I Getting started with Python for science 2
1 Scientific computing with tools and workflow
1.1 Why Python? . . . . . . . . . . . . . . . . . .
. 2 Scientific Python building blocks
1.3 The interactive workflow: IPython and a text editor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 The Python language 10
2.1 First steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Basic types .
2.4 Defining functions

Reusing code. scripts and modules
2.6 mput and Output
2.8 Exception handling in Python
. 9 .xcept .
3 NumPy: creating and manipulating numerical data
3.1 The Numpy array object .
3.2 Numerical operations on arrays
3.3 More elaborate arrays
3.4 Advanced operations
3.5 Some exercises

Matplotlib: plotting
4.1 Introduction
. 2 Simple plot
4.3 Figures, Subplots, Axes and Tick
4.4 Other Types of Plots: examples and exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

5 Beyond thi tutorial ....................
6 Quick references
Scipy: high-level scientific computing
5.1 File input/output: scipy.io
5.2 Special functions: scipy. special
5.2 Special functions: scipy . special ....
5.4 Fast Fourier transforms: scipy fftpack
. 4 Fast Fourier transforms: scipy.fftpack
5 Optimization and fit: scipy .optimize . . .
Statistics and random numbers: scipy . st
Interpolation: scipy. interpolate .
Numerical integration: scipy.integrate
Numerical integration: scipy.integrate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
116
118
5.9 Signal processing: scipy.signal
5.9 Signal processing: scipy.signal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120 . 120

Gaë Varoquaux • Emmanuelle Gouillart • Olav Vahtras Valentin Haenel • Nicolas P. Rougier • Ralf Gommers Fabian Pedregosa • Zbigniew Jędrzejewski-Szmek • Pauli Virtanen Christophe Combelles • Didrik Pinte • Robert Cimrman André Espaze • Adrian Chauve • Christopher Burns
5.11 Summary exercises on scientific computing ..... 124
Getting help and finding documentation ..... 137
I Advanced topics ..... 140
Advanced Python Construct ..... 142
7.1 Iterators, generator expressions and generators ..... 142
7.3 Context manage155
8 Advanced Numpy ..... 159
8.1 Life of ndarray ..... 160
8.2 Universal functions
182
8.3 Interoperability features ..... 182
8.4 Array siblings: chararray, maskedarray, matrix ..... 188
8.6 Contributing to Numpy/Scipy ..... 188
9 Debugging code ..... 192
9.1 Avoiding bugs ..... 192
. Using the Python deb ..... 195
.3 Using the Python debugge .....  200
10 Optimizing code ..... 203
10.1 Optimization workflow ..... 203
203
10.2 Profiling Python code ..... 206
10.4 Writing faster numerical code 207
1 Sparse Matrices in SciPy ..... 210
.210
1.1 Introduction ..... 210
.212
11.2 Storage Schemes ..... 212
.224
11.3 Linear System Solvers . .
11.4 Other Interesting Package .....  229
12 Image manipulation and processing using Numpy and Scipy ..... 230
2.1 Opening and writing to image fis ..... $\begin{array}{r}230 \\ 231 \\ \hline 23\end{array}$
2.2 Displaying images232
233
12.4 Image filtering .....  235
2.4 Image filtering
2.4 Image filtering ..... 240
12.6 Measuring objects properties: ndimage . . . . . . . . . . . . . . . . .....  243
3 Mathematical optimization: finding minima of functions ..... 248
13.1 Knowing your problem ..... 249
3.2 Areview of guide to derent optimizer ..... 258
3.4 Special case: non-linear least-squares ..... 260
3.5 Optimization with constraints ..... 261
4 Interfacing with C ..... 263
14.1 Introduction263
.264
4.2 Python-C-Api ..... 264
268
4.4 SWIG
272
14.4 SWIG . ..... 276
4.6 Summary279
.280
14.8 Exercises .
III Packages and applications ..... 282
5 Statistics in Python ..... 284
5.1 Data representation and interaction ..... 285
5.2 Hypothesis testing: comparing two groups289
.292
5.4 More vi liz ..... 297
155 Testing for interaction .....  299
6 Sympy: Symbolic Mathematics in Python ..... 301
16.1 First Steps with SymPy ..... 301
.303
.303
nanipulations .....  304
16.4 Equation solving 306
7 Scikit-image: image processing
308
.308
17.1 Introduction and concepts
17.2 Input/output, data types and colorspaces ..... 310
17.3 Image preprocessing / enhancement . ..... 312
17.4 Image segmentation ..... 312
.315
.318
17.6 Data visuregions' properties .....  318
17.7 Feature extraction for computer visio .....  320
18 Traits: building interactive dialogs ..... 322
18.1 Introduction ..... $\begin{array}{r}322 \\ .323 \\ \hline\end{array}$
8.2 Example ..... 323
.324
19 3D plotting with Mayavi ..... 340
9.1 Mlab: the scripting interface ..... 340
9.2 Interactive work ..... 347
9.4 Animating the data ..... 349
9.5 Making interactive dialogs ..... 350
19.6 Putting it together .....  351
20 scikit-learn: machine learning in Python ..... 353
20.1 Loading an example dataset ..... 354
20.3 Clustering grouping observations together355
20.3 Clustering: grouping observations together 359
20.5 Putting it all together: face recognition ..... 360
0.6 Linear model from regression to sparsity361
362
Index ..... 363

## Part I

Getting started with Python for science

### 1.1 Why Python?

1.1.1 The scientist's needs

- Get data (simulation, experiment control),
- Manipulate and process data
- Visualize results (to understand what we are doing!),
- Communicate results: produce figures for reports or publications, write presentations


### 1.1.2 Specifications

- Rich collection of already existing bricks corresponding to classical numerical methods or basic actions 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: computer science is neither our job nor our education. We want to be able to draw a curve smooth a signal, do a Fourier transform in a few minutes
- Easy communication with collaborators, students, customers, to make the code live within a lab or a company: the code should be as readable as a book. Thus, the language should contain as few syn company: the code should be as readable as a book. Thus, the language should contain as few syn-
tax symbols or unneeded routines as possible that would divert the reader from the mathematical or scientific understanding of the code
- Efficient code that executes quickly... but needless to say that a very fast code becomes useless if we spend too much time writing it. So, we need both a quick development time and a quick execution time
- A single environment/language for everything, if possible, to avoid learning a new software for each new problem.


### 1.1.3 Existing solutions

Which solutions do scientists use to work?

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

- Advantages:
- Very fast. Very optimized compilers. For heavy computations, it's difficult to outperform these languages.
- Some very optimized scientific libraries have been written for these languages. Example: BLAS (vector/matrix operations)
- Drawbacks:
- Painful usage: no interactivity during development, mandatory compilation steps, verbose syntax (\&, ::, \}\}, ; etc.), manual memory management (tricky in C). These are difficult languages for non computer scientists.


## Scripting languages: Matlab

## - Advantages

- 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 well organized help, integrated editor, etc.
- Commercial support is available
- Drawbacks:
- Base language is quite poor and can become restrictive for advanced users.
- Not free.


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

- Advantages:
- Open-source, free, or at least cheaper than Matlab
- Some features can be very advanced (statistics in R, figures in Igor, etc.)
- Drawbacks:
- Fewer available algorithms than in Matlab, and the language is not more advanced
- Some software are dedicated to one domain. Ex: Gnuplot or xmgrace to draw curves. These programs are very powerful, but they are restricted to a single type of usage, such as plotting.


## What about Python

## - Advantages

- Very rich scientific computing libraries (a bit less than Matlab, though)
- Well thought out language, allowing to write very readable and well structured code: we "code what we think".
- Many libraries for other tasks than scientific computing (web server management, serial port ac cess, etc.)
- Free and open-source software, widely spread, with a vibrant community.
- Drawbacks:
- less pleasant development environment than, for example, Matlab. (More geek-oriented).
- Not all the algorithms that can be found in more specialized software or toolboxes.


### 1.2 Scientific Python building blocks

Unlike Matlab, Scilab 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
- Python language: data types (string, int), flow control, data collections (lists, dictionaries), pat terns, etc
- Modules of the standard library.
- A large number of specialized modules or applications written in Python: web protocols, web framework, etc. ... and scientific computing.
- Development tools (automatic testing, documentation generation)

- IPython, an advanced Python shell http://ipython.org/
- Numpy : provides powerful numerical arrays objects, and routines to manipulate them http://www.numpy.org/
- Scipy : high-level data processing routines. Optimization, regression, interpolation, etc http://www.scipy.org/

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

- Mayavi : 3-D visualization http://code.enthought.com/projects/mayavi/


### 1.3 The interactive workflow: IPython and a text editor

nteractive work to test and understand algorithms: In this section, we describe an interactive workflow with IPython that is handy to explore and understand algorithms.

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

## Reference document for this section:

IPython user manual: http://ipython.org/ipython-doc/dev/index.html

### 1.3.1 Command line interaction

Start ipython:
In [1]: print('Hello world')
llo world

Getting help by using the ? operator after an object:

```
In [2]: print?
ype:
String Form:
amespace:
Docstring:
uiltin_function_r_m
type 'builtin_function_or_method'>
Python builtin
    print(value, ..., sep=' ', end='\n', file=sys.stdout)
    Prints the values to a stream, or to sys.stdout by default
    Optional keyword arguments
    file. a file-like object (stream), defauls to the curgt sys.stdout.
    sep: string inserted between values, default a space
    sep: string inserted between values, default a space.
```


### 1.3.2 Elaboration of the algorithm in an editor

Create a file my_file.py in a text editor. Under EPD (Enthought Python Distribution), you can use Scite, available from the start menu. Under Python( $x, y$ ), you can use Spyder. Under Ubuntu, if you don't already have your favorite editor, we would advise installing Stani's Python editor. In the file, add the following ines:
$\mathrm{s}=\mathrm{'Hello}$ world'

Now, you can run it in IPython and explore the resulting variables:
In [1]: \%run my_file.py
Hello world

In [2]: s
Sut [2]: 'Hello world
In [3]: \%whos
ariable Type Data/Info
s 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


### 3.3 IPython Tips and Tricks

The IPython user manual contains a wealth of information about using IPython, but to get you started we want to give you a quick introduction to four useful features: history, magic functions, aliases and tab completion. ike a UNIX shell, IPython supports command history. Type $u p$ and down to navigate previously typed commands:

In [1]: $\mathrm{x}=10$
In [2]: <UP>
In [2]: $\mathrm{x}=10$
Python supports 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

$$
\begin{aligned}
& \text { In [2]: cd /tmp } \\
& \text { /tmp }
\end{aligned}
$$

- \%timeit allows you to time the execution of short snippets using the timeit module from the standard library:
In [3]: timeit $\mathrm{x}=$
10000000 loops, best of 3 : 39 ns per loop
- \%cpaste allows you to paste code, especially code from websites which has been prefixed with the stan dard Python prompt (e.g. >>>) or with an ipython prompt, (e.g. in [3]):

In [5]: cpaste
asting code; enter alone on the line to stop or use Ctrl-D.
:In [3]: timeit $\mathrm{x}=10$
10000000 loops, best of $3: 85.9$ ns per loop
In [6]: cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
>>> timeit $\mathrm{x}=10$
10000000 loops, best of $3: 86$ ns per loop

- \%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 [7]: $x===10$
File "<ipython-input-6-12fd421b5f28>", line 1
$x===10$
SyntaxError: invalid syntax

In [8]: debug
/.../IPython/core/compilerop.py (87) ast_parse()
86 and are passed to the built-in compile function."""
--> 87 return compile(source, filename, symbol, self.flags | PyCF_ONLY_AST, 1)
ipdb>locals()
\{'source': u'x === 10\n', 'symbol': 'exec', 'self':
<IPython.core.compilerop.CachingCompiler instance at 0x2ad8ef0>,
filename': '<ipython-input-6-12fd421b5f28>'\}

## IPython help

- The built-in IPython cheat-sheet is accessible via the \%qui ckref magic function
- A list of all available magic functions is shown when typing \%magic.

Furthermore IPython ships with various aliases which emulate common UNIX command line tools such as 1 s to list files, cp to copy files and rm to remove files. A list of aliases is shown when typing alias:

```
In [1]: alias
Total number of aliases: 16
Out [1]:
[('cat', 'cat'),
('clear', 'clear')
('cp', 'cp-i'),
('ldir', 'ls -F -o --color %l | grep /$')
('less', 'less'),
(1), ls -F -o --color %l | grep --')
'lk', '1s -F -o --color % | grep -1'),
(1l', 'ls -F -o --color')
'lx','ls -F -o --color %l | grep --...x'),
('man', 'man'),
('mkdir', 'mkdir'),
('more', 'more'),
('mv', 'mv -i'),
(rm', 'rm -i')
```

Lastly, we would like to mention the tab completion feature, whose description we cite directly from the IPython manual:

Tab completion, especially for attributes, 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 [1]: $\mathrm{x}=10$
In [2]: x . <TAB>
x.rea

In [3]: x.real
$\begin{array}{lll}\text { In [3]: X.real. } & \\ \text { x.real.bit_length } & \text { x.real.denominator } & \text { x.real.numerator } \\ \text { x.real.conjugate } & \text { x.real.imag } & \text { x.real.real }\end{array}$
In [4]: x.real.

## 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/tutorial. Dedicated books are also available, such a http://www.diveintopython.net/

## 2 python

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 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 $\mathrm{C}++$
- 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. in the Python( $x, y$ ) or EPD menu if you have installed one of these scientific-Python suites
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!
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
>>> $a=3$
$\ggg b=2 * a$
>>
>>
type (b)
<type 'int'>
>> print(b)
6
18
18
>> b = 'hello
>> type(b)
<type 'str'>
$\ggg \mathrm{b}+\mathrm{b}$
'hellohello'
>> $2 * \mathrm{~b}$
hellohe
Two variables $a$ and $b$ have been defined above. Note that one does not declare the type of an 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

Python supports the following numerical, scalar types:

## Integer

$$
\begin{aligned}
& \text { >>> } 1+1 \\
& 2 \\
& \ggg a=4 \\
& \ggg \text { type }(a) \\
& \text { <type 'int') }
\end{aligned}
$$

Floats

$$
\begin{aligned}
& \text { >>> } \mathrm{c}=2.1 \\
& \text { >>> type(c) } \\
& \text { <type 'float'> }
\end{aligned}
$$

## Complex

$\ggg \mathrm{a}=1.5+0.5 \mathrm{j}$
>>> a.real
1.5
$\ggg$
0.5
0.5
<type 'complex

## Booleans

>> $3>4$
False
>> test $=(3>4)$
F> test
>>> type(test)
<type 'bool'>
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

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

Lists
$A$ list is an ordered collection of objects, that may have different types. For example:
>>> l = ['red', 'blue', 'green', 'black', 'white']
>> type (1)
<type 'list'>
Indexing: accessing individual objects contained in the list
>>> 1[2]
Counting from the end with negative indices:

$$
\begin{aligned}
& \ggg 1[-1] \\
& \text { 'white' } \\
& \ggg 1[-2] \\
& \text { 'black' }
\end{aligned}
$$

## Indexing starts at $\mathbf{0}$ (as in C), not at 1 (as in Fortran or Matlab)!

Slicing: obtaining sublists of regularly-spaced elements:

```
>>> l ['red', 'blue', 'green', 'black', 'white'
['red', 'b
['green', 'black']
```

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

Slicing syntax: 1 [start:stop:stride]
All slicing parameters are optional:

```
['red', 'blue , green', 'black', 'white']
>> 1[3:]
['black', 'white']
>>> 1[:3]
['red', 'blue', 'green']
>> 1[::2]
['red','green', 'white']
```

Lists are mutable objects and can be modified:

```
\(\gg 1[0]=\) 'yellow
>> 1
'yellow', 'blue', 'green', 'black', 'white']
>> 1 [2:4] = ['gray', 'purple']
>> 1
['yellow', 'blue', 'gray', 'purple', 'white']
```

The elements of a list may have different types:
>>> $1=[3,-200$, 'hello' $]$
>> 1
[3, -200 , 'hello
>>> 1[1], l[2]
$(-200$, 'hello')
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.

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/tutorial/datastructures.html\#more-on-lists

Add and remove elements.

```
>> L = ['red' 'blwe', 'sreen' 'black', 'white']
>> L.append('pink')
\(\gg L\)
['red', 'blue', 'green', 'black', 'white' 'pink']
>> L.
pink'
['red', 'blue', 'green', 'black', 'white']
>> L. extend(['pink', 'purple']) \# extend \(L\), in-place
>> L
'red', 'blue', 'green', 'black', 'white', 'pink', 'purple']
    > \(\mathrm{L}=\mathrm{L}[:-2]\)
>> L
['red', 'blue', 'green', 'black', 'white']
```

Reverse:

```
>> r=L[::-1]
['white', 'black', 'green', 'blue', 'red']
>> r2 = list(L)
'red', 'blue', 'green', 'black', 'white']
>> r2.reverse() # in-place
>> r2
['white', 'black', 'green', 'blue', 'red']
```

Concatenate and repeat lists:

```
['white', 'black', 'green', 'blue', 'red', 'red', 'blue', 'green', 'black', 'white']
>> r * 2
['white', 'black', 'green', 'blue', 'red', 'white', 'black', 'green', 'blue', 'red']
```

Sort:

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


## Methods and Object-Oriented Programming

The notation r.method() (e.g. r. append (3) and L.pop()) is our first example of object-oriented pro gramming (OOP). Being a list, the object $r$ 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 [28] : r.<TAB> |  |  |
| :---: | :---: | :---: |
| r.__add_- | r._-iadd_- | r._-setattr_- |
| r.__class_- | r._-imul_- | r._-setitem_- |
| r._-contains_- | r. ._init_- | r.__setslice__ |
| r.__delattr_- | r._-iter_- | r.__sizeof _- |
| r._-delitem_- | r. ._le_- | r._-str_- |
| r.__delslice__ | r.__len_- | r.__subclasshook_ |
| r._-doc_- | r. ${ }^{1 t}$ _- | r.append |
| r. -_eq_- | r._-mul_- | r.count |
| r.__format__ | r. ._ne_- | r.extend |
| r._-ge_- | r. ._new_- | r.index |
| r._-getattribute_- | r.__reduce_- | r.insert |
| r. ._getitem_- | r. ._reduce_ex_- | r.pop |
| r.__getslice__ | r._-repr_- | r.remove |
| r._-gt | r.__reversed_- | r.reverse |
| r.__hash_- | r. ._rmul_- | r.sort |

## Strings

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


The newline character is $\backslash \mathrm{n}$, and the tab character is $\backslash \mathrm{t}$.
Strings are collections like lists. Hence they can be indexed and sliced, using the same syntax and rules.
Indexing:
$\ggg a=$ "hello"
$\ggg \mathrm{a}[0]$
'h'
>>> a[1]
'e'
>> a[-1]
'o'
(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
10,'
>> a[2:10:2] \# Syntax: a [start:stop:step.
10 o'
>> a[::3] \# every three characters, from beginning to end
https://docs.python.org/tutorial/introduction.html\#unicode-strings). trings from the original one
In [53]: a = "hello, world!
In [54]: a [2] \(=\) 'z'
Traceback (most recent call last)
raceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
In [55]: a.replace('1', 'z', 1)
Dut [55]: 'hezlo, world!'
In [56]: a.replace('1', 'z')
In [56] : a.replace( 'hezzo, worzd!'
```

Accents and special characters can also be handled in Unicode strings (see

A string is an immutable object and it is not possible to modify its contents. One may however create new

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/library/stdtypes.html\#string-methods and https://docs.python.org/library/string.html\#new-string-formatting

## String formatting:

> >> 'An integer: \%i; a float: \%f; another string: \% ' \% (1, 0.1, 'string')
> 'An integer: 1; a float: 0.100000 ; another string: string'
>>> i = 102
>> filename = 'processing_of_dataset_\%d.txt' \% i
>> filename
'processing_of_dataset_102.txt'

## Dictionaries

A dictionary is basically an efficient table that maps keys to values. It is an unordered container

```
>> tel \(=\{\) 'emmanuelle': 5752, 'sebastian': 5578
>> tel['francis'] = 5915
〔'sebastian': 5578, 'francis': 5915, 'emmanuelle': 5752\}
>> tel['sebastian']
578
'f
['sebastian', 'fr
[5578, 5915, 5752]
\(\ggg\) 'francis' in tel
True
```

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/tutorial/datastructures.html\#dictionaries for more information. A dictionary can have keys (resp. values) with different types:

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


## More container types

## Tuples

Tuples are basically immutable lists. The elements of a tuple are written between parentheses, or just separated by commas:
$\ggg \mathrm{t}=12345,54321$, 'hello!'
>> $t$ [0]
12345
(12345, 54321, 'hello!')
(12345, $54321, ~$
$\ggg \mathrm{u}=(0,2)$
'hello! ')

Sets: unordered, unique items:

$$
\begin{aligned}
& \text { >>> } s=\operatorname{set}\left(\left(' a^{\prime}, \quad\right.\right. \text { 'b', 'c', 'a')) } \\
& \gg \mathbf{s} \\
& \text { set(['a', 'c', 'b']) } \\
& \text { >>> s.difference(('a', 'b')) } \\
& \text { set(['c']) }
\end{aligned}
$$

## Dbvious

## locks are delimited by indentation

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 column : 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:
lot
ndentation 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

terating with an index:
$\gg$
... for i in range (4):
print(i)
0
1
1
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 powerful
Python is powerful
```


### 2.3.3 while/break/continue

Typical C-style while loop (Mandelbrot problem):

$$
\begin{aligned}
& \ggg z=1+1 j \\
& \ggg \text { while abs }(z) \text { < } 100: \\
& \cdots \quad z=z * * 2+1 \\
& \ggg z \\
& (-134+352 j)
\end{aligned}
$$

## More advanced features

break out of enclosing for/while loop:

```
>> z=1+1j
>> while abs(z) < 100,
    if z.imag == 0:
        f z.imag == 0;
        = z**2 +
```

continue the next iteration of a loop.:

```
\(\ggg a=[1,0,2,4]\)
    >> 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 ( $0,0.0,0+0 \mathrm{j}$ )
- an empty container (list, tuple, set, dictionary, ...)
- False, None


## Evaluates to True:

- everything else
$\mathrm{a}==\mathrm{b}$ Tests equality, with logics:
>>> 1 == 1
True
a is b Tests identity: both sides are the same object:

$$
\begin{aligned}
& \text { False } \\
& \ggg \mathrm{a}=1 \\
& \ggg=1 \\
& \ggg \mathrm{a}=1 \\
& \text { > is } \mathrm{b}
\end{aligned}
$$

$$
\begin{aligned}
& \ggg \\
& \text { True }
\end{aligned}
$$

in b For any collection b : b contains a

$$
\begin{aligned}
& \ggg \mathrm{b}=[1,2,3] \\
& \ggg 2 \text { in } \mathrm{b} \\
& \text { True } \\
& \ggg 5 \text { in } \mathrm{b} \\
& \text { False }
\end{aligned}
$$

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'
    f i in vowels:
    print(i)
```

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

Few languages (in particular, languages for scientific computing) allow to loop over anything but inte gers/indices. With Python it is possible to loop exactly over the objects of interest without bothering with indices you often dont care about. This feature can often be used to make code more readable.
© 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)):
        words[i]))
    (0, 'cool')
    (1, 'powerful')
(1, 'powerful')
```

- 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:
>>> $d=\left\{a^{\prime}: 1, ' b ': 1.2, c^{\prime}: 1 \mathrm{j}\right\}$
>> for key, val in sorted(d.items()): print('Key: \%s has value: \%s ' \% (key, val)
Key: a has value: 1
ey: b has value: 1.2
Key: c has value: 1 j
The ordering of a dictionary in random, thus we use sorted () which will sort on the keys.

### 2.3.6 List Comprehensions

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

## Exercise

Compute the decimals of Pi using the Wallis formula:

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

### 2.4 Defining functions

2.4.1 Function definition

```
In [56]: def test():
    print('in test function')
In [57]: test()
in test function
```

$\triangle$ Function blocks must be indented as other control-flow blocks.

### 2.4.2 Return statement

Functions can optionally return values
In [6]: def disk_area(radius):
return 3.14 * radius * radius

In [8]: disk area (1.5)
Out [8]: 7.0649999999999995
By default, functions return None.

## 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 [81]: def double_it(x):
return $\mathrm{x} * 2$

In [82]: double_it(3)
Dut [82]: 6
In [83]: double_it()

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: double_it() takes exactly 1 argument ( 0 given)
Optional parameters (keyword or named arguments)

$$
\begin{aligned}
& \text { In [84]: def double_it }(x=2) \\
& \text { return } \mathrm{x} * 2
\end{aligned}
$$

In [85]: double_it()
Out [85] : 4
In [86]: double_it(3)
Out [86] : 6

## Keyword arguments allow you to specify default values.

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 [124]: bigx = 10
In [125]: def double_it(x=bigx):
```

$$
\text { return } x * 2
$$

In [126]: bigx $=1 \mathrm{e} 9$ \# Now really big
In [128]: double_it()
Out [128]: 20
Using an mutable type in a keyword argument (and modifying it inside the function body)
In [2]: def add_to_dict(args=\{'a': 1, 'b': 2\}):
for $i$ in args.keys():
$\operatorname{args}[\mathrm{i}]+=1$
print args

In [3]: add_to_dict
Out [3]: <function __main__. add_to_dict>
In [4]: add_to_dict()
$\left\{{ }^{\prime} a^{\prime}: 2, \quad\right.$ 'b': 3\}
In [5]: add_to_dict()
\{'a': 3, 'b': 4\}
In [6]: add_to_dict()
\{'a': 4, 'b': 5\}

More involved example implementing python's slicing:

```
In [98]: def slicer(seq, start=None, stop=None, step=None)
    """Implement basic python slicing."""
    return seq[start:stop:step]
In [101]: rhyme = 'one fish, two fish, red fish, blue fish'.split()
In [102]: rhyme
Out[102]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']
In [103]: slicer(rhyme)
Out[103]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']
In [104]: slicer(rhyme, step=2)
Out[104]: ['one', 'two', 'red', 'blue']
In [105]: slicer(rhyme, 1, step=2)
Out[105]: ['fish,', 'fish,', 'fish,', 'fish']
In [106]: slicer(rhyme, start=1, stop=4, step=2)
Out [106]: ['fish,', 'fish,']
```

The order of the keyword arguments does not matter:
In [107]: slicer(rhyme, step=2, start=1, stop=4) Out [107]: ['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

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 value" and "passing by reference". In Python, such a distinction is somewhat artificial, a
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.
f 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
    x=23
    z = [99] # new reference
    print(x)
    print(y)
    print(z
>> a = 77 # immutable variabl
# mutable variable
>> c= [28]
>> try_to_modify(a, b, c)
99,42
[99]
>> print(a)
```

77

## >> print(b) <br> [99, 42] <br> [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:

$$
\begin{aligned}
& \text { In [114]: } \mathrm{x}=5 \\
& \text { In [115]: } \operatorname{def} \operatorname{addx}(\mathrm{y}): \\
& \ldots \ldots: \text { return } \mathrm{x}+\mathrm{y} \\
& \ldots \ldots: \\
& \text { In [116]: } \operatorname{addx}(10) \\
& \text { Out [116]: } 15
\end{aligned}
$$

But these "global" variables cannot be modified within the function, unless declared global in the function.
This doesn't work:

| In [117]: | def |
| ---: | :--- |
| setx $(\mathrm{y}):$ |  |
| $\ldots \ldots:$ | $\mathrm{x}=\mathrm{y}$ |
| $\ldots \ldots:$ | $\operatorname{print}\left(' \mathrm{x}\right.$ is $\left.\% d^{\prime} \% \mathrm{x}\right)$ |

In [118]: $\operatorname{setx}(10)$
x is 10
n [120]: $x$
Dut [120]: 5
This works:

| In [121]: def setx(y):$\quad \ldots \ldots$ global $x$ |  |
| :---: | :---: |
|  |  |
|  |  |
|  |  |

In [122]: setx(10)
x is 10
In [123]: x
Out [123]: 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 [35]: def variable_args(*args, **kwargs)
    rint 'args is', args
    print 'kwargs is', kwargs
```

In [36]: variable_args('one', 'two', $x=1, y=2, \quad z=3$ )
args is ('one', 'two')
kwargs is \{'y': 2, 'x': 1, 'z': 3\}

### 2.4.7 Docstrings

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

```
n [67]: def funcname(params)
    """Concise one-line sentence describing the function
    Extended summary which can contain multiple paragraphs.
    # function body
    pass
```

In [68] : funcname?
Type:
$\begin{array}{ll}\text { Base Class: } & \text { function } \\ \text { type 'function'> }\end{array}$
String Form: <function funcrame at oxean
Iamespace: Interactive
ile: <ipython console>
Definition
Concise one-line sentence describing the function.
Extended summary which can contain multiple paragraphs.

## 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 tions, that you may want to follow for your own functions, with a Parameters section, an Examples
section, etc. See http://projects.scipy.org/numpy/wiki/CodingStyleGuidelines\#docstring-standard and http:// projects.scipy.org/numpy/browser/trunk/doc/example py\#L37

### 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 [38]: va = variable_args
In [39]: va('three', x=1, y=2)
rgs is ('three',)
kwargs is {'y': 2, 'x': 1}
```


### 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:

- $u_{-} 0=1 ; u_{-} 1=1$
$u_{-}(n+2)=u_{-}(n+1)+u_{-} n$


## Exercise: Quicksort

Implement the quicksort algorithm, as defined by wikipedia

```
unction quicksort(array)
    var list less, greate
    if length(array) < 2
        return array
    al 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 rack 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 (e.g., Scite with Python(x,y)),

### 2.5.1 Scripts

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
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.
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
ut [2]: '
The script has been executed. Moreover the variables defined in the script (such as message) are now available inside the interpreter's namespace.
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?
Standalone scripts may also take command-line arguments
In file.py:
import sys
print sys.argv
\$ python file.py test arguments
Don't implement option parsing yourself. Use modules such as optparse, argparse or docopt

### 2.5.2 Importing objects from modules

```
In [1]: import os
In [2]: os
Out[2]: <module 'os' from '/usr/lib/python2.6/os.pyc'>
In [3]: os.listdir('. ')
Out [3]:
['conf.py',
    'basic_types.rst',
    'control_flow.rst'
    'python_language.rst',
    'reusing.rst',
    'file_io.rst',
    'exceptions.rst'
    'workflow.rst'
    'index.rst']
```

And also:
In [4]: from os import listdir
Importing shorthands:
In [5]: import numpy as np

## $\wedge_{\text {from os import * }}$

## This is called the star import and please, Use it with caution

- 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-vers
- Creates possible name clashes between modules
- Makes the code impossible to statically check for undefined symbols.

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:

$$
\begin{aligned}
& \text { >> import numpy as np \# data arrays } \\
& \ggg \text { np. in inspace }(0,10,6) \\
& \text { >rray([ 0., 2., 4., 6., } 8 ., \quad 10 .]) \\
& \text { >>> import scipy \# scientific computing }
\end{aligned}
$$

In Python( $\mathrm{x}, \mathrm{y}$ ), Ipython( $\mathrm{x}, \mathrm{y}$ ) executes the following imports at startup:
>>> import numpy
>>> import numpy as np
>> fron pylab
and it is not necessary to re-import these modules

### 2.5.3 Creating modules

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

$$
\begin{gathered}
\text { "A demo module." } \\
\text { def print_b(): } \\
\text { "Prints b." } \\
\text { print 'b' }
\end{gathered}
$$

def print_a():
"Prints a."

$$
\begin{aligned}
& c=2 \\
& d=2
\end{aligned}
$$

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 [1]: import demo
In [2]: demo.print_a()
a
n [3]: demo. print_b()
mporting 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


Importing objects from modules into the main namespace

| In [9]: from demo import print_a, print_b |  |
| :---: | :---: |
| In [10]: 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 [11]: print_a() |  |

## Module caching <br> Modules are cached: if you modify demo. py and re-import it in the old session, you will get the old one.

## In [10]: reload(demo)

In Python3 instead reload is not builtin, so you have to import the importlib module first and then do: In [10]: importlib.reload(demo)

### 2.5.4 '__main__' and module loading

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():
    print 'a'
    \# print_b() runs on impor
    print_b()
    if __name__ == '__main _ '
        \# print_a() is only executed when the module is run directly.
```

        print_a()
    Importing it:
In [11] : import demo2

In [12]: import demo2
Running it:
In [13]: \%run demo2
In
b
a
a

### 2.5.5 Scripts or modules? How to organize your code

## 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 direc tories. This list includes a list of installation-dependent default path (e.g., /usr/lib/python) 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 [1]: import sys
In [2]: sys.path
Out [2]:
[' ' ',
'/home/varoquau/.local/bin \({ }^{\prime}\)
    '/uss/lib/python2.7',
    '/home/varoquau/.local/lib/python2.7/site-packages ',
    '/usr/1ib/python2 7/dist-packages ',
    '/usr/local/lib/python2.7/dist-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/.local/lib/python2.7/dist-packages). You may use symbolic links (on Linux) to \$HOME/. 1ocal/1ib/python2.
- modify the environment variable PYTHONPATH to include the directories containing the user-defined modules
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, http:// support.microsoft.com/kb/310519 explains how to handle environment variables.
- or modify the sys. path variable itself within a Python script

$$
\begin{aligned}
& \text { import sys } \\
& \text { new_path = '/home/emma/user_defined_modules } \\
& \text { if new_path not in sys.path: } \\
& \text { sys.path. append(new_path) }
\end{aligned}
$$

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/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.
\$ 1s


| doccer.pye | fourier.pyc | interpolation.py@ | morphology.pyc | setup.pyc |
| :---: | :---: | :---: | :---: | :---: |
| doccer.pyc | info.py@ | interpolation.pyc | _nd_image.so |  |
| setupscons.py@ |  |  |  |  |
| filters.py@ | info.pyc | measurements.py@ | _ni_support.py@ |  |
| setupscons.pyc |  |  |  |  |
| ters.pyc | _-init__•py@ | measurements.pyc | _ni_support.pyc | tests/ |
| fourier.py@ | _-init_-.pyc | morphology.py@ | setup.py@ |  |

From Ipython:
In [1]: import scipy
In [2]: scipy._-file
Out [2]: '/usr/lib/python2.6/dist-packages/scipy/__init__.pyc'
In [3]: import scipy.version
[n [4]: scipy.version.versio
Out [4]: '0.7.0
In [5]: import scipy.ndimage.morphology
In [6]: from scipy.ndimage import morphology
In [17]: morphology.binary_dilation?
type:
function
Base Class: <type 'function'>
String Form: <function binary_dilation at 0x9bedd84>
Tamespace: Interactive
Jusr/lib/python2.6/dist-packages/scipy/ndimage/morphology.py
, mork=None $=1$,
brute_force=False)
Docstring:
Multi-dimensional binary dilation with the given structure.
An output array can optionally be provided. The origin parameter
controls the placement of the filter. If no structuring element is
provided an element is generated with a squared connectivity equal
iterations is less than 1, the dilation is repeated until the
result does not change anymore. If a mask is given, only tho
elements with a true value at the corresponding mask element are modified at each iteration.

### 2.5.7 Good practices

## - Use meaningful object names

## Indentation: no choicel

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 space $(1,2,3,4, \ldots)$. 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. In Python( $x, y$ ), the editor is already configured this way.


## - Style guidelines

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

## $\gg$ long_line $=$ "Here is a very very long line \} <br> 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 Scipy 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:

```
>> type(f)
<type 'file'>
>> f.write('This is a test \nand another test')
>> f.close()
```

To read from a file
In [1]: $\mathrm{f}=\operatorname{open}($ 'workfile', $\quad \mathrm{r}$ ')
In [2]: s $=\mathrm{f} \cdot \mathrm{read}($ )
In [3]: print(s)
his is a test
and another test
In [4]: f.close()

## See also:

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

### 2.6.1 Iterating over a file

In [6]: $\mathrm{f}=\operatorname{open}($ 'workfile', 'r')
In [7]: for line in f:
This is a test
and another test
In [8]: 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: $\mathrm{r}+$
- Binary mode: b
- Note: Use for binary files, especially on Windows.


### 2.7 Standard Library

Reference document for this section:

- The Python Standard Library documentation: https://docs.python.org/library/index.htm
- 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 [17]: os.getcwd()
Out [17]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source

## List a directory:

In [31]: os.listdir(os.curdir)
Out [31] :
'. index.rst.swo',
'. python_language.rst.swp' ,
.view_array.py.swp',
-static',
'basic_types.rst'
'conf.py',
'control_flow.rst',

## 'debugging.rst',

## Make a directory:

In [32]: os.mkdir('junkdir')
In [33]: 'junkdir' in os.listdir(os.curdir)
Out [33]: True

## Rename the directory:

In [36]: os.rename('junkdir', 'foodir')

In [37]: 'junkdir' in os.listdir(os.curdir)
Out [37]: False
In [38]: 'foodir' in os.listdir(os.curdir)
Out [38]: True
In [41]: os.rmdir('foodir')
In [42]: 'foodir' in os.listdir(os.curdir)
Out [42]: False
Delete a file:
In [44]: fp = open('junk.txt', 'w')
In [45]: fp.close()
In [46]: 'junk.txt' in os.listdir(os. curdir)
Dut [46]: True
In [47]: os.remove('junk.txt')
In [48]: 'junk.txt' in os.listdir(os.curdir)
Out [48]: False

## s.path: path manipulations

os. path provides common operations on pathnames.
In [70]: $\mathrm{fp}=$ open('junk.txt', 'w')
In [71]: fp.close()
In [72]: a = os.path.abspath('junk.txt')
In [73]: a
but [73]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/junk.txt'
In [74]: os.path.split(a)
Out [74]: ('/Users/cburns/src/scipy2009/scipy_2009_tutorial/source' 'junk.txt')

In [78]: os.path.dirname(a)
Dut [78]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
In [79]: os.path. basename (a)
Dut [79]: 'junk.txt
In [80]: os.path.splitext(os.path. basename(a))

```
Dut [80]:('junk', '.txt')
In [84]: os.path.exists('junk.txt')
Out [84]: True
In [86]: os.path.isfile('junk.txt')
Dut [86]: True
In [87]: os.path.isdir('junk.txt')
Out [87]: False
In [88]: os.path.expanduser(' ~/local')
Sut [88]: '/Users/cburns/local'
In [92]: os.path.join(os.path.expanduser('~'), 'local', 'bin')
Out [92]: '/Users/cburns/local/bin
```


## Running an external command

| In [8]: os.system('ls') |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- |
| basic_types.rst | demo.py | functions.rst | python_language.rst | standard_library.rst |
| control_flow.rst | exceptions.rst | io.rst | python-logo.png |  |
| demo2.py | first_steps.rst | oop.rst | reusing_code.rst |  |

## Alternative to os system

A noteworthy alternative to os.system is the sh module. Which provides much more convenient ways to A noteworthy alternative to os.system is the sh module. Which provid

In [20]: import sh
In [20]: $\operatorname{com}=\operatorname{sh} .1 \mathrm{~s}()$
In [21]: print con
basic_types.rst exceptions.rst oop.rst standard_libery $\begin{array}{lll}\text { control_flow.rst } & \text { first_steps.rst } \\ \text { demo2.py } & \text { function_language.rst }\end{array}$ $\begin{array}{lll}\text { demo2.py } & \text { functions.rst } & \begin{array}{l}\text { python-logo.png } \\ \text { reusing_code.rst }\end{array} \\ \text { demo.py } & \text { io.rst }\end{array}$

In [22]: print com.exit_code
In
In [23]: type(com)
Out [23]: sh.RunningCommand

## Walking a directory

os . path. walk generates a list of filenames in a directory tree.
In [10]: for dirpath, dirnames, filenames in os.walk(os.curdir):

$$
\text { for } \mathrm{fp} \text { in filenames: }
$$

print os. path. abspath (fp)

Users/cburns/src/scipy2009/scipy_2009_tutorial/source/.index.rst.swo Users/cburns/src/scipy2009/scipy_2009_tutorial/source/.view_array.py.sw
users/cburns/src/scipy2009/scipy_2009_tutorial/source/basic_types.rst
Users/cburns/src/scipy2009/scipy_2009_tutorial/source/conf.py
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/control_flow.rst

## Environment variables:

In [9]: import os
In [11] : os.environ.keys()
out [11]:
['- ${ }^{\text {FSLD }}$ '
'TERM_PROGRAM_VERSION',
' FSLREMOTECALL' ,
'USER',
'PATH',
PATH'
'SHELL',
'EDITOR',
'WORKON_HOME',
'PYTHONPATH',
In [12]: os.environ['PYTHoNPATH']
Out [12]: $\quad .: /$ Users/cburns/src/utils:/Users/cburns/src/nitools
/Users/cburns/local/lib/python2.5/site-packages/
/usr/local/lib/python2.5/site-packages/
/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5'
In [16]: os getenv('PYTHoNPATH')
Out [16]: ' $:$ :/Users/cburns/src/utils:/Users/cburns/src/nitools:
Users/cburns/local/lib/python2.5/site-packages/
/usr/local/lib/python2.5/site-packages/:
/Library/Frameworks/Python.framework/Versions/2.5/1ib/python2.5'

### 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 [18]: import glob
In [19]: glob.glob('*.txt')
but [19]: ['holy_grail.txt', 'junk.txt', 'newfile.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:

$$
\begin{aligned}
& \text { In [117]: sys.platform } \\
& \text { Out [117]: 'darwin' } \\
& \\
& \text { In [118]: } \text { sys.version } \\
& \text { Out [118]: } \text { '2.5.2 (r252:60911, Feb } 22 \text { 2008, } 07: 57: 53 \text { ) } \backslash \text { n } \\
& \text { [GCC } 4.0 .1 \text { (Apple Computer, Inc. build } 5363 \text { )]' } \\
& \text { In [119]: } \\
& \text { Out [119]: }: \text { '/Librefix }
\end{aligned}
$$

- List of command line arguments passed to a Python script

Out [100]: ['/Users/cburns/local/bin/ipython']
sys. path is a list of strings that specifies the search path for modules. Initialized from PYTHONPATH:
In [121]: sys. path
Out [121]:
cburns/local/bin',
Sers/cburns/local/lib/python2.5/site-packages/grin-1.1-py2.5.egg
/Users/cburns/local/lib/python2.5/site-packages/argparse-0.8.0-py2.5.egg
'/Users/cburns/local/lib/python2.5/site-packages/urwid-0.9.7.1-py2.5.egg'
Users/cburns/local/lib/python2.5/site-packages/yolk-0.4.1-py2.5.egg',

### 2.7.5 pickle: easy persistence

Useful to store arbitrary objects to a file. Not safe or fast
In [1]: import pickle
In [2]: $1=[1$, None, 'Stan']
In [3]: pickle.dump(1, file('test.pkl', 'w'))
In [4]: pickle.load(file('test.pkl'))
Dut [4]: [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. Fo 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: integer division or modulo by zero
In [2]: $1+{ }^{\prime} e^{\prime}$
ypeError: unsupported operand type(s) for +: 'int' and 'str'
In [3]: $\mathrm{d}=\{1: 1,2: 2\}$
In [4]: $\mathrm{d}[3]$
KeyError: 3
In [5]: $1=[1,2,3]$
In [6]: 1[4]
IndexError: list index out of range
In [7]: 1.foobar
AttributeError: 'list' object has no attribute 'foobar
As you can see, there are different types of exceptions for different errors

### 8.8.2 Catching exceptions

try/except

```
In [10]: while True:
            try: 
            break
            except ValueError:
            print('That was no valid number. Try again...')
```

Pease enter a number: a
That was no valid number. Try again..
Please enter a number:
In [9]: x
Out [9]: 1
try/finally

```
In [10]: try: 
```

$$
\begin{aligned}
& \mathrm{x}=\operatorname{int}(\text { raw_input('Please enter a } \\
& \text { finally: } \\
& \text { print('Thank you for your input') }
\end{aligned}
$$

lease enter a number:
Thank you for your input
ValueError: invalid literal for int() with base 10: 'a

## mportant for resource management (e.g. closing a file)

Easier to ask for forgiveness than for permission
print(collection)

In [12]: print_sorted([1, 3, 2])
$[1,2,3]$
In [13]: print_sorted(set((1, 3, 2))
$\operatorname{set}([1,2,3])$
n [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, e:
if name $==$ 'Gaël' ':
else:
return name

In [16]: filter_name('Gaë1')
ok, Gaë
Dut [16]: 'Ga\xc3\xabl'
In [17]: filter_name('Stéfan')
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 2: ordinal not in range(128)

- Exceptions to pass messages between parts of the code:

In [17]: def achilles_arrow $(x)$ :
if abs $(\mathrm{x}-1)<1 \mathrm{e}-3$ :
$\mathrm{x}=1-(1-\mathrm{x}) / 2$.
$x=1-1$
return $x$

In [18]: $\mathrm{x}=0$
In [19]: while True:
try:
$\mathrm{x}=$ achilles_arrow $\left.^{\mathrm{x}} \mathrm{x}\right)$
except StopIteration:
break

$$
\begin{aligned}
& \text { In [11]: def print_sorted(collection): } \\
& \text { try: } \\
& \text { collection. sort() } \\
& \text { xcept AttributeError: } \\
& \text { pass }
\end{aligned}
$$

In [20]: x
Out [20]: 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 re-use code in similar contexts.

Here is a small example: we create a Student class, which is an object gathering several custom function (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 at rributes are name, age and major. We can call these methods and attributes with the following notation lassinstance,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)
>>>
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 correspond ing to different objects we encounter (an Experiment class, an Image class, a Flow class, etc.), with their own ing to different objects we encounter (an Experiment class, an Image class, a Flow class, etc.), with their own 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 array
- closer to hardware (efficiency)
- designed for scientific computation (convenience)
- Also known as array oriented computing

$$
\begin{aligned}
& \text { >> import numpy as np } \\
& \ggg \mathrm{a}=\mathrm{np} . \operatorname{array}([0,1,2,3]) \\
& \ggg \mathrm{a} \\
& \operatorname{array}([0,1,2,3])
\end{aligned}
$$

or 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 $=\operatorname{range}(1000)$
In [2]: \%timeit [i**2 for i in L]
1000 loops, best of 3: 403 us per loop
In [3]: a = np.arange (1000)
In [4]: \%timeit a**2
100000 loops, best of 3: 12.7 us per loop

Numpy Reference documentation

- On the web: http://docs.scipy.org/
- Interactive help:

In [5]: np.array?
String Form:<built-in function array>
Docstring:
array (object, dtype=None, copy=True, order=None, subok=False, ndmin=0,

- Looking for something
>> np. lookfor('create array')
Search results for 'create array'
numpy .array
Create an array.
numpy.memmap
Create a memory-map to an array stored in a *binary* file on disk.

