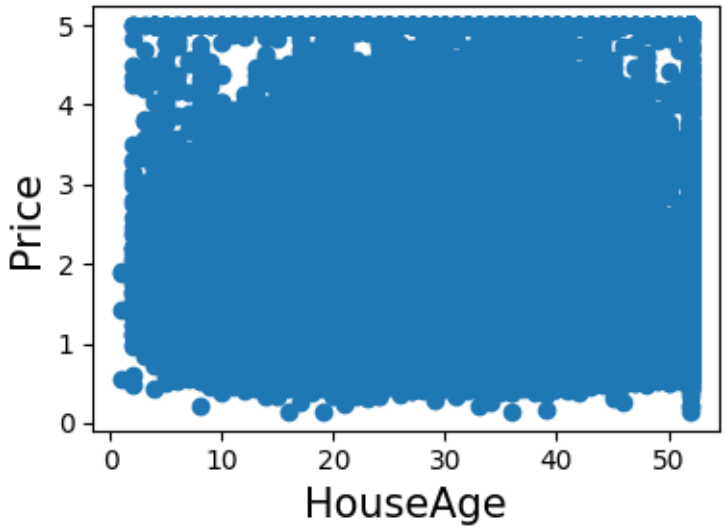


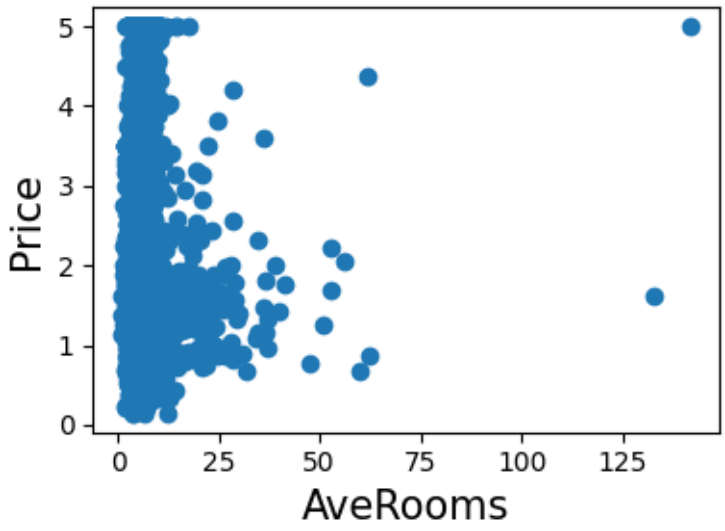
Print the join histogram for each feature

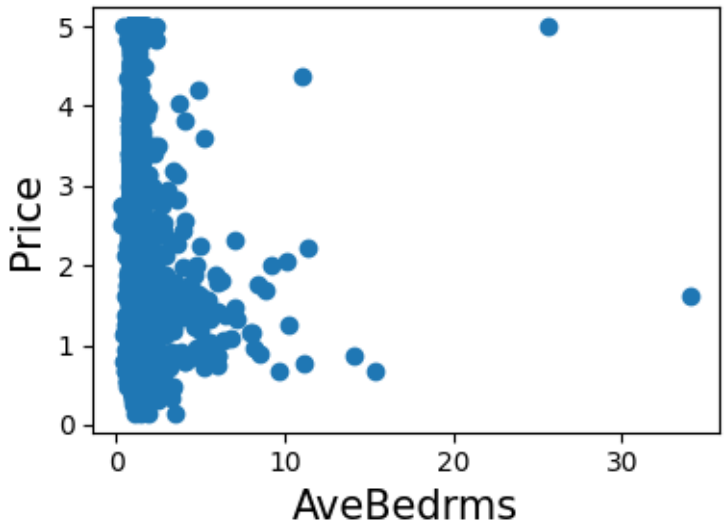
```
for index, feature_name in enumerate(data.feature_names):
    plt.figure(figsize=(4, 3))
    plt.scatter(data.data[feature_name], data.target)
    plt.ylabel("Price", size=15)
    plt.xlabel(feature_name, size=15)
    plt.tight_layout()
```

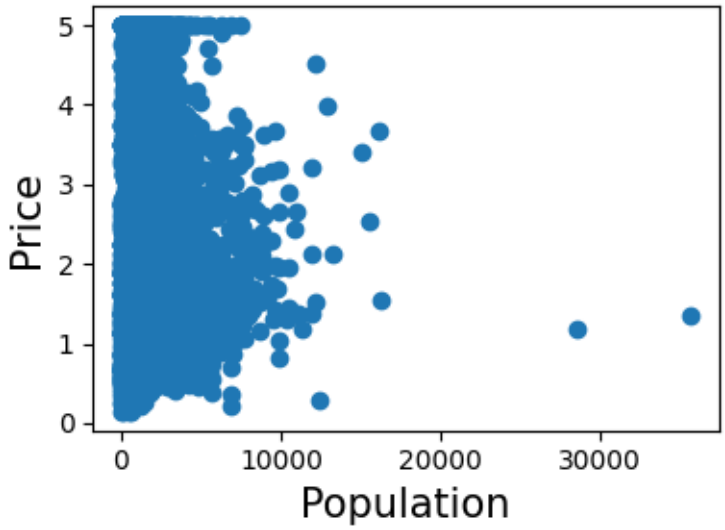


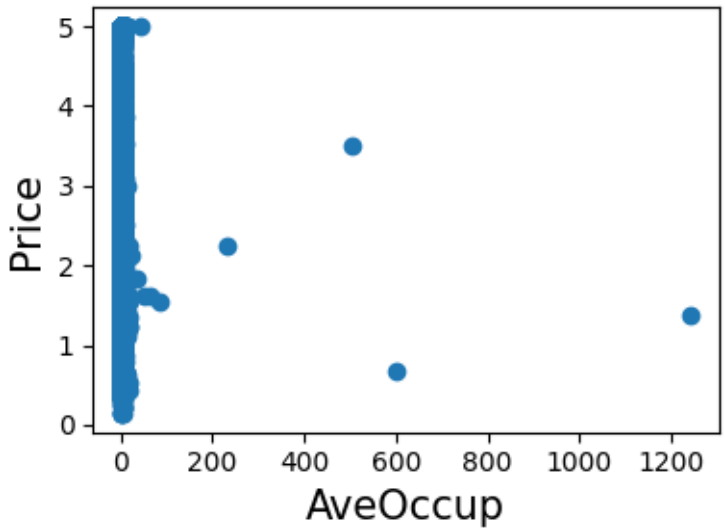
-

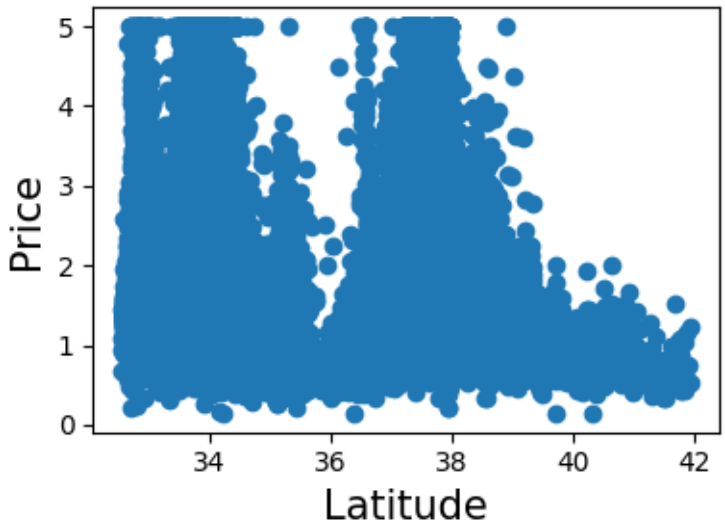

Price
HouseAge
0
1
2
3
4
5
0
10
20
30
40
50

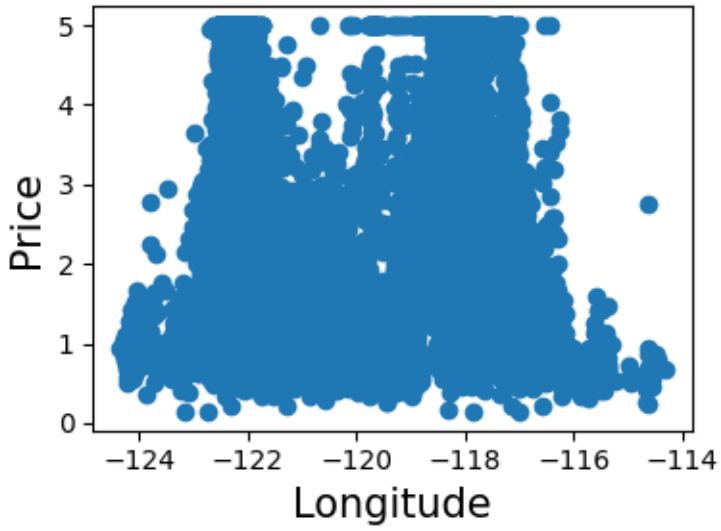

Price
AveRooms
0
1
2
3
4
5
0
25
50
75
100
125


Price
AveBedrms
0
1
2
3
4
5
0
10
20
30


Price
Population
0
1
2
3
4
5
0
10000
20000
30000


Price
AveOccup
0
1
2
3
4
5
0
200
400
600
800
1000
1200

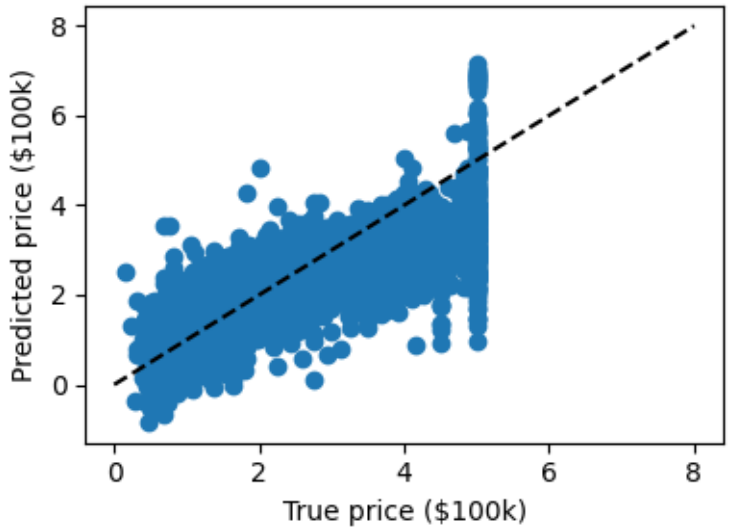

Price
Latitude
0
1
2
3
4
5
34
36
38
40
42



-

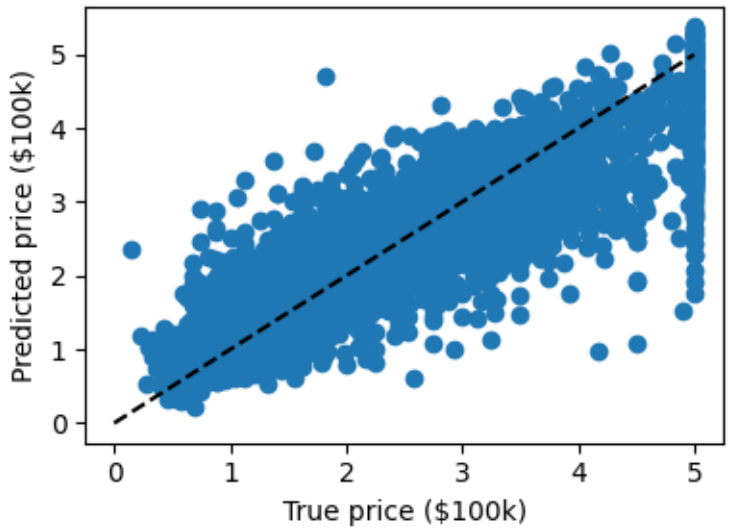
Simple prediction

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(data.data, data.target)

from sklearn.linear_model import LinearRegression

clf = LinearRegression()
clf.fit(X_train, y_train)
predicted = clf.predict(X_test)
expected = y_test

plt.figure(figsize=(4, 3))
plt.scatter(expected, predicted)
plt.plot([0, 8], [0, 8], "--k")
plt.axis("tight")
plt.xlabel("True price ($100k)")
plt.ylabel("Predicted price ($100k)")
plt.tight_layout()
```



Prediction with gradient boosted tree

```
from sklearn.ensemble import GradientBoostingRegressor

clf = GradientBoostingRegressor()
clf.fit(X_train, y_train)

predicted = clf.predict(X_test)
expected = y_test

plt.figure(figsize=(4, 3))
plt.scatter(expected, predicted)
plt.plot([0, 5], [0, 5], "--k")
plt.axis("tight")
plt.xlabel("True price ($100k)")
plt.ylabel("Predicted price ($100k)")
plt.tight_layout()
```



Print the error rate

```
import numpy as np

print(f"RMS: {np.sqrt(np.mean((predicted - expected) ** 2))!r} ")

plt.show()
```

```
RMS: np.float64(0.5314909993118918)
```

Total running time of the script: (0 minutes 4.815 seconds)

18.8.12 Nearest-neighbor prediction on iris

Plot the decision boundary of nearest neighbor decision on iris, first with a single nearest neighbor, and then using 3 nearest neighbors.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors, datasets
```

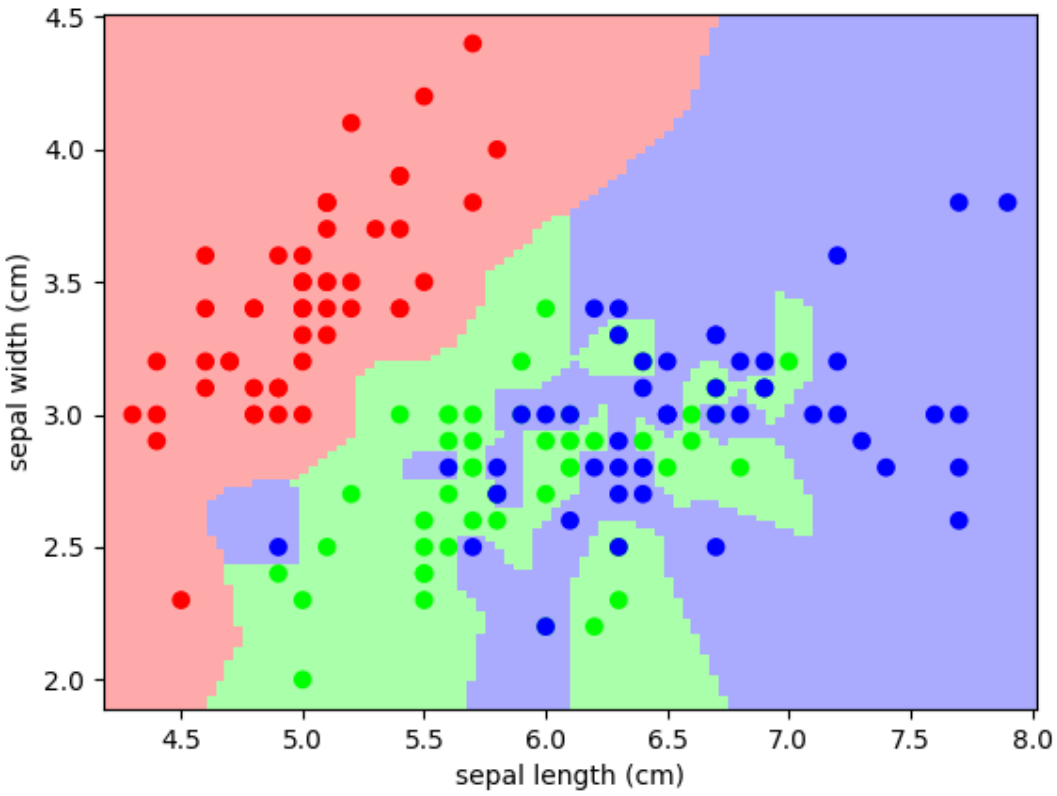
(continues on next page)

(continued from previous page)

```
from matplotlib.colors import ListedColormap

# Create color maps for 3-class classification problem, as with iris
cmap_light = ListedColormap(["#FFAAAA", "#AAFFAA", "#AAAAFF"])
cmap_bold = ListedColormap(["#FF0000", "#00FF00", "#0000FF"])

iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features. We could
# avoid this ugly slicing by using a two-dim dataset
y = iris.target

knn = neighbors.KNeighborsClassifier(n_neighbors=1)
knn.fit(X, y)

x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
```

Put the result into a color plot

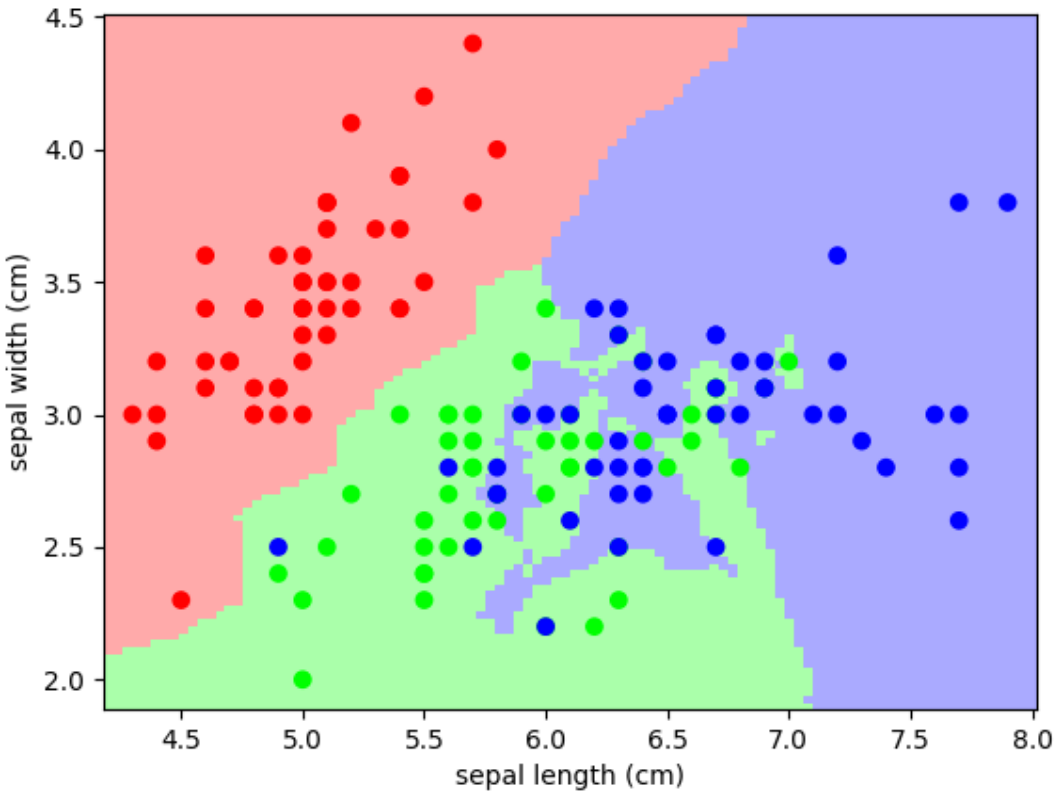
```
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
plt.xlabel("sepal length (cm)")
plt.ylabel("sepal width (cm)")
plt.axis("tight")
```




```
(np.float64(4.180808080808081), np.float64(8.019191919191918), np.float64(1.
↪8868686868686868), np.float64(4.513131313131313))
```

And now, redo the analysis with 3 neighbors

```
knn = neighbors.KNeighborsClassifier(n_neighbors=3)
knn.fit(X, y)

Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
plt.xlabel("sepal length (cm)")
plt.ylabel("sepal width (cm)")
plt.axis("tight")

plt.show()
```



Total running time of the script: (0 minutes 0.745 seconds)

18.8.13 Simple visualization and classification of the digits dataset

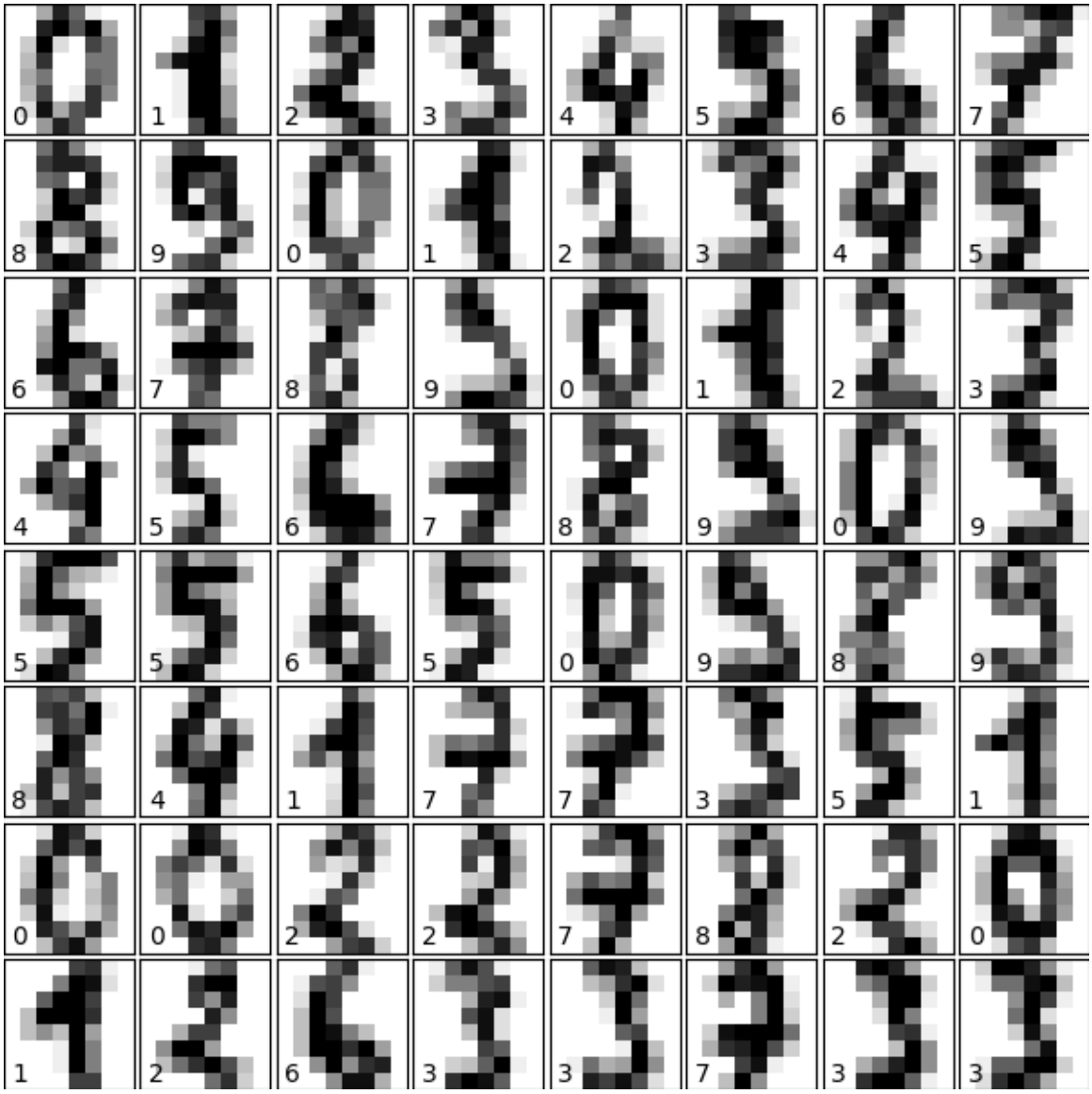
Plot the first few samples of the digits dataset and a 2D representation built using PCA, then do a simple classification

```
from sklearn.datasets import load_digits

digits = load_digits()
```