```
In [6]: np.con* 
np.concatena
np.conj
np.conjugate
np.convolve
```


## mport 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. \(\operatorname{array}([0,1,2,3])\)
>> a
array([0, 1, 2, 3])
>>> a.ndim
\(\stackrel{1}{\ggg}\)
>> a.shape
4,)
>>> len(a)
```


## - 2-D, 3-D, ...

```
>>> b \(=\mathrm{np}\).array ([[0, 1, 2], [3, 4, 5] ]) \#2 x 3 array
>> b
array([[0, 1, 2],
>>> b. ndim
>> b. shape
(2, 3)
>> len(b) \# returns the size of the first dimension
2
>>> \(c=n p\).array([[[1], [2]], [[3], [4]]] \()\)
>ray([L[
        [2]],
            \begin{tabular}{c}
{\(\left[\begin{array}{c}{[3],} \\
[4]] \text { ] }\end{array}\right.\)} \\
\hline
\end{tabular}
        [4] \({ }^{\text {[3) }}\)
```

$(2,2,1)$

## Exercise: Simple arrays

- Create a simple two dimensional array. First, redo the examples from above. And then create you 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

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

- Evenly spaced:

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

- or by number of points:
>>> $\mathrm{c}=\mathrm{np}$.linspace ( $0,1,6$ ) \# start, end, num-points
>>> c
array([ 0., 0.2, 0.4, 0.6, 0.8, 1. ])
$\ggg \mathrm{d}=$ np.linspace ( $0,1,5$, endpoint=False)
>>> d
$\operatorname{array}([0 ., 0.2,0.4,0.6,0.8])$
- Common arrays:

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

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

$$
\text { >> a = np.random.rand(4) \# uniform in }[0,1]
$$

$$
\begin{array}{llll} 
\\
\text { array a }([0.95799151, & 0.14222247,0.08777354, & 0.51887998])
\end{array}
$$

>> b
>> b
array ([ $0.37544699,-0.11425369,-0.47616538,1.79664113]$ )
>>> np.random.seed(1234) \# Setting the random seed

## 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:
>>> $\mathrm{a}=\mathrm{np} . \operatorname{array}([1,2,3])$
>> a.dtype
dtype('int64')
>>> b $=$ np.
>>> b.dtype
dtype('float64')
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:
>>> $\mathrm{c}=\mathrm{np}$.array ([1, 2, 3], dtype=float)
>> c.dtype
type('float64')
The default data type is floating point:
>>> a $=$ np.ones ( $(3,3)$ )
>> a.dtype
dtype('float64')
There are also other types

## Complex

>> d.dtype
dtype('complex128')
Bool
>>> e = np.array([True, False, False, True])
>> e.dtype
dtype('bool')
Strings
>> f = np.array(['Bonjour', 'Hello', 'Hallo',]) dtype('S7')

## 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 the notebook:
\$ ipython 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( $\mathrm{x}, \mathrm{y}$ ) \# line plot
>> plt.show()
Or, if you have enabled interactive plots with \%matplotlib:
>>> plot( $\mathrm{x}, \mathrm{y}$ )

## \# line plot

- 1D plotting
>> $\mathrm{y}=$ np.linspace (0, $0,9,20)$
>> plt.plot(x, y) \# line plo
[<matplotlib.lines.Line2D object at ...>]
>> plt.plot(x, y, 'o') \# dot plot
[<matplotlib.lines.Line2D object at ...>]

- 2D arrays (such as images):
>> image $=n p . r a n d o m . r a n d(30,30)$
>> plt.imshow(image, cmap=plt.cm.hot)
lorbar (
<matplotlib.colorbar.Colorbar instance 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)

$$
\begin{aligned}
& \ggg \mathrm{a}=\text { np.arange (10) } \\
& \ggg \mathrm{a} \\
& \text { array }([0,1,2,3,4,5,6,7,8,9]) \\
& \ggg \mathrm{a}[0, \mathrm{a}, \mathrm{a} 2 \mathrm{a}, \mathrm{a}[-1] \\
& (0,2,9)
\end{aligned}
$$

© Indices begin at 0 , like other Python sequences (and $\mathrm{C} / \mathrm{C}++$ ). 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, indexes are tuples of integers:
>> a = np.diag(np.arange (3))
>>> a
$\operatorname{array}([[0,0,0]$,
$[0,1,0]$,
$[0,0,2]]$ )
>> a[1, 1]
1
>> a[2, 1] = 10 \# third line, second column
>> a
array([[ $0,0,0]$,
$[0,1,0]$,
a[1]
>>> a[1] $\operatorname{array([0,1,0])}$

- In 2D, the first dimension corresponds to rows, the second to columns.
- for multidimensional $\mathrm{a}, \mathrm{a}[0]$ 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
aray ( $[0,1,2,3,4,5,6,7,8,9])$
>> a[2:9:3] \# [start:end:step]
$\operatorname{array}([2,5,8])$
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
>>> a[1:3]
$\operatorname{array}([1,2])$
>> a[::2]
$\operatorname{array}([0,2,4,6,8])$
>>> a[3:]
$\operatorname{aray}([3,4,5,6,7,8,9])$
A small illustrated summary of Numpy indexing and slicing...

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

```
>>> a[4:,4:]
array([[44, 45],
    [54, 55]])
>> a[:,2]
array([2,12,22,32,42,52])
>>> a[2::2,::2]
array([[20,22,24]
    [40,42,44]])
```



You can also combine assignment and slicing:
>>> $a=n p$. arange (10)
$\ggg>$ a $5:]=10$
$\ggg a[5:]=10$
rray([ $0,1,2,3,4,10,10,10,10,10])$
$\gg \mathrm{b}=\mathrm{np}$.arange (5)

$\operatorname{array}([0,1,2,3,4,4,3,2,1,0])$

## 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] $\operatorname{array}([[0,1,2,3,4,5]$,
$[10,11,12,13,14,15]$
$[20,21,22,23,24,25]$,
$[30,31,32,33,34,35]$
$[50,51,52,53,54,55]]$


## Exercise: Array creation

Create the following arrays (with correct data types)
$[[1,1,1,1]$,
$[1,1,1,1]$,
$[1,6,1,1]]$
[[0., 0., 0., 0., 0.]
$[2 ., 0 ., 0 ., 0 ., 0$.
[0., 3., 0., 0., 0.]
[0., 0., 4., 0., 0.],
$[0 ., 0 ., 0 ., 5 ., 0.1$,
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 ame 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
rray([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 # # (!)
>> a = np.arange(10)
>> c=a[::2].copy() # force a copy
>> c[0] = 1
>> 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 num <br> 012345678910111213    <br>  <br> Compute prime numbers in 0-99, with a sieve <br> - 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 $=\operatorname{int}($ np.sqrt(len(is_prime)))
>>> for j in range (2, $\mathrm{N}_{\mathrm{f}}$ max)
is_prime $[2 * \mathrm{j}:: \mathrm{j}]=$ 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 $j$

### 3.1.7 Fancy indexing

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

```
>> np.random.seed (3)
>> a = np.random.random_integers(0, 20, 15)
>> a
array([10, 3, 8, 0, 19, 10, 11, 9, 10, 6, 0, 20, 12, 7, 14])
>> (a % 3 == 0)
array([False, True, False, True, False, False, False, True, False,
```

True, True, False, True, False, False], dtype=bool)
>> mask $=(\mathrm{a} \% 3==0)$
$\gg$ extract_from_a $=a[$
>> extract_from_a $=\mathrm{a}$ [mask] \# or, $a[a \% 3==0]$
>> extract_from_a $\quad$ \# extract $a$ sub-array with the mask
>>
$\operatorname{array}\left(\left[\begin{array}{llll}3, & 0, & 9, & 6,\end{array} 0,12\right]\right)$
Indexing with a mask can be very useful to assign a new value to a sub-array
>> a[a \% 3 == 0] = -
>> a
$\operatorname{rray}([10,-1,8,-1,19,10,11,-1,10,-1,-1,20,-1,7,14])$

## Indexing with an array of integers

>> a = np.arange ( $0,100,10$ )
>> a
Indexing can be done with an array of integers, where the same index is repeated several time:
$\ggg \mathrm{a}[[2,3,2,4,2]]$ \# 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

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

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

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

>>> idx. shape
$(2,2)$
$\ggg>\mathrm{a}$ [idx]
array([[3, 4],

$$
\begin{aligned}
& {[[3,4],} \\
& [9,7]])
\end{aligned}
$$

The image below illustrates various fancy indexing applications
$\ggg \mathrm{a}[(0,1,2,3,4),(1,2,3,4,5)]$
$\operatorname{array}([1,12,23,34,45])$
$\ggg$ a[3:,[0, 2, 5]]
$\operatorname{array}([[30,32,35]$,
$[40,42,45]])$
[50, 52, 55]])
>>> mask $=\operatorname{array}([1,0,1,0,0,1]$
>>> a[mask,2]
$\operatorname{array}([2,22,52])$ dtype=bool)

|  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 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: 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:

$$
\begin{aligned}
& \ggg a=n p . \operatorname{array}([1,2,3,4]) \\
& \ggg a+1 \\
& \text { array([2, 3, 4, 5]) } \\
& \text { >> 2**a } \\
& \text { array([ } 2,4,8,16])
\end{aligned}
$$

All arithmetic operates elementwise:

|  |
| :---: |
|  |  |
|  |  |
|  |  |
|  |  |

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
>> \% timeit [i+1 for in in 1]
1000 loops, best of $3: 861$ us per loop

## Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3)) # NoT matrix multiplication
array([[ 1., 1., 1.],
    [[1.,
```


## Matrix multiplication:

> >>> c.dot (c)
> $\operatorname{array}([[3 ., 3 ., 3],$.
> $\left.\left[\begin{array}{llll}{[3 .,} & 3 ., & 3 .], \\ 3 ., & 3 . & 3\end{array}\right]\right]$

## Exercise: Elementwise operations

- Try simple arithmetic elementwise operations: add even elements with odd elements

Time them against their pure python counterparts using \%timeit.
Generate

$$
\begin{aligned}
& -[2 * * 0,2 * * 1,2 * * 2,2 * * 3,2 * * 4] \\
& -\mathrm{a}_{\mathrm{j}}=2^{\wedge}(3 * j)-j
\end{aligned}
$$

## Other operations

## Comparisons:

>>> $\mathrm{a}=\mathrm{np}$.array ([1, 2, 3, 4])
$\ggg \mathrm{b}=$ np.array $([4,2,2,4])$
$\gg \mathrm{a}==\mathrm{b}$
rray([False, True, False, True], dtype=bool)
$\ggg \mathrm{a}>\mathrm{b}$
Array-wise comparisons:
$\ggg \mathrm{a}=\mathrm{np}$.array ( $[1,2,3,4]$ )
$\ggg \mathrm{b}=$ np.array $(4,2,2,4])$
>>> np. array equal (a, b)
False
>> np. array_equal (a, c)

## ogical 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], dtype=bool
>> np. $\operatorname{logical\_ and(a,~b)~}$
array([ True, False, False, False], dtype=bool)

## Transcendental functions:

| >>> a = np.arange (5) |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: |
| >>> np.sin(a) |  |  |  |  |
| array([ 0 . | , 0.84147098, | 0.90929743 , | 0.14112001 , | -0.7568025 ]) |
| array ( ${ }^{\text {d }}$ | -inf, | 0.69314718 , |  |  |
| >>> np. $\exp (\mathrm{a})$ |  |  |  |  |

$\ggg$ np. $\log (a)$
rray([ -inf, 0. , 0.69314718, 1.09861229, 1.38629436])
array([ 1. $\quad 2.71828183,7.3890561,20.08553692,54.59815003]$ )

## Shape mismatches

>>> a = np. arange (4)
>> a + np.array ([1, 2] )
Fraceback (most recent call last):
File "<stdin>", line 1, in <modul
ValueError: operands could not be broadcast together with shapes (4) (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$)), 1) \quad$ \# see help(np.triu)
>> a
ay ([[ 0., 1., 1.]
$\left[\begin{array}{lll}{\left[\begin{array}{lll}0 ., & 1 ., & 1 .\end{array}\right],} \\ 0 . & 0 ., & 1 .\end{array}\right]$,
>> a.T $\left[\begin{array}{lll}0 ., & 0.1\end{array}\right.$
array([llllll $0 ., 0 ., \quad 0$.
$\left.\begin{array}{lll}{[1 .,} & 0 ., & 0 .], \\ {[1 .,} & 1 ., & 0 .]\end{array}\right]$,
The transposition is a view
As a results, the following code is wrong and will not make a matrix symmetric:
>>> a += a.T
It will work for small arrays (because of buffering) but fail for large one, in unpredictable ways.

## 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 reduction

Computing sums
>> $\mathrm{x}=\mathrm{np} . \operatorname{array}([1,2,3,4])$
>>>
>> x.sum
10
Sum by rows and by columns:
>> $x$
array ([ [1, 1],
>> x.sum(axis=0) \# columns (first dimension)
array ([3, 3])
>> $x[:, 0]$.sum(), $x[:, 1]$ sum()
$(3,3)$
$\ggg \mathrm{x} \cdot \operatorname{sum}(\mathrm{axis}=1)$ \# rows (second dimension)
array ([2, 4])
>> $x[0,:]$ sum (), x[1, :] sum()
$(2,4)$
Same idea in higher dimensions:
>> $\mathrm{x}=\mathrm{np}$. random. rand ( $2,2,2$ )
$\ggg \mathrm{x}$. sum (axis=2)
>>> x. sum (axis=2) $[0,1]$
1.14764
>>> $x[0,1,:]$.sum()
1.14764

Other reductions

- works the same way (and take axis=)

Extrema:
>>> x = np.array ([1, 3, 2])
>> $x=\min ()$
>>
1
$x \cdot m a x(0)$
3
>> x.argmin() \# index of minimu
0
>> x.argmax() \# index of maximu

## Logical operations:

>> np. all([True, True, False])
False
>> np.any ([True, True, False])
2. Numerical operations on arrays

Can be used for array comparisons

|  |
| :---: |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

$\ggg \mathrm{c}=\mathrm{np} . \operatorname{array}([6,4,4,5])$
>> ( $(\mathrm{a}<=\mathrm{b}) \&(\mathrm{~b}<=\mathrm{c}))$.all

## tatistics:

>> $x=n p . \operatorname{array}([1,2,3,1])$
>> $\mathrm{y}=\mathrm{np}$.array ([[1, 2, 3], [5, 6, 1]])
>> x.mean()
. 75
np.median(x)
1.5
$\ggg n$ n.median(y, axis=-1) \# last axis
$\operatorname{array}([$ 2., 5.])
>> x.std() \# full population standard dev.
0. 8291561975888499
... 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: data statistics

Data in populations.txt describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.
You can view the data in an editor, or alternatively in IPython (both shell and notebook):
In [1]: [cat data/populations.txt
First, load the data into a Numpy array:
>>> data $=$ np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data. T \# trick: columns to variables
Then plot it:
>>> from matplotlib import pyplot as plt
>>> plt.axes([0.2, 0.1, $0.5,0.81$ )
>>> plt.axes([0.2, 0.1, $0.5,0.8]$ )
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))


The mean populations over time:
>> populations $=\operatorname{data}[:, 1:]$
>>> populations = mean(axis=0)
array ([ $34080.95238095,20166.66666667,42400$
])
The sample standard deviations:
>>> populations.std(axis=0)
array([ 20897.90645809, 16254.59153691, 3322.50622558])
Which species has the highest population each year?:
>>> np. argmax(populations, axis=1)
$\operatorname{array}([2,2,0,0,1,1,2,2,2,2,2,2,0,0,0,1,2,2,2,2,2])$

## Worked Example: diffusion using a random walk algorithm

${ }^{1 / 21 / 2}$
$\qquad$
et 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:
shuffled jumps
Position : cumulated jumps sum

>> 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:

## >>> $\mathrm{t}=\mathrm{np}$. arange(t_max)

>>> steps $=2$ * np.random.random_integers( 0,1 , (n_stories, t_max)) -
>> np.unique(steps) \# Verification: all steps are 1 or -1
$\operatorname{array}([-1,1])$
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))
<matplotlib.figure. Figure object at
<matplotlib.figure.Figure object at ...>
[<matplotlib.lines.Line2D object at
>>> plt.xlabel(r"\$t\$")
<matplotlib.text.Text object at
S>> plt.ylabel(r"\$\sqrt\{<br>angle (\delta x)~2 \rangle\}\$")
<matplotlib.text.Text object at ...>


### 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:

```
\(\gg \mathrm{a}=\mathrm{np} . \operatorname{tile}(\mathrm{np} . \operatorname{arange}(0,40,10),(3,1)) \cdot T\)
>> a
rray ([[ 0, 0, 0],
    \([10,10,10]\),
    \([20,20,20]\),
\([30,30,30]]\)
>> b = np.array ([0, 1, 2])
>>> a + b
\(\operatorname{array}\left(\left[\begin{array}{lll}0, & 1, & 2] \text {, }\end{array}\right.\right.\)
    \([10,11,12]\),
    \(20,21,22]\),
    \([30,31,32]]\) )
```

We have already used broadcasting without knowing itt:

```
\(\gg \mathrm{a}=\mathrm{np}\). ones \(((4,5))\)
\(\gg \mathrm{a}[0]=2\) \# we assign an array of dimension 0 to an array of dimension 1
>> a
\(\operatorname{rray}\left(\left[\left[\begin{array}{lllll}2 . & 2 ., & 2 ., & 2 . & 2 .\end{array}\right]\right.\right.\)
    \(\left[\begin{array}{lllll}1 ., & 1 ., & 1 ., & 1 ., & 1 .],\end{array}\right.\)
```


## $\left[\begin{array}{lll}1 ., & 1 ., & 1 ., \\ {[1 .,} & 1 .], \\ \text { 1., } & 1 ., & 1 ., \\ 1 ., & 1 . j])\end{array}\right.$

## An useful trick:

```
>> a = np.arange (0, 40, 10)
    >> a.shape
(4,)
>> a = a[:, np.newaxis] \# adds a new axis -> 2D array
    >> a.shape
\((4,1)\)
array([[ 0],
    [10],
    [20],
    [30]J)
>> a + b
\(\operatorname{array}\left(\left[\begin{array}{ll}0, & 1,\end{array}\right]\right.\),
    \(\left[\begin{array}{lll}{[10,} & 21, & 22] \text {, }\end{array}\right.\)
    [30, 31, 327 ]
```

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
>> distance_array $=$ np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
$\operatorname{array}([[\quad 0,198,303,736,871,1175,1475,1544,1913,2448]$
[ 198, 0, 105, 538, 673, 977, 1277, 1346, 1715, 2250]
[ 303, 105, 0, 433, 568, 872, 1172, 1241, 1610, 2145],
$\left.\begin{array}{rrrrrrrr}{[736,} & 538, & 433, & 0, & 135, & 439, & 739, & 808, \\ {[871,} & 673, & 568, & 135, & 0, & 304, & 604, & 673, \\ {[1042} & 1712] \\ 1577\end{array}\right]$
$[1175,977,872,439,304,0,300,369,738,1273]$
[1475, 1277, 1172, 739, 604, 300, $0, \quad 69,438, ~ 973]$
$[1475,1274,1172,139,604,300, \quad 0,69,438,973]$
$[1913,1715,1610,1177,1042,738,438,369,00,535]$,
[2448, 2250, 2145, 1712, 1577, 1273, 973, 904,


A lot of grid-based or network-based problems can also use broadcasting. For instance, if we want to compute he distance from the origin of points on a $10 x 10$ grid, we can do
>> $\mathrm{x}, \mathrm{y}=\mathrm{np}$.arange(5), np.arange (5) [:, np.newaxis]
>> distance $=$ np $\operatorname{sqrt}\left(x_{* *} 2+y * * 2\right)$

Or in color:
>>> plt.pcolor(distance)
>>> plt.colorbar()


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

```
>> \(x, y=n p . o g r i d[0: 5,0: 5]\)
>> \(\mathrm{x}, \mathrm{y}\)
    [1],
        [2],
        [4]]), \(\operatorname{array}([[0,1,2,3,4]])\)
    >> x.shape, y.shape
```

$((5,1),(1,5))$
>> distance = np.sqrt( x ** 2 + y ** 2)

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
$\operatorname{array}\left(\left[\begin{array}{lll}0 & 0, & 0\end{array}\right) 0\right]$
$[1,1,1,1]$,
$2,2,2,2]$,
$3,3,3,3]]$
>> y
$\operatorname{array}([[0,1,2,3]$
$\left[\begin{array}{lll}{[0,1,} & 2, & 3] \\ {[0,} & 1, & 2, \\ 3\end{array}\right]$,
$\left[\begin{array}{l}{[0,1,2,3],} \\ {[0,1,2,3],}\end{array}\right.$
$\left[\begin{array}{l}{[0,1,2,3],} \\ {[0,1,}\end{array}, 3\right]$

### 3.2.4 Array shape manipulation

## Flattening

```
\(\ggg\) a \(=\) navel () .array \(([[1,2,3],[4,5,6]])\)
rray([1, 2, 3, 4, 5, 6])
>> a.T
\(\operatorname{array}([[1,4]\),
    \([2,5]\),
    \([3,6]])\)
>> a.T.ravel(
\(\operatorname{array}([1,4,2,5,3,6])\)
```

Higher dimensions: last dimensions ravel out "first".

## Reshaping

The inverse operation to flattening

```
>> a.shape
\((2,3)\)
\(\ggg \mathrm{b}=\mathrm{a}\)
\(\gg \mathrm{b}\)
>> b = a.ravel()
>> b = b.reshape ( ( 2 ,
\(\ggg b\)
\(\operatorname{array}([[1,2,3]\),
    \([4,5,6]\) )
Or,
>> a.reshape((2, -1)) \# unspecified (-1) value is inferred
\(\operatorname{array}([1,2,3]\),
\[
\begin{gathered}
{[1,2,3]} \\
[4,5,6]])
\end{gathered}
\]
```

$\triangle$ ndarray. reshape may return a view (cf help(np.reshape))), or copy

$$
\begin{aligned}
& \ggg \mathrm{b}[0,0]=99 \\
& \ggg \mathrm{a} \\
& \operatorname{array}\left(\left[\begin{array}{ccc}
{[49,} & 2, & 3],
\end{array}\right.\right.
\end{aligned}
$$

Beware: reshape may also return a copy!

```
>> a = np.zeros((3, 2))
>> b = a.T.reshape(3*2)
```

>> b[0] = 9
>>> a
array ([lllll $0 ., 0$.$] ,$

$$
\left.\begin{array}{ll}
{\left[\begin{array}{ll}
1 & 0, \\
{[0.1,} & 0
\end{array}\right.} \\
{[0 .,} & 0 .
\end{array}\right]
$$

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):
>> $\mathrm{z}=\mathrm{np}$. $\operatorname{array}([1,2,3])$
array ([1, 2, 3])
>> z [:, np. newaxis]
array ([1]
[2],
[3]])
>> z [np.newaxis, :]
$\operatorname{array}([[1,2,3]])$

Dimension shuffling

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

Also creates a view:
>> $\mathrm{b}[2,1,0]=-1$
$\ggg \mathrm{a}[0,2,1]$
$-1$

## 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:
$\ggg \mathrm{b}=\mathrm{a}$
Fiaceback (most recent call last)
File "<stdin>", line 1, in <module>
alueError: cannot resize an array that has been referenced or is
array in this was
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: cannot resize an array that has been referenced or is

## 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
rray([[3, 4, 5],
    [1,1, 2]])
```

Sorts each row separately!
In-place sort:
>>> a. sort(axis=1)
>>> a
rray ([ [3, 4, 5],
Sorting with fancy indexing:

```
\(\gg \mathrm{a}=\mathrm{np} . \operatorname{array}([4,3,1,2])\)
>> j = np.argsort(a)
\(\gg j\)
\(\operatorname{rray}([2,3,1,0])\)
>>> a[j]
\(\operatorname{array}([1,2,3,4])\)
```

Finding minima and maxima:

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


## 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(), lookfor())!
- 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 Scipy lectures to learn the ecosystem, you can directly kip 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 exercices.

### 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
$\operatorname{array}([2.5,3.5,4.5])$
Assignment never changes the type

```
>> a = np.array([1, 2, 3])
*)
>> a[0] = 1.9 # <-- float is truncated to integer
>>> a a
\[
\operatorname{array}([1,2,3])
\]
```

Forced casts:
>>> $\mathrm{a}=\mathrm{np} . \operatorname{array}([1.7,1.2,1.6])$
>> $\mathrm{b}=\mathrm{a}$ astype(int) \# <-- truncates to integer
>> b
rray $([1,1,1])$
Rounding:
>> $\mathrm{a}=\mathrm{np}$. $\operatorname{array}([1.2,1.5,1.6,2.5,3.5,4.5]$ )
$\gg \mathrm{b}=\mathrm{np}$.around (a)
\# still floating-point
2., 2., 2., 4., 4.])
>> $\mathrm{c}=\mathrm{np}$.around (a) astype (int)
$\operatorname{array}([1,2,2,2,4,4])$

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) |
| :--- | :--- |

], dtype=int).dtype
dtype('int64')
nin.info(np.int32).max, 2**31 -
(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)

## Long integers

Python 2 has a specific type for 'long' integers, that cannot overflow, represented with an 'L' at the end. in Python 3, all integers are long, and thus cannot overflow.
>>> np.iinfo(np.int64).max, $2 * * 63$ - 1
(9223372036854775807, 9223372036854775807 L$)$

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
1.1920929e-07
.1020 five
np.float64).eps
$2.2204460492503131 \mathrm{e}-16$
> np.float32(1e-8) + np.float32(1) ==
>> np.float64(1e-8) + np.float64(1) ==
False
Complex floating-point numbers:

| complex64 | two 32-bit floats |
| :--- | :--- |
| complex128 | two 64-bit floats |

complex192 $\quad$ two 96-bit floats, platform-dependent
complex256 $\quad$ 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((1e6,), dtype=np.float64)
In [2]: b = np.zeros((1e6,), dtype=np.float32)
In [3]: \%timeit a*a
1000 loops, best of $3: 1.78 \mathrm{~ms}$ per loop
In [4]: \%timeit b*b
1000 loops, best of $3: 1.07 \mathrm{~ms}$ per loop

- 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


samples = np.zeros((6,), dtype=[( sensor_code, $\left.{ }^{\prime} 4^{\prime}\right)$
>> samples.ndim
>> samples.shape
'sensor_code', 'position', 'value')
>> samples

dtype=[('sensor code', 'S4'), ('position', '<f8'), ('value', '<f8')])

## Field access works by indexing with field names:

>> samples ['sensor_code']
array(['ALFA', 'BETA', 'TAU', 'ALFA', 'ALFA', 'TAU'],
dtype='|S4')
$\begin{array}{lllll}\gg & \text { samples ['value'] } \\ \text { array([ } 0.37, & 0.11, & 0.13, & 0.37, & 0.11,\end{array}$
array([0.37, 0 .
('ALFA', 1.0, 0.37)
>> samples[0]['sensor_code'] = 'TAU
>> samples [0]
('TAU', 1.0, 0.37)
Multiple fields at once:
>> samples[['position', 'value']]
$\operatorname{array}([(1.0,0.37),(1.0,0.11),(1.0,0.13),(1.5,0.37),(3.0,0.11)$
(1.2, 0.13)],
dtype=[('position', '<f8'), ('value', '<f8')])
Fancy indexing works, as usual:
>> samples[samples['sensor_code'] == 'ALFA']
$\operatorname{array}([(' A L F A ', 1.5,0.37),(' A L F A ', 3.0,0.11)]$,

$$
\begin{aligned}
& \text { dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')]) }
\end{aligned}
$$

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:

$$
\begin{aligned}
& \ggg x=n p . \text { ma. array }([1,2,3,4] \text {, mask }=[0,1,0,1]) \\
& \ggg x
\end{aligned}
$$

masked_array (data $=\left[\begin{array}{lll}1 & -- & 3\end{array}\right]$,
mask $=$ [False True False True],
fill_value $=999999$ )

$$
\begin{aligned}
& \ggg y=n p . m a . \operatorname{array}([1,2,3,4], \operatorname{mask}=[0,1,1,1]) \\
& \ggg \mathrm{x}+\mathrm{y} \\
& \text { masked_array (data }=\left[\begin{array}{ll}
2-- & --
\end{array}\right],
\end{aligned}
$$

$$
\begin{aligned}
\text { y (data } & =[\text { malse True True True }],
\end{aligned}
$$

$$
\begin{aligned}
\text { mask } & =\text { [False } \\
\text { fill_value } & =999999)
\end{aligned}
$$

- Masking versions of common functions:

$$
\begin{aligned}
\gg \text { np.ma.sqrt }([1, & -1,2,-2]) \\
\text { masked_array(data } & =[1.0--1.41421356237 \ldots-\text { - }] \\
\text { mask } & =[\text { False } \quad \text { True False True }], \\
\text { fill_value } & =1 \mathrm{e}+20)
\end{aligned}
$$

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 Con ventions page (to manage help strings).

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


### 3.4 Advanced operations

## Section contents <br> - Polynomials <br> - Loading data files

### 3.4.1 Polynomials

Numpy also contains polynomials in different bases:
For example, $3 x^{2}+2 x-1$ :

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

>>> $x=n p$. linspace ( $0,1,20$ )
>> $y=\cos (x)+0$ 3*np random. rand (20
>> $\mathrm{p}=\mathrm{np} . \operatorname{poly1d}(\mathrm{np} . \operatorname{polyfit}(\mathrm{x}, \mathrm{y}, 3)$ )
>> t = np.linspace (0, 1, 200)
>> plt.plot(x, y, 'o', t, p(t), '-'
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ....>]


See http://docs.scipy.org/doc/numpy/reference/routines.polynomials.polyld.html for more.

More polynomials (with more bases)
Numpy also has a more sophisticated polynomial interface, which supports e.g. the Chebyshev basis
$3 x^{2}+2 x-1$ :

```
>> p = np.polynomial.Polynomial([-1, 2, 3]) # coefs in different order!
>>> p(0)
>> p.roots()
rray([-1.
>> p.degree() , 0.33333333])
>>>
```

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

```
x = np.linspace(-1, 1, 2000)
    >> y =np.cos(x) + 0.3*np.random.rand (2000)
    >> = np.polynomial.Chebyshev.fit(x, y, 90)
>> t = np.linspace(-1, 1, 200)
>> plt.plot(x,y, 'r.')
    [<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t, p(t), '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
$1900 \quad 30 \mathrm{e} 3 \quad 4 \mathrm{e} 3 \quad 48300$
$1901 \quad 47.2 \mathrm{e} 3 \quad 6.1 \mathrm{e} 3 \quad 48200$
$1902 \quad 70.2 \mathrm{e} 3 \quad 9.8 \mathrm{e} 3 \quad 41500$
$1903 \quad 77.4 \mathrm{e} 3 \quad 35.2 \mathrm{e} 3 \quad 38200$

```
>> data \(=\) np.loadtxt('data/populations.txt')
>> data
array([[ 1900., 30000., 4000., 48300.], \(\begin{array}{llll}{[\text { 1901., }} & 47200 ., & 6100 ., & 48200 .], \\ {[\text { 1902., }} & 70200 ., & 9800 ., & 41500 .],\end{array}\)
```

>> np. savetxt('pop2.txt', data)
> data2 $=$ np.loadtxt('pop2.txt')
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
'/home/user/stuff/2011-numpy-tutorial ${ }^{\text {\# show current direct }}$
In [2]: cd ex
/home/user/stuff/2011-numpy-tutorial/ex'
In [3]: 1s
populations.txt species.txt

## mages

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', 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:
>>> from scipy.misc import imsave
>>> imsave('tiny_elephant.png', img[::6,::6])
>>> 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 ( $(3,3)$ )
>> 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):
$[[1,6,11]$,
$[2, \quad 7,12]$,
$[4, \quad 9,14]$
$[5,10,15]]$
and generate a new array containing its 2 nd and 4 th rows.
2. Divide each column of the array
>>> import numpy as np
>>> $\mathrm{a}=\mathrm{np}$.arange $(25) \cdot$ reshape $(5,5)$
elementwise with the array $\mathrm{b}=\mathrm{np}$.array ([1., 5, 10, 15, 20]). (Hint: np.newaxis).
3. Harder one: Generate a $10 \times 3$ array of random numbers (in range [0,1]). For each row, pick the number closest to 0.5 .

- Use abs and argsort 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 racoon. scipy provides a 2D array of this image with the scipy.misc.face function:
>>> from scipy import misc
>> face = misc.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 pylab to display the image
>> import pylab as plt
>> face = misc.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 ox...>- 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
>> $\mathrm{y}, \mathrm{x}=\mathrm{np}$.ogrid[0:sy, $0: \mathrm{sx}] \# x$ and $y$ indices of pixels
$\gg y$.shape, $x$.shape
>>> centerx, centery $=(660,300)$ \# center of the image
>>> mask $=((\mathrm{y}$ - centery $) * * 2+(\mathrm{x}$ - 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, $0.5,0.8])$
matplotlib.axes...Axes object at >
>> 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 $\mathrm{f}(\mathrm{a}, \mathrm{b}, \mathrm{c})$ that returns $a^{b}-c$. Form a $24 \times 12 \mathrm{x} 6$ array containing its values in parameter ranges $[0,1] \times[0,1] \times[0,1]$.
Approximate the 3-d integral

$$
\int_{0}^{1} \int_{0}^{1} \int_{0}^{1}\left(a^{b}-c\right) d a d b d c
$$

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 $[0: 1: 20 \mathrm{j}]$.)

## Reminder Python functions:

## def $f(a, b, c)$ : <br> 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 $=50$
$c=x+1 j * y$
for $j$ in xrange (N_max):
$z=z * * 2+c$
Point ( $\mathrm{x}, \mathrm{y}$ ) belongs to the Mandelbrot set if $|c|$ < some_threshold.
Do this computation by:

1. Construct a grid of $\mathrm{c}=\mathrm{x}+1 \mathrm{j}^{*} \mathrm{y}$ values in range $[-2,1] \mathrm{x}[-1.5,1.5]$
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:

$$
\begin{aligned}
& \text { >> import matplotlib. pyplot as plt } \\
& \ggg \text { plt.imshow(mask.T, extent }=[-2,1,-1.5,1.5] \text { ) } \\
& \text { >matplotlib.image.AxesImage object at ...> }
\end{aligned}
$$

## Matplotlib: plotting



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: $p_{\text {new }}=P^{T} p_{\text {old }}$
3. $\operatorname{all}(\operatorname{sum}(P, a x i s=1)==1), p . \operatorname{sum}()==1$ 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 le-5

Toolbox: np.random.rand, .dot(), np.linalg.eig, reductions, abs(), argmin, comparisons, all, np. linalg.norm, etc.
Solution: Python source file

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 exercise

Beyond this tutoria

- Quick references


### 4.1 Introduction

Matplotlib is probably the single most used Python package for 2D-graphics. It provides both a very 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 and the matplotlib mode

IPython is an enhanced interactive Python shell that has lots of interesting features including named inputs and outputs, access to shell commands, improved debugging and many more. It is central to the scientific computing workflow in Python for its use in combination with Matplotlib:
For interactive matplotlib sessions with Matlab/Mathematica-like functionality, we use IPython with it's spe cial Matplotlib mode that enables non-blocking plotting.

IPython console When using the IPython console, we start it with the command line argumen --matplotlib (-pylab in very old versions)
IPython notebook In the IPython notebook, we insert, at the beginning of the notebook the following magic:
\%matplotlib inline

### 4.1.2 pyplot

pyplot provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab ${ }^{\mathrm{TM}}$. Therefore, the majority of plotting commands in pyplot have Matlab ${ }^{\mathrm{TM}}$ analogs with similar arguments. Important commands are explained with interactive examples.
from matplotlitb import pyplot as plt

### 4.2 Simple plot

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
= np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, $S=n p \cdot \cos (X), n p \cdot \sin (X)$
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 --pylab
This brings us to the IPython prompt:
IPython 0.13 -- An enhanced Interactive Python
? $\quad->$ Introduction to IPython's features
help $\rightarrow$ Python's own help system
object? -> Details about 'object'. ?object also works, ?? prints more
elcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'
You can also download each of the examples and run it using regular python, but you will loose interactive data manipulation:
\$ python exercice_1.py
You can get source for each step by clicking on the corresponding figure.

### 4.2.1 Plotting with default settings



Documentation

- plot tutorial
- plot(0 command

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

$$
\begin{aligned}
& \text { import numpy as } \mathrm{np} \\
& \text { import matplotlib.pyplot as plt }
\end{aligned}
$$

$\mathrm{X}=\mathrm{np}$. linspace (-np.pi, np.pi, 256, endpoint=True)
C, $\mathrm{S}=\mathrm{np} \cdot \cos (\mathrm{X}), \mathrm{np} \cdot \sin (\mathrm{X})$
plt.plot(X, C)
plt.plot(X, S)
plt.show()

### 4.2.2 Instantiating defaults



## Documentation

- Customizing matplotlib

In the script below, we've instantiated (and commented) all the figure settings that influence the appearance of the plot.
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 $8 x 6$ inches, 80 dots per inch
plt.figure(figsize=(8, 6), dpi=80)
Create a new subplot from a grid of $1 x$
plt.subplot(1, 1, 1)
= np.linspace(-np.pi, np.pi, 256, endpoint=True)
$\mathrm{C}, \mathrm{s}=\mathrm{np} \cdot \cos (\mathrm{X}), \mathrm{np} \cdot \sin (\mathrm{X})$
\# 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)
\# Plot sine with a green continuous line of width 1 (pixels)
\# Set $x$ limits
plt.xlim ( $-4.0,4.0)$
\# Set $x$ ticks
plt.xticks(np.linspace(-4, 4, 9, endpoint=True))
\# Set y limits
plt. $\mathrm{ylim}(-1.0,1.0)$

## \# Set y ticks

plt.yticks(np.linspace(-1, 1, 5, endpoint=True))
\# Save figure using 72 dots per inch
\# plt.savefig("exercice_2.png", dpi=72)
\# Show result on screen
plt.show()

### 4.2.3 Changing colors and line widths



## Documentation

- Controlling line properties
- Line API

First step, we want to have the cosine in blue and the sine in red and a slighty 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



Documentation

- xlim0 comman
- ylim0 command

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.

```
Ml. xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)
```


### 4.2.7 Moving spines



## Documentation

- Spines
- Axis containe
- Transformations tutorial
spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can b
 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.
x = plt.gca() \# gca stands for 'get current axis
x.spines['right'].set_color('none')
ax. spines['top']. set_color('none')
ax. spines ['bottom'].set position(('data'
$x$.spines yaxis set ticks position('left')
ax. spines['left'].set_position(('data',0)


### 4.2.8 Adding a legend



## Documentation

- Legend guid
- legend0 command
- Legend API

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



Documentation

- Annotating axis
- annotate() command

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.
= $2 *$ 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 $\left(r^{\prime} \$ \sin (\backslash f r a c\{2 \backslash p i\}\{3\})=\backslash f r a c\{\backslash \operatorname{sqrt}\{3\}\}\{2\} \$ '\right.$
$x y=(t, n p . \sin (t))$, $x y c o o r d s=1$ data ,
xytext $=(+10,+30)$, textcoords='offset points', fontsize=16,
arrowprops=dict(arrowstyle=" $>$ ", connectionstyle="arc3, rad= $=.2$ "))
plt.plot([t, t$],[0, \mathrm{np} . \sin (\mathrm{t})]$, color='red', linewidth=2.5, linestyle="--")
plt. scatter $[[t],,[n p . \sin (t)], 50,$, color='red')
plt.annotate (r'\$cos (\frac\{2\pi\}f3\})=-\frac\{1\}\{2\}\$',
$\mathrm{xy}=(\mathrm{t}, \mathrm{np} . \cos (\mathrm{t}))$, $\mathrm{xycoords=}{ }^{\prime} \mathrm{data}^{\prime}$,
xytext $=(-90,-50)$, textcoords=' offset points', fontsize $=16$,
arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

### 4.2.10 Devil is in the details



Documentation

- Artists
- BBox

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 "sub plots".
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 $\operatorname{gcf}()$ 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

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 to the normal Python way starting fis

| Argument | Default | Description |
| :--- | :--- | :--- |
| num | 1 | number of figure |
| figsize | figure.figsize | figure size in 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 |

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

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.

| subplot(2,1,1) | subplot(1,2,1) | subplot(1,2,2) | subplot(2, 2, 1) | subplot(2, 2, 2) | Axes 1 |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  |  |  | Axes 2 |  | Axes 3 |
| subplot(2,1,2) |  |  | subplot(2, 2, 3) | subplot(2, 2,4) | Axes 4 | Axes 5 |  |

### 4.3.3 Axes

Axes are very similar to subplots but allow placement of plots at any location in the fig ure. So if we want to put a smaller plot inside a bigger one we do so with axes.


### 4.3.4 Tick

Well formatted ticks are an important part of publishing-ready figures. Matplotlib provides a totally config urable 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:
$\mathrm{ax}=\mathrm{plt} . \mathrm{gca}()$
ax. xaxis.set_major_locator(eval(locator))


There are several locators for different kind of requirements: locators in matplotlib.dates.

### 4.4 Other Types of Plots: examples and exercises



### 4.4.1 Regular Plots



You need to use the fill between command.
Starting from the code below, try to reproduce the graphic on the right taking care of filled areas:
$\mathrm{n}=256$
$\mathrm{X}=\mathrm{np}$. linspace (-np.pi, np.pi, n , endpoint=True)
$\mathrm{Y}=\mathrm{np} \cdot \sin (2 * \mathrm{X})$
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


Color is given by angle of (X,Y).
Starting from the code below, try to reproduce the graphic on the right taking care of marker size, color and ransparency.

## = 102

$\mathrm{X}=\mathrm{np}$. random.normal $(0,1, \mathrm{n})$
$\mathrm{Y}=\mathrm{np}$. random. normal $(0,1, \mathrm{n})$
plt.scatter(X, Y )
Click on figure for solution

### 4.4.3 Bar Plots



You need to take care of text alignment.
Starting from the code below, try to reproduce the graphic on the right by adding labels for red bars.
$\mathrm{n}=12$
$\mathrm{X}=\mathrm{np}$. arange ( n )
$\mathrm{Y}_{1}=(1-\mathrm{X} / \mathrm{float}(\mathrm{n})) * \mathrm{np}$. random.uniform(0.5, 1.0, n)
$\mathrm{Y} 2=(1-\mathrm{X} / \mathrm{float}(\mathrm{n})) * \mathrm{np}$. random.uniform(0.5, 1.0, n$)$
plt. $\operatorname{bar}(\mathrm{X},+\mathrm{Y} 1$, facecolor='\#9999ff', edgecolor='white')
plt. $\operatorname{bar}(\mathrm{X},-\mathrm{Y} 2$, facecolor='\#ff9999', edgecolor='white')
for $\mathrm{x}, \mathrm{y}$ in $\mathrm{zip}(\mathrm{X}, \mathrm{Y} 1)$ :
plt.text( $\mathrm{x}+0.4, \mathrm{y}+0.05,1 \% .2 f \mathrm{y}, \mathrm{y}, \mathrm{ha}=$ 'center', va='bottom')
plt.ylim(-1.25, +1.25)
Click on figure for solution.

### 4.4.4 Contour Plots



You need to use the clabel command.
Starting from the code below, try to reproduce the graphic on the right taking care of the colormap (see Colormaps below)
def $f(x, y):$
$\quad$ return ( $1-x / 2+x * * 5+y * * 3) * n p \cdot \exp (-x * * 2-y * * 2)$
$\mathrm{n}=256$
$\mathrm{x}=\mathrm{np} .1$ inspace $(-3,3, \mathrm{n})$
$\mathrm{y}=\mathrm{np}$. linspace $(-3,3, \mathrm{n})$
$\mathrm{X}, \mathrm{y}=\mathrm{np} \cdot \operatorname{meshgrid}(\mathrm{x}, \mathrm{y})$
plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='jet')
$\mathrm{C}=\mathrm{plt} . \operatorname{contour}(\mathrm{X}, \mathrm{Y}, \mathrm{f}(\mathrm{X}, \mathrm{Y}), 8$, colors='black', linewidth=$=5)$
Click on figure for solution

### 4.4.5 Imshow



You need to take care of the origin of the image in the imshow command and use a col
Starting from the code below, try to reproduce the graphic on the right taking care of colormap, image inter polation and origin.