Plot the data: images of digits

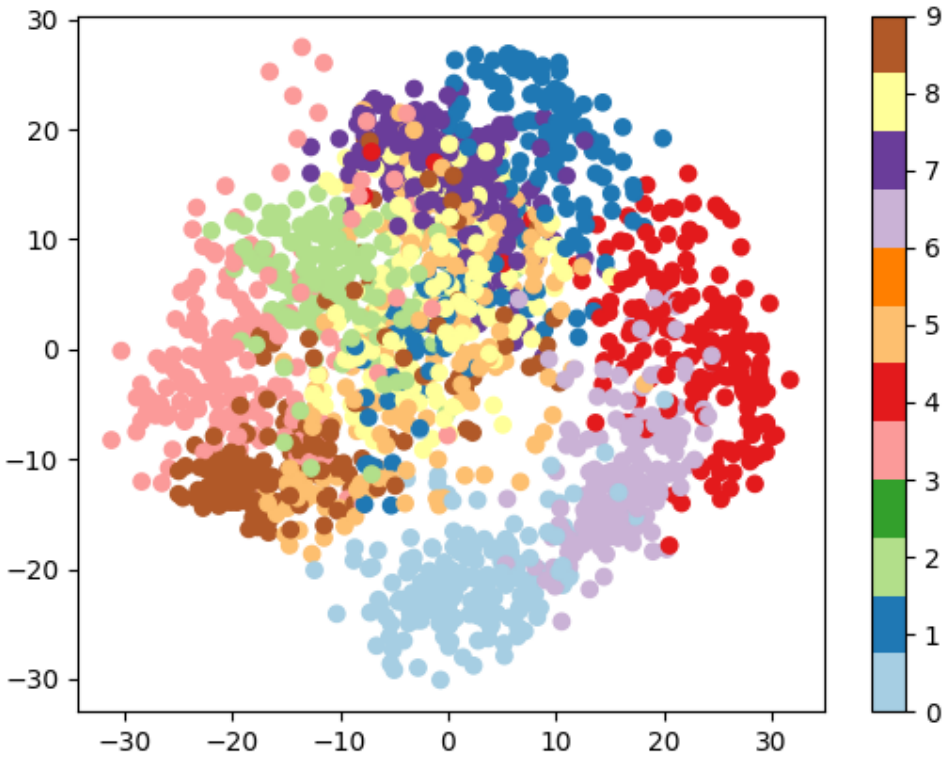
Each data in a 8x8 image

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap="binary", interpolation="nearest")
    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```



Plot a projection on the 2 first principal axis

```
plt.figure()

from sklearn.decomposition import PCA

pca = PCA(n_components=2)
proj = pca.fit_transform(digits.data)
plt.scatter(proj[:, 0], proj[:, 1], c=digits.target, cmap="Paired")
plt.colorbar()
```




```
<matplotlib.colorbar.Colorbar object at 0x7f3b1bd1ede0>
```

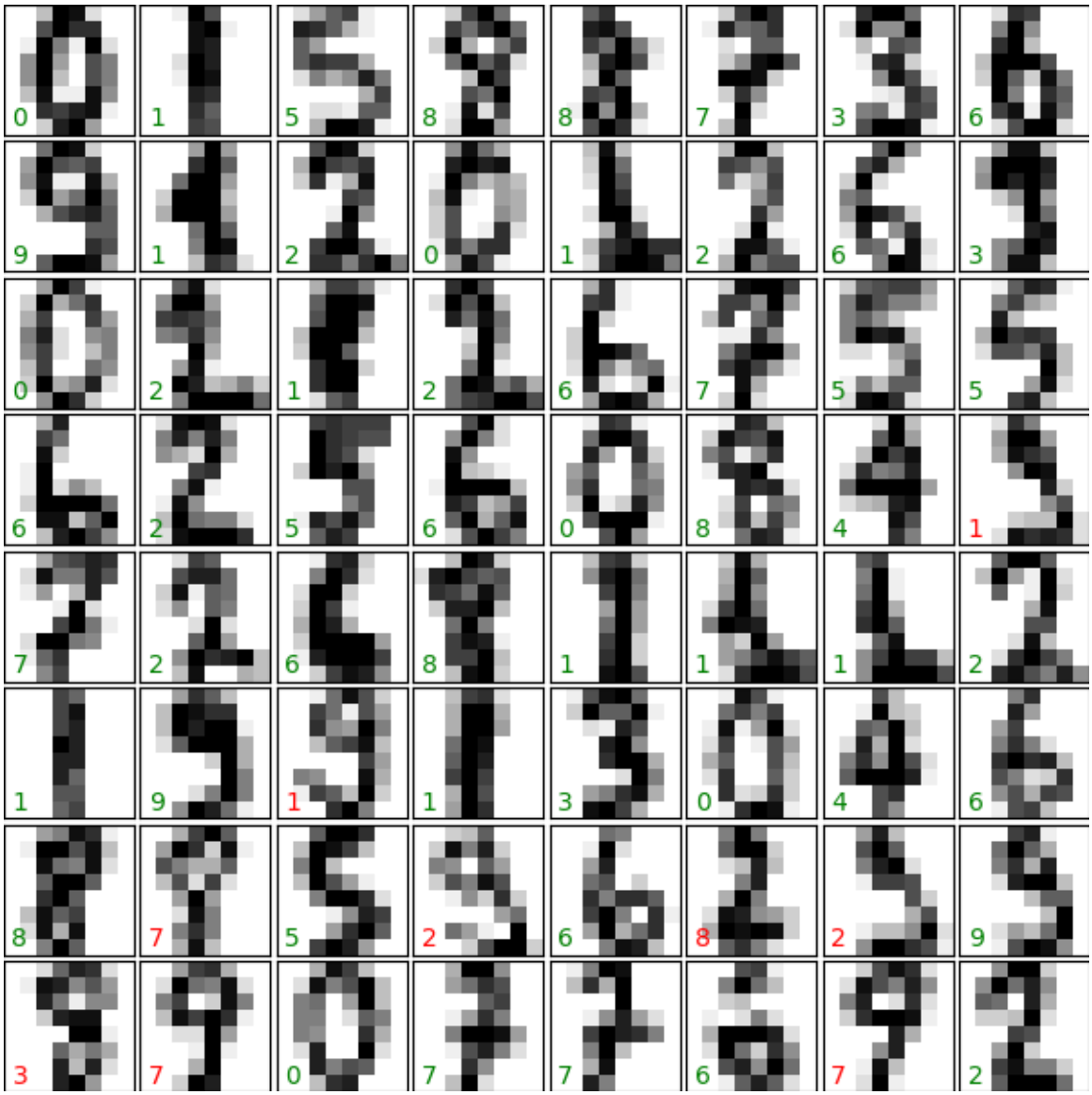
Classify with Gaussian naive Bayes

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split

# split the data into training and validation sets
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target)

# train the model
clf = GaussianNB()
clf.fit(X_train, y_train)

# use the model to predict the labels of the test data
predicted = clf.predict(X_test)
expected = y_test

# Plot the prediction
fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits: each image is 8x8 pixels
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(X_test.reshape(-1, 8, 8)[i], cmap="binary", interpolation="nearest")

    # label the image with the target value
```

(continues on next page)

(continued from previous page)

```
    if predicted[i] == expected[i]:
        ax.text(0, 7, str(predicted[i]), color="green")
    else:
        ax.text(0, 7, str(predicted[i]), color="red")
```



Quantify the performance

First print the number of correct matches

```
matches = predicted == expected
print(matches.sum())
```

```
395
```

The total number of data points

```
print(len(matches))
```

```
450
```

And now, the ration of correct predictions

```
matches.sum() / float(len(matches))
```

```
np.float64(0.8777777777777778)
```

Print the classification report

```
from sklearn import metrics

print(metrics.classification_report(expected, predicted))
```

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96        37
           1       0.83      0.85      0.84        41
           2       0.89      0.84      0.86        49
           3       0.93      0.83      0.88        47
           4       0.93      0.90      0.92        42
           5       0.89      0.95      0.92        42
           6       0.98      0.97      0.97        60
           7       0.81      0.98      0.88        47
           8       0.65      0.87      0.75        39
           9       0.97      0.63      0.76        46

    accuracy                           0.88       450
   macro avg       0.89      0.88      0.87       450
weighted avg       0.89      0.88      0.88       450
```

Print the confusion matrix

```
print(metrics.confusion_matrix(expected, predicted))

plt.show()
```

```
[[35  0  0  0  1  0  0  1  0  0]
 [ 0 35  0  0  0  0  1  1  4  0]
 [ 0  1 41  0  0  0  0  0  7  0]
 [ 0  0  2 39  0  1  0  2  2  1]
 [ 0  1  0  0 38  0  0  2  1  0]
 [ 0  0  0  0  1 40  0  1  0  0]
 [ 0  0  1  0  1  0 58  0  0  0]
 [ 0  0  0  0  0  1  0 46  0  0]
 [ 0  2  0  1  0  1  0  1 34  0]
 [ 1  3  2  2  0  2  0  3  4 29]]
```

Total running time of the script: (0 minutes 1.432 seconds)

18.8.14 The eigenfaces example: chaining PCA and SVMs

The goal of this example is to show how an unsupervised method and a supervised one can be chained for better prediction. It starts with a didactic but lengthy way of doing things, and finishes with the idiomatic approach to pipelining in scikit-learn.