```
ef f(x,y):
```

= 10
= np.linspace $(-3,3,4 * n)$
$y=n p . l i n s p a c e(-3,3,3 * n)$
$\mathrm{X}, \mathrm{y}=\mathrm{np} \cdot \mathrm{meshgrid}(\mathrm{x}, \mathrm{y})$
plt.imshow(f(X, Y)
Click on the figure for the solution

### 4.4.6 Pie Charts



You need to modify Z.
Starting from the code below, try to reproduce the graphic on the right taking care of colors and slices size
= np.random.uniform (0, 1, 20)
lt. pie(Z)
Click on the figure for the solution.

### 4.4.7 Quiver Plots



You need to draw arrows twice
Starting from the code above, try to reproduce the graphic on the right taking care of colors and orientations.
$\mathrm{n}=8$
$\mathrm{X}, \mathrm{Y}=\mathrm{np} \operatorname{mgrid}[0: n, 0: n]$
plt.quiver(X, Y)
Click on figure for solution.

### 4.4.8 Grids

$\square$
aking care of line styles
xes $=\mathrm{plt} . \mathrm{gca}()$
axes.set_xlim( 0,4 )
axes. set_xticklabels([])
xes.set_yticklabels([])
Click on figure for solution.

### 4.9 Multi Plots



You can use several subplots with different partition
Starting from the code below, try to reproduce the graphic on the right.

## plt subplot (2, 2, 1)

plt.subplot $(2,2,4)$
Click on figure for solution.

### 4.4.10 Polar Axis



You only need to modify the axes line
Starting from the code below, try to reproduce the graphic on the right.

```
plt.axes([0, 0, 1, 1])
N = 20
theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
radii = 10* np.random.rand(N)
madii = 10* np.random.rand(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)
for r, bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r / 10.))
    bar.set_facecolor(c)
```

Click on figure for solution.

### 4.4.11 3D Plots



You need to use contourf
Starting from the code below, try to reproduce the graphic on the right.
from mpl_toolkits.mplot3d import Axes3D
fig $=$ plt.figure (
$\mathrm{ax}=\mathrm{Axes3D}(\mathrm{fig})$
$\mathrm{X}=\mathrm{np} . \operatorname{arange}(-4,4,0.25)$
$\mathrm{Y}=\mathrm{np}$. arange $(-4,4,0.25)$
$X, Y=n p . \operatorname{meshgrid}(X, Y)$
$R=n p . \operatorname{sqrt}(X$
$Z=n p \cdot \sin (R)$
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
Click on figure for solution.
See also:
3D plotting with Mayavi

### 4.4.12 Text



Have a look at the matplotlib logo.
Try to do the same from scratch !
Click on figure for solution.

## Quick read

If you want to do a first quick pass through the Scipy 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 tex
- Annotating te
- Introduction
- Customizing your objects
- Object containers
- Figure container
- Axes container
- Axis container
- Tick container
- 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


### 2.5.3 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, **kwargs)
:class: : $\sim$ matplotlib.axes.Axes`. *args* is a variable length
argument, allowing for multiple $*_{x} *$, *y* pairs with an
optional format string For example, each of the following is legal:
plot(x, y) \# plot x and y using default line style and color
plot(x, y, 'bo') \# plot $x$ and $y$ using blue circle markers
plot(y) \# plot $y$ using $x$ as index array 0 ..N-1
plot(y, 'r+') \# ditto, but with red plusses
If $*_{\mathrm{X} *}$ and/or $*_{\mathrm{y}}$ * is 2 -dimensional, then the corresponding columns will be plotted.

### 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 <br> 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 | - - M - - - - |
| 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 | $\square \square \square \square \square \square \square \square \square \square$ |
| markersize (ms) | size of the marker in points | - • व ロ $\quad \square \quad \square \quad \square \square \square \square$ |

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, checks Documenting the matplotlib colormaps.


## Scipy: high-level scientific computing

Authors: Adrien Chauve, Andre Espaze, Emmanuelle Gouillart, Gaël Varoquaux, 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, optimiza tion, image processing, statistics, special functions, etc.
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 imple mented 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 routine are optimized and tested, and should therefore be used when possible.

## Chapters content

- File input/output: scipy.io
- Special functions: scipy. special
- Linear algebra operations: scipy. linalg
- Fast Fourier transforms: scipy.fftpack
- Optimization and fit: scipy.optimize
- Statistics and random numbers: scipy . stats
- Interpolation: scipy interpolate
- Numerical integration: scipy. integrate
- Signal processing: scipy. signal
- Image processing: scipy. ndimad
- Summary exercises on scientific computing
$\triangle$ 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.fftpack | 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 |
| scip. 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 |

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
>>> from scipy import stats \# same for other sub-modules
The main scipy namespace mostly contains functions that are really numpy functions (try scipy.cos is np. cos). Those are exposed for historical reasons only; there's usually no reason to use import scipy in your code

### 5.1 File input/output: scipy.io

- Loading and saving matlab files:
$\ggg$ from scipy import io as spio
>>> $\mathrm{a}=\mathrm{np}$.ones ( $(3,3)$ )
>> spio.savemat('file.mat', \{'a': a\}) \# savemat expects a dictionary
>> data = spio.loadmat('file.mat', struct_as_record=True)
array([ [ 1 .
rray([[ 1., 1., 1.],
$\left[\begin{array}{lll}{[1 .,} & 1 ., & 1 .], \\ {[1 .,} & 1 ., & 1 .]\end{array}\right)$
- Reading images:
>>> from scipy import misc
>>> misc. imread('fnane png')
>> misc.im
array (...)
>> \# Matplotlib also has a similar function
>>> import matplotlib.pyplot as plt
>> plt.imread('fname.png')
array (...)
See also
- Load text files: numpy. loadtxt ()/numpy . savetxt ()
- Clever loading of text/csv files: numpy.genfromtxt ()/numpy .recfromcsv()
- Fast and efficient, but numpy-specific, binary format: numpy . save()/numpy.load()


### 5.2 Special functions: scipy.special

Special functions are transcendental functions. The docstring of the scipy. special module is well-written, so we won't list all functions here. Frequently used ones are

- Bessel function, such as scipy . special.jn() (nth integer order Bessel function)
- Elliptic function (scipy .special .ellipj() for the Jacobian elliptic function, ...)
- Gamma function: scipy.special.gamma(), also note scipy.special.gammaln() which will give th log of Gamma to a higher numerical precision.
- Erf, the area under a Gaussian curve: scipy .special.erf ()


### 5.3 Linear algebra operations: scipy.linalg

The scipy.linalg module provides standard linear algebra operations, relying on an underlying efficient implementation (BLAS, LAPACK)

- The scipy. linalg. det () function computes the determinant of a square matrix
>> from scipy import linalg
>> arr = np. array ( $[1,2]$
$\left[\begin{array}{l}[3,4]]\end{array}\right.$
>> linalg.det(arr)
$-2.0$
$\ggg \operatorname{arr}=\operatorname{np} \cdot \operatorname{array}\left(\left[\begin{array}{ll}{[3,2],} \\ [6,4]])\end{array}\right.\right.$
>> linalg.det(arr)
0.0
>> linalg.det(np.ones ((3, 4))
Traceback (most recent call last)
ValueError: expected square matrix
Traceback (most recent call last)
ValueError: expected square matri
- The scipy. linalg.inv () function computes the inverse of a square matrix

>> iarr $=$ linalg. inv(arr)
>> iarr
$\operatorname{array}([[-2 ., 1$.$] ,$
$[1.5,-0.5]])$
>> np.allclose(np.dot (arr, iarr), np.eye(2))
Finally computing the inverse of a singular matrix (its determinant is zero) will raise LinAlgError:

$$
\begin{aligned}
& \ggg \operatorname{arr}=\text { np } \cdot \operatorname{array}([[3,2], \\
& [6,4]])
\end{aligned}
$$

>> linalg.inv(arr)
ror: singular matrix

LinAlgError: singular matrix

- More advanced operations are available, for example singular-value decomposition (SVD):


The signal frequency can be found by:
>> freq $=$ freqs[power.argmax()]
>> np.allclose(freq, 1./period) \# check that correct freq is found
True
Now the high-frequency noise will be removed from the Fourier transformed signal:
>>> sig_fft[np.abs(sample_freq) > freq] = 0
The resulting filtered signal can be computed by the scipy.fftpack.ifft() function:
>> main_sig = fftpack.ifft(sig_fft)
The result can be viewed with:
>>> import pylab as plt
>> plt.figure()
matplotlib.figure.Figure object at 0x...>
>> plt.plot(time_vec, sig)
[<matplotlib.lines.Line2D object at 0x...>]
>> plt.plot(time_vec, main_sig, linewidth=3)
<matplotilib. ines.Line2D object at 0x...>]
$\gg$ plt.xlabel('Time [s] ')
matplotlib.text.Text object at 0x...>
<matplotlib.text.Text object at 0x...>

numpy.fft
Numpy also has an implementation of FFT (numpy.fft). However, in general the scipy one should be preferred, as it uses more efficient underlying implementations.

## Worked example: Crude periodicity finding




## 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 pylab.imread ().
3. Find and use the 2-D FFT function in scipy.fftpack, and plot the spectrum (Fourier transform of) the image. Do you have any trouble visualising the spectrum? If so, why?
. 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.

### 5.5 Optimization and fit: scipy.optimize

Optimization is the problem of finding a numerical solution to a minimization or equality.
The scipy.optimize module provides useful algorithms for function minimization (scalar or multi dimensional), curve fitting and root finding
>> from scipy import optimize

## Finding the minimum of a scalar function

Let's define the following function:
>>> def $f(x)$ :
.. return $\mathrm{x} * * 2+10 * \mathrm{np} . \sin (\mathrm{x})$
and plot it:
>>> $\mathrm{x}=\mathrm{np}$.arange(-10, 10, 0.1)
>> plt.plot(x, $f(x)$ )
>> plt.show()


This function has a global minimum around -1.3 and a local minimum around 3.8.
The general and efficient way to find a minimum for this function is to conduct a gradient descent starting from a given initial point. The BFGS algorithm is a good way of doing this:
>> optimize.fmin_bfgs(f, 0)
Optimization terminated successfully
Current function value: -7.945823
Iterations: 5
Function evaluations: 24
Gradient evaluations: 8
array ([-1.30644003])
A possible issue with this approach is that, if the function has local minima the algorithm may find these local minima instead of the global minimum depending on the initial point:
>>> optimize $f$ min bfgs(f, 3, disp=0)
rray ([ 3.83746663])
If we don't know the neighborhood of the global minimum to choose the initial point, we need to resort to costlier global optimization. To find the global minimum, we use scipy. optimize basinhopping () (which combines a local optimizer with stochastic sampling of starting points for the local optimizer):
New in version 0.12.0: basinhopping was added in version 0.12 .0 of Scipy
>>> optimize.basinhopping(f, 0$)$
nfev: 1725
minimization_failures: 0
fun: -7.9458233756152845
x: array([-1.30644001])
message: $[$ [requested number of basinhopping iterations completed successfully']
njev: 575
nit: $: 100$

Another available (but much less efficient) global optimizer is scipy.optimize.brute() (brute force optimization on a grid). More efficient algorithms for different classes of global optimization problems exist, but this is out of the scope of scipy. Some useful packages for global optimization are OpenOpt, IPOPT, PyGMO and PyEvolve.
scipy used to contain the routine anneal, it has been deprecated since SciPy 0.14.0 and removed in SciPy 0.16.0.

To find the local minimum, let's constraint the variable to the interval $(0,10)$ using scipy.optimize.fminbound():
>>> xmin_local $=\operatorname{optimize} . f$ fminbound $(f, 0,10)$
>> xmin_loca
. 8374671 .
Finding minima of function is discussed in more details in the advanced chapter: Mathematical optimization: finding minima of functions.

## inding the roots of a scalar function

To find a root, i.e. a point where $f(x)=0$, of the function $f$ above we can use for example sipy.optimize.fsolve()
>> root = optimize.fsolve(f, 1) \# our initial guess is
>> root
Note that only one root is found. Inspecting the plot of $f$ reveals that there is a second root around -2.5 . We find the exact value of it by adjusting our initial guess:
>> root2 $=$ optimize.fsolve(f, -2.5)
>> root2
array ([-2.47948183])

## Curve fitting

Suppose we have data sampled from $f$ with some noise:
>>> xdata $=n$ n. linspace ( $-10,10$, num=20)

Now if we know the functional form of the function from which the samples were drawn $\left(x^{\wedge} 2+\sin (x)\right.$ in his case) but not the amplitudes of the terms, we can find those by least squares curve fitting First we have to define the function to fit:
$\ggg$ def $\mathrm{f} 2(\mathrm{x}, \mathrm{a}, \mathrm{b})$ :
. $\quad$ return $\mathrm{a} * \mathrm{x} * * 2+\mathrm{b} * \mathrm{np} . \sin (\mathrm{x})$
Then we can use scipy.optimize.curve_fit () to find a and b:
>>> guess $=[2,2]$
>> params, params_covariance = optimize.curve_fit(f2, xdata, ydata, guess
>> params
array([ 0.99667386, 10.17808313])
Now we have found the minima and roots of f and used curve fitting on it, we put all those resuls together in a single plot:


In Scipy >= 0.11 unified interfaces to all minimization and root finding algorithms are available: scipy optimize minimize() scipy oltimize minimize scalar() and scipy optimize root(). They allow comparing various algorithms easily through the method keyword.

You can find algorithms with the same functionalities for multi-dimensional problems in scipy.optimize.

## Exercise: Curve fitting of temperature data

The temperature extremes in Alaska for each month, starting in January, are given by (in degrees Celcius):

$$
\begin{array}{lrrrrrrr}
\max : & 17, & 19, & 21, & 28, & 33, & 38, & 37, \\
\min : & -62, & -59, & -56, & -46, & -32, & -18, & -9, \\
\hline
\end{array}
$$

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?
6. 

## Exercise: 2-D minimization

## Six-hump Camelback function



The six-hump camelback function

$$
f(x, y)=\left(4-2.1 x^{2}+\frac{x^{4}}{3}\right) x^{2}+x y+\left(4 y^{2}-4\right) y^{2}
$$

has multiple global and local minima. Find the global minima of this function. Hints:

- Variables can be restricted to $-2<\mathrm{x}<2$ and $-1<\mathrm{y}<1$.
- Use numpy . meshgrid () and pylab. imshow () to find visually the regions.
- Use scipy .optimize.fmin_bfgs() or another multi-dimensional minimizer.

How many global minima are there, and what is the function value at those points? What happens for an initial guess of $(x, y)=(0,0)$ ?

See the summary exercise on Non linear least squares curve fitting: application to point extraction in topo graphical lidar data for another, more advanced example.

### 5.6 Statistics and random numbers: scipy.stats

The module scipy . stats contains statistical tools and probabilistic descriptions of random processes. Random number generators for various random process can be found in numpy. random.

### 5.6.1 Histogram and probability density function

Given observations of a random process, their histogram is an estimator of the random process's PDF (probability density function)

```
>> a = np.random.normal(size=1000)
>> bins = np.arange(-4, 5)
>> bins
array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
>> histogram = np.histogram(a, bins=bins, normed=True)[0]
>> bins = 0.5*(bins[1:] + bins[:-1])
rray([-3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5])
>> from scipy import stats
>>> from = stats.norm.pdf(bins) # norm is a distribution
>> plt.plot(bins, histogram)
[<matplotlib.lines.Line2D object at ...>]
>> plt.plot(bins, b)
[<matplotlib.lines.Line2D object at ...>]
```



If we know that the random process belongs to a given family of random processes, such as normal processes we can do a maximum-likelihood fit of the observations to estimate the parameters of the underlying distribution. Here we fit a normal process to the observed data:

$$
\begin{aligned}
& \text { >>> loc, std = stats.norm.fit(a) } \\
& \ggg 10 c
\end{aligned}
$$

$$
\begin{aligned}
& \text { >>> loc } \\
& \text { 0.0314345570... } \\
& \text { >>> std }
\end{aligned}
$$

0.9778613090

## Exercise: Probability distributions

Generate 1000 random variates from a gamma distribution with a shape parameter of 1 , then plot a histogram from those samples. Can you plot the pdf on top (it should match)?
Extra: the distributions have a number of useful methods. Explore them by reading the docstring or by using IPython tab completion. Can you find the shape parameter of 1 back by using the fit method on your random variates?

### 5.6.2 Percentiles

The median is the value with half of the observations below, and half above:
>>> np. median (a)
0.04041769593.
It is also called the percentile 50 , because $50 \%$ of the observation are below it
>>> stats.scoreatpercentile(a, 50 )
0.0404176959 .

Similarly, we can calculate the percentile 90
>> stats.scoreatpercentile(a, 90)
1.3185699120.

The percentile is an estimator of the CDF: cumulative distribution function.

### 5.6.3 Statistical tests

A statistical test is a decision indicator. For instance, if we have two sets of observations, that we assume are generated from Gaussian processes, we can use a T-test to decide whether the two sets of observations are significantly different:

## >> a = np.random.normal ( 0,1 , size $=100$ ) <br> $\gg \mathrm{b}=\mathrm{np} \cdot$ random.normal (1,, size $=10$ ) <br> >> stats.ttest_ind (a, b)

(array( $-3.177574054 \ldots$...), $0.0019370639 \ldots$ )
The resulting output is composed of:

- The T statistic value: it is a number the sign of which is proportional to the difference between the two
random processes and the magnitude is related to the significance of this difference.
- the p value: the probability of both processes being identical. If it is close to 1 , the two process are almost certainly identical. The closer it is to zero, the more likely it is that the processes have different means.


## See also:

The chapter on statistics introduces much more elaborate tools for statistical testing and statistical data load ng and visualization outside of scipy.

### 5.7 Interpolation: scipy.interpolate

The scipy.interpolate is useful for fitting a function from experimental data and thus evaluating points where no measure exists. The module is based on the FITPACK Fortran subroutines from the netlib project.
By imagining experimental data close to a sine function
>> measured_time $=$ np.linspace $(0,1,10)$
$\ggg$ noise $=($ np. random.random $(10) * 2-1) * 1$
>> noise $=($ np. random. random $(10) * 2-1) * 1 e-1$
$\gg$ measures $=\mathrm{np} \cdot \sin (2 * \mathrm{np} \cdot \mathrm{pi} *$ measured_time $)+$ noise
The scipy .interpolate. interp1d class can build a linear interpolation function:
>> from scipy.interpolate import interp1
>> linear_interp = interp1d(measured_time, measures)
Then the scipy.interpolate.linear_interp instance needs to be evaluated at the time of interest:
>>> computed_time $=$ np.linspace ( $0,1,50$ )
\$>> linear_results = linear_interp(computed_time)
A cubic interpolation can also be selected by providing the kind optional keyword argument:
>> cubic_interp $=$ interp1d(measured_time, measures, kind='cubic')
>> cubic_results $=$ cubic_interp(computed_time)
The results are now gathered on the following Matplotlib figure:

cipy.interpolate.interp2d is similar to scipy.interpolate.interp1d, but for 2-D arrays. Note that for the interp family, the computed time must stay within the measured time range. See the summary exercis on Maximum wind speed prediction at the Sprogo station for a more advance spline interpolation example.

### 5.8 Numerical integration: scipy.integrate

The most generic integration routine is scipy. integrate.quad ():
>>> from scipy.integrate import quad
>> res, err $=$ quad(np.sin, 0 , np.pi/2
rue np.allclose(res, 1)
True
True
Others integration schemes are available with fixed_quad, quadrature, romberg.
scipy. integrate also features routines for integrating Ordinary Differential Equations (ODE). In particula cipy.integrate.odeint () is a general-purpose integrator using 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.
odeint solves first-order ODE systems of the form
dy/dt = rhs(y1, y2, .., t0,...)

As an introduction, let us solve the ODE dy/dt $=-2 y$ between $t=0 . .4$, with the initial condition $y(t=0)$ $=1$. First the function computing the derivative of the position needs to be defined:

```
>> def calc_derivative(ypos, time, counter_arr)
    counter_arr +=
return \(-2 *\) ypos
    return -2 * ypos
```

An extra argument counter_arr has been added to illustrate that the function may be called several times for a single time step, until solver convergence. The counter array is defined as:
$\ggg$ counter $=n \mathrm{n}$. zeros $((1$,$) , dtype=np. uint16)$
The trajectory will now be computed:
>> from scipy.integrate import odeint
>> time_vec = np.linspace (0, 4, 40)
>> yvec, info = odeint(calc_derivative, 1, time_vec,

$$
\begin{aligned}
& \text { calc__errvative, , trime_vec, } \\
& \text { args=(counter, }) \text { full_output=True }
\end{aligned}
$$

Thus the derivative function has been called more than 40 times (which was the number of time steps)
>> counter
array([129], dtype=uint16)
and the cumulative number of iterations for each of the 10 first time steps can be obtained by:
>>> info['nfe'][:10]
$\operatorname{array}([31,35,43,49,53,57,59,63,65,69]$, dtype=int32)
Note that the solver requires more iterations for the first time step. The solution yvec for the trajectory can now be plotted


Another example with scipy.integrate.odeint() will be a damped spring-mass oscillator (2nd order os cillator). The position of a mass attached to a spring obeys the 2nd order ODE y' ' + 2 eps wo $\mathrm{y}^{\prime}+\mathrm{wo}^{-} 2$ $\mathrm{y}=0$ with wo $\mathrm{C}^{2}=\mathrm{k} / \mathrm{m}$ with k the spring constant, m the mass and eps=c/(2 m wo) with c the damping coefficient. For this example, we choose the parameters as
>> mass $=0.5$ \# kg
>> ksping $=0.4 \mathrm{~N} / \mathrm{m}$
so the system will be underdamped, because:

```
>> eps = cviscous / (2 * mass * np.sqrt(kspring/mass)
>> eps < 1
True
```

For the scipy.integrate.odeint () solver the 2nd order equation needs to be transformed in a system of wo first-order equations for the vector $\mathrm{Y}=(\mathrm{y}, \mathrm{y}$ '). It will be convenient to define nu $=2$ eps $*$ wo $=\mathrm{c} /$ m and om $=\mathrm{wo}{ }^{\wedge} 2=\mathrm{k} / \mathrm{m}$ :
>>> nu_coef = cviscous / mass
>>
Thus the function will calculate the velocity and acceleration by:

$$
\begin{aligned}
& \begin{array}{l}
\gg \text { def } \begin{array}{l}
\text { calc_deri (yvec, time, nuc, omc): } \\
\text { return (yvec[1], -nuc * yvec }[1]
\end{array}
\end{array} \\
& \text { return (yvec[1], -nuc * yvec[1] - omc * yvec[0]) } \\
& \text { >>> time_vec }=\text { np.linspace }(0,10,100) \\
& \text { 》> yarr = odeint(calc_deri, (1, 0), time_vec, args=(nu_coef, om_coef)) }
\end{aligned}
$$

The final position and velocity are shown on the following Matplotlib figure:


There is no Partial Differential Equations (PDE) solver in Scipy. Some Python packages for solving PDE's are vailable, such as fipy or SfePy

### 5.9 Signal processing: scipy.signal

>>> from scipy import signal

- scipy . signal. detrend () : remove linear trend from signal:
>>> $\mathrm{t}=\mathrm{np}$.linspace (0, 5, 100)
>> $\mathrm{x}=\mathrm{t}+\mathrm{np}$.random.normal(size=100)
>> plt.plot(t, x, linewidth=3)
[<matplotlib.lines.Line2D object at ...>]
>> plt.plot(t, signal.detrend(x), linewidth=3)
[<matplotlib.lines.Line2D object at ...>]

- scipy .signal.resample (): resample a signal to n points using FFT.
$\ggg \mathrm{t}=\mathrm{np}$.linspace $(0,5,100)$
$\gg \mathrm{x}=\mathrm{np} \cdot \sin (\mathrm{t})$
>>> plt.plot(t, x, linewidth=3
[<matplotlib.lines.Line2D object at ...>]
>> plt.plot(t[::2], signal.resample(x, 50), 'kol)
[<matplotlib.]ines Line2D object at . . >]


Notice how on the side of the window the resampling is less accurate and has a rippling effect.

- scipy.signal has many window functions: scipy.signal.hamming(), scipy.signal.bartlett(), scipy.signal.blackman()..
- scipy.signal has filtering (median filter scipy.signal.medfilt(), Wiene scipy. signal. wiener()), but we will discuss this in the image section.


### 5.10 Image processing: scipy.ndimage

The submodule dedicated to image processing in scipy is scipy .ndimage.
>>> from scipy import ndimage
mage processing routines may be sorted according to the category of processing they perform.

### 5.10.1 Geometrical transformations on images

Changing orientation, resolution, .
>>> from scipy import misc
>> face $=$ misc.face(gray=True)
>> shifted_face = ndimage.shift(face, (50, 50))
>> shifted_face2 $=$ ndimage.shift(face, (50, 50), mode='nearest')
>> rotated_face $=$ ndimage. $\operatorname{rotate}($ face, 30$)$
>>> cropped_face = face[50:-50, 50:-50]
>> zoomed_face = ndimage.zoom(face, 2)
>> zoomed_face.shape
(1536, 2048)

>> plt.subplot(151)
matplotlib.axes. subplots.AxesSubplot object at 0x...
>> plt.imshow(shifted_face, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>
>> plt.axis('off')
$-0.5,1023.5,767.5,-0.5)$
>>> \# etc.

### 5.10.2 Image filtering

>>> from scipy import misc
>> face $=$ misc.face (gray=True)
>> face $=$ face $[: 512,-512$.$] \# crop out square on right$
>> import numpy as np
>> noisy_face = np. copy (face). astype (np.float)
>>> blurred_face = ndimage.gaussian_filter(noisy_face, sigma=3)
>> median_face = ndimage.median_filter(noisy_face, size=5)
>> from scipy import signal
>> wiener_face = signal.wiener(noisy_face, (5,5))
Many 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

Mathematical morphology is a mathematical theory that stems from set theory. It characterizes and transforms geometrical structures. Binary (black and white) images, in particular, can be transformed using this heory: the sets to be transformed are the sets of neighboring non-zero-valued pixels. The theory was als extended to gray-valued images.


Elementary mathematical-morphology operations use a structuring element in order to modify other geometrical structures
Let us first generate a structuring element

```
>> el = ndimage.generate_binary_structure(2, 1)
>> el
array([[False, True, False],
    [...True, True, True]
    [False, True, False]], dtype=bool)
>> el.astype(np.int)
array ([[0, 1, 0],
    [1, 1, 1],
    [0, 1, 0]])
```

    - Erosion
    >> a = np.zeros( \((7,7)\), dtype=np.int)
    \(\ggg a[1: 6,2: 5]=1\)
    >> a
    \(\operatorname{rray}([[0,0,0,0,0,0,0]\),
    \([0,0,1,1,1,0,0]\),
    \([0,0,1,1,1,0,0]\)
    \(0,0,1,1,1,0,0]\),
    \([0,0,1,1,1,0,0]\),
    \([0,0,1,1,1,0,0]\),
    \(\gg\) ndimage binary_erosion (a) .astype (a.dtype)
    array ([ \([0,0,0,0,0,0,0]\)
    \([0,0,0,0,0,0,0]\),
    \(0,0,0,1,0,0,0]\)
    \([0,0,0,1,0,0,0]\),
    \([0,0,0,1,0,0\)
    
## $[0,0,0,0,0,0,0]$,

$[0,0,0,0,0,0,0]]$ )
>> \#Erosion removes objects smaller than the structure
array ([[00, 0, 0, 0, 0, 0,0$]$, structure=np.ones $((5,5)))$ astype (a.dtype)
$[0,0,0,0,0,0,0]$,
$[0,0,0,0,0,0,0]$,
$[0,0,0,0,0,0,0]$,
$[0,0,0,0,0,0,0]$
$[0,0,0,0,0,0,0]$,
$[0,0,0,0,0,0,0]]$ )

## - Dilation

>> $a=n p . z e r o s$
$\gg$
a $[2,2]=1$
>>> a
array ([lllllo., 0., 0., 0.],
$\left[\begin{array}{lllll}0 ., & 0 ., & 0 ., & 0 ., & 0 .\end{array}\right]$,
$\left[\begin{array}{lllll}0 ., & 0 ., & 0 ., & 0.1 & 0 .\end{array}\right]$
$\left[\begin{array}{lllll}{\left[\begin{array}{llll}0 ., & 0 ., & 0 ., & 0 ., \\ {\left[\begin{array}{lll}0 ., & 0 ., & 0 ., \\ 0 ., & 0 .]\end{array}\right]}\end{array}\right]}\end{array}\right.$
>> ndimage.binary_dilation(a) astype (a.dtype)
$\operatorname{array}\left(\left[\begin{array}{lllll}0 ., & 0 ., & 0 ., & 0 ., & 0 .] \text {, }\end{array}\right.\right.$
$\left[\begin{array}{lllll}0 ., & 0 ., & 1 ., & 0 ., & 0 .], \\ 0 . & 1 ., & 1 ., & 1 ., & 0 .\end{array}\right]$,
$\left[\begin{array}{lllll}0 ., & 1 ., & 1 ., & 1 ., & 0 .], \\ 0 ., & 0 ., & 1 ., & 0 ., & 0 .]\end{array}\right.$

- Opening
> $\mathrm{a}=\mathrm{np} . z e r o s((5,5)$, dtype=np.int)
>> a[1:4, 1:4] = 1
>> $a[4,4]=1$
>> a
$\operatorname{rray}\left(\left[\begin{array}{lll}0 & 0, & 0,0,0]\end{array}\right.\right.$
$[0,1,1,1,0]$,
$[0,1,1,1,0]$,
$\left.\left.\begin{array}{l}{[0,1,1,1,0],} \\ {[0,0,} \\ 0,\end{array}, 0,1\right]\right]$
>> \# Opening removes small objects
ndimage.binary_opening(a, structure=np.ones((3, 3))).astype(np.int) array ([L0, 0, 0, 0, 0],
$[0,1,1,1,0]$
$[0,1,1,1,0]$,
$[0,1,1,1,0]$,
>> \# Opening can also smooth corners
>> ndimage binary opening(a) astype (np.int)
$\operatorname{array}([[0,0,0,0,0]$
$[0,0,1,0,0]$,
$[0,1,1,1,0]$,
$[0,0,1,0,0]$,
$0,0,0,0,0] j$
- Closing: 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.


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[1:6, 1:6] = 3
>> a
$\operatorname{array}([[0,0,0,0,0,0,0]$
$[0,3,3,3,3,3,0]$
$[0,3,3,1,3,3,0]$,
$[0,3,3,3,3,3,0]$,
$\left[\begin{array}{l}0,3,3,3,2,3,0], \\ {[0,3,3,3,3,3}\end{array}\right]$
$[0,0,0,0,0,0,0] 1)$
>> ndimage.grey_erosion(a, size=(3, 3))
array ([[0, 0, 0, 0, 0, 0, 0],
$[0,0,0,0,0,0,0]$,
$[0,0,1,1,1,0,0]$,
$[0,1,1,1,0,0]$,
$[0,0,3,2,2,0,0]$,
$[0,0,0,0,0,0,0,0]$ )

### 5.10.4 Measurements on images

Let us first generate a nice synthetic binary image.
>> $\mathrm{x}, \mathrm{y}=\mathrm{np}$.indices((100, 100))
>> sig $=\mathrm{np} \cdot \sin (2 * \mathrm{np} \cdot \mathrm{pi} * \mathrm{x} / 50) * .\mathrm{np} \cdot \sin (2 * \mathrm{np} \cdot \mathrm{pi} * \mathrm{y} / 50) *.(1+\mathrm{x} * \mathrm{y} / 50 \cdot * * 2) * * 2$
>> mask $=$ sig >
Now we look for various information about the objects in the image:
>>> labels, nb = ndimage.label(mask)
>> nb
>>> areas = ndimage.sum(mask, labels, range(1, labels. $\max ()+1)$ )
>>> areas
array([ 190., 45., 424., 278., 459., 190., 549., 424.])
>>> maxima = ndimage maximum(sig, $459 ., 1$., $424 .$, 424.])
>>> maxima $=$ ndimage.maximum(sig, labels, range (1, labels.max ()
$\operatorname{array}\left(\left[\begin{array}{llrl}1.80238238, & 1.13527605, & 5.51954079, & 2.49611818, \\ 6.71673619, & 1.80238238, & 16.76547217, & 5.51954079]\end{array}\right)\right.$
>> ndimage.find_objects(labels==4)
[(slice(30L, 48L, None), slice(30L, 48L, None))]
>>> sl = ndimage.find_objects(labels==4)
>> import pylab as pl
<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 sci entific computing with Python. Now that the basics of working with Numpy and Scipy have been introduced, entific computing with Python. Now that the basic

### 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 cipy.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 no 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 upposed 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 the probability distribution of an annual maxima. In the exercise, the cumulative probability p_i for a given 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/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

n 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 an SQUnivariateSpline 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 For tran subroutines, that's why a lower library access is available through the splrep and splev functions for espectively representing and evaluating a spline. Moreover interpolation functions without the use of FIT ACK parameters are also provided for simpler use (see interp1d, interp2d barycentric interpolat and so on).
For the Sprogø maxima wind speeds, the UnivariateSpline will be used because a spline of degree 3 seems o correctly fit the data:
>>> from scipy.interpolate import UnivariateSpline
>> quantile_func = UnivariateSpline(cprob, sorted_max_speeds)
The quantile function is now going to be evaluated from the full range of probabilities
>> nprob $=$ np.linspace ( $0,1,1$ e2)
>> 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:

## 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 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.


Figure 5.1: Solution: Python source file

- The first step will be to find the annual maxima by using numpy and plot them as a matplotlib bar figure.


Figure 5.2: Solution: Python source file

- The second step will be to use the Gumbell distribution on cumulative probabilities p_i defined a $-\log \left(-\log \left(p_{-}\right)\right)$for fitting a linear quantile function (remember that you can define the degre of the UnivariateSpline). Plotting the annual maxima versus the Gumbell distribution should give you the following figure
- The last step will be to find $34.23 \mathrm{~m} / \mathrm{s}$ 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.

Introduction
Lidars systems are optical rangefinders that analyze property of scattered light to measure distances. Most of hem 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 be tween the platform and the Earth, so as to deliver information on the Earth's topography (see ${ }^{1}$ for more de-
${ }^{1}$ 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.isprsiprs.2008.99.00


Annual wind speed maxima $[\mathrm{m} / \mathrm{s}]$

Figure 5.3: Solution: Python source file
tails).
In this tutorial, the goal is to analyze the waveform recorded by the lidar system ${ }^{2}$. Such a signal contains peak 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 1 m on the Earth surface, the beam can hit multiple targets during the wo-way propagation (for example the ground and the top of a tree or building). The sum of the contributions f each target hit by the laser beam then produces a complex signal with multiple peaks, each one containing information about one targe

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('data/waveform_1.npy')
and visualize it:
>> import matplotlib. pyplot as plt
>> $\mathrm{t}=\mathrm{np}$.arange(len(waveform_1)
>> plt.plot(t, waveform_1)
DD object at ...>]
>> plt.show()


As you can notice, this waveform is a 80 -bin-length signal with a single peak.
${ }^{2}$ 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.

## 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

Gaussian function defined by

$$
B+A \exp \left\{-\left(\frac{t-\mu}{\sigma}\right)^{2}\right\}
$$

can be defined in python by:
>>> def model(t, coeffs) return coeffs[0] + coeffs[1] $*$ np.exp $(-((t-c o e f f s[2]) / c o e f f s[3]) * * 2)$
where

- coeffs [0] is $B$ (noise)
- coeffs [1] is $A$ (amplitude)
- coeffs [2] is $\mu$ (center)
- coeffs [3] is $\sigma$ (width)
nitial solution
An approximative initial solution that we can find from looking at the graph is for instance:
>> $\mathrm{x} 0=\operatorname{np} . \operatorname{array}([3,30,15,1]$, 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
>> from scipy.optimize import leastsq
>> x, flag = leastsq(residuals, x 0 , args=(waveform_1, t))
>> print(x)
$\left[\begin{array}{llll}2.70363341 & 27.82020742 & 15.47924562 & 3.05636228]\end{array}\right]$
And visualize the solution:
$\gg$ plt.plot $(t$, waveform_1, $t, \operatorname{model}(t, x))$
<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
<matplotlib.legend.Legend object at ...>
>> 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 data/waveform_2.npy) that contains three significan 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).
With the following initial solution:
>>> x0 = np.array([3, 50, 20, 1], dtype=float)
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
. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the doc tring 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

Proposed solution
>>> import numpy as np
>>> import pylab as pl
>>> from scipy import ndimage
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 "
>>> dat = pl.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 = 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 pix els, glass pixels and bubble pixels. Other option (homework): write a function that determines automat ically the thresholds from the minima of the histogram.

$$
\begin{aligned}
& \text { >> void }=\text { filtdat <= } 50 \\
& \text { >> sand }=\text { np.logical_and }(\text { filtdat }>50 \text {, filtdat <= 114) } \\
& \text { >>> glass }=\text { filtdat > } 114
\end{aligned}
$$

5. Display an image in which the three phases are colored with three different colors
>>> phases $=$ void.astype(np. int) $+2 *$ glass.astype(np.int) $+3 *$ sand.astype (np. int)

6. Use mathematical morphology to clean the different phases.
>>> sand_op = 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 ndimage. sum or np. bincount to compute the grain sizes.
>> sand_labels, sand_nb = ndimage.label(sand_op)
>>> sand_areas = np.array(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.

[^0]>> mean_bubble_size, median_bubble_size

## 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 [204]: help np.v
np.vdot np.version np.voido
np.vectorize np.void np.vsplit
In [204]: help np.vander
In Ipython it is not possible to open a separated window for help and documentation; how ever 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 http://docs.scipy.org/doc The search button is quite useful inside the reference documentation of the two package (http://docs.scipy.org/doc/numpy/reference/ and http://docs.scipy.org/doc/scipy/reference/).
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 wik http://docs.scipy.org/doc/numpy/. 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!
- Scipy central http://central.scipy.org/ gives recipes on many common problems frequently encoun

- Matplotlib's website http:// 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.
Finally, two more "technical" possibilities are useful as wel
- 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
n [4]: \%psearch np.diag*
np.diagfla
.die

- numpy.lookfor looks for keywords inside the docstrings of specified modules

In [45]: numpy. lookfor(' convolution')
Search results for 'convolution
numpy. convolve
the discrete, linear convolution of two one-dimensional
sequences.
Return the Bartlett window
numpy. correlate
[46]:
[46]: numpy. lookfor('remove', module=' os')
Search results for 'remove'
s. remove
remove(path
s.removedirs
removedirs (path)
os.rmdir
rmdir(path)
os.unlink
unlink(path
os.walk
Directory tree generator.

- If everything listed above fails (and Google doesn't have the answer)... don't despair! Write to the mailing If everything listed above fails (and Google doesn't have the answer)... don't despair! Write to the mailing
list suited to your problem: you should have a quick answer if you describe your problem well. Experts on scientific python often give very enlightening explanations on the mailing-list.
- Numpy discussion (numpy-discussion@scipy.org): all about numpy arrays, manipulating them, indexation questions, etc.

SciPy Users List (scipy-user@scipy.org): scientific computing with Python, high-level data processing, in particular with the scipy package.

- matplotlib-users@lists.sourceforge.net for plotting with matplotlib.


## Part II

## Advanced topics

Author Zbigniew Jędrzejewski-Szmek
This section covers some features of the Python language which can be considered advanced - in the sense hat not every language has them, and also in the sense that they are more useful in more complicated prorams 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 hrough clever external modules

The process of developing the Python programming language, its syntax, is very transparent; proposed hanges are evaluated from various angles and discussed via Python Enhancement Proposals - PEPs. As 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 expression

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 fea ture 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 othe 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 ge hold of an iterator. The iter function does that for us, saving a few keystrokes.
$\ggg$ 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)
>> next(it)
3 next(it)
Traceback (most recent call last):
File "<stdin>", line 1, in <module
StopIteration
raceback (most recent call last)
File "<stdin>", line 1, in <module>
topIteration
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.

Jsing 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 eturns 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.
>>> $f=\operatorname{open}(1 /$ etc/fstab'
$\gg \mathrm{f}$ 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 equential access is allowed.

### 7.1.2 Generator expressions

second way in which iterator objects are created is through generator expressions, the basis for list compre hensions. To increase clarity, a generator expression must always be enclosed in parentheses or an expression. fround parentheses are used, then a generator iterator is created. If rectangular parentheses are used, the pro ess is short-circuited and we get a list.

## >> (i for $i$ in nums)

generator object <genexpr> at 0x...>
>> [i for
1, 2, 3]
i for i in nums)

The list comprehension syntax also extends to dictionary and set comprehensions. A set is created when the enerator expression is enclosed in curly braces. A dict is created when the generator expression contain "pairs" of the form key: value:

```
\(\ggg\) \{i for \(i\) in range (3) \}
\(\operatorname{set}([0,1,2])\)
in range(3)
```

\{0: $0,1: 1,2: 4\}$
fixed.

### 1.1.3 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
    خ. \(\gg\) f()
generator object fat 0x...
\(\ggg\) gen \(=f()\)
\(\ggg\) next \((\) gen \()\)
>> next (gen)
>> next (gen)
)
>>> next (gen)
raceback (most recent call last):
", line 1 , in <module>
StopIteration
```

Let's go over the life of the single invocation of the generator function.
>>> $\operatorname{def} f()$ :
print ("-- start --")
yield 3
print("-- middle --")
yield 4
gen $=f()$
>> next(gen)

- start --

3 $\ggg$ n
>> next(gen)

- middle --
\gg>
> next(gen)
Traceback (most recent call last)
topIteration
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 b next, the statements up to the first yield are executed. The second next prints -- middle -- and execution halts on the second yield. The third next prints -- finished -- and falls of the end of the function. Sinc 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 generato 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 iterato 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 stat 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 (implemented in Python 2.2). 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 (implemented in 2.5). It is achieved by turning the previousl oring yield statement into an expression. When the generator resumes execution after a yield statemen he caller can call a method on the generator object to either pass a value into the generator, which then is eturned 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 generato 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 resume 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 xcept 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--')
or i in itertools count():
print('--yielding \%i--' \% i)
ans $=$ yield $i$
except GeneratorExit:
print('--closing--')
raise
except Exception as e:
print('--yield raised \% r--' \% e)
else:
print('--yield returned \%s--' \% ans)
>> it = g()
-start--
--yielding 0--
-
$\gg$ it. send (11)
-yield returned 11 -
-yielding 1
>> it.throw(IndexError)
-yield raised IndexError()
-yielding 2--
>> it.close()
-closing-

## next or __next__

In Python 2.x, the iterator method to retrieve the next value is called next. It is invoked implicitly through the global function next, which means that it should be called next Just like the global function iter calls iter. This inconsistency is corrected in Python 3.x, where it.next becomes it. _next_. For other generator methods - send and throw - the situation is more complicated because they are not called implicitly by the interpreter. Nevertheless, there's a proposed syntax exten sion to allow continue to take an argument which will be passed to send of the loop's iterator. If this extension is accepted, it's likely that gen. send will become gen.__send__. The last of generator methods, close, is pretty obviously named incorrectly, because it is already invoked implicitly

### 7.1.5 Chaining generators

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 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 ry..except..finally structure similar to the one defined in the previous section to "debug" the generator funcion. Such code is provided in PEP 380, here it suffices to say that new syntax to properly yield from a subgen erator 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 no be that useful.

Bruce Eckel - An Introduction to Python Decorators

Since a function or a class 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:


- A function is defined in the standard way. (C)
- An expression starting with @ placed before the function definition is the decorator $\Theta$. The part after @ must be a simple expression, usually this is just the name of a function or class. This part is evaluated must be a simple expression, usually this is just the name of a function or class. This part is evaluated 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 origina 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:

$$
\begin{array}{ll}
\text { def function(): } & \text { \# (1) } \\
\text { pass } \\
\text { function }=\text { decorator(function) }
\end{array}
$$

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 he 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 modifyin he object, for example register the decorated class in a global registry. In the second case, virtually anything is ossible: when something different is substituted for the original function or class, the new object can be com
 with a different function, the new function usually calls the original function, after doing some preparater work. Likewise, when a class is "d corated" by replacing if with a new class, the new class is usually derived om the original class. When the purpose of the decorator is to domething "every time" like to logevery cal decorted function, only the ${ }^{\circ}$ econe of decors if the a decorated function, only the second type of the the thand, 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

he only requirement on decorators is that they can be called with a single argment. This means that decorators can be implemented as normal functions, or as classes with a __call__ method, or in theory, even as ambda functions.
Let's compare the function and class approaches. The decorator expression (the part after ©) can be either jus 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 thos wo cases:

```
>> def simple_decorator(function)
    print("doing decoration")
    return function
>> @simple_decorato
    def function():
        print("inside function")
oing decoratio
>> function()
inside function
>> def decorator_with_arguments(arg):
    print("defining the decorator")
    def _decorator(function)
        in this inner function, arg is avail
        print("doing decoration, %r" % arg)
            return function
```

```
. return decorator
    >> @decorator with arguments("abc")
    def function():
        print("inside function")
efining the decorator
doing decoratio
is function()
The two trivial decorators above fall into the category of decorators which return the original function. If they The two trivial decorators above fall into the category of decorators which return the original function. If they nested functions.
>> def replacing_decorator_with_args(arg):
print defining the decorator
def _decorator(function):
in thrs inner function, arg is available too
print("doing decoration, \%r" \% arg)
def -wrapper (*args, **kwargs):
print (
, **kwargs)
return -wrapper
>> @replacing_decorator_with_args("abc")
def function(*args, \(* *\) kwargs
print("inside function, or \%r" \% (args, kwargs)) return 14
defining the decorator
doing decoration, 'abc'
)
inside function, \((11,12)\}\)
```

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 he 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 decorato construction.

$$
\begin{gathered}
\text { class decorator_class (object) } \\
\text { def } \quad \text { (init_(_(self, arg) } \\
\text { \# this method is called }
\end{gathered}
$$

("ino called in the decorator expression self.arg = arg
def __call__(self, function):
\# this method is called to do the job rint("in decorator call, \%s" \% self.arg) return function
>> deco_instance = decorator_class('foo')
in decorator init, foo
>> Qdeco_instance
ef function(*args, **kwargs)
print("in function, \%s \%s" \% (args, kwargs))
in decorator call
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 return a new 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)
        \(=\) function
        eturn 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)
        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 he 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 conse quence 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 informaion 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")
(function):
print(doing decoration, $\%$ r $1 \%$ arg
_wrapper (*args, **kwargs)
print("inside wrapper, $\% r \% r^{\prime \prime} \%$ (args, kwargs))
return function(*args, **kwargs)
return functools update_wrapper(_wrapper, function)
return -decorator
ator with args("abc")
def function()
extensive documentation"
return 14
defining the decorator
doing decoration, 'abc'
>> function
<function function at $0 x \ldots$...>
extensive documentation

One important thing is missing from the list of attributes which can be copied to the replacement func tion: 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 unction. 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 copy ing 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 interpeter insert the in bject 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:

```
(object):
```

def __init__(self, data)
self.data $=$ data

## @classmethod

def fromfile(cls, file):
data $=$ numpy. load (file) return cls(data)

This is cleaner then using a multitude of flags to __ init __

- staticmethod is applied to methods to make them "static", i.e. basically a normal function, but acces sible through the class namespace. This can be useful when the function is only needed inside this clas (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.

$$
\begin{aligned}
& \begin{array}{l}
\text { class A(object } \\
\text { @property }
\end{array} \\
& \text { def a(self): } \\
& \text { an important attribute" } \\
& \text { return "a value" } \\
& \text { > A.a } \\
& \ggg \mathrm{A}() \text { object at } 0 \mathrm{x} . \ldots \\
& \text { a value' }
\end{aligned}
$$

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. Since Python 2.6 the following syntax is preferred:

$$
\begin{gathered}
\text { ass Rectangie (object): } \\
\text { def } \quad \text { init_(self, edge) } \\
\text { self.edge }=\text { edge }
\end{gathered}
$$

## @property

def area(self):
"""Computed area.
Setting this updates the edge length to the proper value.
return self.edge**2
@area.setter
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 attribute get, fset, and fael). The getter canbe set tike in he exanple above, when crating the object. Whe defining the setter, we already have the property object under area, and we add the setter to it by usin he 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)
def a(self):
print("getting 1")
return 1
def a(self, value)
print("setting \% \% " \% value)
@a. deleter
def a(self)

```
>> 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
>>> d.a = 2
etting 2
>> del d.a
deleting
>> d.a
getting 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 (__1t__ __gt __, __le__, ...) based on a single available one (Python 2.7)


### 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 nymore. If we don't want to modify the function, we can use a decorator:
class deprecated (object):

$$
\begin{array}{ll}
\text { >> Qdeprecated() } & \text { \# doctest: }+ \text { SKIP } \\
\ldots \text { def } f(): & \\
\ldots \text { pass } & \text { \# doctest: }+ \text { SKIP }
\end{array}
$$

$$
f \text { is deprecated }
$$

def __call__(self, func) self.func = func self.count $=0$ return self.-wrapper
def $\quad$ wrapper (self, *args, $* *$ kwargs) :
self. count $+=1$
if self. count $=1$,
f self. count $==1$ :
name__, 'is deprecated'

It can also be implemented as a function:

$$
\begin{aligned}
& \text { def deprecated(func): } \\
& \text { """Print a deprecation warning once on first use of the function. } \\
& \text { >>> @deprecated } \quad \text { \# doctest: +SKIP }
\end{aligned}
$$

```
    def f():
    f( pass
f is deprecated
# doctest: +SKIP
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

```
ef find_answers()
    answers = []
        while True:
        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 hen the user would have to explicitly call list (find_answers())
We can define a decorator which constructs the list for us:

```
ef 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()
        f ans is None:
        ield 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
```


#### Abstract

@classmethod def plugin(cls, plugin): cls. PLUGINS append (plugin) class CleanMdashesExtension(object) def cleanup(self, text): return text.replace('\—', u'\M\{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 endash in the source of a program.

## See also:

## More examples and reading

- PEP 318 (function and method decorator syntax)
- PEP 3129 (class decorator syntax)
- http://wiki.python.org/moin/PythonDecoratorLibrary
- https://docs.python.org/dev/library/functools.html
- http://pypi.python.org/pypi/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
$\mathrm{var}=$ manager.__enter__()
try:
do_som
inally:
manager.__exit__()
In other words, the context manager protocol defined in PEP 343 permits the extraction of the boring part of a ry..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 f inally 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):

$$
\begin{aligned}
& \text { def __init__(self, obj): } \\
& \begin{array}{c}
\text { self.obj }=o b j \\
\text { def__enter__(self) }
\end{array} \\
& \text { return self.obj } \\
& \text { def __exit__(self, *args): } \\
& \text { self. obj. close() } \\
& \text { > with closing(open('/tmp/file', 'w')) as } f \text { : }
\end{aligned}
$$

f. write('the contents $\backslash n$ ')

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 hrown, is propagated. As with files, there's often a natural operation to perform after the object has been used nd it is most convenient to have the support built in. With each release, Python provides support in mor places:

- all file-like objects:
- file automatically closed
- fileinput, tempfile (py >= 3.2)
- bz2.BZ2File, gzip.GzipFile, tarfile.TarFile, zipfile.ZipFile
- ftplib, nntplib close connection (py >=3.2 or 3.3)
- locks
- multiprocessing. RLock $\rightarrow$ lock and unlock
- multiprocessing.Semaphore
- memoryview $\rightarrow$ automatically release (py >=3.2 and 2.7)
- decimal.localcontext $\rightarrow$ modify precision of computations temporarily
- _winreg. PyHKEY $\rightarrow$ open and close hive key
- warnings.catch_warnings $\rightarrow$ kill warnings temporarily
- contextlib.closing $\rightarrow$ the same as the example above, call close
- parallel programming
- concurrent.futures.ThreadPoolExecutor $\boldsymbol{\rightarrow}$ invoke in parallel then kill thread pool (py>=3.2)
- concurrent.futures.ProcessPoolExecutor $\boldsymbol{*}$ invoke in parallel then kill process pool (py $>=$ 3.2
- nogil $\boldsymbol{\omega}$ 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 $\underset{\text { pass }}{ }$
def __exit__(self, type, value, traceback)
fype is None:
('exception expected')
f 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 hey are shorter, sweeter, and the state is stored as local, not instance, variables. On the other hand, as de cribed in Bidirectional communication, the flow of data between the generator and its caller can be bidirec ional. This includes exceptions, which can be thrown into the generator. We would like to implement contex 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 gener ator 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 con ext 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 wrap per function then throws it at the point of the yield statement. Through the use of generators, the contex manager is shorter and simpler.

Let's rewrite the closing example as a generator:
ocontextlib.contextmanager
ef closing(obj):
try:
finally:
obj. close()
Let's rewrite the assert_raises example as a generator:
@contextlib.contextmanager
def assert_raises(type):
try: $\quad$ 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!

## 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 lexibility, 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 unnec essary copies.
- Recently added features, and what's in them: PEP 3118 buffers, generalized ufuncs, ..


## Prerequisites

- Numpy

Cytho

- Pillow (Python imaging library, used in a couple of examples)


## Chapter contents

- Life of ndarray
- Its...
- Block of memory
- Data types
- Indexing scheme: strides
- Findings in dissectio
- 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 siblings: chararray, maskedarray, matrix

- chararray: vectorized string operations
masked_array missing data
recarray: purely convenienc
matrix: convenience?
Summary
Contributing to Numpy/Scip
Why
Reporting bugs
Contributing to documentation
Contributing features
- How to help, in general

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 descripto

- raw data
- how to locate an element
- how to interpret an element

ndarray
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

>> $\mathrm{x}=\mathrm{np}$.array ([1, 2, 3], dtype=np.int32)
>>
<...data
at
<... at ...>
$\ggg \operatorname{str}(x . d a t a)$
$\backslash 00 \backslash x 00 \backslash x 00 \backslash x 02 \backslash x 00 \backslash x 00 \backslash x 00 \backslash x 03 \backslash x 00 \backslash x 00 \backslash x 00$ '
Memory address of the data:
>> x._-_array_interface__['data'] [0]
64803824
The whole __array_interface__:

$$
\begin{aligned}
& \text { >>> x.-_-array_interface_- } \\
& \text { \{'data': (35828928, False) } \\
& \text { 'descr': [('), '<i4')], } \\
& \text { 'shape': (4,), } \\
& \text { 'strides': None, } \\
& \text { 'typestr': '<i4', } \\
& \text { 'version': 3\} }
\end{aligned}
$$

Reminder: two ndarrays may share the same memory:
>>> $\mathrm{x}=\mathrm{np}$.array ([1, 2, 3, 4])
>>> $y=x[:-1]$

| 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
    ("format", "S4"), # 4-byte string
    ("fmt_id", "S4"),
    ("fmt_size", "<u4"),
    ("audio_fmt", "<u2"), #
    ("num_channels", "<u2"), #- #
    ("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')
dict_proxy(\{'block_align': (dtype('uint16'), 32), 'format': (dtype('S4'), 8), 'data_id': (dtype(('S1', (2, 2))) >> 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:
$\ggg$ 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).

```
>> \(\mathrm{f}=\) open('data/test.wav', '
close()
>>> wav header['sample rate'
array ([16000], dtype=uint32)
Let's try accessing the sub-array:
>>> wav_header['data_id']
array ([[[['d', 'a'],
[type='ls1')
>> wav_header.shape
(1,)
>> wav_header['data_id'] .shape
\((1,2,2)\)
```

>> wav_header = np.fromfile(f, dtype=wav_header_dtype, count=1)
('RIFF', 17402L 'WAVE', 'fmt ', 16L, 1, 1, 16000L, 32000L, 2, 16, [['d', 'a'], ['t', 'a']], 17366L)]

When accessing sub-arrays, the dimensions get added to the end!
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 matter
- largest "safe" type able to represent both is picked
- scalars can "lose" to arrays in some situations
- Casting in general copies data:

```
>>> x m np.array([1, 2, 3, 4], dtype=np.float)
    >>> x
    array([ 1., 2., 3., 4.])
    >>> y = x.astype(np.int8)
    array([1, 2, 3, 4], dtype=int8)
    >> y + 1
    array([2, 3, 4, 5], dtype=int8)
    >>> y + 256
    array([257, 258, 259, 260], dtype=int16)
    >> 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:

$$
\begin{aligned}
& \gg y \mathrm{y}[:]=\mathrm{y}+1.5 \\
& \ggg \mathrm{y} \\
& \operatorname{array}([2,3,4,5] \text {, dtype=int8) }
\end{aligned}
$$

Exact rules: see numpy documentation

Re-interpretation / viewing

- Data block in memory (4 bytes)

| $0 \times 01$ | $\\|$ | $0 x 02$ | $\\|$ | $0 x 03$ | II |
| :---: | :---: | :---: | :---: | :---: | :---: |

- 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"
array ([ 513, 1027],
>> $0 \times 0201,0 \times 0403$
$(513,1027)$

\section*{| $0 \times 01$ | $0 x 02$ | $\\|$ | $0 x 03$ | $0 x 04$ |
| :--- | :--- | :--- | :--- | :--- |}

little-endian: least significant byte is on the left in memory
2. Create a new view:
>>> y = x.view("<i4")
array ([67305985], dtype=int32)
>>> $0 \times 04030201$
67305985

| $0 \times 01$ | $0 \times 02$ | $0 \times 03$ | $0 \times 04$ |
| :---: | :---: | :---: | :---: |

- .view () makes views, does not copy (or alter) the memory block
- only changes the dtype (and adjusts array shape)
$\ggg \mathrm{x}[1]=5$
$\ggg \mathrm{y}$
>>> y
array ([328193], dtype=int32)
>> y base is
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)
ll
>> x[:,:, 2] = 3
```

where the last three dimensions are the $\mathrm{R}, \mathrm{B}$, and G , and alpha channels.
How to make a $(10,10)$ structured array with field names 'r', 'g', 'b', 'a' without copying data?
>>> $\mathrm{y}=$
>> assert (y['r'] == 1).all()
>> assert (y['g'] == 2).all()
>> assert $\left(y\left[1 b^{\prime}\right]==3\right) \cdot a l l()$
$\gg$ assert (y['a'] $==4) \cdot a l l()$
Solution

```
>>> \(\mathrm{y}=\mathrm{x} \cdot \operatorname{view}\left(\left[\left(\mathrm{Ir}^{\prime}, \quad\right.\right.\right.\) 'i1'),
    ('g', 'i1'),
    ('b', 'i1'),
    ('a', 'i1')]
) \([:, ~:, ~ o] ~\)
```


## Another array taking exactly 4 bytes of memory:

```
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
```

>>> $\mathrm{x}=\mathrm{y}$.copy ()
>>> x
$\operatorname{array}([[1,2]$,
$[3,4]]$, dtype=uint8)
>> y
$\operatorname{array}([[1,2]$,
[3, 4]], dtype=uint8)
>> x.view(np.int16)
array ([[513],
>>> 0x0201, 0x0403pe=int16
(513, 1027)
>>> y.view(np.int16)
$\operatorname{array}([[769,1026]]$, dtype=int16)

- What happened?
- ... we need to look into what $\mathrm{x}[0,1]$ actually means
>> 0x0301, 0x0402
(769, 1026)


### 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)
```

```
[7, 8, 9]], dtype=np.int8)
```

>> $\operatorname{str}(x$ data)
'\x01\x02\x03\x04 $\times \mathrm{x} 05 \backslash \mathrm{x} 06 \backslash \mathrm{x} 07 \backslash \mathrm{x} 08 \backslash \mathrm{t}$ '
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


## (3, 1)

>> byte_offset $=3 * 1+1 * 2$ \# to find $x[1,2]$
>> x.flat [byte_offset]
$\stackrel{6}{\ggg} \mathrm{x}[1,2]$
6
simple, **flexible**

C and Fortran order

| [4, 5, 6]], dtype=np.int16, order='C') |  |
| :---: | :---: |
|  |  |
|  |  |
|  |  |
|  |  |

>> $x$ x.stride
$(6,2)$

## >> $\operatorname{str}(\mathrm{x}$ data)

$\backslash x 01 \backslash x 00 \backslash x 02 \backslash x 00 \backslash x 03 \backslash x 00 \backslash x 04 \backslash x 00 \backslash x 05 \backslash x 00 \backslash x 06 \backslash x 00$

- Need to jump 6 bytes to find the next row
- Need to jump 2 bytes to find the next column
>>> $\mathrm{y}=\mathrm{np} . \operatorname{array}\left(\mathrm{x}\right.$, order=' $\mathrm{F}^{\prime}$ )
>>> y .strides
$(2,4)$
$\ggg \operatorname{str}(\mathrm{y} \cdot$ data)

- 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

$$
\begin{aligned}
\text { shape } & =\left(d_{1}, d_{2}, \ldots, d_{n}\right) \\
\text { strides } & =\left(s_{1}, s_{2}, \ldots, s_{n}\right) \\
s_{j}^{C} & =d_{j+1} d_{j+2} \ldots d_{n} \times \text { itemsize } \\
s_{j}^{F} & =d_{1} d_{2} \ldots d_{j-1} \times \text { itemsize }
\end{aligned}
$$

Now we can understand the behavior of .view ():

$$
\begin{aligned}
& \ggg y=n p . \operatorname{array}([[1,3],[2,4]], \text { dtype=np. uint8).transpose() } \\
& \ggg x=y \cdot \operatorname{copy}()
\end{aligned}
$$

Transposition does not affect the memory layout of the data, only strides
$\ggg$
x.strides
$(2,1)$
$\ggg$
y. strides
$(1,2)$
>>> $\operatorname{str}(\mathrm{x}$.data)

<br>x01\x03\x02\x04

- the results are different when interpreted as 2 of int 16
- . copy () creates new arrays in the C order (by default)


## Slicing with integers

- Everything can be represented by changing only shape, strides, and possibly adjusting the data pointer!
- Never makes copies of the data

```
>> \(\mathrm{x}=\mathrm{np}\).array ([1, 2, 3, 4, 5, 6], dtype=np.int32)
>> \(\mathrm{y}=\mathrm{x}[::-1]\)
>> y
\(\operatorname{array}([6,5,4,3,2,1]\), dtype=int32)
\(\ggg\) y.strides
\((-4\),
```

```
>> y = x[2:]
>>> y. array_interface__['data'][0] - x.__array_interface__['data'] [0]
>>> y
>>> x = np.zeros((10, 10, 10), dtype=np.float)
>>> x.strides
(800, 80, 8)
>> x[::2,::3,::4] .stride
(1600, 240, 32)
```

- Similarly, transposes never make copies (it just swaps strides):

$$
\begin{aligned}
& \begin{array}{l}
\text { >> } x=n \text { n.zeros }((10,10,10) \text {, dtype=np.float) } \\
\ggg \mathrm{x} \text {.strides }
\end{array} \\
& \text { ( } 800,80,8 \text { ) } \\
& \text { >> x.T.strides } \\
& (8,80,800)
\end{aligned}
$$

But: not all reshaping operations can be represented by playing with strides:

```
>> a = np.arange(6, dtype=np.int8).reshape(3, 2)
>>> b.stride
(1, 2)
```

So far, so good. However:
>>> str(a.data)
\x00\x01\x02\x03\x04\x05
>> b
rray ([10, 2, 4],
[1, 3, 5]], dtype=int8)
>> $\mathrm{c}=\mathrm{b}$. 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)
as_strided (x, shape=None, strides=None)
Make an ndarray from the given array with the given shape and strides
$\triangle$ as_strided does not check that you stay inside the memory block bounds...
>>> $x=n p$.array ( $[1,2,3,4]$, dtype=np.int16)
$\ggg$ as_strided $(x, \operatorname{strides}=(2 * 2$,$) , shape =(2)$,
$\operatorname{array}([1,3]$, dtype $=$ int 16 )
>> $x[:: 2]$
array ([1, 3], dtype=int16)

## See also:

stride-fakedims.py
Exercise

```
array([1, 2, 3, 4], dtype=np.int8)
    \(\rightarrow \operatorname{array}([[1,2,3,4]\)
    \(\left[\begin{array}{lll}1,2, & 3 & 4] \text {, } \\ 1,2, & 3 & 4]\end{array}\right.\)
    \([1,2,3,4]\),
\([1,2,3,4]]\), dtype=np.int8)
```


## using only as_strided.:

Hint: byte_offset $=$ stride $[0] *$ index[0] + stride[1]*index[1] +

## Spoiler

Stride can also be 0 :
>>> $\mathrm{x}=\mathrm{np}$.array ([1, 2, 3, 4], dtype=np.int8)
>>> $y=\operatorname{as}$ _strided $(x, \operatorname{strides}=(0,1)$, shape $=(3,4))$
>> y
$1,2,3,4]$,
$1,2,3,4]$,
$[1,2,3,4]]$, dtype=int8)
$\gg \mathrm{y}$.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
$\operatorname{array}([[1,2,3,4]$,
$[1,2,3,4]$,
$[1,2,3,4]]$, dtype $=$ int16
>>> $\mathrm{y}=\mathrm{np} \cdot \operatorname{array}([5,6,7]$, dtype=np. int16)
>> $\mathrm{y} 2=$ as_strided $(\mathrm{y}$, strides $=(1 * 2,0)$, shape $=(3,4)$ )
>>> y 2
array ([ [5, 5, 5, 5],
$[7,7,7,7]]$, dtype=int16)

## >> x2 * y2 <br> $\operatorname{array}([[5,10,15,20]$ <br> $[6,12,18,24]$

[ $7,14,21,28]$ ], dtype=int16)

## ... seems somehow familiar ...

```
x = np.array([1, 2, 3, 4], dtype=np.int16)
>> y = np.array([5, 6, 7], dtype=np.int16)
>> x[np.newaxis,:] * y[:,np.newaxis]
array([[ 5, 10, 15, 20]
    [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)

$$
\begin{aligned}
& \ggg x=\operatorname{np} . \operatorname{array}([[1,2,3], \\
& \ldots \\
& \cdots
\end{aligned} \quad[4,5,6], \text { dtype=np.int32) }
$$

>>> 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, )) rray $([4,8]$, dtype=int32)

## Using np.diag

>>> $\mathrm{y}=\mathrm{np} \cdot \operatorname{diag}(\mathrm{x}, \mathrm{k}=1$ )
>>> y
$\operatorname{array}([2,6]$, dtype=int32)
However,
>> y.flags.owndata
False
Note This behavior has changed: before numpy 1.9, np.diag would make a cop

## See also

## stride-diagonals.py

## Challenge

Compute the tensor trace
>>> $\mathrm{x}=\mathrm{np}$.arange ( $5 * 5 * 5 * 5$ ) .reshape ( $5,5,5,5$ )
$\ggg \mathrm{s}=0$
>> for i in range(5):

$$
\begin{aligned}
& r \mathrm{j} \text { in range (5): } \\
& \mathrm{s}+=\mathrm{x}[\mathrm{j}, \mathrm{i}, \mathrm{j}, \mathrm{i}]
\end{aligned}
$$

by striding, and using sum () on the result.
>>> y = as_strided(x, shape=(5,5), strides=(TODO, TODO))
$\ggg$ assert $\mathrm{s}==\mathrm{s} 2$
Solution

```
>>> y = as_strided(x, shape=(5,5), strides=((5*5*5 + 5)*x.itemsize,
>>> s2 = y.sum()
(5*5 + 1)*x.itemsize))
```

CPU cache effects
Memory layout can affect performance:
In [1]: $x=n p . z e r o s((20000)$,
In [2]: y $=n$ n. $\operatorname{zeros}((20000 * 67)),[:: 67]$
In [3]: x.shape, y.shap
$(20000),,(20000)$,
In [4]: \%timeit x.sum()
100000 loops, best of 3: 0.180 ms per loop
In [5]: \%timeit y.sum()
100000 loops, best of 3: 2.34 ms per loop
In [6]: x.strides, y strides
$((8),,(536)$,

Smaller strides are faster?

\section*{$x$ <br> |  |  |  |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | <br> $y$


|  |  |  |  |  |  |  |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |}

## cache block size

- 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)
- $\Rightarrow$ fewer transfers needed
- $\Rightarrow$ faster

See also:
numexpr is designed to mitigate cache effects in array computing

Example: inplace operations (caveat emptor)

- Sometimes,
$\square$


## is not the same a

>>> a -= b.copy()

```
>> x = np.array([[1, 2], [3, 4]]
>> x -= x.transpose()
array([[ 0, -1],
>>y =np.array([[1, 2], [3, 4]])
>>> y -= y.T.copy()
>> y
rray([[ 0, -1],
    [ 1, 0]j)
```

- x and $\mathrm{x} \cdot$ transpose () share data
- x -= $\mathrm{x} . \operatorname{tr}$ ranspose() modifies the data element-by-element...
- because x and x .transpose () have different striding, modified data re-appears on the RHS
8.1.5 Findings in dissection

- 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) ${ }^{\text {void }}$

* ${ }^{*}$ 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.
*/
(har input_1 (char*)args[0],
char *input_2 $=$ (char*)args[1];
int i;
for ( $\mathrm{i}=0$; $\mathrm{i}<$ dimensions [0]; ++i) \{
*output $=$ elementwise_function(*input_1, *input_2);
input_1 $+=$ steps $[0]$;
output += steps [2];
\}

2. The Numpy part, built by
char types [3]
types [0] $=$ NPY_BYTE $\quad / *$ 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_FromFuncAndDatal
ufunc_loop,
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) |
| :--- | :--- |
| Py |  |

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 $\quad$ elementwise_func(npy_cdouble *input, npy_cdouble* output)

| PyUfunc_D_D | elementwise_func(npy_cdouble *input, npy_cdouble* output) |
| :--- | :--- |
| PyUfunc_DD_D | $\begin{array}{l}\text { elementwise_func (npy_cdouble } \\ \text { out) }\end{array}$ |

- 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 $c=x+i y$ 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 ( $\mathrm{zO}, \mathrm{c}$ ) that computes:

$$
\begin{aligned}
& z=z 0 \\
& \text { for } k \text { in range(iterations): }
\end{aligned}
$$

$$
\mathrm{z}=\mathrm{z} * \mathrm{z}+\mathrm{c}
$$

Mandelbrot set.

- Our function is a simple one, so make use of the PyUFunc * helpers
- Write it in Cython


## ee also:

mandel.pyx, mandelplot.py
\# Fix the parts marked by TODO
\# Fix
\#
\#
\# 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 $(0,1+2 \mathrm{j})$
\#
The elementwise functio
cdef void mandel_single_point (double complex *z_in, double complex *c_in,
\#
\# The Mandelbrot iteration
$\stackrel{\text { \# }}{\#}$
\#
\# 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
cdef double complex $z=z_{-}$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 $* * 2+\mathrm{z}$.imag**2

TODO: mandelbrot iteration should go here
\# Return the answer for this point
$z_{-}$out $[0]=z$

## Boilerplate Cython definition

\#

* The litany below is particularly long, but you don't really need to
\# read this part; it just pulls in stuff from the Numpy C headers.

def extern from "numpy/arrayobject.h"
void import_array()
def enum NPY TYPES:
NPY_DOUBLE
NPY_CDOUBLE
NPY_LONG
cdef extern from "nump
ctypedef void (*PyuFuncGenericFunction)(char**, npy_intp*, npy intp*, void*)
object PyUFunc_FromFuncAndData(PyUFuncGenericFunction* func, void** data, char* types, int ntypes, int nin, int nout,
int identity, char* name, char* doc, int c)
\# List of pre-defined loop functions
void PyUFunc_f_f_As_d_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_d_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_f_f (char** args, npy_intp* dimensions, npy_intp* steps, void* func)
void PyuFunc_g_g (char** args, npy_intp* dimensions, npy_intp* steps, void* func)
void PyUFunc_F_F_As_D_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_F_F(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_D_D (char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_G_G(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_ff_f_As_dd_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_ff_f (char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_dd_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_FF_FAs_DD_D(char** args, npy intp* dimensions, npy intp* steps, void* func) void PyUFunc_DD_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_FF_F(char** args, npy_intp* dimensions, npy_intp* steps, void* func) void PyUFunc_GG_G(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

Required module initialization
import_array()

```
import_ufunc()
The actual ufunc declaration
def PyUFuncGenericFunction loop_func[1]
def char input_output_types[3]
cdef void *elementwise_funcs[1]
#
Reminder: some pre-made Ufunc loops:
# Ren
```


\# - - PyUfunc_ff_f
\# ' ${ }^{-P y U f u n c \_d \_d `}$
\# - - Pyufunc_dd_d
*- PyUfunc_D_D
\# - PyUfunc_DD_D
\# =
The full list is abo

| \# |
| :--- |
| \# |
| \# |

\# 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] = ... TODD: 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"
"mandel (z, c) -> computes $z * z+c$ ", \# docstring
0 \# unused

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 $\quad$ elementwise_func (complex_double *input, complex_double* output)
PyUfunc_DD_D elementwise_func (complex_double *in1, complex_double *in2,
Type codes
PPY BOOL, NPY BYTE, NPY_UBYTE, NPY SHORT, NPY USHORT, NPY INT, NPY UINT,
PY_LONG, NPY ULONG, NPY LONGLONG, NPY_ULONGLONG, NPY_FLOAY, NPY DOUBLE,
PPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETTME,
NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID

### 8.2.3 Solution: building an ufunc from scratch

```
\# The elementwise function
def 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 \({ }^{\text {- nogil }}\)
    \# - 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 aroun
    cdef double complex \(z=z_{-}\)in [0]
    def double copmplex \(c=c_{-}\)in [0]
    cdef int k \# the integer we use in the for loop
    \# Straightforward iteration
    for \(k\) in range (100)
        f z.real \(* * 2+z\).imag**2 \(>1000\)
            break
\# Return the answer for this point
\(z_{-}\)out \([0]=\) z
\# Boilerplate Cython definitions
\#
\# You don't really need to read this part, it just pulls in
\# stuff from the Numpy C headers.
```

```
def extern from "numpy/arrayobject.h"
    void import_array()
    cdef enum NPY TYPES:
```

        NPY_CDOUBLE
    cdef extern from "numpy/ufuncobject.h"
void import_ufunc()
ctypedef void (*PyUFuncGenericFunction)(char**, npy_intp*, npy_intp*, void*)
object PyUFunc_FromFuncAndData(PyUFuncGenericFunction* func, void** data, har* types, int ntypes, int nin, int nout,
int identity, char* name, char* doc, int c)
void PyUFunc_DD_D(char**, npy_intp*, npy_intp*, void*)

* Required module initialization
import_array()
import_ufunc ()
\# The actual ufunc declaration
def PyUFuncGenericFunction loop_func[1]
def char input_output_types [3]
def void *elementwise_funcs[1]
loop_func[0] = PyUFunc_DD_D
input_output_types [0] = NPY_CDDUBLE
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
"""
import numpy as n
import mandel
$x=$ np.linspace $(-1.7,0.6,1000)$
$\mathrm{y}=\mathrm{np} . \operatorname{linspace}(-1.4,1.4,1000)$

plt.imshow(abs(z)**2 < 1000, extent=[-1.7, 0.6, -1.4, 1.4])

Most of the boilerplate could be automated by these Cython modules: http://wiki.cython.org/MarkLodato/CreatingUfuncs

Several accepted input types
E.g. supporting both single- and double-precision versions
def void mandel_single_point(double complex *z_in,

$$
\begin{aligned}
& \text { double complex *c_in, } \\
& \text { double complex *z_out) nogil: }
\end{aligned}
$$

cdef void mandel_single_point_singleprec (float complex $*$ __in,

$$
\begin{aligned}
& \text { float complex } *_{\text {c_ }} \text { in, } \\
& \text { float complex } * z_{\text {_ out })} \text { nogil: }
\end{aligned}
$$

cdef PyUFuncGenericFunction loop_funcs[2]
def char input_output_types [3*2]
def void *elementwise_funcs [1*2]
loop_funcs [0] = PyUFunc_DD_D
input_output_types [0] = NPY_CDOUBLE
nput output types [1] $=$ NPYY
nput_output_types[1] = NPY_CDOBLE
elementwise_funcs[0] = <void*>mandel_single_point
oop_funcs [1] = PyUFunc_FF-F
nput_output_types [3] = NPY_CFLOA
nput_output_types [4] = NPY_CFLOA
nput_output_types [5] $=$ NPY_CFLOAT
lementwise_funcs[1] = <void*>mandel_single_point_singlepre
mandel $=$ PyUFunc FromFuncAndData(
loop_func,
elementwise_funcs,
input_output_types,

```
2, # number of supported input types
    2, # number of input 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 $=(m, p)$
$(m, n),(n, p) \rightarrow(m, p)$

- 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
- ... but we don't ship with public g-ufuncs, except for testing, ATM
>>> import numpy.core.umath_tests as ut
>> ut.matrix_multiply.signature
$(m, n),(n, p)->(m, p){ }^{\prime}$

```
x> = np.ones((10, 2, 4))
>> y = np.ones((10, 4, 5))
>> ut.matrix_multiply(x, y).shap
(10, 2, 5)
```

- the last two dimensions became core dimensions, and are modified as per the signature
- otherwise, the g-ufunc operates "elementwise"
- matrix multiplication this way could be useful for operating on many small matrices at once

```
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 input_2_strides_p = steps[6];
    int output_strides_p = steps [8];
    int m=dimension[1]; /* core dimensions are added after */
    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];
        nput_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 protoco

- 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.int8
>> data[:, :] = [255, 0, 0, 255] \# Red
>> \# In PIL, Ra
>> 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 protoco

```
From buffer
Show how to exchange data between numpy and a library that only knows
the buffer interface
mport numpy as np
mport Image
# Let's make a sample image, RGBA format
x = np.zeros((200, 200, 4), dtype=np.int8)
x[:,:,0] = 254 # red
[:,:,3] = 255 # opaque
data = x.view(np.int32) # Check that you understand why this is OK!
mg = Image.frombuffer("RGBA", (200, 200), data)
img.save('test.png')
##
Modify the original data, and save again.
# It turns out that PIL, which knows next to nothing about N
# happily shares the same data
#
[:,:,1] = 254
img.save('test2.png')
```

```
>> from PIL import Image
>> img = Image.open('data/test.png')
>> img._-array_interface_
    'shape':(200, 200, 4)
    'typestr': 'lu1'}
    >> x = np.asarray(img)
    >> x.shape
(200, 200, 4)
```

A more C-friendly variant of the array interface is also defined

### 8.4 Array siblings: chararray, maskedarray, matrix

8.4.1 chararray: vectorized string operations

```
>> x = np.array(['a',' bbb', ' ccc']).view(np.chararray)
>> x.1strip(
    *)
        dtype='...'
>> d.upper()
chararray(['A', ' BBB',' CCC']
        dtype='...')
```

.view() has a second meaning: it can make an ndarray an instance of a specialized ndarray subclass

### 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:

$$
\ggg x=\operatorname{np} . \operatorname{array}([1,2,3,-99,5])
$$

One way to describe this is to create a masked array:

```
>> mx = np.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
masked_array(data \(=\left[\begin{array}{llll}1 & 2 & 3 & -- \\ \hline\end{array}\right]\),
mask \(=\) [False False False True False]
fill_value \(=999999\) )
```


## Masked mean ignores masked data

$$
\begin{aligned}
& \gg \mathrm{mx} \cdot \text { mean () } \\
& 2.75 \\
& \ggg \mathrm{np} \text {. mean (mx) } \\
& 2.75
\end{aligned}
$$

2.75

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])
```


## 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) $)$
$\cdots \quad \mid(($ year $>=1917) \&($ year <= 1918)))
$\ggg$ \# 'छ' means 'and' and 'I' means 'or
>>> populations[bad_years, 0] =np.ma.masked
populations[bad_years, 1] $=$ np.ma.masked
$\gg$ populations.mean (axis=0)
masked_array(data $=\left[\begin{array}{lll}40472.72727272727 & 18627.27272727272842400 .0\end{array}\right]$,
mask $=$ [False False False]
fill_value $=1 \mathrm{e}+20$ )
>> populations.std(axis=0)
masked_array(data $=[21087.65648900671715625 .7998142402543322 .5062255844787]$ mask $=[$ False False False $]$
fill value $=1 \mathrm{e}+20$ )
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
chararray(['a', 'b']
ar2.y
rray ([1, 2]

### 8.4.4 matrix: convenience?

- always 2-D
- $*$ is the matrix product, not the elementwise one
>> np.matrix([[1, 0], [0, 1]]) * np.matrix([[1, 2], [3, 4]])
matrix $([1,2]$,


### 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: http://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)
- http://projects.scipy.org/numpy
- http://projects.scipy.org/scipy
- Click the "Register" link to get an accoun
- Mailing lists ( scipy.org/Mailing_Lists )
- 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
hen calling numpy.random.permutation with non-integer arguments
it fails with a cryptic error message:
>>> np.random.permutation(12) $\quad$ array([ $6,11,4,10,2,8,1,7,9,3,0,5]$ )
>>> np.random. permutation(12.)
Traceback (most recent call last)
File "<stdin>", line 1, in <module>
File "mtrand.pyx", line 3311, in mtrand.RandomState.permutation
File "mtrand.pyx", line 3254, in mtrand.RandomState.shuffle
TypeError: len() of unsized object
This also happens with long arguments, and so
np.random.permutation(X.shape [0]) where X is an array fails on 6
bit windows (where shape is a tuple of longs).
It would be great if it could cast to integer or at least raise 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.
. 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, $\mathrm{x} 86 / \mathrm{PPC}, . .$.
3. Version of Numpy/Scipy
>>> print(np.__version__)

## Check that the following is what you expec

>>> 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

- http://docs.scipy.org/doc/numpy
- Registration
- Register an account
- Subscribe to scipy-dev mailing list (subscribers-only)


## - Problem with mailing lists: you get mai

* But: you can turn mail delivery off
* "change your subscription options", at the bottom of
http://mail.scipy.org/mailman/listinfo/scipy-dev
- 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
Cheers

- Check the style guide
- http://docs.scipy.org/doc/numpy/
- 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

0 . Ask on mailing list, if unsure where it should go

1. Write a patch, add an enhancement ticket on the bug tracket
2. OR, create a Git branch implementing the feature + add enhancement ticket

- Especially for big/invasive addition
- http://projects.scipy.org/numpy/wiki/GitMirror
- http://www.spheredev.org/wiki/Git_for_the_lazy
it clone --origin svn http://projects.scipy.org/git/numpy.git numpy
cd numpy
Create a feature branch
git checkout -b name-of-my-feature-branch svn/trunk
<edit stuff>
git commit -a
- Create account on https:// github.com (or anywhere)

Create a new repository @ Github

- Push your work to github
git remote add github git@github:USERNAME/REPOSITORYNAME.git
git push github name-of-my-feature-branch


### 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

* Needs to be done eventually
* Want to think? Come up with a Table of Contents http:// scipy.org/Developer_Zone/UG_Toc
- Ask on communication channels.
- numpy-discussion list
- scipy-dev list


## - 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
- pep8
- 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-0>: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 http://www.vim.org/scripts/script.php?script_id=2441
2. extract the files in $\sim / . v i m / f t p l u g i n / p y t h o n ~$

## 3. make sure your vimrc has filetype plugin indent on

def _compute_log_likelihood (obs) : return self._log_emissionprob[:, obs].T

- Alternatively: use the syntastic plugin. This can be configured to use flake8 too and also handles on-the-fly checking for many other languages.


$$
\begin{aligned}
& \quad \text { (file-name-directory buffer-file-name))) } \\
& \quad \text { (list "pyflakes" (list local-file)))) } \\
& \text { (add-to-list 'flymake-allowed-file-name-masks } \\
& \text { '("\\
.py\\
'" flymake-pyflakes-init))) }
\end{aligned}
$$

(add-hook 'find-file-hook 'flymake-find-file-hook)

### 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-runfail 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.

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/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.