Here we'll take a look at a simple facial recognition example. Ideally, we would use a dataset consisting of a subset of the Labeled Faces in the Wild data that is available with `sklearn.datasets.`

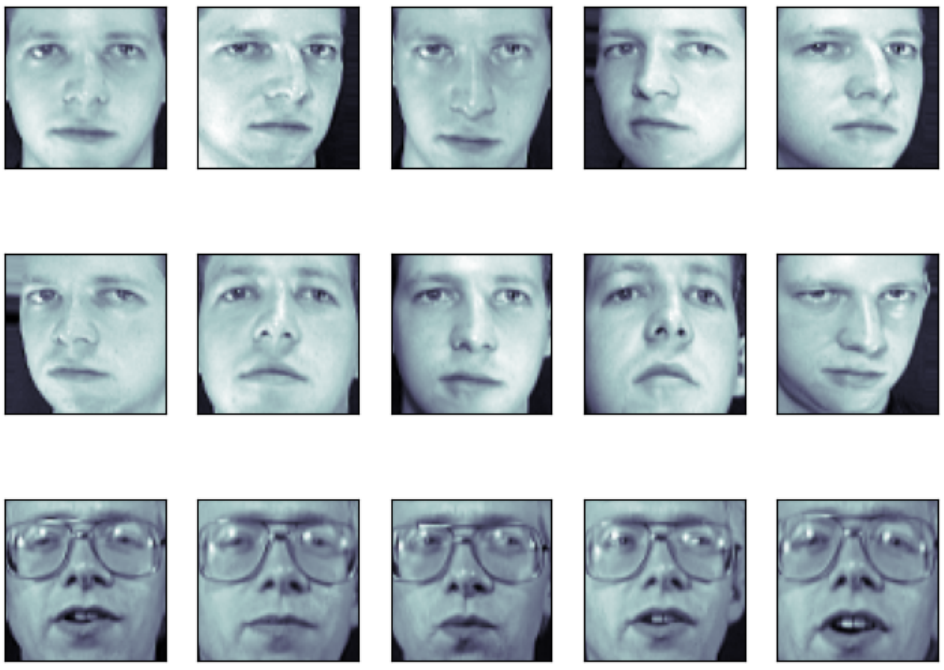
`fetch_lfw_people()`. However, this is a relatively large download (~200MB) so we will do the tutorial on a simpler, less rich dataset. Feel free to explore the LFW dataset.

```
from sklearn import datasets

faces = datasets.fetch_olivetti_faces()
faces.data.shape
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to /
↪home/runner/scikit_learn_data

(400, 4096)
```

Let's visualize these faces to see what we're working with

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8, 6))
# plot several images
for i in range(15):
    ax = fig.add_subplot(3, 5, i + 1, xticks=[], yticks=[])
    ax.imshow(faces.images[i], cmap="bone")
```



Tip

Note is that these faces have already been localized and scaled to a common size. This is an important preprocessing piece for facial recognition, and is a process that can require a large collection of training data. This can be done in scikit-learn, but the challenge is gathering a sufficient amount of training

data for the algorithm to work. Fortunately, this piece is common enough that it has been done. One good resource is OpenCV, the *Open Computer Vision Library*.

We'll perform a Support Vector classification of the images. We'll do a typical train-test split on the images:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    faces.data, faces.target, random_state=0
)

print(X_train.shape, X_test.shape)
```

```
(300, 4096) (100, 4096)
```

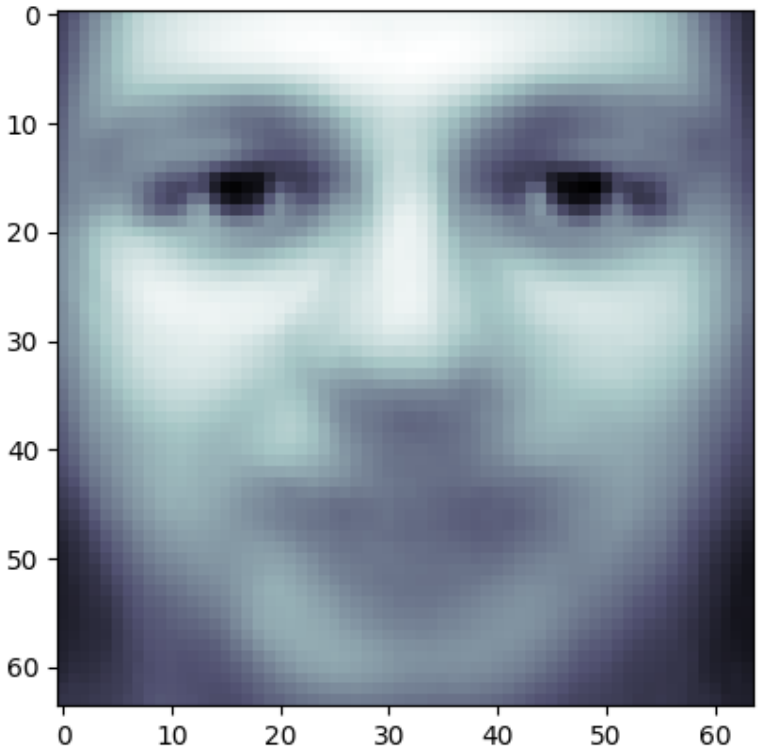
Preprocessing: Principal Component Analysis

1850 dimensions is a lot for SVM. We can use PCA to reduce these 1850 features to a manageable size, while maintaining most of the information in the dataset.

```
from sklearn import decomposition

pca = decomposition.PCA(n_components=150, whiten=True)
pca.fit(X_train)
```

One interesting part of PCA is that it computes the "mean" face, which can be interesting to examine:

```
plt.imshow(pca.mean_.reshape(faces.images[0].shape), cmap="bone")
```




```
<matplotlib.image.AxesImage object at 0x7f3b19c3eff0>
```

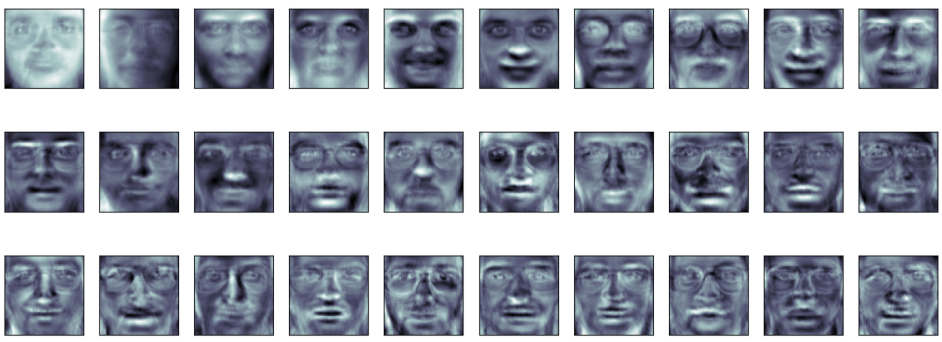
The principal components measure deviations about this mean along orthogonal axes.

```
print(pca.components_.shape)
```

```
(150, 4096)
```

It is also interesting to visualize these principal components:

```
fig = plt.figure(figsize=(16, 6))
for i in range(30):
    ax = fig.add_subplot(3, 10, i + 1, xticks=[], yticks=[])
    ax.imshow(pca.components_[i].reshape(faces.images[0].shape), cmap="bone")
```



The components ("eigenfaces") are ordered by their importance from top-left to bottom-right. We see that the first few components seem to primarily take care of lighting conditions; the remaining components pull out certain identifying features: the nose, eyes, eyebrows, etc.

With this projection computed, we can now project our original training and test data onto the PCA basis:

```
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print(X_train_pca.shape)
```

```
(300, 150)
```

```
print(X_test_pca.shape)
```

```
(100, 150)
```

These projected components correspond to factors in a linear combination of component images such that the combination approaches the original face.

Doing the Learning: Support Vector Machines

Now we'll perform support-vector-machine classification on this reduced dataset:

```
from sklearn import svm

clf = svm.SVC(C=5.0, gamma=0.001)
clf.fit(X_train_pca, y_train)
```

Finally, we can evaluate how well this classification did. First, we might plot a few of the test-cases with the labels learned from the training set:

```
import numpy as np

fig = plt.figure(figsize=(8, 6))
for i in range(15):
    ax = fig.add_subplot(3, 5, i + 1, xticks=[], yticks=[])
    ax.imshow(X_test[i].reshape(faces.images[0].shape), cmap="bone")
    y_pred = clf.predict(X_test_pca[i, np.newaxis])[0]
    color = "black" if y_pred == y_test[i] else "red"
    ax.set_title(y_pred, fontsize="small", color=color)
```

The classifier is correct on an impressive number of images given the simplicity of its learning model! Using a linear classifier on 150 features derived from the pixel-level data, the algorithm correctly identifies a large number of the people in the images.

Again, we can quantify this effectiveness using one of several measures from `sklearn.metrics`. First we can do the classification report, which shows the precision, recall and other measures of the "goodness" of the classification:

```
from sklearn import metrics

y_pred = clf.predict(X_test_pca)
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.50      0.67         6
           1       1.00      1.00      1.00         4
           2       0.50      1.00      0.67         2
           3       1.00      1.00      1.00         1
           4       0.33      1.00      0.50         1
           5       1.00      1.00      1.00         5
           6       1.00      1.00      1.00         4
           7       1.00      0.67      0.80         3
           9       1.00      1.00      1.00         1
          10       1.00      1.00      1.00         4
          11       1.00      1.00      1.00         1
          12       0.67      1.00      0.80         2
          13       1.00      1.00      1.00         3
          14       1.00      1.00      1.00         5
```

(continues on next page)

(continued from previous page)

```
          15       1.00      1.00      1.00         3
          17       1.00      1.00      1.00         6
          19       1.00      1.00      1.00         4
          20       1.00      1.00      1.00         1
          21       1.00      1.00      1.00         1
          22       1.00      1.00      1.00         2
          23       1.00      1.00      1.00         1
          24       1.00      1.00      1.00         2
          25       1.00      0.50      0.67         2
          26       1.00      0.75      0.86         4
          27       1.00      1.00      1.00         1
          28       0.67      1.00      0.80         2
          29       1.00      1.00      1.00         3
          30       1.00      1.00      1.00         4
          31       1.00      1.00      1.00         3
          32       1.00      1.00      1.00         3
          33       1.00      1.00      1.00         2
          34       1.00      1.00      1.00         3
          35       1.00      1.00      1.00         1
          36       1.00      1.00      1.00         3
          37       1.00      1.00      1.00         3
          38       1.00      1.00      1.00         1
          39       1.00      1.00      1.00         3

    accuracy                           0.94       100
   macro avg       0.95      0.96      0.94       100
weighted avg       0.97      0.94      0.94       100
```

Another interesting metric is the *confusion matrix*, which indicates how often any two items are mixed-up. The confusion matrix of a perfect classifier would only have nonzero entries on the diagonal, with zeros on the off-diagonal:

```
print(metrics.confusion_matrix(y_test, y_pred))
```

```
[[3 0 0 ... 0 0 0]
 [0 4 0 ... 0 0 0]
 [0 0 2 ... 0 0 0]
 ...
 [0 0 0 ... 3 0 0]
 [0 0 0 ... 0 1 0]
 [0 0 0 ... 0 0 3]]
```

Pipelining

Above we used PCA as a pre-processing step before applying our support vector machine classifier. Plugging the output of one estimator directly into the input of a second estimator is a commonly used pattern; for this reason scikit-learn provides a `Pipeline` object which automates this process. The above problem can be re-expressed as a pipeline as follows:

```
from sklearn.pipeline import Pipeline

clf = Pipeline(
    [
        ("pca", decomposition.PCA(n_components=150, whiten=True)),
        ("svm", svm.LinearSVC(C=1.0)),
    ]
```

(continues on next page)

(continued from previous page)

```
)

clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
print(metrics.confusion_matrix(y_pred, y_test))
plt.show()
```

```
[[4 0 0 ... 0 0 0]
 [0 4 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 ...
 [1 0 0 ... 3 0 0]
 [0 0 0 ... 0 1 0]
 [0 0 0 ... 0 0 3]]
```

A Note on Facial Recognition

Here we have used PCA "eigenfaces" as a pre-processing step for facial recognition. The reason we chose this is because PCA is a broadly-applicable technique, which can be useful for a wide array of data types. Research in the field of facial recognition in particular, however, has shown that other more specific feature extraction methods are can be much more effective.