[^1]
### 9.3.1 Invoking the debugger

Ways to launch the debugger:

1. Postmortem, launch debugger after module error
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
/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in <module>()
    7 if _-nnme_- == '_-main__':
---> 8 --ndex_error()
/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in index_error()
    3 def index_error()
    4 lst = list('foobar')
    ---> 5 print lst[len(1st)]
    6}7\mathrm{ if __name__ == '__main__'
IndexError: list index out of range
In [2]: %debug
/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py(5)index_error()
4 lst = list('foobar')
---> 5 print lst[len(lst)]
ipdb> list
    1 """Small snippet to raise an IndexError."""
    2 (ef index error().
        3 def index_error():
        lst = 11st('foobar
    --> 5 print lst[len(lst)]
        7 if __name__ == '__main_-'
        8 index_error()
    8
ipdb> len(1st)
6
ipdb> print lst[len(1st)-1]
r ipdb> quit
In [3]:
```


## 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/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(1)<module>()
-> """Small snippet to raise an IndexError."""
(Pdb) continue
Traceback (most recent call last):
File "/usr/lib/python2.6/pdb.py", line 1296, in main
pdb._runscript(mainpyfile)
File "/usr/lib/python2.6/pdb.py", line 1215, in _runscript
self.run(statement)
File "/usr/lib/python2.6/bdb.py", line 372 , in run
exec cmd in globals, locals
le "String>", line 1 , in <module>
index_error. $\cdot$ py", line 8 , in <module>
1e "index_error.py", line 5, in index_error
print lst [1en(1st)]
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
>/home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py (5) index_error() $\rightarrow$ print lst [len(1st)

## 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. p:

In [1]: \%run -d wiener_filtering.py
*** Blank or comment
*** Blank or comment
reakpoint 1 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py: NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

- Set a break point at line 34 using b 34
pdb> n /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(4)<module>()
3
---> 4 import numpy as np
5 import scipy as sp
ipdb> b 34
Breakpoint 2 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py:34
- Continue execution to next breakpoint with c (ont (inue)):
ipdb> c
$\xrightarrow[33]{\text { ipab }}$ /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py (34)iterated_wiener()
$\begin{aligned} & \text { 2--> } 34 \\ & 35 \text { noisy_img }=\text { noisy_img } \\ & \text { denoised_img }=\text { local_ }\end{aligned}$
- Step into code with $n$ (ext) and $s(t e p)$ : 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/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(35)iterated wiener()
$34 \quad$ noisy_img $=$ noisy_img
mean(noisy_img, size=size)
36 l_var = local_var(noisy_img, size=size)
/home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py (36)iterated_wiener () $\rightarrow 36$ denoised_img = local_mean(noisy_img, size=size)
37 _var = local_var(noisy_img, size=size)
- Step a few lines and explore the local variables
ipdb> n

> home/varoquau/dev/scipy-1ecture-notes/advanced/optimizing/wiener_filtering.py(37)iterated_wiener()
> 36 l_var = local_var(noisy_img, size=size)
> 37 for $i$ in range (3):
> res = noisy_img - denoised_img
> pdb> print 1_var
> $\left[\begin{array}{lllllll}5868 & 5379 & 5316 & \ldots & 50714799 & 5149\end{array}\right]$
> $\left.\begin{array}{llllllll}5013 & 363 & 437 & \ldots & 346 & 262 & 4355\end{array}\right]$
> $\left[\begin{array}{llllllll}{[5379} & 410 & 344 & \ldots, & 392 & 604 & 3377]\end{array}\right.$
> $\left[\begin{array}{lllllll}435 & 362 & 308 & \ldots & 275 & 198 & 1632\end{array}\right]$
> $\left[\begin{array}{llllll}{\left[\begin{array}{lllll}548 & 392 & 290 & \ldots, & 248 \\ \hline 466 & 789 & 736 & 1653]\end{array}\right]} \\ \hline & 73, & 1835 & 1725 & 1940\end{array}\right]$
> pdb> print l_var.min()

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
wiener_filtering.py:40: 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': 'print', 'invalid': 'print', 'over': 'print', 'under': 'ignore'\}
In [4]: \%run wiener_filtering.py
FloatingPointError
/home/esc/anaconda/lib/python2.7/site-packages/IPython/utils/py3compat.pyc in execfile(fname, *where) 176
$177 \quad$ filename $=$ fname
/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.py in <module>() 55 pl.matshow(noisy_face [cut], cmap=p1.cm.gray) 56
-> 57 denoised_face = iterated_wiener(noisy_face)
58 pl .matshow(denoised_face[cut], cmap=pl.cm.gray)
59
59
/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener filtering py in iterated win

$$
\begin{aligned}
& 38 \text { res = noisy_img - denoised_img } \\
& \begin{aligned}
39 & \text { noise }=(\text { res } * * 2) \text {.sum }() / \text { res } . \text { size }
\end{aligned} \\
& 41 \text { noise_level[noise_level<0] = } 0 \\
& 42
\end{aligned}
$$

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 a he 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()

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

- For stepping through code and inspecting variables, you might find it more convenient to use graphical debugger such as winpdb
- Alternatively, pudb is a good semi-graphical debugger with a text user interface in the console.
- Also, the pydbgr project is probably worth looking at.
9.3.2 Debugger commands and interaction

| l(list) | Lists the code at the current position |
| :--- | :--- |
| $\mathrm{u}(\mathrm{p})$ | Walk up the call stack |
| d (own) | Walk down the call stack |
| $\mathrm{n}($ ext $)$ | Execute the next line (does not go down in new functions) |
| $\mathrm{s}(\mathrm{tep})$ | 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 |

A 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 debugge

Type $h$ or help to access the interactive help:
ipdb> help
Documented commands (type help <topic>):

| F | bt | cont | enable jump | pdef | r | tbreak |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |

$\begin{array}{lllllllll} & c & \text { continue } & \text { exit } & l & \text { pdoc } & \text { restart } & \text { u } & \text { what } \\ \text { lias } & \text { cl } & d & \text { h } & \text { list } & \text { pinfo } & \text { return } & \text { unalias }\end{array}$
rgs clear debug help n wher

reak condition down j puit step up
Miscellaneous help topics:
xec pdb
Undocumented commands:
retval rv

### 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 ou $/$. gbdinit. 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
tarting program:/usr/bin/python segfault.py
Thread debuging using libthread do 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
(gdb)
_FAST_MOVE(Int32);
```

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
Ox004ar4f5 in _copy_from_same_shape (dest=<value optimized out> src=<va
swap=0)
at nu
748
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 riggers 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 15 at ../Python/ceval.c:3750
3750 . Python/ceval c: No such file or directory
in ../Python/ceval.c
gdb) up
Frame 0x85371ec, for file/home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.py, line 156 at ../Python/ceval.c:241
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 ca 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 0 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
    return big_array
shape=(2e6, 2e6), strides=(32, 32))
def print_big_array(small_array):
    big_array = make_big_array(small_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.
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作基 for numerical computing, but it does not work... Can you debug it Python source code: to_debug.py

## 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 faste
- Algorithmic optimizatio
* Example of the SVD

Writing faster numerical cod

- 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 bottle neck, finding a better algorithm or implementation. Keep in mind that a trade off should be found be tween 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


### 0.2.1 Timeit

In IPython, use timeit (https://docs.python.org/library/timeit.html) to time elementary operations:
In [1]: import numpy as np
in [2]: a = np.arange (1000)
[n [3]: \%timeit a ** 2
100000 loops, best of 3 : 5.73 us per loop
In [4]: \%timeit a ** 2.1
1000 loops, best of 3: 154 us per loop
n [5]: \%timeit a * a
100000 loops, best of 3: 5.56 us per loop
Use this to guide your choice between strategies.
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
mport numpy as \(n p\)
rom scipy import linal
from ica import fastica
def test()
    data \(=\) np. \(\operatorname{random.random((5000,~100))~}\)
    u, s, v = linalg.svd(data)
    \(\mathrm{pca}=\mathrm{np} \cdot \operatorname{dot}(\mathrm{u}[:,: 10] . \mathrm{T}\), data \()\)
    results = fastica(pca.T, whiten=False)
if __name__ == '__main__'
    test()
```

This is a combination of two unsupervised learning techniques, principal component analysis (PCA) and in dependent component analysis (ICA). PCA is a technique for dimensionality reduction, i.e. an algorithm explain the observed variance in your data using less dimensions. ICA is a source seperation technique, for example to unmix multiple signals that have been recorded through multiple sensors. Doing a PCA first and the 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 [1]: \%run -t demo.py
Pyython CPU timings (estimated)

User : $\quad 14.3929 \mathrm{~s}$.
System: $\quad 0.256016 \mathrm{~s}$.
and profile it:


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)

## ef test()

data $=$ np.random. $\operatorname{random}((5000,100))$
$u, s, v=l i n a l g . s v d(d a t a)$
data)

Then we run the script using the kernprof.py program, 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.py -l -v demo.py
Wrote profile results to demo.py.lprof
Timer unit: 1e-06 s
ile: demo.py
Function: test at line 5
Total time: 14.2793 s
\begin{tabular}{cc} 
Line \# Hits \\
\(=================================================\) \\
5 & Cprofile
\end{tabular}
\begin{tabular}{|c|c|c|c|}
\hline 19015 & 19015.0 & 0.1 &  \\
\hline 14242163 & 14242163.0 & 99.7 & \(\mathrm{u}, \mathrm{s}, \mathrm{v}=\mathrm{linalg} . \mathrm{svd}\) (data) \\
\hline 10282 & 10282.0 & 0.1 & \(\mathrm{pca}=\mathrm{np} . \operatorname{dot}(\mathrm{u}[: 10, \mathrm{l}, \mathrm{data})\) \\
\hline
\end{tabular}
```

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 [3]: \%timeit np.linalg.svd(data)
loops, best of 3: 14.5 s per loop
In [4]: from scipy import linalg
In [5]: \%timeit linalg.svd(data)
1 loops, best of $3: 14.2 \mathrm{~s}$ per loop
In [6]: \%timeit linalg.svd(data, full_matrices=False)
1 loops, best of 3: 295 ms per loop

In [7]: \%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()
    data = np.random.random((5000, 100))
    u, s, v = linalg.svd(data, full_matrices=False)
    pca =np.dot(u[:,:10].T, data)
    results = fastica(pca.T, whiten=False)
```


## In [1]: import demo

In [2]: \%timeit demo
emo.fastica demo.np
emo.linalg demo.prof
demo.prof.pdf demo.py demo.pyc

In [2]: \%timeit demo.test()
ica.py:65: RuntimeWarning: invalid value encountered in sqrt
$\mathrm{W}=(\mathrm{u} * \mathrm{np} . \operatorname{diag}(1.0 / \mathrm{np} . \mathrm{sqrt}(\mathrm{s})) * \mathrm{u} \cdot \mathrm{T}) * \mathrm{~W} \quad \mathrm{~W}=(\mathrm{W} * \mathrm{~W} \cdot \mathrm{~T})-\{-1 / 2\} * \mathrm{~W}$
1 loops, best of 3: 17.5 s per loop
In [3]: import demo_opt
In [4]: \%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, usin 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 [1]: a = np.zeros(1e7)
In [2]: \%timeit global a ; a = 0*a
10 loops, best of 3: 111 ms per loop

In [3]: \%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 [1]: a = np.zeros(1e7)
In [2]: \%timeit a.copy ()
10 loops, best of 3: 124 ms per loop
In [3]: \%timeit a + 1
10 loops, best of 3: 112 ms per 100

## - Beware of cache effect

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 [1]: c = np.zeros((1e4, 1e4), order='C')
In [2]: \%timeit c.sum(axis=0)
1 loops, best of 3: 3.89 s per loop
In [3]: \%timeit c.sum(axis=1)
loops, best of $3: 188 \mathrm{~ms}$ per loop
In [4]: c.strides
Dut [4]: (80000, 8)
This is the reason why Fortran ordering or C ordering may make a big difference on operations:
In [5]: a $=\mathrm{np}$. random. $\boldsymbol{r a n d}(20,2 * * 18)$
In [6]: $\mathrm{b}=$ np. random. $\begin{aligned} \text { and }(20,2 * * 18)\end{aligned}$
In [7]: \%timeit np.dot(b, a.T)
1 loops, best of 3: 194 ms per loop
In [8]: $\mathrm{c}=\mathrm{np}$.ascontiguousarray(a.T)
In [9]: \%timeit np. dot(b, c)
10 loops, best of $3: 84.2 \mathrm{~ms}$ per loop
Note that copying the data to work around this effect may not be worth it:
(10. timeit c = np.ascontiguousarray(a.T)
loops, best of 3: 106 ms per loop
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.
$\triangle$ For all the above: profile and time your choices. Don't base your optimization on theoretical considerations.

- 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 performace of your code across time, i.e. as you make new commits to your repository, you could try: vbench
- If you need some interactive visualization why not try RunSnakeRun

Author: Robert Cimrman

### 11.1 Introduction

## dense) matrix is:

- mathematical object
- data structure for storing a 2 D array of values


## mportant 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, that grows like n**
- small example (double precision matrix)
>> import numpy as np
>>> import matplotlib.pyplot as plt
>> $x=n p$.linspace $(0,1 e 6,10)$
>> plt.plot(x, 8.0 * (x**2) / 1e6, lw=5)
<<matplotlib.lines.Line2D object at ...>>

>> plt.ylabel('memory [MB] ')
<matplotlib.text. Text object at ...>
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 item
- think compression
- pros: huge memory savings
- cons: depends on actual storage scheme, (*) usually does not hold


### 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 $j$
- ...


### 11.1.4 Prerequisites

recent versions of

- 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 matrix types in scipy.sparse:

1. csc_matrix: Compressed Sparse Column format
2. csr_matrix: Compressed Sparse Row format
3. bsr_matrix: Block Sparse Row format
4. lil_matrix: List of Lists format
5. dok matrix: Dictionary of Keys format
6. coo_matrix: COOrdinate format (aka IJV, triplet format)
7. dia_matrix: DIAgonal format

- each suitable for some tasks
- many employ sparsetools C++ module by Nathan Bel
- assume the following is imported:
>>> import numpy as np
>>> import scipy.sparse as sps
>> import matplotlib.pyplot as plt


## - warning for NumPy users

- the multiplication with '*' is the matrix multiplication (dot product)
- not part of NumPy!
* passing a sparse matrix object to NumPy functions expecting ndarray/matrix does not work


### 11.2.1 Common Methods

## - all scipy.sparse classes are subclasses of spmatrix

- default implementation of arithmetic operations


## * always converts to CSR

* subclasses override for efficiency
- shape, data type set/get
- nonzero indices
- format conversion, interaction with NumPy (toarray (), todense()
- ...
- attributes:


## - mtx.A - same as mtx.toarray()

- mtx. T - transpose (same as mtx.transpose ())
- mtx. H - Hermitian (conjugate) 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 usually stored in NumPy arrays


### 11.2.2 Sparse Matrix 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 matrix 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 operation

- manipulate data array directly (fast NumPy machinery)
- constructor accepts
- dense matrix (array)
- sparse matrix
- shape tuple (create empty matrix)
- (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 matrices

```
>> data = np.array([[1, 2, 3, 4]]).repeat(3, axis=0)
    data
    [1,2,3,4],
    [1, 2, 3, 4],
    >> offsets = np.array([0, -1, 2])
    >> mtx = sparse.dia_matrix((data, offsets), shape=(4, 4))
    >> mtx
    <4x4 sparse matrix of type '<... 'numpy.int64'>'
            with 9 stored elements (3 diagonals) in DIAgonal format>
>>> mtx.todense()
natrix([[1, 0, 3, 0]
            [1, 2, 0, 4],
            [0, 0, 3, 4]]
>> data = np.arange(12).reshape((3,4)) + 1
>> data
array([[ 1, 2, 3, 4],
    [ 5, 6, 7, 8],
    >> mtx = sparse.dia_matrix((data, offsets), shape=(4, 4))
    >> mtx.data
    array([[ 1, 2, 3, 4],
        [ [ 5, 6, 7, 8],
    >> mtx.offsets
    array([ 0, -1, 2], dtype=int32)
    > print(mtx)
        (0,0)
        (1,1)
        (2, 2)
    (3,3
    (1,0)
    (3, 2
    (3,2)
    (1, 3)
    (1, mtx.todense()
matrix([[ 1, 0, 11, 0],
    [ 5, 2, 0, 12],
    0, 0, 7, 4]7)
```

- explanation with a scheme:

```
offset: row
    2: 9
    0: 1 . 11 .
    -1: 5 2 . 12
    -2: . }
    -3: : .-. 
```

- matrix-vector multiplication

$$
\begin{aligned}
& \text { >>> vec }=n p . \operatorname{ones}((4,)) \\
& \text { >> vec }
\end{aligned}
$$

$$
\begin{aligned}
& \gg \text { vec } \\
& \text { array([ } 1 ., \text { 1., } 1 ., 1 .]) \\
& \ggg \text { mtx } ~ \text { v vec }
\end{aligned}
$$

$$
\ggg \operatorname{mtx} \text { v vec }
$$

$$
\begin{aligned}
& \text { >>> mtx * vec } \\
& \text { 19rray([ 12., 19., 9., 11.]) } \\
& \ggg \text { mtx.toarray() * vec }
\end{aligned}
$$

$$
\begin{aligned}
& \gg \text { mtx.toarray () * vec } \\
& \text { array([] 1., }
\end{aligned}
$$

$$
\operatorname{array}\left(\left[\begin{array}{cccc}
{\left[\begin{array}{lll}
1 ., & 0 ., & 11 ., \\
{\left[\begin{array}{ll}
2 ., & 0 .
\end{array}\right],} & 0 ., & 12 .],
\end{array},\right.}
\end{array}\right.\right.
$$

$$
\left[\begin{array}{ccccc}
{[ } & 5 ., & 2 ., & 0 ., & 12 .], \\
{[ } & 0 ., & 6 ., & 3 ., & 0 .],
\end{array}\right.
$$

$$
\left[\begin{array}{llll}
0 ., & 0 ., & 7 ., & 4 .]
\end{array}\right]
$$

## 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 matrices incrementally
- constructor accepts:
- dense matrix (array)
- sparse matrix
- shape tuple (create empty matrix)
- flexible slicing, changing sparsity structure is efficient
- slow arithmetics, slow column slicing due to being row-based
- use:
- when sparsity pattern is not known apriori or changes
- example: reading a sparse matrix from a text file


## Examples

- create an empty LIL matrix:
>>> mtx = sparse.lil_matrix $((4,5))$


## - prepare random data:

> >>> from numpy.random import rand
> >> data $=$ np. round $($ rand $(2,3))$
> $\ggg$ data
> $\operatorname{array}\left(\left[\begin{array}{lll}\text { 1., } & 1 ., & 1 .], \\ \\ {[1 .,} & 0 ., & 1 .]])\end{array}\right.\right.$

- assign the data using fancy indexing:

```
>> mtx[:2, [1, 2, 3]] = data
    >> mtx
    <4x5 sparse matrix of type '<... 'numpy.float64'>'
    with 5 stored elements in LInked List format>
    > print(mtx)
    0,2) 1.0
    (0,3) 1.0
    (1,1) 1.0
    (1, 3) 1.0
>>> mtx.todense()
matrix([[ 0., 1., 1., 1., 0.],
    [0.,}1.
        [ 0., 0., 0., 0., 0.]])
    >> mtx.toarray(),
    array([[ 0., 1., 1., 1., 0.],
    [[0.,}1.,\mp@code{0.,
    [ll,
```

- more slicing and indexing
>> mtx = sparse.lil_matrix([[0, 1, 2, 0], [3, 0, 1, 0], [1, 0, 0, 1]])
$\ggg$ mtx.todense()
matrix $\left(\left[\begin{array}{ll} & 1, \\ 1,2,0]\end{array}\right]\right.$,
$[3,0,1,0]$,
$[1,0,0,1]] \ldots)$
>> print(mtx)
$(0,1)$
$(1,0)$
$(1,2)$
1, 2)
$(2,0)$
$(2,3)$
>> mtx[:2,:]
$2 \times 4$ sparse matrix of type '<... 'numpy.int64'>
with 4 stored elements in LInked List format>
$\gg \operatorname{mtx}[: 2,:]$ todense()
natrix([[0, 1, 2, 0],
> $[1,0,1,0]] \ldots$ )
>> mtx[1:2, $[0,2]]$ todense(
natrix $([12,1]] \ldots$
matrix ([[0, 1, 2, 0],
$[3,0,1,0]$,

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 matrices incrementally


## constructor accepts

- dense matrix (array)
- sparse matrix


## - shape tuple (create empty matrix)

- efficient $O(1)$ access to individual elements
- flexible slicing, changing sparsity structure is efficient
- can be efficiently converted to a coo_matrix once constructed
- slow arithmetics (for loops with dict.iteritems())
- use:
- when sparsity pattern is not known apriori or changes


## Examples

- create a DOK matrix element by element:
>>> mtx $=$ sparse.dok_matrix((5, 5), dtype=np.float64)
>> mtx
<5x5 sparse matrix of type '<... 'numpy.float64'>'
with 0 stored elements in Dictionary Of Keys format>
for ir in range (5):
for ic in range(5):
$\quad$ mtx[ir, ic] $=1.0 *(i r ~!=~ i c)$
>>> mtx
$5 \times 5$ sparse matrix of type '<... 'numpy.float64'>
with 20 stored elements in Dictionary Of Keys format>
>>> mtx.todense()
natrix([[ 0., 1., 1., 1., 1.],
$\left[\begin{array}{llllll}{[1 .,} & 0 ., & 1 ., & 1 ., & 1 .],\end{array}\right.$
[1., 1., 1., 0., 1.]
$\left[\begin{array}{llll}1 ., & 1 ., & 1 ., & 1 ., \\ 1 ., & 1 ., & 1 ., & 1 ., \\ 0 .]\end{array}\right]$


## - slicing and indexing:

>>> mtx[1, 1]
0.0

1x2 sparse matrix of type '<... 'numpy.float64'>
with 1 stored elements in Dictionary Of Keys format>
$\gg \operatorname{mtx}[1,1: 3]$ todense()
natrix([[ 0., 1.]])
$\ggg \operatorname{mtx}[[2,1], 1: 3]$.todense()
natrix([[ $1 ., 0$.$] ,$

## Coordinate Format (COO)

## - also known as the ' $\mathbf{i j v}$ ' or 'triplet' format

- three NumPy arrays: row, col, data
- 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 matrices
- constructor accepts
- dense matrix (array)
- sparse matrix
- shape tuple (create empty matrix)
- (data, ij) tuple
- very fast conversion to and from CSR/CSC format
- fast matrix * vector (sparsetools)


## - fast and easy item-wise operation

- manipulate data array directly (fast NumPy machinery)
- no slicing, no arithmetics (directly)
- 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 matrix
>>> mtx $=$ sparse. coo_matrix ( $(3,4)$, dtype=np.int8)
>>> mtx.todense()
matrix([[0, 0, 0, 0]
$[0,0,0,0]$
$[0,0,0,0]]$, dtype=int8)
- create using (data, ij) tuple:
>>> row $=$ np. $\operatorname{array([0,3,1,0])}$
>>> col $=$ np.
>> data $=$ np.array $([4,5,7,9])$
>> mtx $=$ sparse.coo matrix ((data, (row, col)), shape=(4, 4))
>> mtx
x4 sparse matrix of type '<... 'numpy.int64'>'
with 4 stored elements in CoOrdinate format
todense()
matrix([[4, 0, 9, 0],
$[0,7,0,0]$,
$[0,0,0,0]$,
$[0,0,0,5]]$ )
- duplicates entries are summed together:

```
\(\ggg\) col \(=\) np. \(\operatorname{array}([0,2,1,3,1,0,0])\)
>> data \(=\) np.array \(([1,1,1,1,1,1,1])\)
\(\gg\) mtx \(=\) sparse.coo_matrix ((data, (row, col)), shape=(4, 4))
>> mtx.todense()
natrix([[3, 0, 1, 0]
    \(\left.\left.\begin{array}{l}{[0,} \\ {[0,} \\ {[0,} \\ 0,\end{array} 0,0\right], 0\right]\),
    \([0,0,0,1]]\)
```

- no slicing...

$$
\begin{aligned}
& \ggg \text { mtx }[2,3] \\
& \text { Traceback (most recent call last): } \\
& \text { TypeError: 'coo_matrix' object ... }
\end{aligned}
$$

```
Traceback (most recent call last)
TypeError: 'coo_matrix' 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 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 in dices indices [indptr[i]: indptr[i+1]]
* item ( $i, j$ ) can be accessed as data[indptr [ i$]+\mathrm{k}]$, 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 matrix classes with data attribute
- fast matrix vector products and other arithmetics (sparsetools


## - constructor accepts

- dense matrix (array)
- sparse matrix
- shape tuple (create empty matrix)
- (data, ij) 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 matrix:

> >>> mtx = sparse.csr_matrix ((3, 4), dtype=np.int8)
> >> mtx.todense()
> matrix([[0, 0, 0, 0]
> $\left.\begin{array}{lll}{\left[\begin{array}{llll}{[0,} & 0 & 0, & 0\end{array}\right],} \\ {[0,} & 0, & 0\end{array}, 0\right]$, dtype=int8)

- create using (data, ij) tuple:
>>> row $=$ np.array ([0, 0, 1, 2, 2, 2])
$\ggg c o l=n p$. $\operatorname{array}([0,2,2,0,1,2])$
>> mtx = sparse.csrmatrix((data, (row, col)), shape=(3, 3))
>> mtx

```
<3x3 sparse matrix of type '<... 'numpy.int64'>'
    vith 6 stored elements in Compressed Sparse Row format>
    mtx.todense()
    matrix([[1, 0, 2]
        [0,0, 3],
    [4,5,6]]...)
    >> mtx.data
    array([1, 2, 3, 4, 5, 6]\ldots)
    >> mtx.indices
    array([0, 2, 2, 0, 1, 2], dtype=int32)
    >>> mtx.indptr
    array([0, 2, 3, 6], dtype=int32)
```

- create using (data, indices, indptr) tuple:
>>> data $=$ np. array $([1,2,3,4,5,6])$
$\gg$ indices $=n p$.array ( $[0,2,2,0$,
$\gg$ indptr $=n p$.array $([0,2,3,6])$
>>> mtx = sparse.csr_matrix ((data, indices, indptr), shape=(3,3))
atrix $\left(\left[\begin{array}{lll}1 & 0 & 2\end{array}\right]\right.$
natrix $([11,0,2]$
$\left[\begin{array}{ll}{[0,} & 0, \\ {[4,5} & 5\end{array}\right]$,

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 value
* 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 in dices 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 matrix classes with data attribute)
- fast matrix vector products and other arithmetics (sparsetools)
- constructor accepts:
- dense matrix (array)
- sparse matrix
- shape tuple (create empty matrix)
- (data, ij) 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 matrix:

```
>> mtx.todense()
atrix([L0
    \([0,0,0,0]\),
    \([0,0,0,0]]\), dtype=int8
```

- create using (data, ij) tuple:
>> row $=$ np.array([0, 0, 1, 2, 2, 2])

>> mtx = sparse.csc_matrix ((data, (row, col)), shape=(3, 3))
>> mtx
$<3 \times 3$ sparse matrix of type '<... 'numpy.int64'>'
with 6 stored elements in Compressed Sparse Column format>
>> mtx.todense()
atrix( $\left[\begin{array}{lll}{[1,} & 0, & 2] \\ & & 3]\end{array}\right.$
$[4,5,6]] \ldots)$
>> mtx.data
$\operatorname{array}([1,4,5,2,3,6] \ldots$ )
$\ggg$ mtx.indices
$\operatorname{array}([0,2,2,0,1,2]$, dtype=int32)
>> mtx.indptr
array ([0, 2, 3, 6], dtype=int32)
- create using (data, indices, indptr) tuple:
>> data $=$ np.array([1, 4, 5, 2, 3, 6])
>> indices $=n p$.array ( $[0,2,2,0,1,2]$ )
>> indptr $=$ np.array $([0,2,3,6])$
>> mtx = sparse.csc_matrix((data, indices, indptr), shape=(3, 3))
>> mtx.todense()
matrix([[1, 0, 2]
$[4,5,6]$ )

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 ( $\mathrm{nnz}, \mathrm{R}, \mathrm{C}$ )
* ...


## - subclass of _cs_matrix (common CSR/CSC functionality)

* subclass of data matrix (sparse matrix classes with data attribute)
- fast matrix vector products and other arithmetics (sparsetools)


## - constructor accepts:

- dense matrix (array)
- sparse matrix
- shape tuple (create empty matrix)
- (data, ij) tuple
- (data, indices, indptr) tuple
- many arithmetic operations considerably more efficient than CSR for sparse matrices with dense submatrices
- use:
- like CSR
- vector-valued finite element discretizations

Examples

- create empty BSR matrix with $(1,1)$ block size (like CSR...):
>>> mtx = sparse.bsr_matrix((3, 4), dtype=np.int8)
>> mtx
<3x4 sparse matrix of type '<... 'numpy.int8'>
with 0 stored elements (blocksize $=1 \times 1$ ) in Block Sparse Row format>
todense()
natrix([[0, 0, 0, 0]
$\left[\begin{array}{llll}{[0,} & 0, & 0, & 0\end{array}\right]$
$\left.\left[\begin{array}{lll}1, & 0, & 0, \\ 0\end{array}\right]\right]$, dtype=int8
- create empty BSR matrix with $(3,2)$ block size:
>>> mtx = sparse.bsr_matrix((3, 4), blocksize=(3, 2), dtype=np.int8)
<3x4 sparse matrix of type '<... 'numpy.int8'>
with 0 stored elements (blocksize $=3 \times 2$ ) in Block Sparse Row format>
>> mtx.todense()
natrix([[0, 0, 0, 0]
$[0,0,0,0]$
$[0,0,0,0]]$, dtype=int8)
- a bug?
- create using (data, ij) tuple with (1, 1) block size (like CSR...):
$\ggg$ row $=$ np. $\operatorname{array}([0,0,1,2,2,2])$
>>> col = np. array([0, 2, 2, 0, 1, 2])
>>> mtx = sparse.bsr_matrix((data, (row, col)), shape=(3, 3))
>> mtx
<3x3 sparse matrix of type '<... 'numpy.int64'>'
with 6 stored elements (blocksize $=1 \times 1$ ) in Block Sparse Row format>
>>> mtx.todense()
matrix([[1, 0, 2]
$[1,0,3]$,
$[4,5,6]] \ldots)$
$\gg$ mtx. data
[[2]],
[ [3]]
[ [4]]
[[5]],

[[6]]]..

array ([0, 2, 2, 0, 1, 2], dtype=int32)

$\operatorname{array}([0,2,3,6]$, dtype=int32)

- create using (data, indices, indptr) tuple with $(2,2)$ block size:
>>> indptr $=n p$.array ( $[0,2,3,6]$ )
>> indices = hp.aara( $1,2,2,2,1$ ),
6]).repeat (4).reshape (6, 2, 2)
>> mtx = sparse.bsr_matrix((data, indices, indptr), shape=(6, 6))
》> mtx
matrix([[1, 1, 0, 0, 2, 2],
$\left.\begin{array}{l}{[1,1,0,0,2,2],} \\ {[0,0,0,} \\ 0,\end{array}, 3,3\right]$,
$[0,0,0,0,3,3]$,
$[4,4,5,5,6,6]$,
$[4,4,5,5,6,6]]$
>> data
$\operatorname{array}([[[1,1]$,
$[1,1]]$
$\left[\begin{array}{ll}{[2,2],} \\ {[2,2]}\end{array}\right.$,
$\left[\begin{array}{ll}{[3,3]} \\ {[3,3]}\end{array}\right.$
$[3,3]]$
$[4,4]$,
$\left[\begin{array}{l}{[5,5],} \\ [5,5]]\end{array}\right.$,
[6, 6],
[6, 6]j])


### 11.2.3 Summary

| Table 11.1: Summary of storage schemes. |  |  |  |  |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :---: |
| for- <br> mat | matrix <br> vector | get <br> item | fancy <br> get | set <br> item | fancy <br> set | solvers | note |  |
| DIA | sparse- <br> tools | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ | itera- <br> tive | has data array, specialized |  |
| LIL | via CSR | yes | yes | yes | yes | itera- <br> tive | arithmetics via CSR, <br> incremental construction |  |
| DOK | python | yes | one axis <br> only | yes | yes | itera- <br> tive | O(1) item access, incremental <br> construction |  |
| COO | sparse- <br> tools | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ | itera- <br> tive | has data array, facilitates fast <br> conversion |  |
| CSR | sparse- <br> tools | yes | yes | slow | $\cdot$ | any | has data array, fast row-wise ops |  |
| CSC | sparse- <br> tools | yes | yes | slow | $\cdot$ | any | has data array, fast column-wise <br> ops |  |
| BSR | sparse- <br> tools | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ | spe- <br> cialized | has data array, specialized |  |

### 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.sparse.linalg as spla
>>> spla. _-all_-
'LinearOperator', 'Tester', 'arpack', 'aslinearoperator', 'bicg',
'bicgstab', 'cg', 'cgs', 'csc_matrix', 'csr_matrix', 'dsolve'
'isolve', 'iterative', 'lgmres', 'linsolve', 'lobpcg', 'lsqr',
'minres', 'np', 'qmr', 'speigs', 'spilu', 'splu', 'spsolve', 'svd'
'test', 'umfpack', 'use_solver', 'utils', 'warnings']


### 11.3.1 Sparse Direct Solvers

## - default solver: SuperLU 4.0

- 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.umf pack
- check-out the new scikits. suitesparse by Nathaniel Smith


## Examples

- import the whole module, and see its docstring
>> from scipy.sparse.linalg import dsolve
>>> help(dsolve)
- both superlu and umfpack can be used (if the latter is installed) as follows:
- prepare a linear system:
>>> import numpy as np
>> from scipy import sparse
>> mtx.todense()
matrix([[ 1, 5, 0, 0, 0],
$[0,2,8,0,0]$,
$[0,0,3,9,0]$,
$[0,0,0,4,10]$,
>>> rhs = np.array([1, 2, 3, 4, 5], dtype=np.float32)
- solve as single precision real:

$$
\begin{aligned}
& \begin{array}{l}
\text { s>> mtx1 }=\text { mtx.astype(np.float } 32 \text { ) } \\
\ggg \gg=\text { dsolve.spsolve(mtx1, rhs }
\end{array} \\
& \text { >>> } \mathrm{x}=\text { dsolve.spsolve(mtx1, rhs, use_umfpack=False) } \\
& \text { >>> print (x) } \\
& \text { 106. }-21 . \quad 5.5-1.51 .] \\
& \text { [ } 0 \text { (mor: } \mathrm{Ks}^{n} \% \text { (mtx1 * x - rhs)) }
\end{aligned}
$$

- solve as double precision real:

$$
\begin{aligned}
& \begin{array}{l}
\text { >> mtx2 }=\text { mtx.astype (np.float64) } \\
\ggg \mathrm{x}=\text { dsolve. }
\end{array} \\
& \text { >> } \mathrm{x}=\text { dsolve.spsolve(mtx2, rhs, use_umfpack=True) } \\
& \text { 106. }-21 \text {. } \\
& \text { 106. }-21 .
\end{aligned}
$$

## - solve as single precision complex:

>>> mtx1 = mtx.astype (np.complex64)
>>> $\mathrm{x}=$ dsolve.spsolve(mtx1, rhs, use_umf pack=False)
print(x)
106.0+0.j $-21.0+0 . j \quad 5.5+0 . j \quad-1.5+0 . j \quad 1.0+0 . j]$
> 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:

$$
\begin{aligned}
& \text { >>> mtx2 = mtx.astype(np. complex128) } \\
& \text { >> } \mathrm{x}=\text { dsolve.spsolve(mtx2, rhs, use umfpack=True) } \\
& \text { >> print(x) } \\
& \text { 106.0+0.j }-21.0+0 . j \quad 5.5+0 . j \quad-1.5+0 . j \quad 1.0+0 . j] \\
& \text { [rint(Error: \%s" \% (mtx2 * x - rhs)) } \\
& \text { Error: [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j] }
\end{aligned}
$$

```
Solve a linear system
Construct a 1000x1000 lil_matrix and add some values to it, convert it
to CSR format and solve A }x=b\mathrm{ for x:and solve a linear system with a
direct solver.
direct
import numpy as np
mport scipy.sparse as sps
from matplotlib import pyplot as plt
from scipy.sparse.linalg.dsolve import linsolve
rand = np.random.rand
mtx = sps.lil_matrix((1000, 1000), dtype=np.float64
tx[0, :100] = rand(100)
mtx[1, 100:200] = mtx[0, :100]
mtx.setdiag(rand(1000))
plt.clf()
plt.spy(mtx, marker='.', markersize=2)
plt.show()
tx = mtx.tocsr()
rhs = rand(1000)
\(\mathrm{x}=\) linsolve.spsolve(mtx, rhs)
```


## print('rezidual: $\% \mathrm{r}$ 「 \% np.linalg.norm(mtx $* x-r h s)$ )

- examples/direct_solve.py


### 11.3.2 Iterative Solvers

## - the isolve module contains the following solvers:

## - bicg (BIConjugate Gradien

- 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 matrix, dense 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 ( $\mathrm{N}, 1$ ).

- optional
$\mathbf{x 0}$ [\{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 matrix, dense matrix, LinearOperator\}] Preconditioner for A. The preconditioner should ap proximate 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

from scipy.sparse.linalg.interface import LinearOperator

- 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
>> from scipy.sparse.linalg import LinearOperator
>> def $m v(v)$ :
return np.array ([2*v[0], $3 * v[1]]$ )
>> $\mathrm{A}=$ LinearOperator ( $(2,2)$, matvec=mv $)$


## >>> A

<2x2 LinearOperator with unspecified dtype>
>>> A.matvec (np. ones (2))
$\operatorname{array}([2 ., 3]$.
> A * np. ones (2)
rray([ $2 ., \quad 3]$.

## Few Notes on Preconditioning

- problem specific
- often hard to develop
- if not sure, try ILU
- available in dsolve as spilu()


### 11.3.3 Eigenvalue Problem Solvers

## The eigen module

- arpack * a collection of Fortran 77 subroutines designed to solve large scale eigenvalue problems
- lobpcg (Locally Optimal Block Preconditioned Conjugate Gradient Method) * works very well in combi nation 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 scipy
from scipy.sparse.linalg import lobpcg
from pyamg import smoothed_aggregation_solver
from pyamg.gallery import poisson
$\mathrm{N}=100$
= poisson((N,N), format='csr')
\# create the AMG hierarchy
$\mathrm{ml}=$ smoothed_aggregation_solver(A)
initial approximation to the $K$ eigenvectors
preconditioner based on m
= m1.aspreconditioner()
\# compute eigenvalues and eigenvectors with LOBPCG
$\mathrm{W}, \mathrm{V}=\operatorname{lobpcg}(\mathrm{A}, \mathrm{X}, \mathrm{M}=\mathrm{M}$, tol $=1 \mathrm{e}-8$, largest=False $)$
plot the eigenvectors
import pylab

## pylab.figure(figsize=(9,9))

for $i$ in range (K):
pylab.subplot (3, 3, i+1)
pylab.subplot
pylab.title('Eigenvector $\%{ }^{1}$ ' \% i) $) ~$
pylab. pcolor(V[:, i] reshape(N,N))
pylab.axis('equal')
pylab.axis('off'
pylab.show()

- examples/pyamg_with_lobpcg.py
- example by Nils Wagner:
- examples/lobpcg_sakurai.py
- output:
\$ python examples/lobpcg_sakurai.py
Results by LOBPCG for $n=2500$
$\left[\begin{array}{lll}0.06250083 & 0.06250028 & 0.06250007\end{array}\right]$
Exact eigenvalues
$\left[\begin{array}{lll}0.06250005 & 0.0625002 & 0.06250044\end{array}\right]$
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
- http://pysparse.sourceforge.net/

Authors: Emmanuelle Gouillart Gaël Varoquaux
his section addresses basic image manipulation and processing using the core scientific modules NumPy and ciPy. Some of the operations covered by this tutorial may be useful for other kinds of multidimensional arra processing than image processing. In particular, the submodule scipy .ndimage provides functions operating on n-dimensional NumPy array
See also:
For more advanced image processing and image-specific routines, see the tutorial Scikit-image: image pro essing, 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

## ools used in this tutorial

- numpy: basic array manipulation
- scipy: scipy.ndimage submodule dedicated to image processing (n-dimensional images). See the documentation:
>> from scipy import ndimage


## Common tasks in image processing:

- Input/Output, displaying image
- 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: ndimage.measurements


### 12.1 Opening and writing to image files

Writing an array to a file:
from scipy import misc
= misc.face()
isc.imsave('face.png', f) \# uses the Image module (PIL)
mport matplotlib.pyplot as plt
plt.imshow(f)
plt.show()


Creating a numpy array from an image file:

```
>> from scipy import mis
>> face = misc.face()
>> misc.imsave('face.png', face) # First we need to create the PNG file
>> face = misc.imread('face.png')
    >> type(face)
<... 'numpy ndarray '>
>> face.shape, face.dtype
((768, 1024, 3), dtype('uint8'))
```

dtype is uint8 for 8-bit images (0-255)

## pening 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)) $\qquad$
(data are read from the file, and not loaded into memory)
Working on a list of image files
>>> for i in range(10):
im = np.random.random_integers(0, 255, 10000).reshape((100, 100))
misc.imsave('random_\%o2d.png' \% i, 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 = misc.face(gray=True) \# retrieve a grayscale image
>>> import matplotlib. pyplot as plt
<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 $0 x . . .>$
>>> \# Remove axes and ticks
>> plt.axis('off')
$(-0.5,1023.5,767.5,-0.5)$
Draw contour lines:
>>> plt.contour(f, [50, 200])
matplotlib.contour. QuadContourSet


For fine inspection of intensity variations, use interpolation='nearest':
$\gg$ plt.imshow $(\mathrm{f}[320: 340,510: 530]$, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>
》> ppletlent


See also:
3-D visualization: Mayavi
See 3D plotting with Mayavi

- Image plane widgets
- Isosurfaces
- ...



### 12.3 Basic manipulations

Images are arrays: use the whole numpy machinery.

>>> face = misc.face(gray=True)
$\gg$ face $=$ misc
$\gg$ face $[0,40]$

>> \# Slicing
>>> face[10:13, 20:23]
array([[141, 153, 145],

$$
[133,134,125] \text {, }
$$

[ $96,92,94]$ ], dtype=uint8)
>> face[100:120] = 255
>>>
>> $1 \mathrm{x}, 1 \mathrm{ly}=$ face.shape
>> mask $=(\mathrm{X}-1 \mathrm{x} / 2) * * 2+(\mathrm{Y}-1 \mathrm{y} / 2) * * 2>1 \mathrm{x} * 1 \mathrm{y} / 4$
>> \# Masks
>>> face[mask] = 0
>> \# Fancy indexing
face[range(400), range(400)] $=255$


### 2.3.1 Statistical information

```
>> face = misc.face(gray=True)
>>> face.mean()
>>> face.mean()
>> face.max(), face.min()
(250, 0)
```


## Exercise

- Open as an array the scikit-image logo (http://scikit-image.org/_static/img/logo.png), or an im age 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.
- 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)


## scikits-image

image processing in python
12.3.2 Geometrical transformations


### 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:

>> from scipy import misc
>> face = misc. face(gray=True)
>> blurred_face = ndimage.gaussian_filter(face, sigma=3)

## Uniform filter

>>> local_mean = ndimage.uniform_filter(face, size=11)


### 12.4.2 Sharpening

## Sharpen a blurred image:

>> from scipy import mis
face = misc.face(gray=True). astype(float)
>>> blurred_f = ndimage.gaussian_filter(face, 3)
increase the weight of edges by adding an approximation of the Laplacian
>>> filter_blurred_f = ndimage.gaussian_filter(blurred_f, 1)
>> alpha = 30
>> sharpened = blurred_f + alpha * (blurred_f - filter_blurred_f)


### 12.4.3 Denoising

Noisy face:
>>> from scipy import misc
>>
$\gg \mathrm{f}=\mathrm{misc}$.face(gray=True)
l
>> noisy $=\mathrm{f}+0.4 * \mathrm{f}$.std() $*$ np.random.random( f .shape)

## Gaussian filter smoothes the noise out... and the edges as well

>>> gauss_denoised = ndimage.gaussian_filter (noisy, 2)
Most local linear isotropic filters blur the image (ndimage.uniform_filter

## A median filter preserves better the edges:

>>> med_denoised = ndimage.median_filter(noisy, 3)


Median filter: better result for straight boundaries (low curvature):
>> im $=$ np. zeros $((20,20))$
$\ggg$ im $[5:-5,5:-5]=1$
>> $\operatorname{im}=$ np. zeros ( 20,
>>> im = ndimage.distance_transform_bf(im)
>> im_noise $=\mathrm{im}+0.2$ * np.random.randn(*im.shape)
>> im_med = ndimage.median_filter(im_noise, 3)


Other rank filter: ndimage.maximum_filter, 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.

### 2.4.4 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 = ndimage.generate_binary_structure (2, 1)
```

>> el
array([[False, True, False]
[ True, True, True],
[False, True, False]], dtype=bool)
>> el.astype(np.int)
$\operatorname{array}([[0,1,0]$,
$\left.\begin{array}{ll}{[1,} & 1, \\ {[0,} & 1, \\ 0\end{array}\right]$ )


Erosion = minimum filter. Replace the value of a pixel by the minimal value covered by the structuring ele ment.:

```
>> a = np.zeros((7,7), dtype=np.int)
    >>> 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, 1, 0, 0]
    [0,0,1,1,1, 0, 0]
    [0, 0, 0, 0, 0, 0, 0]])
, astype(a.dtype)
array([[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
>> ndimage.binary_erosion(a, structure=np.ones(( }5,5))).\mathrm{ astype (a.dtype)
\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: maximum filter:


Also works for grey-valued images:
>> np.random.seed(2)
>> $\mathrm{x}, \mathrm{y}=(63 * \mathrm{np}$. random.random $((2,8)))$.astype (np. int
>> im $[x, y]=n p$.arange(8)
>>> bigger_points = ndimage.grey_dilation(im, size=(5, 5), structure=np.ones((5, 5)))
>> square $=$ np.zeros $((16,16)$ )
>> square $[4:-4,4:-4]=$
>> dist = ndimage.distance_transform bf (square)
>> dilate_dist = ndimage.grey_dilation(dist, size=(3, 3), structure=np. ones ( $(3,3)$ ))


Opening: erosion + dilation:
>>> a = np.zeros((5,5), dtype=np.int)
$\gg \mathrm{a}=\mathrm{np} . \operatorname{zeros}((5,5)$, dtype=np
$\gg \mathrm{a}[1: 4,1: 4]=1 ; \mathrm{a}[4,4]=1$
>>> a[1
$\operatorname{array}([[0,0,0,0,0]$
$[0,1,1,1,0]$,
$[0,1,1,1,0]$
$[0,1,1,1,0]$ $\left[\begin{array}{llll}{[0,} & 0 & 0, & 0, \\ \hline\end{array}\right]$ )
>> \# Opening removes small objects
>> ndimage binary opening(a, structure=np ones(( 3,3$))$ ) astype(np.int)
$\operatorname{array}\left(\left[\begin{array}{llll}{[0,} & 0, & 0, & 0, \\ 0\end{array}\right]\right.$.
$[0,1,1,1,0]$,
$[0,1,1,1,0]$,
$[0,1,1,1$,
$[0$,,
$[0,1,1,1,0]$,
$[0,0,0,0,0]]$
>>> \# Opening can also smooth corners
>>> ndimage. binary_opening(a) astype(np.int)
$\operatorname{array}([[0,0,0,0,0]$,
$[0,0,1,0,0]$,
$[0,1,1,1,0]$,
$[0,0,1,0,0]$,

## Application: remove noise

>>> square $=$ np.zeros( $(32,32)$ )
>> square $[10:-10,10:-10]=1$
>> np. random. seed (2
$\gg \mathrm{x}, \mathrm{y}=(32 * \mathrm{np}$. random. $\operatorname{random}((2,20)))$.astype(np.int)
>> square $[x, y]=1$
>> open_square = ndimage.binary_opening(square)
>>> eroded_square = ndimage.binary_erosion(square)
>> reconstruction = 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] =
>>
>> im = ndimage.rotate(im, 15, mode='constant')
im = ndimage.gaussian_filter(im, 8)
```

Use a gradient operator (Sobel) to find high intensity variations
>>> sx = ndimage.sobel(im, axis=0, mode=' constant')
>> sy = 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
＞＞points \(=1 *\) np．random． \(\operatorname{random}((2, \mathrm{n} * * 2))\)
＞＞im＝ndimage ．astype（np．int），（points［1］）．astype（np．int）］＝1
1m＝ndimage．gaussian＿filter（im，sigma＝1／（4．＊n））
＞＞mask \(=(\mathrm{im}>\) im．mean（））．astype（np．float）
＞＞mask＋＝ 0.1 ＊im
＞＞ \(\mathrm{img}=\) mask \(+0.2 * \mathrm{np}\). random． randn （＊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＝ndimage．binary＿opening（binary＿img）
＞＞\＃Remove small black hole
＞＞close＿img＝ndimage．binary＿closing（open＿img）

## 又双运

## Exercise

Check that reconstruction operations（erosion + propagation）produce a better result than open ing／closing：

```
M>> recod_ing = ndimage.binary_erosion(binary_img)
    *)
    $> tmp =np.logical_not(reconstruct_img)
    > eroded_tmp = ndimage.binary_erosion(tmp)
    reconstruct_final = np.logical_not(ndimage.binary_propagation(eroded_tmp, mask=tmp))
    > np.abs(mask - close_img).mean()
.00727836
>>> np.abs(mask - reconstruct final).mean()
00059502
```


## 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
＞＞ $1=10$
＞＞ $\mathrm{x}, \mathrm{y}=\mathrm{np}$ ．indices（ $(1,1)$ ）
＞＞center $1=(28,24)$
＞＞center2 $=(40,50)$
＞＞center3 $=(67,58)$
$\ggg$ center4 $4=(24,70)$
＞＞＞radius1，radius2，radius3，radius4 $=16,14,15,14$
＞＞circle1 $=(\mathrm{x}-$ center1［0］）$* * 2+(\mathrm{y}-$ center1［1］$) * * 2<$ radius $1 * * 2$ ＞＞circle2 $=(\mathrm{x}-$ center2［0］）$* * 2+(\mathrm{y}-$ center2［1］）$* * 2<$ radius2 $2 * 2$ $\ggg$ circle3 $=(x-$ center3［0］$* * 2+(y-$ center3 3 ［1］$) * * 2<$ radius3 $* 2$
＞＞\＃ 4 circles
＞＞＞img＝circle1＋circle2＋circle3＋circle4
＞＞mask＝img．astype（bool）
＞＞img＝img．astype（float）
＞＞img＋＝ $1+0.2 * n$ p．random．randn（＊img．shape）
＞＞＞\＃Convert the image into a graph with the value of the gradient a ＞＞\＃the edges
》 graph＝image．img＿to＿graph（img，mask＝mask）
\＞＞\＃Take a decreasing function of the gradient：we take it weakly
＞＞＞\＃dependant from the gradient the segmentation is close to a vorono
＞＞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: ndimage.measurements

Synthetic data:

>
im $=$ np.
eros $((1,1))$
>> points $=1 *$ np.random.random ( $(2, \mathrm{n} * * 2)$ )
>>> $\operatorname{im}[($ points[0]).astype(np.int), (points[1]).astype(np.int)] $=1$
im = ndimage.gaussian_filter(im, sigma $=1 /(4 . * n)$
mask $=$ im $>$ im.mean()

## - Analysis of connected components

Label connected components: ndimage.label:
>>> label_im, nb_labels = ndimage.label(mask)
>> nb labels \# how many regions?
$\ggg$
16
$\ggg>$
>> plt.imshow(label_im)
<matplotlib.image.AxesImage object at 0x...>


Compute size, mean_value, etc. of each region:
>>> sizes = ndimage.sum(mask, label_im, range(nb_labels + 1)) mean_vals = 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]
>> 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 = ndimage.find_objects(label_im==4) [0]
>> roi $=$ im[slice x , slice y ]
plt.imshow(roi)
<matplotlib.image.AxesImage object at 0x...>


Other spatial measures: ndimage. center_of_mass, ndimage.maximum_position, etc.
Can be used outside the limited scope of segmentation applications
Example: block mean:
>>> from scipy import misc
>> f = misc.face(gray=True)
>>> sx, sy $=\mathrm{f}$.shape
$\gg X, Y=n p . o g r i d[0: s x, 0: s y]$
>>> regions $=(\mathrm{sy} / / 6)$ * ( $\mathrm{X} / / 4)+(\mathrm{Y} / / 6)$ \# note that we use broadcasting
ock_mean = ndimage.mean(f, labels=regions, index=np.arange(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
> $\mathrm{X}, \mathrm{Y}=\mathrm{np}$.ogrid[0:sx, $0: \mathrm{sy}]$
$\mathrm{r}=\mathrm{np} . \operatorname{hypot}(\mathrm{X}-\mathrm{sx} / 2, \mathrm{Y}-\mathrm{sy} / 2$
$\ggg$ rbin $=(20 * r / r \cdot \max ())$ astype (np.int)
》> radial mean = ndimage. mean(f, labels=rbin, index=np.arange (1, rbin $\cdot \max ()+1)$ )

## - 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))
    (y - n)**2 <= n**2
```

    struct [mask] =
    struct[mask] \(=1\)
    >>
>> def granulometry(data, sizes=None)
$\mathbf{s}=\max ($ data. shape $)$
if sizes $==$ None:
sizes $=$ range ( $1, \mathrm{~s} / 2,2$ )
granulo = [ndimage.binary_opening(data,
structure=disk_structure(n)).sum() for $n$ in sizes]
eturn granulo
>>>


```
>> np.random. seed(1)
>> n=10
>> l=256
>> im = np.zeros((1, 1))
>> points = l*np.random.random((2,n**2))
>> im[(points[0]).astype(np.int),(points[1]).astype(np.int)] =1
>>> im = ndimage.gaussian_filter(im, sigma=1/(4.*n))
>>
>> mask = im > im.mean()
>> granulo = granulometry(mask, sizes=np.arange(2, 19, 4))
```



## See also

More on image-processing:

- The chapter on Scikit-image


## 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 mathe matical expression of the function that we are optimizing. Note that this expression can often be used for more matical expression of the function that

## Prerequisites

- Numpy, Scip
- 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 explainations


## Chapters content

- 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

Gradient-less methods
Global optimizers

- Practical guide to optimization with scipy
- Choosing a method

Making your optimizer faster

- Computing gradient
- Special case: non-linear least-squares
- Minimizing the norm of a vector function
- Minimizing

Optimization with constraint

- Box bounds
- General constraints


### 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


Optimizing convex functions is easy. Optimizing non-convex functions can be very hard.
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


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 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



### 13.2 A review of the different optimizers

### 3.2.1 Getting started: 1D optimization

Jse scipy .optimize .brent () to minimize 1D functions. It combines a bracketing strategy with a parabolic approximation

| Brent's method on a quadratic function: it converges in 3 iterations, as the quadratic approximation is then exact. |  |  |
| :---: | :---: | :---: |
| Brent's method on a non-convex function: note that the fact that the optimizer avoided the local minimum is a matter of luck. |  |  |

is a matter of luck
>> from scipy import optimize
$\gg \operatorname{def} f(x)$ :
$\therefore$ return -np. $\exp (-(\mathrm{x}-.7) * * 2)$
>> x_min = optimize.brent(f) \# It actually converges in 9 iterations!
>> $x$ _min
>> x min -
>> x_min - -7
$-2.1605 \ldots$. ${ }^{2}$-10
Brent's method can also be used for optimization constrained to an interval using scipy.optimize.fminbound ()
In scipy 0.11, scipy.optimize.minimize_scalar() gives a generic interface to 1D scalar minimization

### 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 he steepest descent.

Table 13.1: Fixed step gradient descent

| A well-conditionned quadratic function. |  |  |
| :---: | :---: | :---: |
| An ill-conditionned quadratic function. <br> The core problem of gradient-methods on ill-conditioned problems is that the gradient tends not to point in the direction of the minimum. |  | $\underset{y}{x}$ |

We can see that very anisotropic (ill-conditionned) 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.


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, i that it tends to oscillate across a valley, each time following the direction of the gradient, that makes it cross
he 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.

| An ill-conditionned non-quadratic function. |  |  |
| :---: | :---: | :---: |
| An ill-conditionned very non-quadratic function. |  |  |

Methods based on conjugate gradient are named with ' $c g^{\prime}$ ' in scipy. The simple conjugate gradient method to minimize a function is scipy.optimize.fmin_cg():
$\gg$ def $f(x)$ : The rosenbrock function
>> optimize.fmin_cg(f, [2, 2])
Optimization terminated successfully.
Current function value: 0.000000
Iterations: 13
Function evaluations: 120
Gradient evaluations: 30
array ([ $0.99998968,0.99997855]$ )
These methods need the gradient of the function. They can compute it, but will perform better if you can pas hem the gradient:
$\ggg$ def
fprime( x$):$
return $\mathrm{np} . \operatorname{array}((-2 * .5 *(1-\mathrm{x}[0])-4 * \mathrm{x}[0] *(\mathrm{x}[1]-\mathrm{x}[0] * * 2), 2 *(\mathrm{x}[1]-\mathrm{x}[0] * * 2)))$
>> optimize.fmin_cg(f, [2, 2], fprime=fprime)
ptimization terminated successfully.
Current function value: 0.000000
Iterations: 13
Function evaluations: 30
rray([ $0.99999199,0.99998336])$
Note that the function has only been evaluated 30 times, compared to 120 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-conditionned quadratic function: <br> Note that, as the quadratic approximation is exact, the Newton method is blazing fast |  |  |
| :---: | :---: | :---: |
| An ill-conditionned non-quadratic function: <br> Here we are optimizing a Gaussian, which is always below its quadratic approximation. As a result, the Newton method overshoots and leads to oscillations. | (1) |  |
| An ill-conditionned very non-quadratic function: | (a) |  |

In scipy, the Newton method for optimization is implemented in scipy.optimize.fmin_ncg() (cg here refers to that fact that an inner operation, the inversion of the Hessian, is performed by conjugate gradient). scipy.optimize.fmin_tnc() can be use for constraint problems, although it is less versatile:

```
>> def f(x): # The rosenbrock function
.. return.5*(1-x[0])**2+(x[1] - x[0]**2)**2
> def fprime(x)
    return np.array((-2*.5*(1-x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] - x[0]**2)))
>> optimize.fmin_ncg(f, [2, 2], fprime=fprime)
ptimization terminated successfully
    current function value: 0.000000
    Iterations: 9
    Function evaluations: 11
    Gradient evaluations: 51
    Hessian evaluations: 0
array([[ 1., 1.])
```

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 o the algorithm:
>> def hessian(x): \# Computed with sympy
return np.array(( $(1-4 * x[1]+12 * x[0] * * 2,-4 * x[0]),(-4 * x[0], 2)))$
optimize.fmin_ncg(f, [2, 2], fprime=fprime, fhess=hessian)
optimization terminated successfully.
Current function value: 0.000000
Iterations: 9
Gradient
Hessian evaluations: 9
$\operatorname{array}([1 ., ~ 1]$.
At very high-dimension, the inversion of the Hessian can be costly and unstable (large scale $>250$ ).
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.

| An ill-conditionned quadratic function: On a exactly quadratic function, BFGS is not as fast as Newton's method, but still very fast. |  |  |
| :---: | :---: | :---: |
| An ill-conditionned 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-conditionned very non-quadratic function: | (6) |  |

return $5 *(1-x[0]) * * 2+(x[1]-x[0] * * 2) * * 2$
>> def feturn (5 ${ }^{\text {rime }}$ )
return np.array ((-2*.5*(1-x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] -x[0]**2)))
optimize.fmin_bfgs(f, [2, 2], fprime=fprime)
ptimization terminated successfully.
Current functi
Iterations: 16
Function evaluations: 24
Gradient evaluations: 24
$\operatorname{array}([1.00000017,1.00000026]$ )
L-BFGS: Limited-memory BFGS Sits between BFGS and conjugate gradient: in very high dimensions (> 250
he Hessian matrix is too costly to compute and invert. L-BFGS keeps a low-rank version. In addition, the scipy version, scipy .optimize.fmin_1_bfgs b(), includes box bounds:
>> def $\mathrm{f}(\mathrm{x})$ : \# The rosenbrock function
return $5 *(1-x[0]) * * 2+(x[1]-x[0] * * 2) * * 2$
>> def fprime(x):
$\because \quad$ return np.array $((-2 * .5 *(1-\mathrm{x}[0])-4 * \mathrm{x}[0] *(\mathrm{x}[1]-\mathrm{x}[0] * * 2), 2 *(\mathrm{x}[1]-\mathrm{x}[0] * * 2)))$
>> optimize.fmin_l_bfgs_b(f, [2, 2], fprime=fprime)
(array ([ $1.00000005,1.00000009]), 1.4417677473011859 \mathrm{e}-15,\{\ldots\})$
If you do not specify the gradient to the L-BFGS solver, you need to add approx_grad=1

### 3.2.4 Gradient-less methods

A shooting method: the Powell algorithm
Almost a gradient approach

| An ill-conditionned quadratic function: Powell's method isn't too sensitive to local ill-conditionning in low dimensions |  |  |
| :---: | :---: | :---: |
| An ill-conditionned very non-quadratic function: | ctit |  |

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.


In scipy, scipy.optimize.fmin() implements the Nelder-Mead approach:
>>> def $f(x)$ : \# The rosenbrock function
return $5 *(1-x[0]) * * 2+(x[1]-x[0] * * 2) * * 2$
>> optimize.fmin(f, $[2,2]$ )
Optimization terminated successfully
Current function value: 0.000000
Iterations: 46
rray([ 0.99998568 , 0.99996682])

### 3.2.5 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:
$\gg$ def
$\mathrm{f}(\mathrm{x})$ : \# The rosenbrock function
return $.5 *(1-\mathrm{x}[0]) * * 2+(\mathrm{x}[1]-\mathrm{x}[0] * * 2) * * 2$
return $5 *(1-\mathrm{x}[0]) * * 2+(\mathrm{x}[1)-\mathrm{optimize} . \operatorname{brute}(\mathrm{f},((-1,2),(-1,2)))$
$\operatorname{array}([1.00001462,1.00001547])$

### 13.3 Practical guide to optimization with scipy

13.3.1 Choosing a method


## Without knowledge of the gradient

- In general, prefer BFGS (scipy.optimize.fmin_bfgs()) or L-BFGS (scipy.optimize.fmin_1_bfgs_b()), even if you have to approximate numerically gradients
- On well-conditioned problems, Powell (scipy.optimize.fmin_powell()) and Nelder-Mead (scipy.optimize.fmin()), both gradient-free methods, work well in high dimension, but they collapse for ill-conditioned problems.


## With knowledge of the gradien

- BFGS (scipy .optimize.fmin_bfgs()) or L-BFGS (scipy .optimize.fmin_l_bfgs_b()).
- Computational overhead of BFGS is larger than that L-BFGS, itself larger than that of 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 (scipy.optimize.fmin_ncg()).


## If you have noisy measurements

- Use Nelder-Mead (scipy.optimize.fmin()) or Powell (scipy.optimize.fmin_powell()).


### 13.3.2 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


### 13.3.3 Computing gradients

Computing gradients, and even more Hessians, is very tedious but worth the effort. Symbolic computation with Sympy may come in handy.
A 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:
>> optimize.check_grad(f, fprime, [2, 2])
2. $384185791015625 \mathrm{e}-07$

See also scipy .optimize .approx_fprime() to find your errors.
13.3.4 Synthetic exercices


## Exercice: A simple (?) quadratic function

Optimize the following function, using $\mathrm{K}[0]$ as a starting point
.random.seed(0)
$\mathrm{K}=\mathrm{np} \cdot \mathrm{random} \cdot \mathrm{normal}(\operatorname{size}=(100,100))$
$\operatorname{def} f(x)$
return np. sum $((\operatorname{np} \cdot \operatorname{dot}(\mathrm{K}, \mathrm{x}-1)) * * 2)+\mathrm{np} \cdot \operatorname{sum}(\mathrm{x} * * 2) * * 2$
Time your approach. Find the fastest approach. Why is BFGS not working well?

## Exercice: A locally flat minimum

Consider the function $\exp (-1 /(.1 * x * * 2+\mathrm{y} * * 2)$. This function admits a minimum in $(0,0)$. Starting from an initialization at $(1,1)$, try to get within $1 \mathrm{e}-8$ of this minimum point.



### 13.4 Special case: non-linear least-squares

### 3.4.1 Minimizing the norm of a vector function

east square problems, minimizing the norm of a vector function, have a specific structure that can be used in he Levenberg-Marquardt algorithm implemented in scipy.optimize. leastsq().
Lets try to minimize the norm of the following vectorial function:

```
>> def f(x):
\[
\text { return np. } \arctan (x)-n p . \arctan (n p . l i n s p a c e(0,1, \operatorname{len}(x)))
\]
```

>> x0 = np.zeros(10)
>>> optimize.leastsq(f, x0)
array ([0. $\quad, \quad 0.11111111,0.22222222,0.33333333,0.44444444$

$$
\begin{array}{llllll}
0 . & 0.11111111, & 0.22222222, & 0.33333333, & 0.44444444, \\
0.55555556, & 0.66666667, & 0.77777778, & 0.88888889, & 1 . & 1),
\end{array}
$$

This took 67 function evaluations (check it with 'full_output=l'). What if we compute the norm ourselves and se a good generic optimizer (BFGS):

```
>> def g(x):
return np.sum(f(x)**2)
Optimize.fmin_bfgs(g, x0)
ptimization terminated successfully
    Current function value: 0.000000
    Iterations: 11
    Function evaluations: 14
array([ -7.4\ldots-09, 1.1...e-01, 2.2\ldotse-01,
    3.3...e-01, 4.4..e-01, 5.5\ldotse-01,
    6.6\ldotse-01, 7.7..e-01, 8.8...e-01,
    1.0...e+00])
```

BFGS needs more function calls, and gives a less precise result.
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.

A If the function is linear, this is a linear-algebra problem, and should be solved with scipy. linalg. 1stsq().

### 13.4.2 Curve fitting



Least square problems occur often when fitting a noninear to data. While it is possible to construct our optimization problem ourselves, scipy provides a helper function for this purpose: scipy . optimize. curve_fit():
$\begin{array}{ll}\ggg \text { def } & f(t, \text { omega, phi): } \\ \text { return np. cos (omega } * t+\text { phi) }\end{array}$
>>> x = np.linspace(0, 3, 50)
>> $\mathrm{y}=\mathrm{f}(\mathrm{x}, 1.5,1)+.1 * \mathrm{np}$. random.normal(size=50)
>> optimize.curve_fit (f, x, y)
(array([ 1.51854577, 0.92665541]), array([[ 0.00037994, -0.00056796]
$[-0.00056796,0.00123978]])$

## Exercise

Do the same with omega $=3$. What is the difficulty?

### 13.5 Optimization with constraints

### 13.5.1 Box bounds

Box bounds correspond to limiting each of the individual parameters of the optimization. Note that som problems that are not originally written as box bounds can be rewritten as such via change of variables.


- scipy.optimize.fminbound () for 1D-optimization
- scipy.optimize.fmin_1_bfgs_b() a quasi-Newton method with bound constraints:

```
>> def f(x)
.. return np.sqrt ( \((x[0]-3) * * 2+(x[1]-2) * * 2)\)
>> optimize fmin l bfgs_b(f, np.array([0, 0]), approx_grad=1, bounds=((-1.5, 1.5), (-1.5, 1.5))
```

(array([ 1.5, 1.5]), 1.5811388300841898, \{...\})

### 135.2 General constraint

Equality and inequality constraints specified as functions: $\mathrm{f}(\mathrm{x})=0$ and $\mathrm{g}(\mathrm{x})<0$.

- scipy.optimize.fmin_slsqp() Sequential least square programming: equality and inequality con-

straints:

$$
\begin{aligned}
& f(x): ~
\end{aligned}
$$

$$
\text { return np. sqrt }((x[0]-3) * * 2+(x[1]-2) * * 2)
$$

>> def constraint(x):
return np.atleast_1d(1.5 - np.sum(np.abs(x)))
>> optimize.fmin_slsqp(f, np.array ([0, 0]), ieqcons=[constraint, ])
Optimization terminated successfully. (Exit mode 0)
current function value: 2.4748737350
terations: 5
unction evaluations: 20
radient evaluations: 5
rray ([ $1.25004696,0.24995304]$ )

- scipy.optimize.fmin_cobyla() Constraints optimization by linear approximation: inequality con straints only:
optimize.fmin_cobyla(f, np.array([0, 0]), cons=constraint
array([ 1.25009622, 0.24990378])

The above problem is known as the Lasso problem in statistics, and there exists 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.

## Interfacing with C

## uthor: Valentin Haenel

his chapter contains an introduction to the many different routes for making your native code (primarily $/ \mathrm{C}^{++}$) 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 almos certainly will want to consult the respective documentation for your selected technique.

## Chapters content

Introductio

- Python-C-Api

Ctype
swig
Summar

- 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 havin 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 wo use-cases that come up:

- Existing code in $\mathrm{C} / \mathrm{C}++$ 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 iven to a function, and later to construct the return type.

## Advantage

- 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.

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.
Since refernce 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;
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 [] $=$
${ }^{\text {stat }}$
cos_func", cos_func, MEIH_VARARGS, "evaluate the cosine"\},
\{NULL, NULL, 0, NULL
\};
(* module initialization */
PyMODINIT_FUNC
initcos_module(void)
(void) Py_InitModule("cos_module", CosMethods)

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 distutils supports compiling C-extensions from a setup.py, which is rather convenient:
from distutils.core import setup, Extension
\# define the extension module
cos_module $=$ Extension('cos_module', sources=['cos_module.c'])
\# run the setup
setup(ext_modules=[cos_module])

## This can be compiled:

\$ cd advanced/interfacing_with_c/python_c_api
\$ 1s
cos_module.c setup.py
python setup.py build_ext --inplace
running build_ext
bilding 'cos_module' extension
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an fcc -pthread -shared build/temp.linux-x86_64-2.7/cos_module.o-L/home/esc/anaconda/lib -lpython2.7-o/home/esc \$ 1 s
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:
In [1]: import cos_module
In [2]: cos_module?
ype: module

$\begin{array}{ll}\text { File: } & \text { /home/esc/git } \\ \text { Docstring: } & \text { <no docstring }\end{array}$
In [3]: dir(cos_module)
Sut [3]: ['__doc__', '__file__', '__name__', '__package__', 'cos_func']
In [4]: cos_module.cos_func (1.0)
aut [4]: 0.5403023058681398
In [5]: cos_module. cos_func (0.0)
out [5] : 1.0
In [6]: cos_module.cos_func(3.14159265359)
Out [7]: - -1.0
Now let's see how robust this is:
In [10]: cos_module.cos_func('foo')
ypeError Traceback (most recent call last)
ipython-input-10-11bee483665d> in <module>()
----> 1 cos_module.cos_func('foo')
ypeError: 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: :ref:'advanced_numpy_-
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 *in_array;
    PyObject *out_array
    NpyIter *in_iter;
    NpyIter *out_iter;
    Func *in_iternext;
    NpyIter_IterNextFunc *out_iternext
    /* parse single numpy array argument *
    if (!PyArg_ParseTuple(args, "0!", &PyArray_Type, &in_array))
        return NULL;
    /* construct the output array, like the input array */
    out_array = PyArray_NewLikeArray(in_array, NPY_ANYORDER, NULL, 0)
    if (out_array == NULL)
        return NULL;
    in_iter = NpyIter_New(in_array, NPY_ITER_READONLY, NPY_KEEPORDER,
    NPY_NO_CASTING, NULL);
    if (in_iter == NULL
        goto fail;
    out_iter = NpyIter_New((PyArrayObject *)out_array, NPY_ITER_READWRITE,
    NPY_KEEPORDER, NPY_NO_CASTING, NULL);
    if (out_iter == NULL) {
        NpyIter_Deallocate(in_iter)
        goto fail;
    }
    in_iternext = NpyIter_GetIterNext(in_iter, NULL);
    out_iternext = NpyIter_GetIterNext(out_iter, NULL)
    if (in_iternext == NULL || out_iternext == NULL) {
        NpyIter_Deallocate(in_iter)
        eallocate(out_iter)
        goto fail;
    double ** in_dataptr = (double **) NpyIter_GetDataPtrArray(in_iter).
    double ** out_dataptr = (double **) NpyIter_GetDataPtrArray(out_iter);
    ** iterate over the arrays */
    do {
        **out_dataptr = cos(**in_dataptr)
    } while(in_iternext(in_iter) && out_iternext(out_iter))
    /* clean up and return the result */
    NpyIter_Deallocate(in_iter);
    NpyIter_Deallocate(in_iter);
    Py_INCREF(out_array);
    return out_array;
    /* in case bad things happen *
    fail:
        Py_XDECREF(out_array);
        return NULL;
```

```
* define functions in module */
static PyMethodDef CosMethods[] =
    {"cos_func_np", cos_func_np, METH_VARARGS,
    on a numpy array"}
    NULL, NULL, 0, NULL}
};
ization */
PyMODINIT_FUNC
nitcos_module_np(void)
(void) Py_InitModule("cos_module_nP", CosMethods)
    \* IMPORTANT: this must be called */
    import_array();
\}
```

To compile this we can use distutils again. However we need to be sure to include the Numpy headers by using func:numpy.get_include.
from distutils.core import setup, Extension
import numpy
\# define the extension module
cos_module_np = Extension('cos_module_np', sources=['cos_module_np.c'],
nclude_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 scrip
import cos_module_np
import numpy as np
import pylab
$x=n p$.arange ( $0,2 *$ np. pi, 0.1)
$y=$ cos_module_np.cos_func_np(x)
pylab.plot(x, y
pylab.show()
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. """
port ctypes
from ctypes.util import find_library
\# find and load the library
libm = ctypes.cdll.LoadLibrary(find_library('m')
\# set the argument type
libm.cos.argtypes $=$ [ctypes.c_double]
\# set the return type
ibm cos. restype $=$ ctypes.c_doubl
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 [1]: import cos_module
In [2]: cos_module?
ype: module
String Form:<module 'cos_module' from 'cos_module.py'>
File: /home/esc/git-working/scipy-lecture-notes/advanced/interfacing_with_c/ctypes/cos_module.py
Docstring: <no docstring>
In [3]: dir (cos_module)
Out [3]:
['__builtins__',
'--_doc_-
    ,--doc--
    ----name_- ',
    '-_package_-',
    'cos_func'
    'ctypes',
    'ftypes'library'
    'libm']
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
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 [7]: cos_module.cos_func('foo')

ArgumentError $\quad$ Traceback (most recent call last)
ipython-input-7-11be483665d> in <module>()
----> 1 cos module.cos_func('foo')
/home/esc/git-working/scipy-lecture-notes/advanced/interfacing_with_c/ctypes/cos_module.py in cos_func(arg) 12 def cos_func (arg)
13 ' $'$ Wrapper for cos from math.h ' $'$
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 ( $\mathrm{i}=0$; $\mathrm{i}<$ size $; \mathrm{i}++$ )
out_array[i] $=$ cos(in_array[i]);
\}
And since the library is pure C, we can't use distutils to compile it, but must use a combination of make and gcc :

```
m.PHONY : clean
ibcos_doubles.so : cos_doubles.o
    gcc -shared -wl,-soname,libcos_doubles.so -o libcos_doubles.so cos_doubles.o
cos_doubles.0 : 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:
1s
cos_doubles.c cos_doubles.h cos_doubles.py makefile test_cos_doubles.p \$ make
cc -c -fPIC cos_doubles.c -o cos_doubles.o
cc -shared -W1,--soname,libcos_doubles.so -o libcos_doubles.so cos_doubles.o
os_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
mport 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
ibcd.cos_doubles.restype $=$ None
ibcd.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 re quires this kind of buffer
- Also note that the output array must be preallocated, for example with numpy $\cdot$ zeros () and the function will write into it's buffer
- Although the original signature of the cos_doubles function is ARRAY, ARRAY, int the fina cos_doubles func takes only two Numpy arrays as arguments.

And, as before, we convince ourselves that it worked
import numpy as np
import pylab
mport pylab
import cos_doubles
np. arange (0, 2 * np.pi, 0.1)
$y=n p . e m p t y \_$like (x)
cos_doubles.cos_doubles_func (x, y
pylab.plot(x, y)
pylab.show()


### 14.4 SWIG

WWIG, 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 importan hing 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 huma eadable and the multiple levels of indirection which are a result of the wrapping process, may be a bit tricky o understand
The autogenerated C code uses the Python-C-Api.

## dvantages

- 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 cand consists of the source file cos_module.c:
\# include <math.h>
double cos_func(double arg) t
return $\cos (\mathrm{arg})$;
\}
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. */
module cos_module
$\%$
(ald
\#define SWIG_FILE_WITH_INIT
/* Includes the header in the wrapper code */
\#
\%

* Parse the header file to generate wrappers *
include "cos_module.h"
s you can see, not too much code is needed here. For this simple example it is enough to simply includ he header file in the interface file, to expose the function to Python. However, SWIG does allow for more fine rained 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, distutils knows how to handle SWIG interface files, so that our setup.py is simply:

## from distutils.core import setup, Extension

setup(ext_modules=[Extension("_cos_module"
sources=["cos_module.c", "cos_module.i"])]

## cd advanced/interfacing_with_c/swig

1s
cos_module.c cos_module.h cos_module.i setup.py
python setup.py build_ext --inplace
running build_ext
bilding '_cos_module' extension
wigging cos_module.i to cos module wrap.c
wig -python -o cos_module_wrap.c cos_module.i
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an gcc -pthread -fno-strict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an 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/

1s
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:


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 [7]: cos_module.cos_func('foo')

TypeError
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>
* 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 $i$;

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

\%modu
/* the resulting C file sho
\#define SWIG_FILE_WITH_INI
/* 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) \{ 1* calls the original funcion, 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 distutils to wrap this:
from distutils.core import setup, Extension
import numpy
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.
\$ 1s
cos_doubles.c cos_doubles.h cos_doubles.i numpy.i setup.py test_cos_doubles.py
python setup.py build_ext -i
unning build_ext
building '_cos_doubles' extension
igging cos_doubles.i to cos_doubles_wrap.c
wig -python -o cos_doubles_wrap.c cos_doubles.i
os_doubles.1:24: Warning(490) : Fragment 'NumPy_Backward_Compatibility' not found.
os_doubles.i:24: Warning(490): Fragment 'NumPy_Backward_Compatibility' not found.
creating build
reating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an gcc -pthread -fno-strict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an In file included from /home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/ndarraytypes.h:17: from/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/ndarrayobject.h:1 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: \#w gcc -pthread -shared build/temp.1inux-x86_64-2.7/cos_doubles.o build/temp.1inux-x86_64-2.7/cos_doubles_wrap.o 1s
cos doubles.h cos doubles.py cos doubles_wrap.c setup.py


## And, as before, we convince ourselves that it worked

import numpy as np
import pylab
import cos_doubles
$x=n p . \operatorname{arange}(0,2 * n p . p i, 0.1)$
$\mathrm{y}=\mathrm{np}$. empty_like $(\mathrm{x})$
cos_doubles.cos_doubles_func(x, y)
pylab.plot(x, y
pylab.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 The Cython language is a superset of Python, which comes with additional constructs that allow you call C 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 ypes 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.
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 th 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. """
def extern from "math.h"
double cos(double arg)
def cos_func(arg):
return $\cos (a r g)$
Note the additional keywords such as cdef and extern. Also the cos_func is then pure Python.
Again we can use the standard distutils module, but this time we need some additional pieces from the Cython.Distutils:
from distutils.core import setup, Extension
from distutils.core import setup, Exten
from Cython. Distutils import build_ext
setup(
ext_modules=[Extension("cos_module", ["cos_module.pyx"])]

## Compiling this

## cd advanced/interfacing_with_c/cython

\$ 1 s
python setup.py build_ext --inplace
running build_ext
cythoning cos_module.pyx to cos_module.c
building 'cos_module' extension
reating build
centinux-x86_64-2.7
cc -pthread -sharst buict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an cc -pthread -shared build/temp.linux-x86_64-2.7/cos_module.0-L/home/esc/anaconda/lib -lpython2.7-o/home/esc
build/ cos_module.c cos_module.pyx cos_module.so* setup.py
And running it:
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/scipy-lecture-notes/advanced/interfacing_with_c/cython/cos_module.so
Docstring: <no docstring>
In [3]: $\operatorname{dir}($ cos_module)
Out [3]:
['_-_builtins__',
'--doc_-',
--_file_-',
'_-package__',
'--test_-',
'cos_func']
In [4]: cos_module.cos_func(1.0)
In [4]: cos_module.cos_fun
In [5]: cos_module.cos_func ( 0.0 )
Dut [5]: 1.0
In [6]: cos_module.cos_func (3.14159265359)
Out [6] : -1.0
And, testing a little for robustness, we can see that we get good error messages:


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 \(\cos (a r g)\)
```

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 you Cython code. I.e. like specifying that variable $i$ is of type int, you can specify that variable a is of typ umpy. 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 o your Cython wrapped C functions, there is a section about this in the Cython wiki.
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>
/* 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;
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()
def 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
<double*> np PyArray DATA(out array),
in array. shape [0])
And can be compiled using distutils:
from distutils.core import setup, Extension
mport numpy
from Cython.Distutils import build_ext

```
setup(
    cmdclass={'build_ext': build_ext},
    ext_modules=[Extension("cos_doubles",
        sources=["_cos_doubles.pyx", "cos_doubles.c"],
        include_dirs=[numpy.get_include()])],
```

- As with the previous compiled Numpy examples, we need the include_dirs option.

1s
cos_doubles.c cos_doubles.h _cos_doubles.pyx setup.py test_cos_doubles.py
\$ python setup.py build_ext -i
funning build_ext
ythoning _cos_doubles.pyx to _cos_doubles.c
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -g -02 -DNDEBUG -g -fwrapv -03 -Wall -Wstrict-prototypes -fPIC -I/home/esc/an In file included from/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/ndarraytypes.h:17 from/home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/ndarrayobject.h:1 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: \#wa home/esc/anaconda/lib/python2.7/site-packages/numpy/core/include/numpy/__ufunc_api.h:236: warning: '_import_un cc -pthread -shared build/temp.linux-x86 64-2.7/ cos dopv -03 -Wall -Wp linux-x86 64-2.7/cos doubles.0-L/h $\begin{array}{ll}\$ 1 s \\ \text { build } & \text { cos_doubles.c cos doubles.c cos doubles.h cos_doubles.pyx cos_doubles.so* setup.py test cos dou }\end{array}$
nd, as before, we convince ourselves that it worked:
import numpy as np
mport pylab
import cos_doubles
= np.arange(0, 2 * np.pi, 0.1)
$\mathrm{y}=\mathrm{np}$. empty_like(x)
cos_doubles.cos_doubles_func (x, y
pylab.plot(x, y)
pylab.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 potentia solution. Here are some ideas:
(a) Numerical overflow.
(b) Input and output arrays that have different lengths
(c) Multidimensional array
(d) Empty array
(e) 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 solu tion, 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. Doe 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 mak ing 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 mak ing 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 ypemaps? 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

. 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 mak ing it more like the Numpy-C-API example.

## Packages and applications

## Author: Gaël Varoquaux

## Requirements

- Standard scientific Python environment (numpy, scipy, matplotlib)
- Panda
- Statsmode
- Seaborn

To install Python and these dependencies, we recommend that you download Anaconda Python or Enthought Canopy, 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.

## 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 panda data-frame
- Hypothesis testing: comparing two groups

Student's t -test: the simplest statistical tes

- Paired tests: repeated measurements on the same indivuals
- 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 matrice
- lmplot: plotting a univariate regression
- Testing for interactions

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 question 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 set of different attributes or features. The data can than be seen as a 2D table, or matrix, with column 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 panda data-frame

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

## Separato

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:
>> data = pandas.read_csv('examples/brain_size.csv', sep=';', na_values=".")
>> data

| Unnamed: | 0 | Gender | FSIQ | VIQ | PIQ | Weight | Height |
| ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | MRI_Count

## Missing value

The weight of the second individual is missing in the CSV file. If we don't specify the missing value ( $\mathrm{NA}=\mathrm{not}$ available) marker, we will not be able to do statistical analysis.
reating 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:

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

$\rightarrow \gg \cos t=n p \cdot \cos (t)$

We can expose them as a pandas. DataFrame:

$$
\begin{aligned}
& \text { >> pandas. DataFrame(\{'t': t, 'sin': sin_t, 'cos': cos_t }\} \text { ) } \\
& \begin{array}{lrrr} 
& \text { cos } & \text { sin } & \text { t } \\
0 & 0.960170 & 0.279415 & -6.000000
\end{array} \\
& \begin{array}{lll}
0.960170 & 0.279415 & -6.000000 \\
0.609977 & 0.792419 & -5.368421
\end{array} \\
& \begin{array}{llll}
0.024451 & 0.999701 & -4.736842
\end{array} \\
& \text {-0.570509 } 0.821291-4.105263 \\
& -0.945363 \quad 0.326021-3.473684 \\
& \text {-0.955488-0.295030 }-2.842105 \\
& \begin{array}{lll}
-0.056979 & -0.80225 & -1.878947
\end{array} \\
& \text { 3 } 0.583822-0.811882-0.947368
\end{aligned}
$$

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([u'Unnamed: $0^{\prime}$, u'Gender', u'FSIQ', u'VIQ', u'PIQ', u'Weight', u'Height', u'MRI_Count'], dtype='object')
>> print(data['Gender']) \# Columns can be addressed by name
Female
Male
Male
Male
Female
>> \# Simpler selector
data[data['Gender'] == 'Female']['viQ'].mean()
109.45

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']
$\therefore \quad$ print ((gender, value.mean()))
'Female', 109.45
groupby_gender is a powerful object that exposes many operations on the resulting group of dataframes

$\begin{array}{lllllllll}\text { emale } & 19.65 & 111.9 & 109.45 & 110.45 & 137.200000 & 65.765000 & 862654 . \\ \text { Male } & 21.35 & 115.0 & 115.25 & 111.60 & 166.444444 & 71.431579 & 954855 .\end{array}$
Use tab-completion on groupby_gender to find more. Other common grouping functions are median, count 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 VQ 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?
groupby_gender . boxplot is used for the plots above (see this example)

Plotting data
Pandas comes with some plotting tools (pandas . tools . plotting, using matplotlib behind the scene) to display statistics of the data in dataframes:

## Scatter matrices

>>> from pandas.tools import plotting
>>> plotting.scatter_matrix(data[['Weight', 'Height', 'MRI_Count']]

>> plotting.scatter_matrix(data[['PIQ', 'VIQ', 'FSIQ']])

## 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:
>>> from scipy import stats

## See also

Scipy is a vast library. For a quick summary to the whole library, see the scipy chapter.

### 5.2.1 Student's t-test: the simplest statistical test

1 -sample $t$-test: testing the value of a population mean

scipy. stats.ttest 1 samp () tests if the population mean of data is likely to be equal to a given value (tech nically if observations are drawn from a Gaussian distributions of given population mean). It returns the T statistic, and the p -value (see the function's help):
>>> stats.ttest_1samp(data['VIQ'], 0)
With a p-value of $10^{\wedge}-28$ we can claim that the population mean for the IQ (VIQ measure) is not 0 .

2-sample $\mathbf{t}$-test: testing for difference across populations
We have seen above that the mean VIQ in the male and female populations were different. To test if this is significant, we do a 2 -sample $t$-test with scipy. stats. ttest_ind ():
>>> female_viq = data[data['Gender'] == 'Female']['VIQ']
>> male_viq = data[data['Gender'] == 'Male']['VIQ']
>>> stats.ttest_ind(female_viq, male_viq)
$(\ldots-0.77261617232 \ldots, 0.4445287677858 \ldots$.
15.2.2 Paired tests: repeated measurements on the same indivuals


PIQ, VIQ, and FSIQ give 3 measures of IQ. Let us test if FISQ and PIQ are significantly different. We can use a 2 sample test
>> stats.ttest_ind(data['FSIQ'], data['PIQ'])
(...0.46563759638..., 0.64277250...)

The problem with this approach is that it forgets that there are links between observations: FSIQ and PIQ are measured on the same individuals. Thus the variance due to inter-subject variability is confounding, and can be removed, using a "paired test", or "repeated measures test":
>>> stats.ttest_rel(data['FSIQ'], data['PIQ'])
(...1.784201940.... $0.082172638183 \ldots$ )


This is equivalent to a 1 -sample test on the difference:
S>> stats.ttest_1samp(data['FSIQ'] - data['PIQ'], 0)
(...1.784201940..., 0.082172638...)

1-tests assume Gaussian errors. We can use a Wilcoxon signed-rank test, that relaxes this assumption:
>> stats.wilcoxon(data['FSIQ'], data['PIQ'])
( $274.5,0.106594927 \ldots$ )
The corresponding test in the non paired case is the Mann-Whitney U test, scipy. stats. mannwhitneyu()

## Exercice

- 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
he hypothesis that y is a linear function of x . In other terms:

$$
y=x * \operatorname{coef}+\text { intercept }+e
$$

where e is observation noise. We will use the statmodels 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)
>> np. random.seed (1)
>>> np.random. Seed(1)
>> y = -5 + 3*x + 4* np.random. 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
>> from statsmodels.formula.api import ols
>>> model = ols("y ~ x", data).fit()
We can inspect the various statistics derived from the fit


## Terminology:

Statsmodel uses a statistical terminology: the $y$ variable in statsmodel 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 relevent attribute.

## Categorical variables: comparing groups or multiple categorie

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
$\ggg$ model $=01 \mathrm{~s}($ "VIQ $\sim$ Gender +1 ", data).fit()
$\ggg>\operatorname{print}($ model .summary())

| Dep. Variable: | viQ | R-squared: | 0.015 |
| :---: | :---: | :---: | :---: |
| Model: | OLS | Adj. R-squared: | -0.010 |
| Method: | Least Squares | F-statistic: | 0.5969 |


| Date: | $\ldots$ |  | Prob (F- | -statistic): |  | 0.445 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Time: |  |  | Log-Like | elihood: |  | -182.42 |
| No. Observations |  | 40 | AIC: |  |  | 368.8 |
| Df Residuals: |  | 38 | BIC: |  |  | 372.2 |
| Df Model: |  | 1 |  |  |  |  |
|  | coef | std err | t | $p>\|t\|$ | [95.0\% Conf | Int.] |
| 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-W | Watson: |  | 1.709 |
| Prob(Omnibus) : |  | 0.000 | Jarque-B | Bera (JB) : |  | 3.703 |
| Skew: |  | 0.010 | Prob(JB) |  |  | 0.157 |
| Kurtosis: |  | 1.510 | Cond. No |  |  | 2.62 |

## Tips on specifying model

Forcing categorical: the 'Gender' is automatical detected as a categorical variable, and thus each of it different values are treated as different entities.
An integer column can be forced to be treated as categorical usin
>>> 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
By default, statsmodel treats a categorical variable with K possible values as $\mathrm{K}-1$ '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 variable (http://statsmodels.sourceforge.net/devel/contrasts.html)

## Link to $t$-tests between different FSIQ and PIQ

To compare different type 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'\})
$\ggg$ data_long = pandas.concat((data_fisq, data_piq))
print(data_long)
$\begin{array}{rl}\text { iq } & \text { type } \\ 0 & 133 \\ \text { fsiq }\end{array}$
$1 \begin{array}{lll}140 & \text { fsiq }\end{array}$
2139 fsiq
$\begin{array}{lll}31 & 137 & \text { piq }\end{array}$
$\begin{array}{ccc}32 & 110 & \text { piq } \\ 33 & 86 & \text { piq }\end{array}$
>> model = ols("iq ~ type", data_long).fit()
>> print(model.summary())
OLS Regression Results

|  | coef | std err | t | $p>\|t\|$ | [95.0\% Conf | Int.] |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 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:
>>> stats.ttest_ind(data['FSIQ'], data['PIQ'])
..0.46563759638..., 0.64277250...)
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 $(\mathrm{x}, \mathrm{y}, \mathrm{z})$ points.

Example: the iris data (examples/iris.csv)
Sepal and petal size tend to be related: bigger flowers are bigger! But is there in addition a systematic effect of species?
blue: setosa, green: versicolor, red: virginica


| ```>>> data = pandas.read_csv('examples/iris.csv') >>> model = ols('sepal_width ~ name + petal_length', data).fit() >>> print(model.summary())``` |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| 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) : |  |  | $1.58 \mathrm{e}-20$ |  |
| Time: |  |  | Log-Likelihood: |  |  | -38.185 |  |
| No. Observations: |  | 150 | AIC: |  |  |  | . 37 |
| Df Residuals: |  | 146 |  |  |  | 96.41 |  |
| Df Model: |  | 3 |  |  |  |  |  |
|  | coef | std err |  | t | P> $\|t\|$ | [95.0\% Conf | Int.] |
| 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 |  | que-Bera (J) |  | 2.885 |  |
| Skew: |  | -0.082 | Prob (JB) : |  |  | 0.236 |  |
| Kurtosis: |  | 3.659 | Cond. No. |  |  | 54.0 |  |

### 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, afte emoving the effect of sepal width. This can be formulated as testing the difference between the coefficient ociated 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]解 name [T virginica] " with an F-test:

<F test: $F=\operatorname{array}([[3.24533535]]), p=[[0.07369059]]$, df_denom=146, df_num=1>
Is this difference significant?

## Exercice

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)
The full code loading and plotting of the wages data is found in corresponding example.

|  |  |  |  |  |  |  |  |  |
| ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: |
| $\ggg$ | print data |  |  |  |  |  |  |  |
|  | EDUCATION | SOUTH | SEX | EXPERIENCE | UNION | WAGE | AGE | RACE |
| 0 | 8 | 0 | 1 | 21 | 0 | 0.707550 | 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 |

### 5.4.1 Pairplot: scatter matrices

We can easily have an intuition on the interactions between continuous variables using seaborn. pairplot () to display a scatter matrix:

```
>>
    >> seaborn.pairplot(data, vars=['WAGE', 'AGE', 'EDUCATION'],
    kind='reg')
```



Categorical variables can be plotted as the

## Look and feel and matplotlib settings

Seaborn changes the default of matplotlib figures to achieve a more "modern", "excel-like" look. It doe that upon import. You can reset the default using:
>>> from matplotlib import pyplot as plt
>>> plt.rcdefaults()
To switch back to seaborn settings, or understand better styling in seaborn, see the relevent section of the seaborn documentation.
15.4.2 Implot: plotting a univariate regression

A regression capturing the relation between one variable and another, eg wage and eduction, can be plotted using seaborn. 1mplot ():
>>> seaborn. 1mplot ( $\mathrm{y}=$ 'WAGE', $\mathrm{x}=$ 'EDUCATION', data=data)


## Robust regression

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 sentive 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


males than females?

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 to population. This is done via an "interaction"

|  | coef | std err | t | $p>\|t\|$ | [95.0\% Conf | f. Int.] |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 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-value 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 simple model fits matters!
Conditionning (adding factors that can explain all or part of the variation) is important modeling aspect that changes the interpretation

## 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 system 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 http://www.sympy.org/

## Chapters contents

- First Steps with SymPy
- Using SymPy as a calculator
- Exercises
- Algebraic manipulation
- Expand
- Simplify
- Calculus
- Limits
- Differentiation
- Series expansion
- Integration
- Exercises
- Equation solving
- Exercises
- Linear Algebra
- 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 $1 / 2$, $\operatorname{Rational}(5,2) 5 / 2$ and so on:
>> from sympy import
>> $\mathrm{a}=$ Rational $(1,2)$
-
$1 / 2$
>>> a*2

SymPy uses mpmath in the background, which makes it possible to perform computations using arbitraryprecision arithmetic. That way, some special constants, like e, pi, oo (Infinity), are treated as symbols and can be evaluated with arbitrary precision:
>> pi**2
$\mathrm{pi}^{2}$
>> pi.evalf()
3. 14159265358979
>> ( $\mathrm{pi}+\exp (1)$ ).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:
>>> 00 > 99999
True
>> 00 + 1
-0

### 16.1.2 Exercises

1. Calculate $\sqrt{2}$ with 100 decimals
2. Calculate $1 / 2+1 / 3$ in rational arithmetic.

### 16.1.3 Symbols

In contrast to other Computer Algebra Systems, in SymPy you have to declare symbolic variables explicitly:
>> from sympy import *
>> $\mathrm{x}=$ Symbol(' x '
Then you can manipulate them

$$
\begin{aligned}
& \ggg x+y+x-y \\
& 2 * x \\
& \ggg(x+y) * * 2
\end{aligned}
$$

$(x+y){ }^{2}$
Symbols can now be manipulated using some of python operators: +, -, *, ** (arithmetic), \&, |, ~ , >>, << (boolean).

## Printing

Here we use the following setting for printing
>>> sympy.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 requently used: expand and simplify.

### 16.2.1 Expand

Use this to expand an algebraic expression. It will try to denest powers and multiplications
$\begin{array}{ll}\gg & \operatorname{expand}((\mathrm{x}+\mathrm{y}) * * 3) \\ 3 & 2 \\ 3 & 2\end{array}$
x $+3 * x * y+3 * x * y+y$
> ${ }_{3} \quad 3 * \mathrm{x} * \mathrm{y} * * 2+3 * \mathrm{y} * \mathrm{x} * * 2+\mathrm{x} * * 3+\mathrm{y} * * 3$
$3{ }^{2}{ }^{2}{ }^{2}$

Further options can be given in form on keywords
$\ggg \operatorname{expand}(\mathrm{x}+\mathrm{y}$, complex=True $)$
$\mathrm{re}(\mathrm{x})+\mathrm{re}(\mathrm{y})+\mathrm{I} * \mathrm{im}(\mathrm{x})+\mathrm{I} * \mathrm{im}(\mathrm{y})$
$\mathrm{re}(\mathrm{x})+\mathrm{re}(\mathrm{y})+\mathrm{I} * \mathrm{im}(\mathrm{x})+\mathrm{I} * \mathrm{im}(\mathrm{y})$
$\ggg \mathrm{I} * \mathrm{im}(\mathrm{x})+\mathrm{I} * \mathrm{im}(\mathrm{y})+\mathrm{re}(\mathrm{x})+\mathrm{re}(\mathrm{y})$
$r e(x)+r e(y)+I * i m(x)+I * i m(y)$
>> expand (cos(x+y), trig=True)
$-\sin (\mathrm{x}) * \sin (\mathrm{y})+\cos (\mathrm{x}) * \cos (\mathrm{y})$
>> $\cos (\mathrm{x}) * \cos (\mathrm{y})-\sin (\mathrm{x}) * \sin (\mathrm{y})$
$-\sin (\mathrm{x}) * \sin (\mathrm{y})+\cos (\mathrm{x}) * \cos (\mathrm{y})$

### 16.2.2 Simplify

Use simplify if you would like to transform an expression into a simpler form:

```
>> 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)}\mp@subsup{}{}{6
    1. Calculate the expanded form of (x+y).
```


### 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 $x \rightarrow 0$, you would issue $\operatorname{limit}(f, \mathrm{x}, 0)$
>> $\operatorname{limit}(\sin (\mathrm{x}) / \mathrm{x}, \mathrm{x}, 0)$
you can also calculate the limit at infinity:
>>> $\operatorname{limit}(x, x, \circ \circ)$
○○
>> $\operatorname{limit}(1 / \mathrm{x}, \mathrm{x}, \circ 0)$
>>>
>>> $\operatorname{limit}(x * * x, x, 0)$

### 16.3.2 Differentiation

You can differentiate any SymPy expression using diff (func, var). Examples
$\ggg \operatorname{diff}(\sin (\mathrm{x}), \mathrm{x})$
$\cos (\mathrm{x})$
>> diff $(\sin (2 * x), x)$
$2 * \cos (2 * x)$
$\gg \operatorname{diff}(\tan (\mathrm{x}), \mathrm{x})$
$\tan (x)+$
You can check, that it is correct by:

$$
\begin{aligned}
& \ggg \operatorname{limit}^{2}((\tan (x+y)-\tan (x)) / y, y, 0) \\
& 2
\end{aligned}
$$

Higher derivatives can be calculated using the diff (func, var, n ) method:
>> $\operatorname{diff}(\sin (2 * x), x, 1)$
$2 * \cos (2 * x)$
>> $\operatorname{diff}(\sin (2 * x), x, 2)$
$-4 * \sin (2 * x)$
>> $\operatorname{diff}(\sin (2 * x), x, 3)$
$-8 * \cos (2 * x)$

### 16.3.3 Series expansion

SymPy also knows how to compute the Taylor series of an expression at a point. Use series (expr, var):

```
>> series(cos(x), x)
    2 4
| x / 6\
---+--- + 0\x/
```

|  |
| :---: |
|  |  |
|  |  |
|  |  |

## Exercise

1. Calculate $\lim _{x \rightarrow 0} \sin (x) / x$
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 powerful extended Risch-Norman algorithm and some heuristics and pattern matching. You can integrate elementary functions:

```
>> integrate(6*x**5, x)
```

6
$\ggg$ integrate $(\sin (x), x)$
$-\cos (\mathrm{x})$
>> integrate $(\log (x), x)$
$x * \log (x)-x$
>> integrate (2*x + $\sinh (x), x)$
${ }^{2}+$
${ }^{2}+\cosh (\mathrm{x})$
Also special functions are handled easily:


## It is possible to compute definite integral:

>>> integrate $(\mathrm{x} * * 3,(\mathrm{x},-1,1))$
0
>>> integrate $(\sin (\mathrm{x}),(\mathrm{x}, 0, \mathrm{pi} / 2))$
1
>>> integrate( $(\cos (\mathrm{x}),(\mathrm{x},-\mathrm{pi} / 2, \mathrm{pi} / 2)$
>>> integrate (cos(x), (x, -pi/2, pi/2)

Also improper integrals are supported as well:

$$
\begin{aligned}
& \ggg \text { integrate }(\exp (-\mathrm{x}),(\mathrm{x}, 0, \infty 0)) \\
& 1 \ggg \text { integrate }(\exp (-\mathrm{x} * * 2),(\mathrm{x},-\infty, \circ 0)) \\
& \text { (--- }
\end{aligned}
$$

### 16.3.5 Exercises

### 16.4 Equation solving

SymPy is able to solve algebraic equations, in one and several variables

In [7]: solve ( $\mathrm{x} * * 4-1$, x$)$
out [7]: [I, 1, -1, -I]
As you can see it takes as first argument an expression that is supposed to be equaled to 0 . It is able to solve a arge part of polynomial equations, and is also capable of solving multiple equations with respect to multiple variables giving a tuple as second argument:
In [8]: solve([x+5*y-2,-3*x+6*y-15],[x,y])
Out [8]: $\{y: 1, x:-3\}$
It also has (limited) support for trascendental equations:
In [9]: $\operatorname{solve}(\exp (x)+1, x)$
Out [9]: [pi*I]
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
In [10]: $f=x * * 4-3 * x * * 2+1$
In [11]: factor (f)
Out [11]: $(1+\mathrm{x}-\mathrm{x} * * 2) *(1-\mathrm{x}-\mathrm{x} * * 2$
In [12]: factor(f, modulus=5)
Dut [12]: $(2+x) * * 2 *(2-x) * * 2$
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:
In [13]: satisfiable( $\mathrm{x} \& \mathrm{y}$ )
Out [13]: \{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:
In [14]: satisfiable( $\mathrm{x} \& \sim \mathrm{x}$ )
Out [14]: False

### 16.4.1 Exercises

1. Solve the system of equations $x+y=2,2 \cdot x+y=0$
2. Are there boolean values x , y that make $(\sim \mathrm{x} \mid \mathrm{y})$ \& ( $\sim \mathrm{y} \mid \mathrm{x}$ ) true?

### 16.5 Linear Algebra

### 16.5.1 Matrice

Matrices are created as instances from the Matrix class:

```
>> from sympy import Matrix
    >> Matrix ([[1, 0], \([0,1]]\) )
\(\left[\begin{array}{ll}1 & 0\end{array}\right]\)
\(\left[\begin{array}{cc}{\left[\begin{array}{ll}{[ } & 1\end{array}\right]}\end{array}\right.\)
```

unlike a NumPy array, you can also put Symbols in it

```
>> x = Symbol('x')
>>> y = Symbol('y')
>> A = Matrix([[1, x],[y,1]])
```


## Scikit-image: image processing

### 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 = symbols('f g', cls=Function)

and $g$ are now undefined functions. We can call $f(x)$, and it will represent an unknown function:
> $\mathrm{f}(\mathrm{x})$
$f(x)$
$\gg f(x) \cdot \operatorname{diff}(x, x)+f(x)$
$f(x)+{ }^{d^{2}}{ }^{--(f(x)}$
$\mathrm{dx}^{2}$
$\ggg$ dsolve( $f(x) \cdot \operatorname{diff}(x, x)+f(x), f(x))$
$\mathrm{f}(\mathrm{x})=\mathrm{C} 1 * \sin (\mathrm{x})+\mathrm{C} 2 * \cos (\mathrm{x})$
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:

## Exercises

1. Solve the Bernoulli differential equation

$$
x \frac{d f(x)}{x}+f(x)-f(x)^{2}=0
$$

2. Solve the same equation using hint='Bernoulli'. What do you observe?
uthor: Emmanuelle Gouillart
cikit-image is a Python package dedicated to image processing, and using natively NumPy arrays as image objects. This chapter describes how to use scikit-image on various image processing tasks, and insists on he link with 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 content

- Introduction and concepts
- scikit-image and the SciPy ecosystem
- What's to be found in scikit-image
- Input/output, data types and colorspace
- Data types

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


### 17.1 Introduction and concepts

## mages 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 n
>>> check $=$ np. zeros ( $(9,9)$ )
>>> check $[:: 2,1:: 2]=1$
>> check $[1:: 2,:: 2]=1$
>>> import matplotlib. pyplot as plt
>>> plt.imshow(check, cmap='gray', interpolation='nearest')

17.1.1 scikit-image and the SciPy ecosystem

Recent versions of scikit-image is packaged in most Scientific Python distributions, such as Anaconda or Enthought Canopy. It is also packaged for Ubuntu/Debian.

```
>> import skimage
>> import skimage form from skimage import data # most functions are in subpackages
```

Most scikit-image functions take NumPy ndarrays as arguments
>>> camera = data.camera()
>> camera.dtype
dtype('uint8')
>>> camera. shape
(512, 512)
>> from skimage import restoration
>> filtered_camera $=$ restoration denoise bilateral (camera)
camera)
<type 'numpy.ndarray'>
Other Python packages are available for image processing and work with NumPy arrays:

- scipy .ndimage : for nd-arrays. Basic filtering, mathematical morphology, regions properties
- Mahotas

Also, powerful image processing libraries have Python bindings:

- OpenCV (computer vision)
- ITK (3D images and registration)
- and many others
(but they are less Pythonic and NumPy friendly, to a variable extent).


### 17.1.2 What's to be found in scikit-image

- Website: http://scikit-image.org/
- Gallery of examples (as in matplotlib or scikit-learn): http://scikit-
image.org/docs/stable/auto_examples/
Different kinds of functions, from boilerplate utility functions to high-level recent algorithms.
- Filters: functions transforming images into other images
- NumPy machinery
- Common filtering algorithms
- Data reduction functions: computation of image histogram, position of local maxima, of corners, etc.
- Other actions: I/O, visualization, etc.


### 17.2 Input/output, data types and colorspaces

I/O: skimage.io
>>> from skimage import io
Reading from files: skimage.io.imread()
>>> import os
>>> filename = os.path.join(skimage.data_dir, 'camera.png')
>>> camera = io.imread(filename)


Imaging Library (or any other I/O plugin provided to imread with the plugin keyword argument)
Also works with URL image paths:
>>> logo = io.imread('http://scikit-image.org/_static/img/logo.png')
Saving to files:

## >>> io.imsave('local_logo.png', logo)

(imsave also uses an external plugin such as PIL)

### 17.2.1 Data types



Image ndarrays can be represented either by integer
(signed or unsigned) or floats.
Careful with overflows with integer data types
>> camera = data.camera()
>>> camera.dtype
dtype('uint8')
>> camera_multiply $=3 *$ camera
Different integer sizes are possible: 8-, 16- or 32-bytes, signed or unsigned
A 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
>> from skimage import img_as_float
>>> camera_float = img_as_float (camera)
$(255,1.0)$ max () , camera_float.max
meimage roce ing ype and the data range from the input array

$$
\begin{aligned}
& \ggg \text { try: } \\
& \ldots \quad \text { from skimage import filters } \\
& \ldots \text { except ImportError: } \\
& \ldots \quad \text { from skimage import filter as filters } \\
& \ggg \text { camera_sobel = filters. sobel(camera) } \\
& \ggg \text { camera_sobel } \text {.max () } \\
& 0.591502 \ldots
\end{aligned}
$$

In the example above, we use the filters submodule of scikit-image, that has been renamed from filter to filters between versions 0.10 and 0.11 , in order to avoid a collision with Python's built-in name filter.

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.

### 17.2.2 Colorspaces

Color images are of shape ( $\mathrm{N}, \mathrm{M}, 3$ ) or ( $\mathrm{N}, \mathrm{M}, 4$ ) (when an alpha channel encodes transparency)
$\gg$ face $=$ scipy .misc.face(
>> face. shape

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 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.

### 17.3 Image preprocessing / enhancement

Goals: denoising, feature (edges) extraction, ...

### 7.3.1 Local filters

Local filters replace the value of pixels by a function of the values of neighboring pixels. The function can be inear or non-linear.
Neighbourhood: square (choose size), disk, or more complicated structuring element


Example : horizontal Sobel filter
>>> text = data.text()
>>> hsobel_text = filters.hsobel(text)
Uses the following linear kernel for computing horizontal gradients:

| 1 | 2 | 1 |
| :--- | :--- | :--- |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

$\begin{array}{ccc}0 & 0 & 0 \\ -1 & -2 & -1\end{array}$


### 17.3.2 Non-local filters

Non-local filters use a large region of the image (or all the image) to transform the value of one pixel:
>> from skimage import exposure
>> camera = data.camera()
>> camera_equalized = exposure.equalize_hist(camera)


Enhances contrast in large almost uniform regions

### 17.3.3 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

```
>> from skimage import morpholog,
>> morphology 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 ele ment:

```
>> a = np.zeros((7,7), dtype=np.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]1)
```

```
>> morphology binary_erosion(a, morphology 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]
    0,0,0,0,0, 0, 0
    [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>> #Erosion removes objects smaller than the structur
>> morphology binary_erosion (a, morphology diamond (2)) .astype (np.uint8)
array([[0, 0, 0, 0, 0, 0, 0]
    [0,0,0,0,0, 0, 0]
    llllll
    0, 0, 0, 0, 0, 0, 0]
    0, 0, 0, 0, 0, 0, 0]
    [0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```


## ilation: maximum filter:

>> a = np.zeros ( $(5,5)$ )
$\gg \mathrm{a}[2,2]=1$
> a
array([[ 0., 0., 0., 0., 0.],

$$
\left[\begin{array}{lllll}
{[0 .,} & 0 ., & 0 ., & 0 ., & 0 .], \\
{\left[\begin{array}{llll}
0 ., & 0 ., & 1 ., & 0 ., \\
\hline
\end{array}\right],} & 0 .
\end{array}\right],
$$

>> morphology. binary_dilation(a, morphology.diamond (1)) astype(np.uint8)
rray ([[0, $0,0,0,00]$,
$[0,0,1,0,0]$,
$[0,1,1,1,0]$,
$[0,0,0,0,0]]$, dtype=uint8)
Opening: erosion + dilation:

```
>> \(\mathrm{a}=\mathrm{np}\). zeros \(((5,5)\), dtype=np. int
>> a
\(\operatorname{array}([[0,0,0,0,0]\),
    \([0,1,1,1,0]\),
    \([0,1,1,1,0]\),
    \([0,1,1,1,0]\),
\([0,0,0\),
0,
\(\gg\) morphology binary_opening(a, morphology diamond(1)) astype (np.uint8)
\(\operatorname{array}\left(\left[\begin{array}{lll}{[0,0,0} & 0,0] \text {, }\end{array}\right.\right.\)
    \([0,0,1,0,0]\),
    \([0,1,1,1,0]\),
    0, 0, 0, 0, 0]1, dtype=uints
```

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 mplementation works on arbitrary-dimensional arrays.

## Example of filters comparison: image denoising

>>> from skimage.morphology import disk
coins = data.coins()
\>> coins_zoom $=$ coins [10:80, 300:370]
>>> median_coins = filters.rank.median(coins_zoom, disk(1))
>> from skimage import restoration
>>> tv_coins = restoration.denoise_tv_chambolle(coins_zoom, weight=0.1)
>> gaussian_coins = filters.gaussian_filter (coins, sigma=2)


### 17.4 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.

### 17.4.1 Binary segmentation: foreground + background

Histogram-based method: Otsu thresholding
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
from skimage import data
from skimage import filters
from skimage import fil
camera $=$ data. camera()
val = filters.threshold_otsu(camera)
mask $=$ camera $<$ val


## Labeling connected components of a discrete image

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:

```
l>>n=20
>> im = np.zeros((1, 1))
>> points = 1* np.random.random((2, n ** 2))
>> im[(points[0]) astype(np.int), (points[1]).astype(np.int)] = 1
>> im = filters.gaussian_filter(im, sigma=l / (4. * n))
>> blobs = im > im.mean()
```

Label all connected components:
>>> from skimage import measure
>>> all labels = measure label (blobs)
Label only foreground connected components:
>>> blobs_labels = measure.label(blobs, background=0)


## See also:

scipy.ndimage.find_objects() is useful to return slices on object in an image
17.4.2 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 .morphology. watershed()) is a region-growing approach that fills "basins" in the image
>>> from skimage.morphology import watershed
>> from skimage.feature import peak_local_max
$\ggg$
\# Generate an initial image with two overlapping circles
>> $x, y=n p . \operatorname{indices}((80,80))$
>> $\mathrm{x} 1, \mathrm{y} 1, \mathrm{x} 2, \mathrm{y} 2=28,28,44,52$
>> r1, r2 $=16,20$
>> mask_circle1 $=(\mathrm{x}-\mathrm{x} 1) * * 2+(\mathrm{y}-\mathrm{y} 1) * * 2<\mathrm{r} 1 * * 2$
>> mask_circle2 $=(\mathrm{x}-\mathrm{x} 2) * * 2+(\mathrm{y}-\mathrm{y} 2) * * 2<\mathrm{r} 2 * * 2$
>> 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
>> from scipy import ndimage
>> distance = ndimage.distance_transform_edt(image)
>>> local_maxi = peak_local_max(distance, indices=False, footprint=np.ones((3, 3)), labels=image >> labels_ws = watersh.label(local_maxi)

## Random walker segmentation

The random walker algorithm (skimage.segmentation.random_walker()) is similar to the Watershed, bu with a more "probabilistic" approach. It is based on the idea of the diffusion of labels in the image:
>>> from skimage import segmentation
>> \# Transform markers image so that 0-valued pixels are to
>>> \# be sorm markers image so that o-valued pixels are
>> markers[~image] $=-1$
>> labels_rw = segmentation.random_walker(image, markers)


## Postprocessing label images

skimage provides several utility functions that can be used on label images (ie images where ifferent discrete values identify different regions). Functions names are often self-explaining skimage.segmentation.clear_border() kimage.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 threshold-
ing, adaptive thresholding, and watershed or random walker segmentation.
- If necessary, use a postprocessing function to improve the coins / background segmentation.


### 17.5 Measuring regions' properties

>>> from skimage import measure
Example: compute the size and perimeter of the two segmented regions:
>>> properties = measure.regionprops(labels_rw)
>> [prop.area for prop in properties]
[770.0, 1168.0]
>> [prop.perimeter for prop in properties]
[100.91..., 126.81...]

## See also:

for some properties, functions are available as well in scipy.ndimage.measurements with a different API (a ist 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 an image of labels for the different co


### 17.6 Data visualization and interaction

Meaningful visualizations are useful when testing a given processing pipeline.
Some image processing operations:
>>> coins = data.coins()
>> mask = coins > filters.threshold_otsu(coins)
>> clean_border = segmentation.clear_border(mask)
Visualize binary result:

## >> plt.figure()

<matplotlib.figure.Figure object at 0x...>
>> plt.imshow(clean_border, cmap='gray')
<matplotlib.image.AxesImage object at 0x...>
Visualize contour
>>> plt.figure()
<matplotlib.figure.Figure object at 0x. .
>> 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 = segmentation.mark_boundaries(coins, clean_border.astype(np.int))


The (ex-
perimental) scikit-image viewer
skimage. viewer $=$ matplotlib-based canvas for displaying images + experimental Qt-based GUI-toolkit
>>> from skimage import viewer
>>> new_viewer = viewer. ImageViewer(coins)
>>> new_viewer.show()
Useful for displaying pixel values
For more interaction, plugins can be added to the viewer:
>> new_viewer = viewer.ImageViewer(coins)
>> from skimage.viewer. plugins import lineprofile
new_viewer += lineprofile.LineProfile()
>> new_viewer.show()

Line Profile

## 



28 @ [160, 231]

### 17.7 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

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)

ample 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.

## Traits: building interactive dialogs

## Author: Didrik Pinte

The Traits project allows you to simply add validation, initialization, delegation, notification and a graphical user interface to Python object attributes.
In this tutorial we will explore the Traits toolset and learn how to dramatically reduce the amount of boilerplate code you write, do rapid GUI application development, and understand the ideas which underly other parts oode you write, do rapid GUI
Traits and the Enthought Tool Suite are open source projects licensed under a BSD-style license.

## Intended Audience

Intermediate to advanced Python programmer

## Requirements

- Either wxPython, PyQt or PySide
- Numpy and Scipy
- Enthought Tool Suit
- All required software can be obtained by installing the EPD Free


## Tutorial content

- Introduction

Example

- What are Traits
- Initialisation

Validation

- Documentation
- Visualization: opening a dialos
- Deferral
- Notification
- Some more advanced traits


### 18.1 Introduction

The Enthought Tool Suite enable the construction of sophisticated application frameworks for data analysis, 2D plotting and 3D visualization. These powerful, reusable components are released under liberal BSD-style licenses.


The main package of the Enthought Tool Suite are

- Traits - component based approach to build our applications.
- Kiva - 2D primitives supporting path based rendering, affine transforms, alpha blending and more.
- Enable - object based 2D drawing canvas
- Chaco - plotting toolkit for building complex interactive 2D plots
- Mayavi - 3D visualization of scientific data based on VTK.
- Envisage - application plugin framework for building scriptable and extensible applications
n this tutorial, we will focus on Traits.


### 18.2 Example

Throughout this tutorial, we will use an example based on a water resource management simple case. We will try to model a dam and reservoir system. The reservoir and the dams do have a set of parameters :

- Name
- Minimal and maximal capacity of the reservoir [hm3]
- Height and length of the dam [m]
- Catchment area [km2]
- Hydraulic head [m]
- Power of the turbines [MW]
- Minimal and maximal release [m3/s]
- Efficiency of the turbines

The reservoir has a known behaviour. One part is related to the energy production based on the water released A simple formula for approximating electric power production at a hydroelectric plant is $P=\rho h r g k$, where:

- $P$ is Power in watts,
- $\rho$ is the density of water $(\sim 1000 \mathrm{~kg} / \mathrm{m} 3)$,
- $h$ is height in meters
- $r$ is flow rate in cubic meters per second,
- $g$ is acceleration due to gravity of $9.8 \mathrm{~m} / \mathrm{s} 2$,
- $k$ is a coefficient of efficiency ranging from 0 to 1 .

Annual electric energy production depends on the available water supply. In some installations the water flow rate can vary by a factor of 10:1 over the course of a year

The second part of the behaviour is the state of the storage that depends on controlled and uncontrolled parameters :

$$
\text { storage }_{t+1}=\text { storage }_{t}+\text { inflows }- \text { release }- \text { spillage }- \text { irrigation }
$$

The data used in this tutorial are not real and might even not have sense in the reality

### 18.3 What are Traits

A trait is a type definition that can be used for normal Python object attributes, giving the attributes some additional characteristics:

## - Standardization:

## - Initialization

- Validation
- Deferral
- Notification
- Visualization
- Documentation

A class can freely mix trait-based attributes with normal Python attributes, or can opt to allow the use of only fixed or open set of trait attributes within the class. Trait attributes defined by a class are automatically arited by any subclass derived from the class.

The common way of creating a traits class is by extending from the HasTraits base class and defining class raits
from traits.api import HasTraits, Str, Float
class Reservoir(HasTraits):

## name $=$ Str

max_storage $=$ Float

## For Traits 3.x users

If using Traits 3.x, you need to adapt the namespace of the traits packages:

- traits.api should be enthought.traits.api
- traitsui.api should be enthought.traits.ui.ap

Using a traits class like that is as simple as any other Python class. Note that the trait value are passed using keyword arguments:
reservoir = Reservoir(name='Lac de Vouglans', max_storage=605)

### 8.3.1 Initialisation

All the traits do have a default value that initialise the variables. For example, the basic python types do have he following trait equivalents.

| Trait | Python Type | Built-in Default Value |
| :--- | :--- | :--- |
| Bool | Boolean | False |
| Complex | Complex number | $0+0 \mathrm{j}$ |
| Float | Floating point number | 0.0 |
| Int | Plain integer | 0 |
| Long | Long integer | 0 L |
| Str | String | " |
| Unicode | Unicode | $\mathrm{u}^{\prime \prime}$ |

A number of other predefined trait type do exist : Array, Enum, Range, Event, Dict, List, Color, Set, Expression, Code, Callable, Type, Tuple, etc.
Custom default values can be defined in the code
from traits.api import HasTraits, Str, Floa
class Reservoir(HasTraits):
name $=$ Str
max_storage $=$ Float(100)
reservoir = Reservoir(name='Lac de Vouglans')

## Complex initialisation

When a complex initialisation is required for a trait, a XXX default magic method can be implemented. It will be lazily called when trying to access the XXX trait. For example:

```
def _name_default(self)
    Complex initialisation of the reservoir name. """
    return 'Undefined
```


### 18.3.2 Validation

Every trait does validation when the user tries to set its content:
reservoir = Reservoir(name='Lac de Vouglans', max_storage=605)
reservoir.max_storage $={ }^{\prime} 230$
TraitError
/scipy-luts/<traits/<ipython-input-7-979bdff9974a> in <module>()
----> 1 reservoir max storage $={ }^{\prime} 230$

```
../traits/trait_handlers.pyc in error(self, object, name, value)
lll""
->168 value )
170 def arg_error (self, method, arg_num, object, name, value ):
```

TraitError: The 'max_storage' trait of a Reservoir instance must be a float, but a value of ' 23 ' <type 'str

### 18.3.3 Documentation

By essence, all the traits do provide documentation about the model itself. The declarative approach to the reation of classes makes it self-descriptive
from traits.api import HasTraits, Str, Float
class Reservoir(HasTraits):
name $=$ Str
max_storage $=$ Float(100)
The desc metadata of the traits can be used to provide a more descriptive information about the trait :
from traits.api import HasTraits, Str, Float
lass Reservoir(HasTraits),
name $=$ Str
max_storage $=$ Float (100, desc='Maximal storage [hm3]')

## Let's now define the complete reservoir class

from traits.api import HasTraits, Str, Float, Range
class Reservoir(HasTraits):
name $=$ Str
max_storage $=$ Float (1e6, desc='Maximal storage $[\mathrm{hm} 3]$ ')
max_release $=$ Float (10, desc='Maximal release $[\mathrm{m} 3 / \mathrm{s}]$ ')
head $=$ Float (10, desc='Hydraulic head [m]')
efficiency $=\operatorname{Range}(0,1$.
def energy_production(self, release):
"' Returns the energy production [wh] for the given release [m3/s.
power $=1000 * 9.81 *$ self.head $*$ release $*$ self.efficiency
return power * 3600
if _-_name_- == '_-main_-'
reservoir $=$ Reservoir
name $=$ 'Project $A^{\prime}$,
max_storage $=30$,
max_release $=100.0$,
head $=60$,
ency $=0.8$

## )

re1en
print 'Releasing $\} \mathrm{m} 3 / \mathrm{s}$ produces $\}$ kWh'.format(
release, reservoir.energy_production(release)

### 8.3.4 Visualization: opening a dialog

The Traits library is also aware of user interfaces and can pop up a default view for the Reservoir class:
reservoir1 = Reservoir()
eservoir1.edit_traits()

| Edit properties |  |
| ---: | :--- |
| Efficiency: | 0.0 |
| Head: | 10 |
| Hydraulic head: | 60 |
| Max release: | 100.0 |
| Max storage: | 30.0 |
| Name: | Project A |
|  |  |
|  | OK |

TraitsUI simplifies the way user interfaces are created. Every trait on a HasTraits class has a default editor that will manage the way the trait is rendered to the screen (e.g. the Range trait is displayed as a slider, etc.)

In the very same vein as the Traits declarative way of creating classes, TraitsUI provides a declarative interface o build user interfaces code:
from traits.api import HasTraits, Str, Float, Range
from traitsui.api import View
class Reservoir(HasTraits):
name $=$ Str
max_storage $=$ Float(1e6, desc='Maximal storage [hm3]
max_release $=$ Float (10, desc='Maximal release [m3/s]')
head $=$ Float (10, desc='Hydraulic head [m]')
efficiency $=$ Range $(0,1$.
traits_view $=$ View
'name', 'max_storage', 'max_release', 'head', 'efficiency',
resizable $=$ True,
)
def energy_production(self, release):
Returns the energy production [Wh] for the given release [m3/s]
power $=1000 * 9.81 *$ self.head $*$ release $*$ self.efficiency
return power * 3600
if _-name__ == ' _ main__'
reservoir $=$ Reservoir
name $=$ 'Project A',
max_storage $=30$,
max_release $=100.0$,
head $=60$,
efficiency $=0.8$
)

## reservoir configure traits()

|  | Reservoir |
| ---: | :--- |
| Name: | Project A |
| Max storage: | $\mathbf{3 0 . 0}$ |
| Max release: | 100.0 |
| Head: | 60.0 |
| Efficiency: | 0.0 |
|  |  |

### 18.3.5 Deferral

Being able to defer the definition of a trait and its value to another object is a powerful feature of Traits.
from traits.api import HasTraits, Instance, DelegatesTo, Float, Range
from reservoir import Reservoir
class ReservoirState(HasTraits):
"""Keeps track of the reservoir state given the initial storage
reservoir = Instance(Reservoir, ()
min_storage $=$ Float
max_storage $=$ DelegatesTo('reservoir')
min_release $=$ Float
max_release = DelegatesTo('reservoir')
\# state attributes
storage $=$ Range(low='min_storage', high='max_storage')
\# control attributes
inflows $=$ Float(desc='Inflows [hm3]')
release $=$ Range(low='min_release', high='max_release')
spillage $=$ Float (desc='Spillage [hm3]')
def print_state(self):
print 'Storage\tRelease\tInflows\tSpillage'
str_format $='^{\prime} \backslash t$ '. join(['\{:7.2f\}'for $i$ in range (4)])
print str_format.format(self.storage, self.release, self.inflows,
self.spillage)
print '-1 * 79

If __name__ == '_-main_-_'
projectA $=$ Reservoir
name $=$ Project
max storage $=30$,
max_release $=100$.
hydraulic_head $=60$
efficiency $=0.8$
)
state = ReservoirState(reservoir=projectA, storage=10)
state.release $=90$
state.inflows $=0$
state. print_state()
print 'How do we update the current storage ?'
A special trait allows to manage events and trigger function calls using the magic _xxxx_fired method: from traits.api import HasTraits, Instance, DelegatesTo, Float, Range, Event
from reservoir import Reservoir
class Reservoirstate(HasTraits):
""Keeps track of the reservoir state given the initial storage
For the simplicity of the example, the release is considered in
hm3/timestep and not in $\mathrm{m} / \mathrm{s}$.
reservoir = Instance(Reservoir, ()
min_storage $=$ Float
max_storage $=$ DelegatesTo('reservoir')
min_release $=$ Float
max_release $=$ DelegatesTo('reservoir')
\# state attributes
storage = Range(low='min_storage', high='max_storage')
\# control attributes
inflows $=$ Float(desc='Inflows [hm3]
release $=$ Range(low='min_release', high='max_release')
spillage $=$ Float $($ desc='Spillage $[$ hm3]')
update_storage $=$ Event(desc='Updates the storage to the next time step')
def _update_storage_fired(self)
\# update storage state
ew_storage $=$ self.storage - self.release + self inflows
elf.
self.spillage $=\max ($ overflow, 0$)$
def print_state(self)
print 'Storage\tRelease\tInflows\tSpillage'
str_format $=$ ' $\backslash t$ '.. join(['\{:7.2f $\}$ 'for i in range(4)])
print str_format.format(self.storage, self.release, self.inflows
self.spillage)
if __name__ == '_-main__'
projectA $=$ Reservoir
name $=$ 'Project A',
max_storage $=30$,
max_release $=5.0$,
hydraulic_head $=60$
efficiency $=0.8$
)
state $=$ ReservoirState(reservoir=projectA, storage=15)
state.release $=5$
state.inflows $=0$
\# release the maximum amount of water during 3 time steps
state.update_storage $=$ True
state.print_state()
state. update_storage $=$ True
state.print_state()

## state. update storage

state. print_state()
Dependency between objects can be made automatic using the trait Property. The depends_on attribute expresses the dependency between the property and other traits. When the other traits gets changed, the property is invalidated. Again, Traits uses magic method names for the property :

- _get_XXX for the getter of the XXX Property trai
- _set_XXX for the setter of the XXX Property trait
from traits.api import HasTraits, Instance, DelegatesTo, Float, Range
from traits.api import Property
from reservoir import Reservoir
class ReservoirState(HasTraits)
"""Keeps track of the reservoir state given the initial storage
For the simplicity of the example, the release is considered in
hm3/timestep and not in m3/s.
reservoir $=$ Instance(Reservoir, ())
max_storage $=$ DelegatesTo('reservoir')
min_release $=$ Float
max_release $=$ DelegatesTo('reservoir')
\# state attributes
storage $=$ Property(depends_on='inflows, release')
\# control attributes
inflows $=$ Float (desc='Inflows [hm3] $\left.{ }^{1}\right)$
release $=$ Range(low='min_release', high='max_release')
desc='Spillage [hm3]', depends_on=['storage', 'inflows', 'release']
)
\#\#\# Private traits.
_storage $=$ Float
\#\#\# Traits property implementation.
def _get_storage (self):
new_storage $=$ self. storage - self.release + self.inflows return min(new_storage, self.max_storage)
def _set_storage(self, storage_value)
self._storage = storage_value
def _get_spillage(self):
new_storage $=$ self._storage - self.release + self.inflows overflow = new_storage - self.max_storage return $\max$ (overflow, 0
def print_state(self):
print 'Storage\tRelease\tInflows\tSpillage'

print str_format.format(self.storage, self.release, self.inflows
self.spillage)
print '-' * 79
f __name__ == '_-main__'
projectA $=$ Reservoir
name $={ }^{\prime}$ Project $\mathrm{A}^{\prime}$
max_storage $=30$,

```
max_release = 5,
lic_head = 60
efficiency = 0.8
```

)
state $=$ ReservoirState(reservoir=projectA, storage=25)
state. release $=4$
state.inflows $=0$
state.print_state()

## Caching property

Heavy computation or long running computation might be a problem when accessing a property where the inputs have not changed. The @cached_property decorator can be used to cache the value and only recompute them once invalidated.
Let's extend the TraitsUI introduction with the ReservoirState example
from traits.api import HasTraits, Instance, DelegatesTo, Float, Range, Property from traitsui.api import View, Item, Group, VGroup
from reservoir import Reservoir
class ReservoirState(HasTraits)
"""Keeps track of the reservoir state given the initial storage
For the simplicity of the example, the release is considered in
hm3/timestep and not in m3/s.
reservoir $=$ Instance(Reservoir, ()
name = DelegatesTo('reservoir')
max_storage $=$ DelegatesTo('reservoir')
max_release $=$ DelegatesTo('reservoir')
min_release $=$ Float
\# state attributes
storage = Property(depends on='inflows, release')
\# control attributes
release $=$ Range(low='min_release', high='max_release')
spillage $=$ Property $($
desc='Spillage [hm3]', depends_on=['storage', 'inflows', 'release']
\#\#\# Traits vie
traits vien $=$ Views
Group(
VGroup(Item('name'), Item('storage'), Item('spillage') label = 'State', style = 'readonly'

VGroup(Item('inflows'), Item('release'), label='Control')
,
\#\#\# Private traits.
_storage $=$ Float
\#\# Traits property implementation
def _get_storage(self):
ew_storage $=$ self._storage - self.release + self.inflows
feturn min(new_storage, self.max_storage)