Total running time of the script: (0 minutes 5.572 seconds)

18.8.15 Example of linear and non-linear models

This is an example plot from the tutorial which accompanies an explanation of the support vector machine GUI.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm


rng = np.random.default_rng(27446968)
```

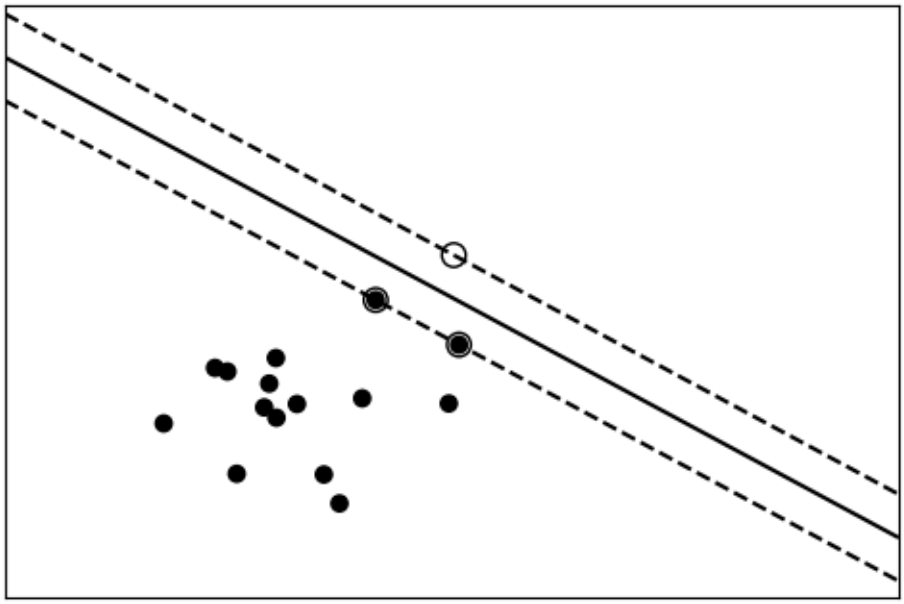
data that is linearly separable

```
def linear_model(rseed=42, n_samples=30):
    "Generate data according to a linear model"
    np.random.seed(rseed)

    data = np.random.normal(0, 10, (n_samples, 2))
    data[: n_samples // 2] -= 15
    data[n_samples // 2 :] += 15

    labels = np.ones(n_samples)
    labels[: n_samples // 2] = -1

    return data, labels


X, y = linear_model()
```

(continues on next page)

(continued from previous page)

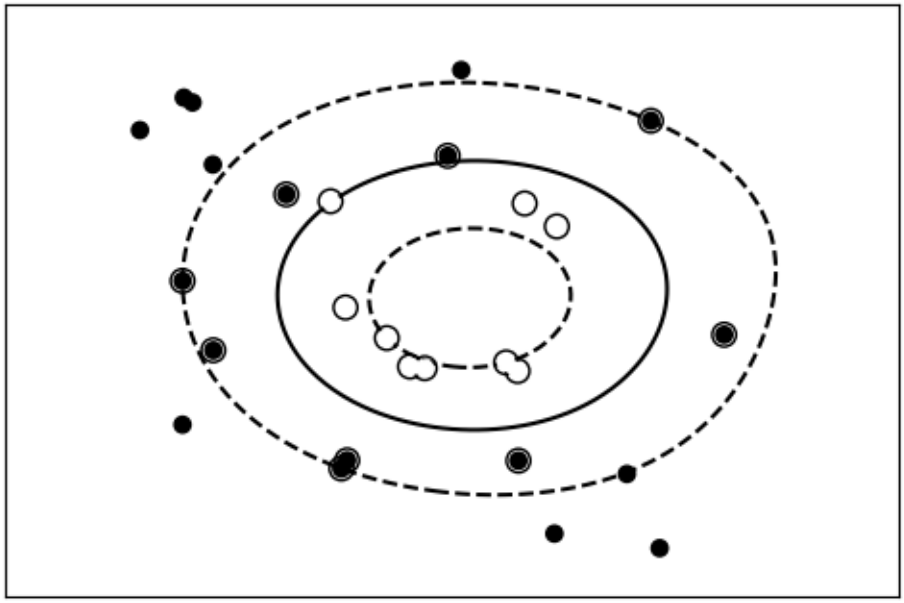
```
clf = svm.SVC(kernel="linear")
clf.fit(X, y)

plt.figure(figsize=(6, 4))
ax = plt.subplot(111, xticks=[], yticks=[])
ax.scatter(X[:, 0], X[:, 1], c=y, cmap="bone")

ax.scatter(
    clf.support_vectors_[:, 0],
    clf.support_vectors_[:, 1],
    s=80,
    edgecolors="k",
    facecolors="none",
)

delta = 1
y_min, y_max = -50, 50
x_min, x_max = -50, 50
x = np.arange(x_min, x_max + delta, delta)
y = np.arange(y_min, y_max + delta, delta)
X1, X2 = np.meshgrid(x, y)
Z = clf.decision_function(np.c_[X1.ravel(), X2.ravel()])
Z = Z.reshape(X1.shape)

ax.contour(
    X1, X2, Z, [-1.0, 0.0, 1.0], colors="k", linestyles=["dashed", "solid", "dashed"]
)
```



```
<matplotlib.contour.QuadContourSet object at 0x7f3b1c1e5fa0>
```

data with a non-linear separation

```
def nonlinear_model(rseed=27446968, n_samples=30):
    rng = np.random.default_rng(rseed)

    radius = 40 * rng.random(n_samples)
    far_pts = radius > 20
    radius[far_pts] *= 1.2
    radius[~far_pts] *= 1.1

    theta = rng.random(n_samples) * np.pi * 2

    data = np.empty((n_samples, 2))
    data[:, 0] = radius * np.cos(theta)
    data[:, 1] = radius * np.sin(theta)

    labels = np.ones(n_samples)
    labels[far_pts] = -1

    return data, labels


X, y = nonlinear_model()
clf = svm.SVC(kernel="rbf", gamma=0.001, coef0=0, degree=3)
clf.fit(X, y)

plt.figure(figsize=(6, 4))
ax = plt.subplot(1, 1, 1, xticks=[], yticks=[])
ax.scatter(X[:, 0], X[:, 1], c=y, cmap="bone", zorder=2)

ax.scatter(
    clf.support_vectors_[:, 0],
    clf.support_vectors_[:, 1],
    s=80,
    edgecolors="k",
    facecolors="none",
)

delta = 1
y_min, y_max = -50, 50
x_min, x_max = -50, 50
x = np.arange(x_min, x_max + delta, delta)
y = np.arange(y_min, y_max + delta, delta)
X1, X2 = np.meshgrid(x, y)
Z = clf.decision_function(np.c_[X1.ravel(), X2.ravel()])
Z = Z.reshape(X1.shape)

ax.contour(
    X1,
    X2,
    Z,
    [-1.0, 0.0, 1.0],
    colors="k",
    linestyles=["dashed", "solid", "dashed"],
    zorder=1,
)

plt.show()
```



Total running time of the script: (0 minutes 0.064 seconds)

18.8.16 Bias and variance of polynomial fit

Demo overfitting, underfitting, and validation and learning curves with polynomial regression.

Fit polynomes of different degrees to a dataset: for too small a degree, the model *underfits*, while for too large a degree, it overfits.

```
import numpy as np
import matplotlib.pyplot as plt


def generating_func(x, rng=None, error=0.5):
    rng = np.random.default_rng(rng)
    return rng.normal(10 - 1.0 / (x + 0.1), error)
```

A polynomial regression

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

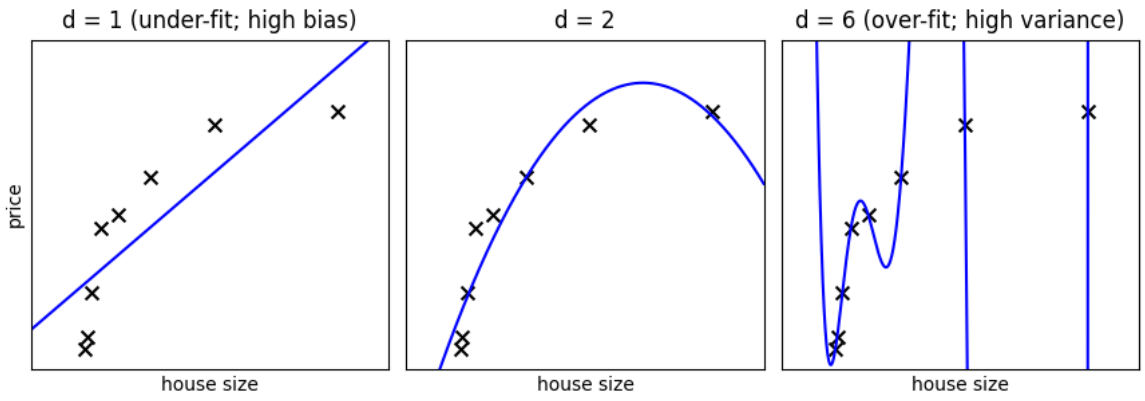
A simple figure to illustrate the problem

```
n_samples = 8

rng = np.random.default_rng(27446968)
x = 10 ** np.linspace(-2, 0, n_samples)
y = generating_func(x, rng=rng)

x_test = np.linspace(-0.2, 1.2, 1000)
```

(continues on next page)

(continued from previous page)

```
titles = ["d = 1 (under-fit; high bias)", "d = 2", "d = 6 (over-fit; high variance)"]
degrees = [1, 2, 6]

fig = plt.figure(figsize=(9, 3.5))
fig.subplots_adjust(left=0.06, right=0.98, bottom=0.15, top=0.85, wspace=0.05)

for i, d in enumerate(degrees):
    ax = fig.add_subplot(131 + i, xticks=[], yticks=[])
    ax.scatter(x, y, marker="x", c="k", s=50)

    model = make_pipeline(PolynomialFeatures(d), LinearRegression())
    model.fit(x[:, np.newaxis], y)
    ax.plot(x_test, model.predict(x_test[:, np.newaxis]), "-b")

    ax.set_xlim(-0.2, 1.2)
    ax.set_ylim(0, 12)
    ax.set_xlabel("house size")
    if i == 0:
        ax.set_ylabel("price")

    ax.set_title(titles[i])
```



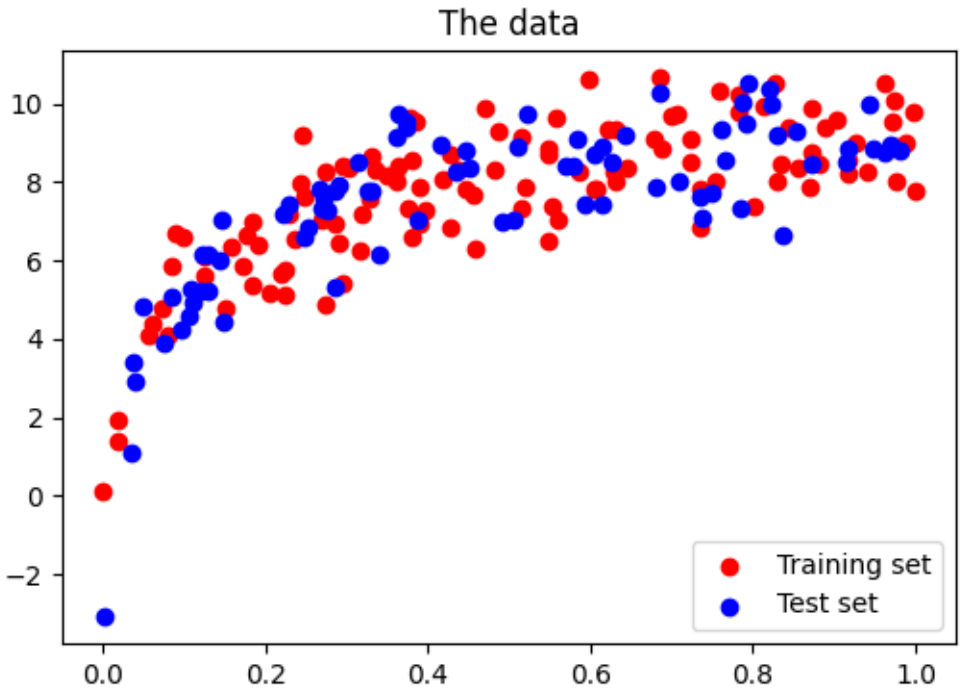
Generate a larger dataset

```
from sklearn.model_selection import train_test_split

n_samples = 200
test_size = 0.4
error = 1.0

# randomly sample the data
x = rng.random(n_samples)
y = generating_func(x, rng=rng, error=error)

# split into training, validation, and testing sets.
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size)

# show the training and validation sets
plt.figure(figsize=(6, 4))
plt.scatter(x_train, y_train, color="red", label="Training set")
```

(continues on next page)

(continued from previous page)

```
plt.scatter(x_test, y_test, color="blue", label="Test set")
plt.title("The data")
plt.legend(loc="best")
```




```
<matplotlib.legend.Legend object at 0x7f3b19d36510>
```

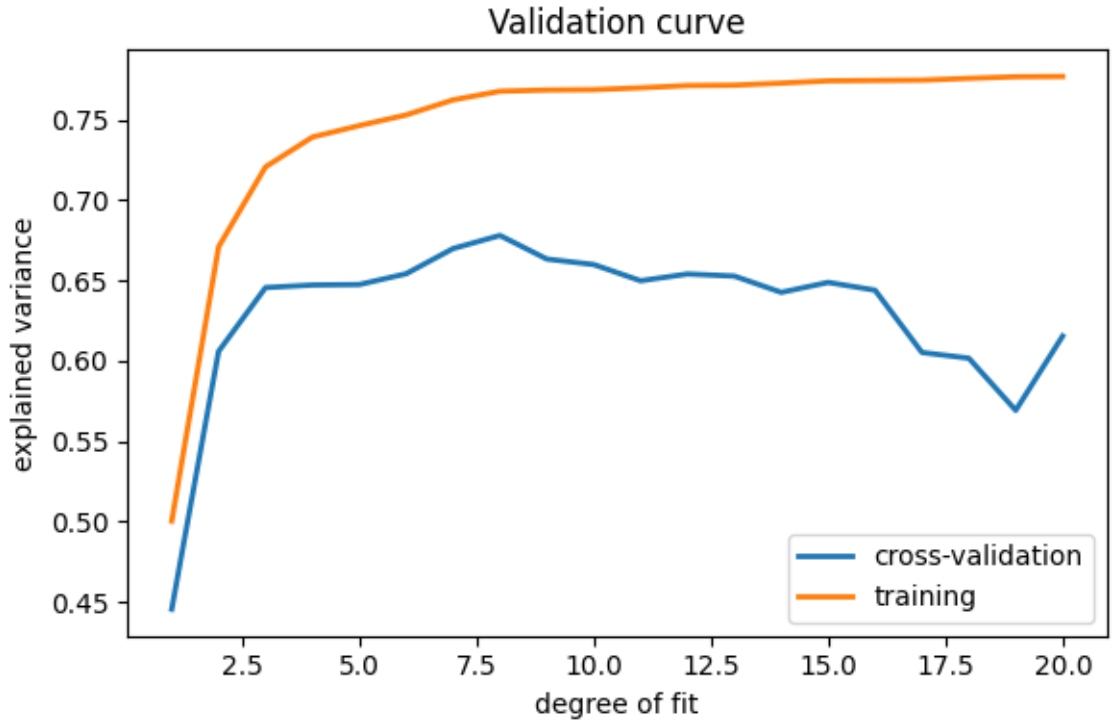
Plot a validation curve

```
from sklearn.model_selection import validation_curve

degrees = list(range(1, 21))

model = make_pipeline(PolynomialFeatures(), LinearRegression())

# The parameter to vary is the "degrees" on the pipeline step
# "polynomialfeatures"
train_scores, validation_scores = validation_curve(
    model,
    x[:, np.newaxis],
    y,
    param_name="polynomialfeatures__degree",
    param_range=degrees,
)

# Plot the mean train error and validation error across folds
plt.figure(figsize=(6, 4))
plt.plot(degrees, validation_scores.mean(axis=1), lw=2, label="cross-validation")
plt.plot(degrees, train_scores.mean(axis=1), lw=2, label="training")

plt.legend(loc="best")
plt.xlabel("degree of fit")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("explained variance")
plt.title("Validation curve")
plt.tight_layout()
```



Learning curves

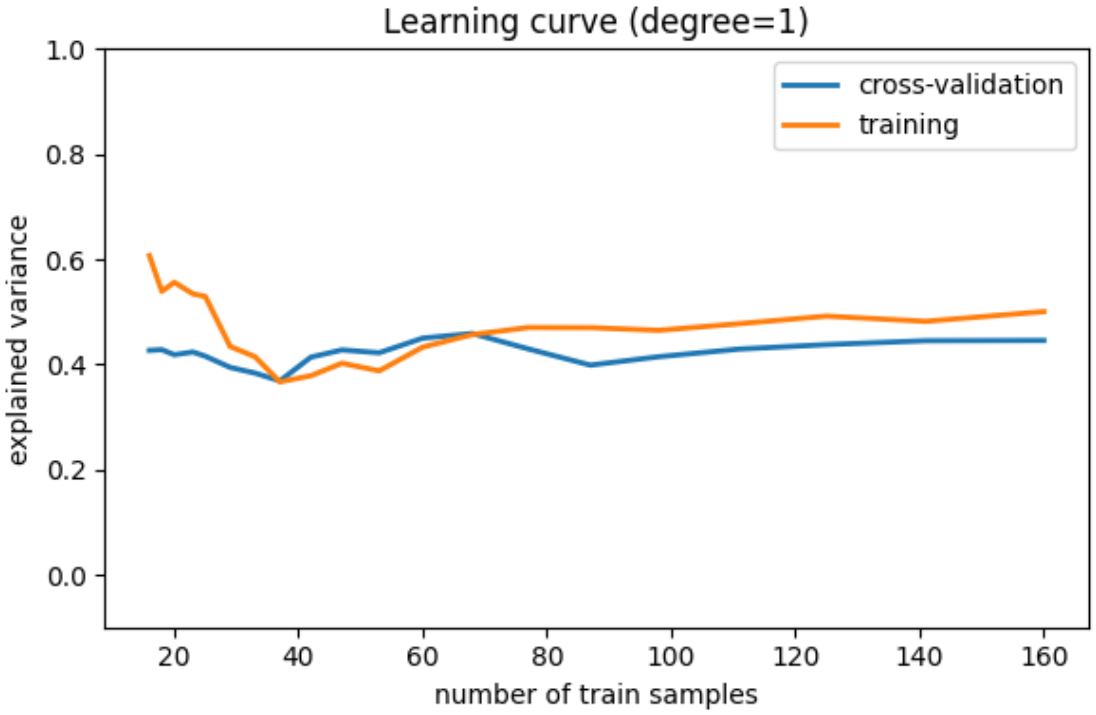
Plot train and test error with an increasing number of samples

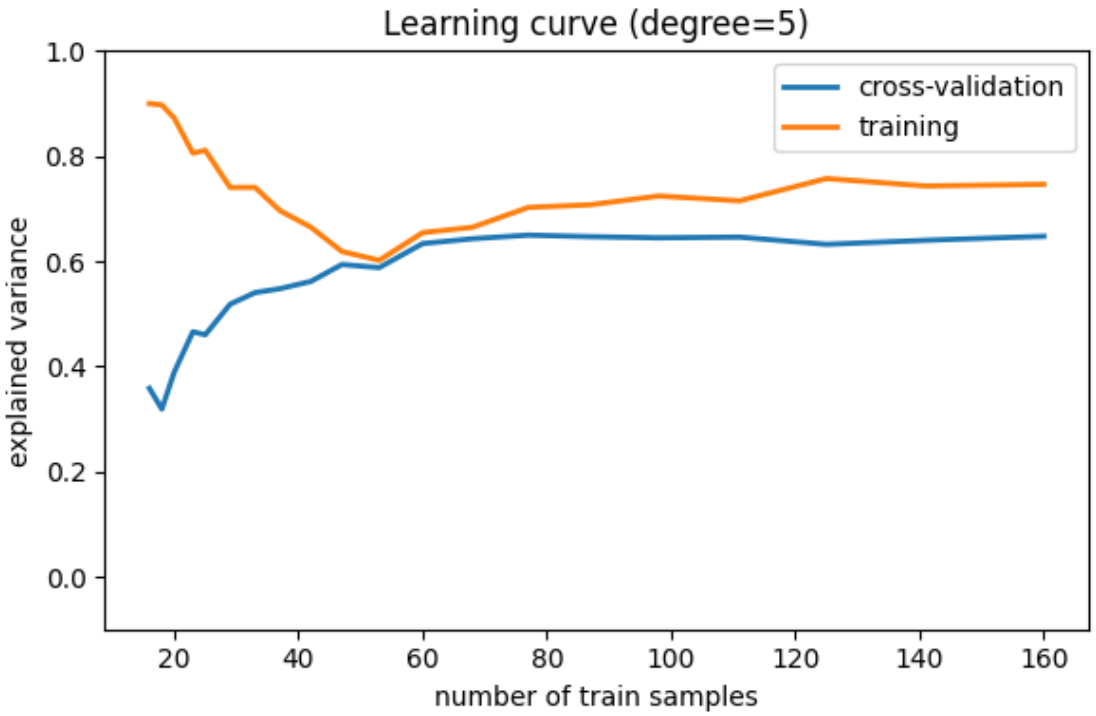
```
# A learning curve for d=1, 5, 15
for d in [1, 5, 15]:
    model = make_pipeline(PolynomialFeatures(degree=d), LinearRegression())

    from sklearn.model_selection import learning_curve

    train_sizes, train_scores, validation_scores = learning_curve(
        model, x[:, np.newaxis], y, train_sizes=np.logspace(-1, 0, 20)
    )