```
def _set_storage(self, storage_value)
    self. storage = storage value
```

    def _get_spillage(self):
        new_storage = self._storage - self.release + self.inflows
        overflow = new_storage - self.max_storage
        return max(overflow, 0)
    def print_state(self)
        print 'Storage\tRelease\tInflows\tSpillage'
        str_format \(=' \backslash t\) '. join(['\{:7.2f\}'for i in range(4)]
        rint str_format.format(self.storage, self.release, self.inflows,
        self.spillage)
        print '-। * 79
    if __name__ == '__main__':
projectA = Reservoir (
name $=$ 'Project A',
max_storage $=30$,
max_release $=5$
hydraulic_head $=60$
efficiency $=0.8$
)
state $=$ ReservoirState(reservoir=projectA, storage=25
state. release $=4$
state. inflows $=0$
state. print_state()
state.configure_traits()

| Edit properties |
| :--- |
| State |
| Name: Project A |
| Storage: 21.0 |
| Spillage: 0 |
| Control |
| Inflows: 0.0 |
| Release: 0.0 |

Some use cases need the delegation mechanism to be broken by the user when setting the value of the trait. The PrototypeFrom trait implements this behaviour
from traits api import HasTraits, Str, Float, Range, PrototypedFrom, Instance
class Turbine(HasTraits):
turbine_type $=$ Str
power = Float(1.0, desc='Maximal power delivered by the turbine [Mw]')
class Reservoir(HasTraits)
name $=$ Str
max_storage = Float (1e6, desc='Maximal storase [hm3] $)$
max_release $=$ Float (10, desc='Maximal release $[\mathrm{m} 3 / \mathrm{s}]$ ')
head $=$ Float(10, desc='Hydraulic head [m]')
efficiency $=\operatorname{Range}(0,1$.
turbine = Instance(furbine)
installed_capacity = PrototypedFrom('turbine', 'power')
_-name_- == '_-_main__'
turbine $=$ Turbine(turbine_type='type1', power=5.0)
reservoir $=$ Reservoir (
name $=$ 'Project A'
max_release $=100$.
head $=60$,
efficiency $=0.8$,
turbine $=$ turbine,
)
print 'installed capacity is initialised with turbine. power'
print reservoir. installed_capacity
print '-' * 15
print 'updating the turbine power updates the installed capacity'
turbine. power $=10$
print reservoir.installed_capacity
print '-' * 15
print 'setting the installed capacity breaks the link between turbine.power print 'and the installed_capacity trait
reservoir.installed_capacity $=8$
print turbine.power, reservoir.installed_capacity

### 18.3.6 Notification

Traits implements a Listener pattern. For each trait a list of static and dynamic listeners can be fed with call backs. When the trait does change, all the listeners are called

Static listeners are defined using the_XXX_changed magic methods:
from traits.api import HasTraits, Instance, DelegatesTo, Float, Range
from reservoir import Reservoir
class ReservoirState(HasTraits)
Keeps track of the reservoir state given the initial storage
reservoir $=$ Instance (Reservoir, ())
min storage = Float
max_storage $=$ DelegatesTo('reservoir')
min_release $=$ Float
max_release $=$ DelegatesTo('reservoir')
\# state attributes
storage $=$ Range(low='min_storage', high='max_storage')
\# control attributes
inflows $=$ Float (desc='Inflows [hm3]')
release $=$ Range(low='min_release' high='max release')
spillage $=$ Float (desc='Spillage [hm3]')
def print_state(self)
print 'Storage \tRelease\tInflows \tSpillage
str_format $=' \backslash t '$ '.join([' $\{: 7.2 \mathrm{ft}$ 'for i in range(4)])
print str_format.format(self.storage, self.release, self.inflows self.spillage)
print '-1 * 79
\#\#\# Traits listeners \#\#\#\#\#\#\#\#\#\#\#
def _release_changed (self, new):
hen the release is higher than zero, warn all the inhabitants of
the valley
if new $>0$
print 'Warning, we are releasing $\} \mathrm{hm} 3$ of water'.format(new)
if _-name__ == '__main__'
projectA $=$ Reservoir $($
name = 'Project A'
max_storage $=30$,
max_release $=100.0$
efficiency $=0.8$
,
state $=$ ReservoirState(reservoir=projectA, storage=10)
state. release $=90$
state. inflows $=0$
state. print_state()
The static trait notification signatures can be

- def_release_changed(self): pass
- def_release_changed(self, new): pass
- def _release_changed(self, old, new): pass
- def _release_changed(self, name, old, new pass


## Listening to all the changes

To listen to all the changes on a HasTraits class, the magic _any_trait_changed method can be implemented.

In many situations, you do not know in advance what type of listeners need to be activated. Traits offers the ability to register listeners on the fly with the dynamic listeners
from reservoir import Reservoi
from reservoir state property import ReservoirState
def wake_up_watchman_if_spillage(new_value):
if new_value > 0
print 'Wake up watchman! Spilling \{\} hm3'.format(new_value)
_-_name_- == '-_main_-
projectA $=$ Reservoir
name $=$ 'Project A',
max_storage $=30$,
max_release $=100.0$
hydraulic_head $=60$
efficiency $=0.8$
state $=$ ReservoirState(reservoir=projectA, storage=10)
register the dynamic listener
state.on_trait_change (wake_up_watchman_if_spillage, name='spillage')
state.release $=90$
state. inflows $=0$
state. print_state()
print 'Forcing spillage'
state. inflows $=100$
state. release $=0$
print 'Why do we have two executions of the callback ?'
The dynamic trait notification signatures are not the same as the static ones :

- def wake_up_watchman (): pass
- def wake_up_watchman(new): pass
- def wake_up_watchman(name, new): pas
- def wake_up_watchman(object, name, new): pass
- def wake_up_watchman(object, name, old, new): pass


## Removing a dynamic listener can be done by:

- calling the remove_trait_listener method on the trait with the listener method as argument,
- calling the on_trait_change method with listener method and the keyword remove=True,
- deleting the instance that holds the listener.

Listeners can also be added to classes using the on_trait_change decorator:
from traits.api import HasTraits, Instance, DelegatesTo, Float, Range from traits.api import Property, on_trait_change
from reservoir import Reservoir
class ReservoirState(HasTraits):
"""Keeps track of the reservoir state given the initial storage
For the simplicity of the example, the release is considered in
hm3/timestep and not in m3/s.
reservoir = Instance(Reservoir, ())
max_storage $=$ DelegatesTo('reservoir'
min_release $=$ Float
max_release = DelegatesTo('reservoir')
\# state attributes
storage $=$ Property(depends_on='inflows, release')
\# control attributes
inflows = Float(desc='Inflows [hm3] ')
release = Range(low='min_release', high='max_release')
spillage $=$ Property
desc='Spillage [hm3]', depends_on=['storage', 'inflows', 'release']
\#\#\# Private traits. \#\#\#\#\#\#\#\#\#\#
_storage $=$ Float
\#\#\# Traits property implementation
def _get_storage(self):
new_storage $=$ self._storage - self.release + self.inflows return $\min$ (new_storage, self.max_storage)
def _set_storage(self, storage_value) self._storage = storage_value
def _get_spillage(self):
new_storage $=$ self._storage - self.release + self.inflows overflow $=$ new storage - self max storage return max (overflow, 0)
©on_trait_change('storage')
def print_state(self)
print Storageltrelease\tinfiows\tSpillage
str_format $=' \backslash t$ '. join([' $\{: 7.2 f\}$ 'for $i$ in range(4)])
print str_format.format(self.storage, self.release, self.inflows
self.spillage)
print '-' * 79
if _-_name_- == '_-_main_- '
projectA $=$ Reservoir (
name $=$ 'Project $\mathrm{A}^{\prime}$,
max_storage $=30$,
max_release $=5$,
hydraulic_head $=60$
hydraulic_head
)
state $=$ ReservoirState(reservoir=projectA, storage=25)
state. release $=4$
state.inflows $=0$
The patterns supported by the on_trait_change method and decorator are powerful. The reader should look at the docstring of HasTraits.on_trait_change for the details.

### 18.3.7 Some more advanced traits

The following example demonstrate the usage of the Enum and List traits :
from traits.api import HasTraits, Str, Float, Range, Enum, List
from traitsui.api import View, Item
Class IrrigationArea(HasTraits)
name $=$ Str
surface $=$ Float (desc='Surface [ha] ')
crop $=$ Enum('Alfalfa', 'Wheat', 'Cotton')
lass Reservoir(HasTraits):
name $=\operatorname{Str}$
max_storage $=$ Float (1e6, desc='Maximal storage [hm3]')
max_release $=$ Float (10, desc='Maximal release [m3/s]')
head $=$ Float ( 10 , desc $=$ Hydraulic head $[m]$ )
efficiency = Range ( $0,1$. )
irrigated_areas $=$ List (IrrigationArea)
def energy_production(self, release):
Returns the energy production [wh] for the given release [m3/s]

```
power = 1000 * 9.81 * self.head * release * self.efficiency
return power * 3600
```

traits_view = View(
Item('name'),
Item('max_storage'),
Item('max_release'),
Item('head'),
Item('efficiency'),
Item('irrigated_areas')
resizable $=$ True
)
if __name_- == ' _-main__'
upper_block = IrrigationArea(name='Section C', surface=2000, crop='Wheat')
reservoir $=$ Reservoir
name= 'Project A'
max_storage $=30$,
max_release $=100.0$,
head $=60$,
efficien
irrigated_areas=[upper_block]
)
release $=80$
print 'Releasing $\} \mathrm{m} 3 / \mathrm{s}$ produces $\} \mathrm{kWh}$ '.format(
, release, reservoir.energy_production(release)
Trait listeners can be used to listen to changes in the content of the list to e.g. keep track of the total crop surface on linked to a given reservoir.
from traits.api import HasTraits, Str, Float, Range, Enum, List, Property
from traitsui.api import View, Item
class IrrigationArea(HasTraits):
name $=\operatorname{Str}$
surface $=$ Float(desc='Surface [ha] ')
crop $=$ Enum('Alfalfa', 'Wheat', 'Cotton')
class Reservoir(HasTraits):
name $=$ Str
max_storage $=$ Float (1e6, desc='Maximal storage [hm3]')
max_release $=$ Float $(10$, desc='Maximal release $[\mathrm{m} 3 / \mathrm{s}]$ ')
head $=$ Float (10, desc='Hydraulic head [m]')
efficiency = Range ( $0,1$. )
irrigated_areas = List(IrrigationArea)
total_crop_surface = Property(depends_on='irrigated_areas.surface')
def _get_total_crop_surface(self):
return sum([iarea.surface for iarea in self.irrigated_areas])
def energy_production(self, release):
"' Returns the energy production [Wh] for the given release [m3/s] power $=1000$ * 9.81 * self.head $*$ release * self.efficiency return power * 3600

```
traits_view = View(
    Item('name'),
    Item('max_storage'),
    tem('max_release')
    tem('head')
    Item('efficiency'),
    Item('total_ced_areas')
    p_surface'),
)
___name_- == '__main__'
upper_block = IrrigationArea(name='Section C', surface=2000, crop='Wheat')
reservoir = Reservoir(
                                    name=' Project A',
                                    nax_storage=30,
                                max_release=100.0
            head=60,
            efficiency=0.8,
            irrigated_areas=[upper_block],
            )
release = 80
print 'Releasing {} m3/s produces {} kWh'.format(
release, reservoir.energy_production(release)
```

The next example shows how the Array trait can be used to feed a specialised TraitsUI Item, the ChacoPlotItem:

```
mport numpy as np
from traits.api import HasTraits, Array, Instance, Float, Property
from traits.api import DelegatesTo
from traitsui.api import View, Item, Group
from chaco.chaco_plot_editor import ChacoPlotIten
from reservoir import Reservoir
lass ReservoirEvolution(HasTraits)
    reservoir = Instance(Reservoir)
    name = DelegatesTo('reservoir')
    inflows = Array(dtype=np.float64, shape=(None))
    releass = Array(dtype=np.float64, shape=(None))
    initial_stock = Float
    stock = Property(depends_on='inflows, releases, initial_stock')
    month = Property(depends_on='stock')
    ### Traits view ###########
    traits_view = View
            Item('name')
            Group(
        ChacoPlotItem('month', 'stock', show_label=False),
            width = 500,
            resizable = Tru
,
```

\#\#\# Traits properties \#\#\#\#
def _get_stock(self):

$$
\begin{aligned}
& \text { fixme: should handle cases where we go over the max storage } \\
& \text { ""n" } \\
& \text { return self.initial_stock + (self.inflows - self.releases).cumsum() }
\end{aligned}
$$

return np.arange(self.stock. size)
f __name__ == '_-main__':
reservoir $=$ Reservoir ( name $=$ 'Project $A^{\prime}$
max_storage $=30$
max_release $=100.0$,
head $=60$,
efficiency $=0.8$
)
initial stock $=10$
inflows_ts = np.array([6., 6, 4, 4, 1, 2, 0, 0, 3, 1, 5, 3])
releases ts = np. $\operatorname{array}([4 ., 5,3,5,3,5,5,3,2,1,3,3])$
view $=$ ReservoirEvolution(
reservoir = reservoir
inflows = inflows_ts,
releases $=$ releases_ts
view.configure_traits()


Edit properties
Name: Project A

Plot Editor


## See also:

References

- ETS repositories
- Traits manual
- Traits UI manual
- Mailing list : enthought-dev@enthought.com


## Author: Gaël Varoquaux

Mayavi is an interactive 3D plotting package. matplotlib can also do simple 3D plotting, but Mayavi relies on a more powerful engine (VTK) and is more suited to displaying large or complex data

## Chapters contents

- Mlab: the scripting interface
- 3D plotting functions
* Points
* Elevation surface
* Arbitrary regular mesh
* Volumetric data
- Figures and decorations

Frgure management

* Changing plot properties
* Decoratio

Interactive work

- The "pipeline dialog"
"ript recording button
Slicing and dicing data: sources, modules and filters
- An example: inspecting magnetic field
rews on data: sources and module
* Different sources: scatters and fields
* Transforming data: filters
ne: the scripting layer
- Animating the data
a
- A simple dialog

Putting it together

### 19.1 Mlab: the scripting interface

The mayavi.mlab module provides simple plotting functions to apply to numpy arrays, similar to matplotlib or matlab's plotting interface. Try using them in IPython, by starting IPython with the switch --gui=wx.

### 19.1.1 3D plotting functions

Points


Points in 3D, represented with markers (or "glyphs") and optionaly different sizes.
$x, y, z$, value $=n p . r a n d o m . r a n d o m((4,40))$
mlab. points3d(x, y, z, value)

Lines
plot3d


A line connecting points in 3D, with optional thickness and varying color
mlab.clf() \# Clear the figure
$t=n p .1$ inspace $(0,20,200)$
mlab. plot3d (np. $\sin (\mathrm{t}), \mathrm{np} \cdot \cos (\mathrm{t}), 0.1 * \mathrm{t}, \mathrm{t})$

Elevation surface
surf


A surface given by its elevation, coded as a 2D array
mlab.clf()
x, $y=n p . m g r i d[-10: 10: 100 j,-10: 10: 100 j]$
$\mathrm{r}=\mathrm{np} . \operatorname{sqrt}(\mathrm{x} * * 2+\mathrm{y} * * 2)$
$=n \mathrm{n} \cdot \sin (\mathrm{r}) / \mathrm{r}$
mlab.surf( $z$, warp_scale='auto')

Arbitrary regular mesh
mesh


A surface mesh given by $\mathrm{x}, \mathrm{y}, \mathrm{z}$ positions of its node points
mlab.clf( $)$
phi, theta $=$ np.mgrid $[0: n p$. pi:11j, $\quad 0: 2 * n p$. pi:11j]
$\mathrm{x}=\mathrm{np} \cdot \sin (\mathrm{phi}) * \mathrm{np} \cdot \cos ($ theta $)$
$\mathrm{z}=\mathrm{np} \cdot \cos (\mathrm{phi})$
lab.mesh $(x, y, z)$ representation='wireframe', color=(0, 0, 0)
A surface is defined by points connected to form triangles or polygones. In mayavi.mlab.surf( and mayavi.mlab.mesh(), the connectivity is implicity given by the layout of the arrays. See also mayavi.mlab.triangular_mesh()
Our data is often more than points and values: it needs some connectivity information

Volumetric data
contour3d


If your data is dense in 3D, it is more difficult to display. One option is to take iso-contours of the data.
mlab.clf()
$x, y, z=n p . \operatorname{mgrid}[-5: 5: 64 \mathrm{j},-5: 5: 64 \mathrm{j},-5: 5: 64 \mathrm{j}]$
values $=x * x * 0.5+\mathrm{y} * \mathrm{y}+\mathrm{z} * \mathrm{z} * 2.0$
mlab. contour3d(values)


This function works with a regular orthogonal grid: the value array is a 3D array that gives the shape of the grid.

### 19.1.2 Figures and decorations

Figure management
Here is a list of functions useful to control the current figure

| Get the current figure: | mlab.gcf() |
| :--- | :--- |
| Clear the current figure: | mlab.clf() |
| Set the current figure: | mlab.figure(1, bgcolor=(1, 1, 1), fgcolor $=(0.5, \quad 0.5,0.5)$ |
| Save figure to image file: | mlab.savef $\lg ($ ' $f o o$. png', size $=(300,300))$ |
| Change the view: | mlab.view(azimuth $=45$, elevation $=54$, distance $=1)$. |

## Changing plot propertie

In general, many properties of the various objects on the figure can be changed. If these visualization ar created via mlab functions, the easiest way to change them is to use the keyword arguments of these functions, as described in the docstrings.

## Example docstring: mlab.mesh

Plots a surface using grid-spaced data supplied as 2D arrays.
Function signatures
$\operatorname{mesh}(x, y, z, \ldots)$
$\mathrm{x}, \mathrm{y}, \mathrm{z}$ are 2D arrays, all of the same shape, giving the positions of the vertices of the surface. The connec
tivity between these points is implied by the connectivity on the arrays.
For simple structures (such as orthogonal grids) prefer the surf function, as it will create more efficient
data structures.
Keyword arguments
color the color of the vtk object. Overides the colormap, if any, when specified This is specified as a triplet of float ranging from 0 to 1, eg $(1,1,1)$ for white.
colormap type of colormap to use.
extent [xmin, xmax, ymin, ymax, zmin, zmax] Default is the $\mathrm{x}, \mathrm{y}, \mathrm{z}$ arrays extents. Use this to change the extent of the object created.
figure Figure to populate.
line_width The with of the lines, if any used. Must be a float. Default: 2.0
mask boolean mask array to suppress some data points
mask_points If supplied, only one out of 'mask_points' data point is displayed. This option is usefull to reduce the number of points displayed on large datasets Must be an integer or None.
mode the mode of the glyphs. Must be '2darrow' or '2dcircle' or '2dcross' or '2ddash' or '2ddiamond' or '2dhooked_arrow' or '2dsquare' o 2dthick arrow' or '2dthick cross' or '2dtriangle' or '2dvertex' or 'arrow' or 'cone' or 'cube' or 'cylinder' or 'point' or 'sphere'. Default: sphere
name the name of the vtk object created.
representation the representation type used for the surface. Must be 'surface' or 'wireframe' or 'points' or 'mesh' or 'fancymesh'. Default: surface
resolution The resolution of the glyph created. For spheres, for instance, this is
the number of divisions along theta and phi. Must be an integer. Default: 8
scalars optional scalar data
scale factor scale factor of the glyphs used to represent the vertices, in fancy_mesh mode. Must be a float. Default: 0.05
scale_mode the scaling mode for the glyphs ('vector', 'scalar', or 'none')
transparent make the opacity of the actor depend on the scalar
tube_radius radius of the tubes used to represent the lines, in mesh mode. If None, simple lines are used.
tube_sides number of sides of the tubes used to represent the lines. Must be an integer. Default: 6
max vmax is used to scale the colormap If None, the max of the data will be used vmin vmin is used to scale the colormap If None, the $\min$ of the data will be used


## Example:

In [1]: import numpy as np
In [2]: r, theta $=n p \cdot m g r i d[0: 10,-n p . p i: n p \cdot p i: 10 j]$
In [3]: $x=r$ np. $\cos$ (theta)
In [4]: y $=\mathrm{r} * \mathrm{np} \cdot \sin ($ theta $)$
n [5]: $\mathrm{z}=\mathrm{np} \cdot \sin (\mathrm{r}) / \mathrm{r}$
In [6]: from mayavi import mlab
In [7]: mlab.mesh(x, y, z, colormap='gist_earth', extent=[0, 1, 0, 1, 0, 1]
Dut [7]: <mayavi.modules.surface.Surface object at 0xde6f08c>
In [8]: mlab.mesh(x,y, $z$, extent $=[0,1,0,1,0,1]$,
representation='wireframe', line_width=1, color=(0.5, 0.5, 0.5))
[87. <mayavi.modules surface. Surface object at 0xdd6a71c>

## Decorations

Different items can be added to the figure to carry extra information, such as a colorbar or a title
In [9]: mlab.colorbar(Out [7], orientation='vertical')
Sut [9]: <tvtk_classes.scalar_bar_actor.ScalarBarActor object at 0xd897f8c>
In [10]: mlab.title('polar mesh')
Jut [10]: <enthought.mayavi.modules.text.Text object at 0xd8ed38c>
In [11]: mlab.outline (Out [7])
Out [11]: <enthought. mayavi. modules.outline.Outline object at 0xdd21b6c>
In [12]: mlab.axes(Out [7])
Dut [12]: <enthought.mayavi. modules axes. Axes object at 0xd2e4bcc>

extent: If we specified extents for a plotting object, mlab.outline' and 'mlab.axes don't get them by default.

### 9.2 Interactive work

The quickest way to create beautiful visualization with Mayavi is probably to interactively tweak the various settings

### 19.2.1 The "pipeline dialog"

Click on the 'Mayavi' button in the scene, and you can control properties of objects with dialogs.


- Set the background of the figure in the Mayavi Scene node
- Set the colormap in the Colors and legends node
- Right click on the node to add modules or filter


### 19.2.2 The script recording button

o find out what code can be used to program these changes, click on the red button as you modify those properties, and it will generate the corresponding lines of code.

### 19.3 Slicing and dicing data: sources, modules and filters

19.3.1 An example: inspecting magnetic fields

Suppose we are simulating the magnetic field generated by Helmholtz coils. The examples/compute_field.py script does this computation and gives you a B array, that is ( 3 x n ), where the first axis is the direction of the field (Bx, By, Bz), and the second axis the index number of the point. Arrays $\mathrm{X}, \mathrm{Y}$ and Z give the positions of these data points.

## Excercise

Visualize this field. Your goal is to make sure that the simulation code is correct.


## Suggestions

- If you compute the norm of the vector field, you can apply an isosurface to it. using mayavi.mlab. quiver3d () you can plot vectors. You can also use the 'masking' options (in the GUI) to make the plot a bit less dense


### 19.3.2 Different views on data: sources and modules

As we see above, it may be desirable to look at the same data in different ways.
Mayavi visualization are created by loading the data in a data source and then displayed on the screen using modules.

This can be seen by looking at the "pipeline" view. By right-clicking on the nodes of the pipeline, you can add new modules.

## Quiz

Why is it not possible to add a VectorCutPlane to the vectors created by mayavi .mlab. quiver3d ()?

## Different sources: scatters and fields

Data comes in different descriptions.

- A 3D block of regularly-spaced value is structured: it is easy to know how one measurement is related to another neighboring and how to continuously interpolate between these. We can call such data a field, borrowing from terminology used in physics, as it is continuously defined in space.
- A set of data points measured at random positions in a random order gives rise to much more difficult and ill-posed interpolation problems: the data structure itself does not tell us what are the neighbors of a data point. We call such data a scatter


Data sources corresponding to scatters can be created with mayavi.mlab.pipeline.scalar_scatter() or mayavi.mlab.pipeline.vector_scatter(); field data sources can be created with mlab.pipeline.scalar_field() ormlab.pipeline.vector_field()

## Exercice:

1. Create a contour (for instance of the magnetic field norm) by using one of those functions and adding the right module by clicking on the GUI dialog
. Create the right source to apply a 'vector_cut_plane' and reproduce the picture of the magnetic field shown previously.
Note that one of the difficulties is providing the data in the right form (number of arrays, shape) to the functions. This is often the case with real-life data.

## See also:

Sources are described in details in the Mayavi manual.

Transforming data: filters
If you create a vector field, you may want to visualize the iso-contours of its magnitude. But the isosurface module can only be applied to scalar data, and not vector data. We can use a filter, ExtractVectorNorm to add this scalar value to the vector field

Filters apply a transformation to data, and can be added between sources and modules

## Excercice

Using the GUI, add the ExtractVectorNorm filter to display iso-contours of the field magnitude.
mlab.pipeline: the scripting layer
The mlab scripting layer builds pipelines for you. You can reproduce these pipelines programmatically with he mlab. pipeline interface: each step has a corresponding mlab. pipeline function (simply convert the name of the step to lower-case underscore-separated: ExtractVectorNorm gives extract_vector_norm). This unction takes as an argument the node that it applies to, as well as optional parameters, and returns the new node.
For example, iso-contours of the magnitude are coded as:
mlab. pipeline.iso_surface(mlab.pipeline.extract_vector_norm(field),

$$
\text { contours=[0.1*Bmax, } 0.4 * \mathrm{Bmax}],
$$

opacity $=0.5$ )


## Excercice

Using the mlab.pipeline interface, generate a complete visualization, with iso-contours of the field mag nitude, and a vector cut plane. (click on the figure for a solution)

### 19.4 Animating the data

To make movies, or interactive application, you may want to change the data represented on a given visualiza tion.
If you have built a visualization, using the mlab plotting functions, or the mlab. pipeline function, we can update the data by assigning new values to the mlab source attributes
$x, y, z=n p . o g r i d[-5: 5: 100 j,-5: 5: 100 j,-5: 5: 100 j]$
scalars $=n p \cdot \sin (x * y * z) /(x * y * z)$
so = mlab.contour3d(scalars, transparent=True, contours=[0.5]
or i in range ( 1,20 ):
(1, in $(i * x * y * z) /(x * y * z)$
iso.mlab_source.scalars = scalars

## See also:

More details in the Mayavi documentation

## Event loops

For the interaction with the user (for instance changing the view with the mouse), Mayavi needs some time to process these events. The for loop above prevents this. The Mayavi documentation details a workaround

### 19.5 Making interactive dialogs

It is very simple to make interactive dialogs with Mayavi using the Traits library (see the dedicated chapter Traits: building interactive dialogs).

### 19.5.1 A simple dialog

from traits.api import HasTraits, Instance
from traitsui.api import View, Item, HGroup
from mayavi.core.ui.api import SceneEditor, MlabSceneModel
def curve(n_turns)
"The function creating the $x, y, z$ coordinates needed to plot"
phi $=$ np.linspace ( $0,2 *$ np.pi, 2000)
return [np. $\cos (\mathrm{phi}) *\left(1+0.5 * \mathrm{np} \cdot \cos \left(\mathrm{n}_{\mathrm{n}}\right.\right.$ turns*phi)$)$ ),
np. $\sin ($ phi $) *\left(1+0.5 * n p \cdot \cos \left(n_{-}\right.\right.$turns*phi) $)$,
$0.5 *$ np. $\sin \left(n_{-}\right.$turns*phi)]
class Visualization(HasTraits)
"The class that contains the dialog"
scene $=$ Instance(MlabSceneModel, ())
def __init__(self):
HasTraits. __init__(self)
x, y, z = curve(n_turns=2)
elf

$$
\text { self.plot }=\text { self.scene.mlab.plot3d }(x, y, z)
$$

\# Describe the dialog
view $=$ View(Item('scene', height=300, show_label=False,
editor=SceneEditor())
HGroup('n_turns'), resizable=True
\# Fire up the dialog
isualization().configure_traits()
Let us read a bit the code above (examples/mlab_dialog.py).
First, the curve function is used to compute the coordinate of the curve we want to plot.
Second, the dialog is defined by an object inheriting from HasTraits, as it is done with Traits. The important point here is that a Mayavi scene is added as a specific Traits attribute (Instance). This is important for embedding it in the dialog. The view of this dialog is defined by the view attribute of the object. In the init of this object, we populate the 3D scene with a curve.
Finally, the conf igure_traits method creates the dialog and starts the event loop.
See also
There are a few things to be aware of when doing dialogs with Mayavi. Please read the Mayavi documentation

### 19.5.2 Making it interactive

We can combine the Traits events handler with the mlab_source to modify the visualization with the dialog. We will enable the user to vary the n_turns parameter in the definition of the curve. For this, we need:

- to define an n_turns attribute on our visualization object, so that it can appear in the dialog. We use a Range type.
- to wire modification of this attribute to a recomputation of the curve. For this, we use th on_traits_change decorator.


## © - Edit properties <br> 园 区



## N turns:0 $\quad 3 0 \longdiv { 1 1 }$

```
from traits.api import Range, on_trait_change
class Visualization(HasTraits)
    n_turns = Range(0,30,11)
    def _-init__(self)
        HasTraits.__init__(self)
        x, y, z = curve(self.n_turns)
        self.plot = self.scene.mlab.plot3d(x, y, z)
```

    @on_trait change('n turns')
    def update_plot(self):
        \(\mathrm{x}, \mathrm{y}, \mathrm{z}=\) curve(self.n turns)
        self.plot.mlab_source. \(\operatorname{set}(x=x, y=y, z=z)\)
    view \(=\) View(Item('scene', height=300, show_label=False,
        editor=SceneEditor()),
        HGroup('n_turns'), resizable=True)
    \# Fire up the dialog
isualization().configure traits()

Full code of the example: examples/mlab_dialog.py.

### 19.6 Putting it together

## Exercise

Using the code from the magnetic field simulation, create a dialog that enable to move the 2 coils: change their parameters.
Hint: to define a dialog entry for a vector of dimension 3
direction $=\operatorname{Array}(f l o a t$, value $=(0,0,1)$, cols=3, shape $=(3)$,

You can look at the example_coil_application.py to see a full-blown application for coil design in 270 lines of code.
scikit-learn: machine learning in Python

## Authors: Fabian Pedregosa, Gael Varoquaux

learn

## Prerequisites

- Numpy, Scipy
- IPython
- matplotlib
- scikit-learn (http:// scikit-learn.org)

Chapters contents

- Loading an example dataset
- Learning and Predicting
- Classification
- k-Nearest neighbors classifier
- Support vector machines (SVMs) for classification
- Clustering: grouping observations together
- K-means clustering
- Dimension Reduction with Principal Component Analysis
- Putting it all together: face recognition
- Linear model: from regression to sparsity
- Sparse models
- Model selection: choosing estimators and their parameters - Grid-search and cross-validated estimators


First we will load some data to play with. The data we will use is a very simple flower database known as the Iris dataset.
We have 150 observations of the iris flower specifying some measurements: sepal length, sepal width, petal length and petal width together with its subtype: Iris setosa, Iris versicolor, Iris virginica.

To load the dataset into a Python object:
>>> from sklearn import datasets
>> iris = datasets.load_iris()
This data is stored in the .data member, which is a (n_samples, n_features) array.
>> iris. data. shape
$(150,4)$
The class of each observation is stored in the .target attribute of the dataset. This is an integer 1D array of length n_samples:
>> iris.target.shape
(150,)
>> import numpy as np
>> np. unique (iris.target
$\operatorname{array}([0,1,2])$

## An example of reshaping data: the digits dataset



The digits dataset consists of 1797 images, where each one is an $8 \times 8$ pixel image representing a hand written digit:
>>> digits = datasets.load_digits (
3>> digits.images.shape
(1797, 8, 8)
>> pl.imshow(digits.images[0], cmap=pl.cm.gray_r)
<matplotlib.image.AxesImage object at ...>
To use this dataset with the scikit, we transform each 8x8 image into a vector of length 64
>>> data $=$ digits.images.reshape ((digits.images.shape[0], -1))

### 20.1.1 Learning and Predicting

Now that we've got some data, we would like to learn from it and predict on new one. In scikit-learn, we learn from existing data by creating an estimator and calling its $f$ it ( $\mathrm{X}, \mathrm{Y}$ ) method
>> from sklearn import svm
>> clf = svm.LinearSvC()
>> clf.fit(iris.data, iris.target) \# learn from the dat
LinearSVC(...)
Once we have learned from the data, we can use our model to predict the most likely outcome on unseen data
>> clf.predict([[ $5.0,3.6,1.3,0.25]]$ )
array([0])
We can access the parameters of the model via its attributes ending with an underscore
>> clf coef
array ([[ $[0-7])$

### 20.2 Classification

### 20.2.1 k-Nearest neighbors classifier

The simplest possible classifier is the nearest neighbor: given a new observation, take the label of the trainin amples closest to it in $n$-dimensional space, where $n$ is the number of features in each sample.


Sepal length
The $k$-nearest neighbors classifier internally uses an algorithm based on ball trees to represent the samples it is trained on.

## KNN (k-nearest neighbors) classification example:

>> \# Create and fit a nearest-neighbor classifier
>> from sklearn import neighbors
>> from sklearn import neighbors
>> knn = neighbors.KNeighborsClassifier ()
>> knn.fit(iris.data, iris.target)
NeighborsClassifier(...)
(01)
array([0])

## Training set and testing set

When experimenting with learning algorithms, it is important not to test the prediction of an estima tor on the data used to fit the estimator. Indeed, with the kNN estimator, we would always get perfect prediction on the training set.
>>> perm = np.random. permutation(iris.target.size)
>>> iris.data = iris.data[perm]
>>> iris.target = iris.target [perm]
>>> knn.fit(iris.data[:100], iris.target[:100])
KNeighborsClassifier (...)
>>> knn.score(iris.data[100:], iris.target [100:])
0.95999 .

Bonus question: why did we use a random permutation?

### 20.2.2 Support vector machines (SVMs) for classification

## inear Support Vector Machines

SVMs try to construct a hyperplane maximizing the margin between the two classes. It selects a subset of the input, called the support vectors, which are the observations closest to the separating hyperplane.

>> from sklearn import svm
>> svc = svm.SVC(kernel='linear')
SVC(...) svit(iris.data, iris.target)
SVC(...)
There are several support vector machine implementations in scikit-learn. The most commonly used one are svm. SVC, svm. NuSVC and svm.LinearSVC; "SVC" stands for Support Vector Classifier (there also exist SVMs for regression, which are called "SVR" in scikit-learn).

## Exercise

Train an svm. SVC on the digits dataset. Leave out the last $10 \%$, and test prediction performance on these observations.

## Using kernels

Classes are not always separable by a hyperplane, so it would be desirable to have a decision function that is not linear but that may be for instance polynomial or exponential:

| Linear kernel | Polynomial kernel | RBF kernel (Radial Basis Function) |
| :---: | :---: | :---: |
|  |  |  |
| >>> svc = svm.SVC(kernel='lin |  | $y \geqslant \ggg s v c=s v m . S V C(k e r n e l=1 r b$ <br> >>> \# gamma: inverse of size <br> e>> \# radial kernel |

Exercise
Which of the kernels noted above has a better prediction performance on the digits dataset?

### 20.3 Clustering: grouping observations together

Given the iris dataset, if we knew that there were 3 types of iris, but did not have access to their labels, we could try unsupervised learning: we could cluster the observations into several groups by some criterion.

### 20.3.1 K-means clustering

The simplest clustering algorithm is k -means. This divides a set into $k$ clusters, assigning each observation to cluster so as to minimize the distance of that observation (in $n$-dimensional space) to the cluster's mean; the means are then recomputed. This operation is run iteratively until the clusters converge, for a maximum for max_iter rounds.
(An alternative implementation of k -means is available in SciPy's cluster package. The scikit-learn im plementation differs from that by offering an object API and several additional features, including smart ini ialization.)
from sklearn import cluster, datasets
>> iris = datasets.load_iris()
>> k_means = cluster.KMeans (n_clusters=3)
>> k_means.fit(iris.data)
kMeans (...)
>> print(k_means.labels_[::10])

[00000111112222


## Application to Image Compression

Clustering can be seen as a way of choosing a small number of information from the observations (like a projection on a smaller space). For instance, this can be used to posterize an image (conversion of a continuous gradation of tone to several regions of fewer tones)
>>> from scipy import misc
>>> face $=$ misc.face (gray=True) .astype (np.float32)
>> $\mathrm{x}=\mathrm{face} \cdot \mathrm{reshape}((-1,1))$ \# We need an ( $n$ sample, $n$ feature) arra
>>> K = k_means = cluster. KMeans (n_clusters=5) \# 5 clusters
>>> k_means.fit(X)
KMeans (...)
>>> values = k_means.cluster_centers_. squeeze()
>>> labels = k_means.labels_
>> face_compressed = np.choose(labels, values)


### 20.4 Dimension Reduction with Principal Component Analysis



The cloud of points spanned by the observations above is very flat in one direction, so that one feature can almost be exactly computed using the 2 other. PCA finds the directions in which the data is not flat and it can reduce the dimensionality of the data by projecting on a subspace.

Depending on your version of scikit-learn PCA will be in module decomposition or pca.
>>> from sklearn import decomposition
>> pca $=$ decomposition. $\operatorname{PCA}\left(\mathrm{n}_{\text {_ }}\right.$ components=2)
>> pca.fit(iris.data)
PCA (copy=True, n_components=2, whiten=False)

Now we can visualize the (transformed) iris dataset:
>>> import pylab as pl
$\mathrm{x}[:, 1]$, $c=$ iris.target
matplotlib.collections...Collection object at


PCA is not just useful for visualization of high dimensional datasets. It can also be used as a preprocessing step to help speed up supervised methods that are not efficient with high dimensions.

### 20.5 Putting it all together: face recognition

An example showcasing face recognition using Principal Component Analysis for dimension reduction and Support Vector Machines for classification.


Stripped-down version of the face recognition example:
mport numpy as np
import pylab as $p$
rom sklearn import cross_val, datasets, decomposition, svm
\# ... load data
fw_people = datasets.fetch_lfw_people(min_faces_per_person=70, resize=0.4)
perm $=$ np.random. permutation(lfw_people.target.size)
fw_people.data $=1 \mathrm{fw}$ _people.data[perm]
$\mathrm{f}_{\mathrm{w}}$ people.target $=1 \mathrm{f} \mathrm{w}_{-}$_people.target [perm]
faces $=n p$. reshape (lfw_people.data, (lfw_people.target.shape[0], -1))
rain, test = iter(cross_val.StratifiedKFold(lfw_people.target, $k=4$ )).next ()
train, y test $=1 \mathrm{f}$ people target [test]
_train, y test = lfw people.target[train], lfw_people.target[test]
pca $=$ decomposition.RandomizedPCA(n_components=150, whiten=True)
pca.fit(X_train)
_train_pca = pca.transform (X_train)
X_test_pca $=$ pca.transform(X_test)
\#.
.. classification
lf $=\operatorname{svm} . \operatorname{SVC}(C=5 .$, gamma=0.001)
lf.fit(X_train_pca, y_train)

```
##...
#or i in range(10):
    print(lfw_people.target_names[clf.predict(X_test_pca[i])[0]])
    _= pl.imshow(X_test[i].reshape(50, 37), cmap=pl.cm.gray)
    = raw_input()
```


### 20.6 Linear model: from regression to sparsity

## Diabetes dataset

The diabetes dataset consists of 10 physiological variables (age, sex, weight, blood pressure) measure on
42 patients, and an indication of disease progression after one year:
>>> diabetes = datasets.load_diabetes()
>>> diabetes_X_train = diabetes.data[:-20]
>> diabetes_X_test = diabetes.data[-20:]
>> diabetes_y_train $=$ diabetes.target $[:-20]$
The task at hand is to predict disease prediction from physiological variables.

### 20.6.1 Sparse models

To improve the conditioning of the problem (uninformative variables, mitigate the curse of dimensionality, as a feature selection preprocessing, etc.), it would be interesting to select only the informative features and set non-informative ones to 0 . This penalization approach, called Lasso, can set some coefficients to zero. Such methods are called sparse method, and sparsity can be seen as an application of Occam's razor: prefer simple models to complex ones.
>> from sklearn import linear_model
>> rear = linear_mort linear_model
>> regr.fit(diabetes_X_train, diabetes_y_train)
lasso(...) reeg \# very sparse coefficients
$\operatorname{array}\left(\left[\begin{array}{ccc}0 & - \\ -0 . & -0 . & -0 . \\ 4 & -197.34075682, & 199.17441034, \\ \hline\end{array}\right.\right.$

| -0.930 .9379595, | -0. | 0. |
| :---: | :---: | :---: |

>> regr.score(diabetes_X_test, diabetes_y_test)
0.5510835453.
being the score very similar to linear regression (Least Squares):
>>> lin = linear_model.LinearRegression()
>>> lin.fit(diabetes_X_train, diabetes_y_train
inearRegression (. .
>>> lin.score(diabetes_X_test, diabetes_y_test)
0.5850753022 .

## Different algorithms for a same problem

Different algorithms can be used to solve the same mathematical problem. For instance the Lasso ob ject in the sklearn solves the lasso regression using a coordinate descent method, that is efficient on large datasets. However, the sklearn also provides the LassoLARS object, using the LARS which is very effi cient for problems in which the weight vector estimated is very sparse, that is problems with very few observations.

### 20.7 Model selection: choosing estimators and their parameters

### 20.7.1 Grid-search and cross-validated estimators

Grid-search
he scikit-learn provides an object that, given data, computes the score during the fit of an estimator on parameter grid and chooses the parameters to maximize the cross-validation score. This object takes an estimator during the construction and exposes an estimator API:
>> from sklearn import svm, grid_search
>> gammas = np.logspace ( $-6,-1,10$ )
>>> svc $=\operatorname{svm} \cdot \operatorname{SVC}()$
>> clf = grid_search.GridSearchCV(estimator=svc, param_grid=dict(gamma=gammas)
$\cdots$ n_jobs=-1)
(>lf.fit(digits.data[:1000], digits.target[:1000])
>> clf.best score
0.9..
clf. best_estimator_ gamm
. 00059948425031894088
By default the GridSearchCV uses a 3-fold cross-validation. However, if it detects that a classifier is passed rather than a regressor, it uses a stratified 3-fold

Cross-validated estimators
Cross-validation to set a parameter can be done more efficiently on an algorithm-by-algorithm basis. This is why, for certain estimators, the scikit-learn exposes " CV " estimators, that set their parameter automatically by cross-validation:
>> from sklearn import linear_model, datasets
>> lasso = linear_model. LassoCV()
>> diabetes = datasets.load_diabetes(
>> X_diabetes = diabetes.data
>> y_diabetes = diabetes.target
>>> lasso.fit(X_diabetes, y_diabetes
LassoCV (alphas=None,
>> \# The estimator chose automatically its lambda:
>> lasso. alpha
-012..
These estimators are called similarly to their counterparts, with ' CV ' appended to their name.

## Exercise

On the diabetes dataset, find the optimal regularization parameter alpha.
diff, 304, 307
differentiation, 304
dsolve, 307
E
uations
algebraic, 305
differential, 307
I
integration, 305
M
Matrix, 306
P
ython Enhancement Proposals
PEP 255, 145
PEP 3118, 182
PEP 3129, 155
PEP 318, 147, 155
PEP 342, 145
PEP 343, 155
PEP 380, 14id13, 147
PEP 380\#id

S
solve, 305


[^0]:    >>> bubbles_labels, bubbles_nb = ndimage.label(void)
    >> bubbles_areas = np.bincount (bubbles_labels.ravel()) [1:]
    >> mean_bubble_size = bubbles_areas.mean()
    >> med bubble size = median(bubbles_areas)

[^1]:    print
    Yes, print statements do work as a debugging tool. However to inspect runtime, it is often more efficien to use the debugger.