    # Plot the mean train error and validation error across folds
    plt.figure(figsize=(6, 4))
    plt.plot(
        train_sizes, validation_scores.mean(axis=1), lw=2, label="cross-validation"
    )
    plt.plot(train_sizes, train_scores.mean(axis=1), lw=2, label="training")
    plt.ylim(ymin=-0.1, ymax=1)

    plt.legend(loc="best")
    plt.xlabel("number of train samples")
    plt.ylabel("explained variance")
    plt.title(f"Learning curve (degree={d})")
    plt.tight_layout()
```

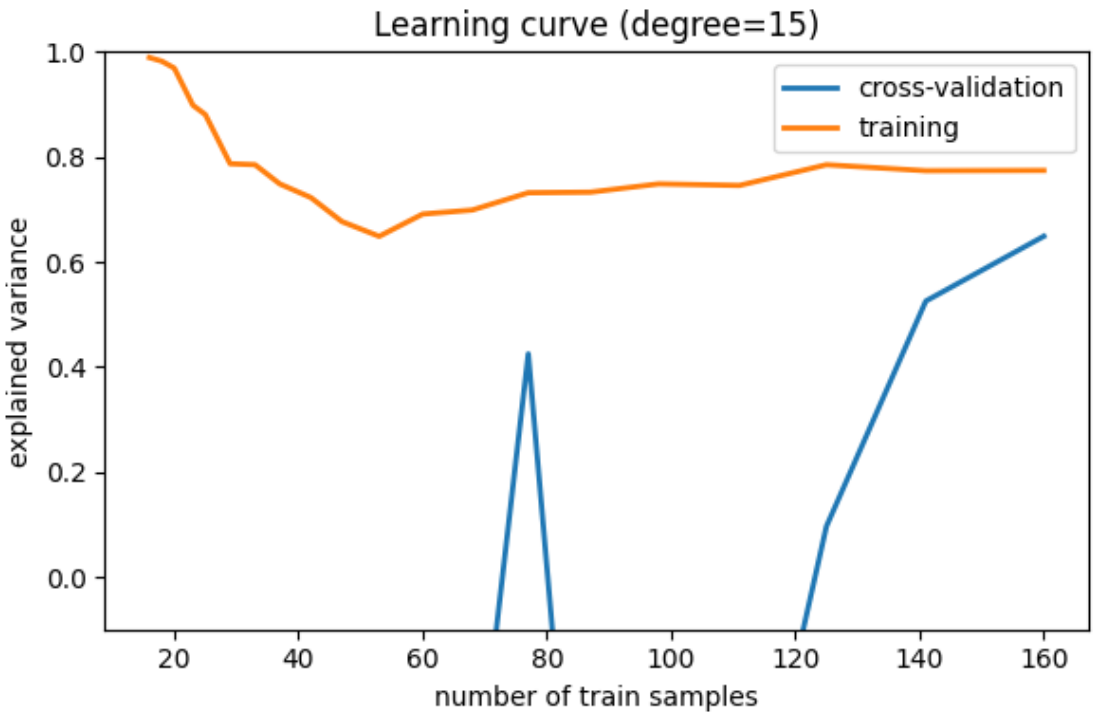
(continues on next page)

(continued from previous page)

```
plt.show()
```

- 

- 

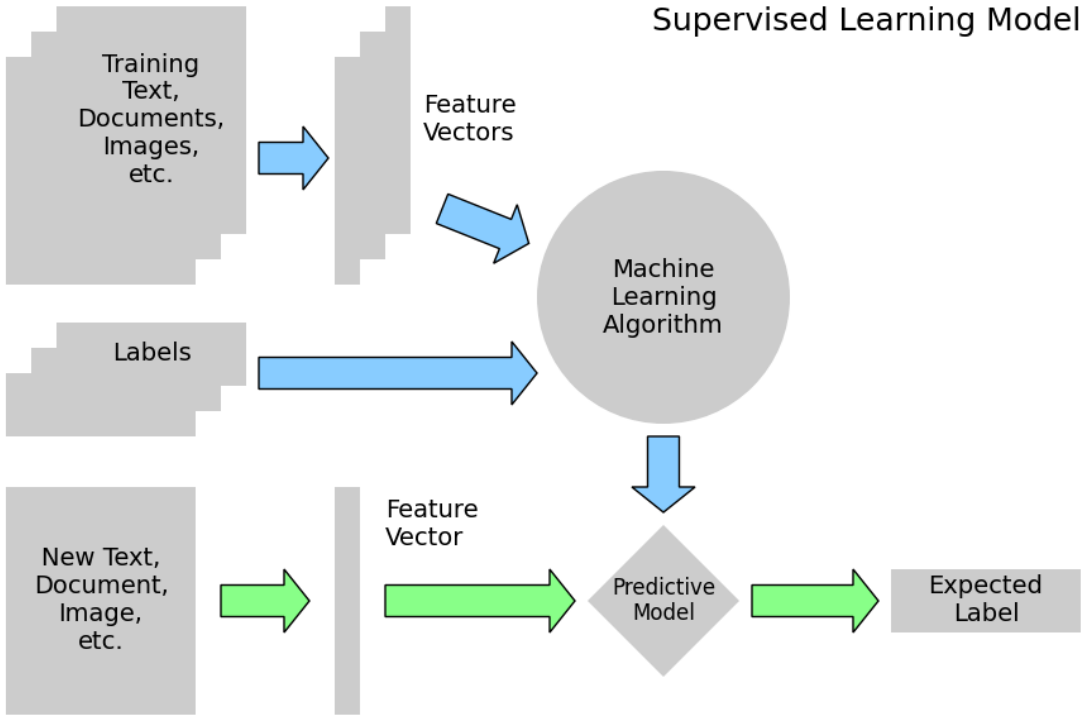


-

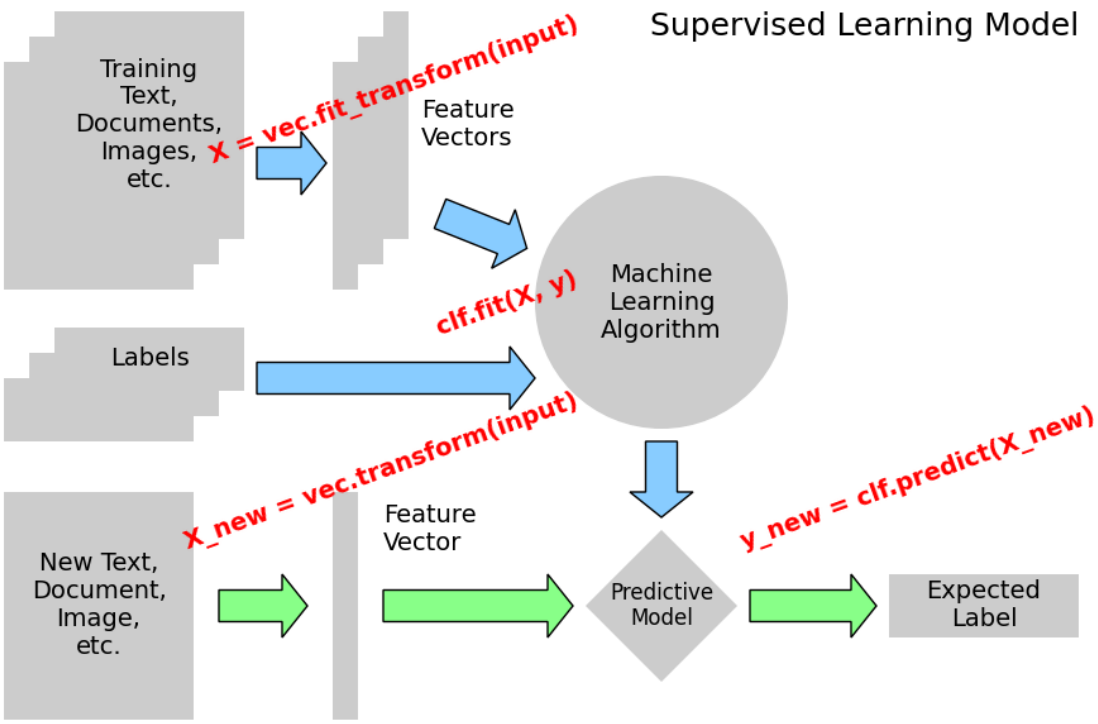
Total running time of the script: (0 minutes 1.714 seconds)

18.8.17 Tutorial Diagrams

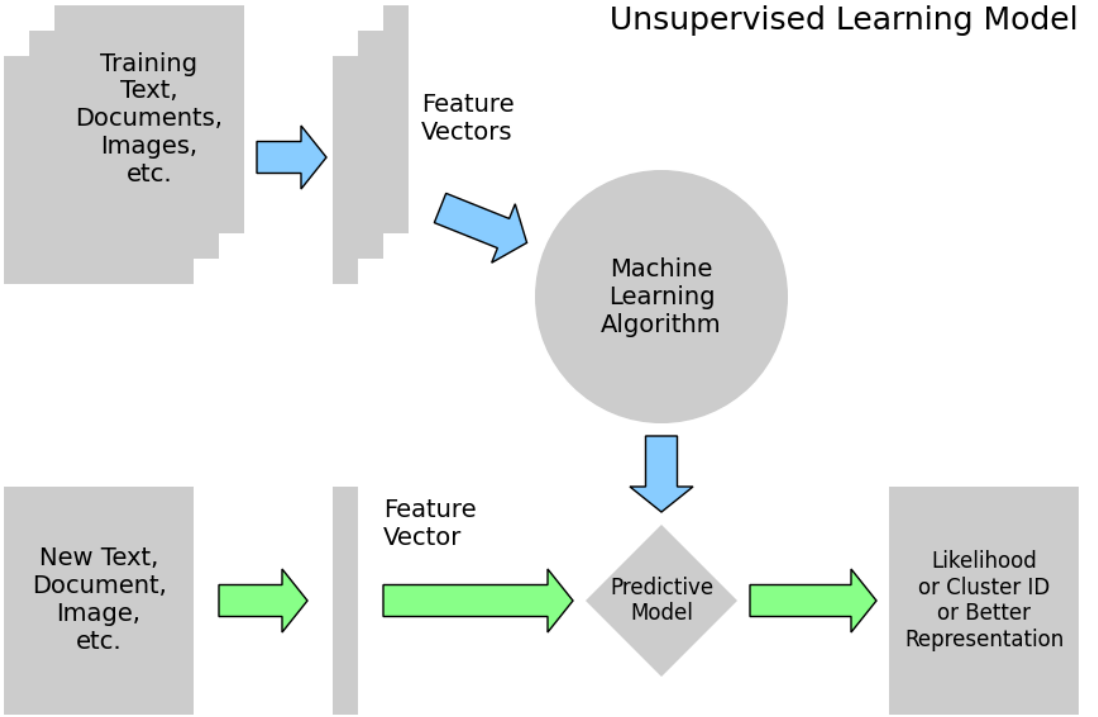
This script plots the flow-charts used in the scikit-learn tutorials.



-



-



-

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, Rectangle, Polygon, Arrow, FancyArrow


def create_base(box_bg="#CCCCCC", arrow1="#88CCFF", arrow2="#88FF88",␣
↪supervised=True):
    fig = plt.figure(figsize=(9, 6), facecolor="w")
```

(continues on next page)

(continued from previous page)

```
    ax = plt.axes((0, 0, 1, 1), xticks=[], yticks=[], frameon=False)
    ax.set_xlim(0, 9)
    ax.set_ylim(0, 6)

    patches = [
        Rectangle((0.3, 3.6), 1.5, 1.8, zorder=1, fc=box_bg),
        Rectangle((0.5, 3.8), 1.5, 1.8, zorder=2, fc=box_bg),
        Rectangle((0.7, 4.0), 1.5, 1.8, zorder=3, fc=box_bg),
        Rectangle((2.9, 3.6), 0.2, 1.8, fc=box_bg),
        Rectangle((3.1, 3.8), 0.2, 1.8, fc=box_bg),
        Rectangle((3.3, 4.0), 0.2, 1.8, fc=box_bg),
        Rectangle((0.3, 0.2), 1.5, 1.8, fc=box_bg),
        Rectangle((2.9, 0.2), 0.2, 1.8, fc=box_bg),
        Circle((5.5, 3.5), 1.0, fc=box_bg),
        Polygon([[5.5, 1.7], [6.1, 1.1], [5.5, 0.5], [4.9, 1.1]], fc=box_bg),
        FancyArrow(
            2.3, 4.6, 0.35, 0, fc=arrow1, width=0.25, head_width=0.5, head_length=0.2
        ),
        FancyArrow(
            3.75, 4.2, 0.5, -0.2, fc=arrow1, width=0.25, head_width=0.5, head_
↪length=0.2
        ),
        FancyArrow(
            5.5, 2.4, 0, -0.4, fc=arrow1, width=0.25, head_width=0.5, head_length=0.2
        ),
        FancyArrow(
            2.0, 1.1, 0.5, 0, fc=arrow2, width=0.25, head_width=0.5, head_length=0.2
        ),
        FancyArrow(
            3.3, 1.1, 1.3, 0, fc=arrow2, width=0.25, head_width=0.5, head_length=0.2
        ),
        FancyArrow(
            6.2, 1.1, 0.8, 0, fc=arrow2, width=0.25, head_width=0.5, head_length=0.2
        ),
    ]

    if supervised:
        patches += [
            Rectangle((0.3, 2.4), 1.5, 0.5, zorder=1, fc=box_bg),
            Rectangle((0.5, 2.6), 1.5, 0.5, zorder=2, fc=box_bg),
            Rectangle((0.7, 2.8), 1.5, 0.5, zorder=3, fc=box_bg),
            FancyArrow(
                2.3, 2.9, 2.0, 0, fc=arrow1, width=0.25, head_width=0.5, head_
↪length=0.2
            ),
            Rectangle((7.3, 0.85), 1.5, 0.5, fc=box_bg),
        ]
    else:
        patches += [Rectangle((7.3, 0.2), 1.5, 1.8, fc=box_bg)]

    for p in patches:
        ax.add_patch(p)

    plt.text(
        1.45,
        4.9,
```

(continues on next page)

(continued from previous page)

```
        "Training\nText,\nDocuments,\nImages,\netc.",
        ha="center",
        va="center",
        fontsize=14,
    )

    plt.text(3.6, 4.9, "Feature\nVectors", ha="left", va="center", fontsize=14)

    plt.text(
        5.5, 3.5, "Machine\nLearning\nAlgorithm", ha="center", va="center",␣
↪fontsize=14
    )

    plt.text(
        1.05,
        1.1,
        "New Text,\nDocument,\nImage,\netc.",
        ha="center",
        va="center",
        fontsize=14,
    )

    plt.text(3.3, 1.7, "Feature\nVector", ha="left", va="center", fontsize=14)

    plt.text(5.5, 1.1, "Predictive\nModel", ha="center", va="center", fontsize=12)

    if supervised:
        plt.text(1.45, 3.05, "Labels", ha="center", va="center", fontsize=14)

        plt.text(8.05, 1.1, "Expected\nLabel", ha="center", va="center", fontsize=14)
        plt.text(
            8.8, 5.8, "Supervised Learning Model", ha="right", va="top", fontsize=18
        )

    else:
        plt.text(
            8.05,
            1.1,
            "Likelihood\nor Cluster ID\nor Better\nRepresentation",
            ha="center",
            va="center",
            fontsize=12,
        )
        plt.text(
            8.8, 5.8, "Unsupervised Learning Model", ha="right", va="top", fontsize=18
        )


def plot_supervised_chart(annotate=False):
    create_base(supervised=True)
    if annotate:
        fontdict = {"color": "r", "weight": "bold", "size": 14}
        plt.text(
            1.9,
            4.55,
            "X = vec.fit_transform(input)",
```

(continues on next page)

(continued from previous page)

```
            fontdict=fontdict,
            rotation=20,
            ha="left",
            va="bottom",
        )
        plt.text(
            3.7,
            3.2,
            "clf.fit(X, y)",
            fontdict=fontdict,
            rotation=20,
            ha="left",
            va="bottom",
        )
        plt.text(
            1.7,
            1.5,
            "X_new = vec.transform(input)",
            fontdict=fontdict,
            rotation=20,
            ha="left",
            va="bottom",
        )
        plt.text(
            6.1,
            1.5,
            "y_new = clf.predict(X_new)",
            fontdict=fontdict,
            rotation=20,
            ha="left",
            va="bottom",
        )


def plot_unsupervised_chart():
    create_base(supervised=False)


if __name__ == "__main__":
    plot_supervised_chart(False)
    plot_supervised_chart(True)
    plot_unsupervised_chart()
    plt.show()
```

Total running time of the script: (0 minutes 0.197 seconds)

See also

Going further

- The documentation of scikit-learn is very complete and didactic.
- Introduction to Machine Learning with Python, by Sarah Guido, Andreas Müller (notebooks available here).

Part IV

About the Scientific Python Lectures

CHAPTER 19

About the Scientific Python Lectures

Release: 2025.1rc0.dev0

The lectures are archived on zenodo: http://dx.doi.org/10.5281/zenodo.594102

All code and material is licensed under a Creative Commons Attribution 4.0 International License (CC-by) http://creativecommons.org/licenses/by/4.0/

19.1 Authors

19.1.1 Editors

- K. Jarrod Millman
- Stéfan van der Walt
- Gaël Varoquaux
- Emmanuelle Gouillart
- Olav Vahtras
- Pierre de Buyl

19.1.2 Chapter authors

Listed by alphabetical order.

- Christopher Burns
- Adrian Chauve
- Robert Cimrman
- Christophe Combelles
- André Espaze
- Emmanuelle Gouillart
- Mike Müller
- Fabian Pedregosa
- Didrik Pinte
- Nicolas Rougier
- Gaël Varoquaux
- Pauli Virtanen
- Zbigniew Jędrzejewski-Szmek
- Valentin Haenel (editor from 2011 to 2015)

19.1.3 Additional Contributions

Listed by alphabetical order

- Osayd Abdu
- arunpersaud
- Ross Barnowski

- Sebastian Berg
- Lilian Besson
- Matthieu Boileau
- Joris Van den Bossche
- Michael Boyle
- Matthew Brett
- BSGalvan
- Lars Buitinck
- Pierre de Buyl
- Ozan Çağlayan
- Lawrence Chan
- Adrien Chauve
- Robert Cimrman
- Christophe Combelles
- David Cournapeau
- Dave
- dogacan dugmeci
- Török Edwin
- egens
- Andre Espaze
- André Espaze
- Loïc Estève
- Corey Farwell
- Tim Gates
- Stuart Geiger
- Olivier Georg
- Daniel Gerigk
- Robert Gieseke
- Philip Gillißen
- Ralf Gommers
- Emmanuelle Gouillart
- Julia Gustavsen
- Omar Gutiérrez
- Matt Haberland
- Valentin Haenel
- Pierre Haessig
- Bruno Hanzen
- Michael Hartmann
- Jonathan Helmus
- Andreas Hilboll
- Himanshu
- Julian Hofer
- Tim Hoffmann
- B. Hohl
- Tarek Hoteit
- Gert-Ludwig Ingold
- Zbigniew Jędrzejewski-Szmek
- Thouis (Ray) Jones
- jorgeprietoarranz
- josephsalmon
- Greg Kiar
- kikocorreoso
- Vince Knight
- LFP6
- Manuel López-Ibáñez
- Marco Mangan
- Nicola Masarone
- John McLaughlin
- mhemantha
- michelemaroni89
- K. Jarrod Millman
- Mohammad
- Zachary Moon
- Mike Mueller
- negm
- John B Nelson
- nicoguaro
- Sergio Oller
- Theofilos Papapanagiotou
- patniharshit
- Fabian Pedregosa
- Philippe Pepiot
- Tiago M. D. Pereira
- Nicolas Pettiaux
- Didrik Pinte
- Evgeny Pogrebnyak
- reverland
- Maximilien Riehl

- Kristian Rother
- Nicolas P. Rougier
- Pamphile Roy
- Rutzmoser
- Sander
- João Felipe Santos
- Mark Setchell
- Helen Sherwood-Taylor
- Shoeboxam
- Simon
- solarjoe
- ssmiller
- Scott Staniewicz
- strpeter
- surfer190
- Bartosz Telenczuk
- tommyod
- Wes Turner
- Akihiro Uchida
- Utkarsh Upadhyay
- Olav Vahtras
- Stéfan van der Walt
- Gaël Varoquaux
- Nelle Varoquaux
- Olivier Verdier
- VirgileFritsch
- Pauli Virtanen
- Yosh Wakeham
- yasutomo57jp

Index

D

diff, 567, 571
differentiation, 567
dsolve, 571

E

equations
    algebraic, 569
    differential, 571

I

integration, 568

M

Matrix, 570

P

Python Enhancement Proposals
    PEP 255, 300
    PEP 3118, 338
    PEP 3129, 309
    PEP 318, 302, 309
    PEP 342, 300
    PEP 343, 310
    PEP 380, 301
    PEP 380#id13, 301
    PEP 8, 304

S

solve, 569