Python Scientific lecture notes

Release 2010

EuroScipy tutorial team
Editors: Emmanuelle Gouillart, Gaël Varoquaux

July 09, 2010
## Contents

1 Scientific computing: why Python? .......................... 1
   1.1 The scientist’s needs ........................................... 1
   1.2 Specifications .................................................. 1
   1.3 Existing solutions .............................................. 1

2 Building blocks of scientific computing with Python ......... 3

3 A (very short) introduction to Python ......................... 6
   3.1 First steps ..................................................... 6
   3.2 Basic types .................................................... 8
   3.3 Control Flow .................................................. 14
   3.4 Defining functions ............................................ 19
   3.5 Reusing code: scripts and modules ......................... 25
   3.6 Input and Output ............................................. 31
   3.7 Standard Library .............................................. 32
   3.8 Exceptions handling in Python ............................ 37
   3.9 Object-oriented programming (OOP) ....................... 39

4 NumPy: creating and manipulating numerical data .......... 41
   4.1 Creating NumPy data arrays ................................. 41
   4.2 Graphical data representation: matplotlib and Mayavi .. 42
   4.3 Indexing ....................................................... 45
   4.4 Slicing ......................................................... 46
   4.5 Manipulating the shape of arrays ......................... 48
   4.6 Exercises: some simple array creations ................ 49
   4.7 Real data: read/write arrays from/to files ............... 50
   4.8 Simple mathematical and statistical operations on arrays 53
   4.9 Fancy indexing .............................................. 55
   4.10 Broadcasting ............................................... 57
   4.11 Synthesis exercises: framing Lena ....................... 61

5 Getting help and finding documentation ...................... 63

6 Matplotlib .................................................... 67
Scientific computing: why Python?

authors Fernando Perez, Emmanuelle Gouillart

1.1 The scientist's needs

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

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 neither is 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 labo or a company: the code should be as readable as a book. Thus, the language should contain as few syntax symbols or unneeded routines 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.

1.3 Existing solutions

Which solutions do the 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. Ex: 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 script 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 softwares 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 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 access, 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 softwares or toolboxes.
Building blocks of scientific computing with Python

author Emmanuelle Gouillart

• **Python**, a generic and modern computing language
  
  – Python language: data types (*string*, *int*), flow control, data collections (lists, dictionaries), patterns, 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 tests, documentation generation)

• **IPython**, an advanced Python shell

• **Numpy**: provides powerful numerical arrays objects, and routines to manipulate them.

```python
>>> import numpy as np
>>> t = np.arange(10)
>>> t
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print t
[0 1 2 3 4 5 6 7 8 9]
>>> signal = np.sin(t)
```

http://www.scipy.org/

• **Scipy**: high-level data processing routines. Optimization, regression, interpolation, etc:

```python
>>> import numpy as np
>>> import scipy
>>> t = np.arange(10)
>>> t
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> signal = t**2 + 2*t + 2 + 1e-2*np.random.random(10)
>>> scipy.polyfit(t, signal, 2)
array([1.00001151, 1.99920674, 2.00902748])
```

http://www.scipy.org/

• **Matplotlib**: 2-D visualization, “publication-ready” plots

http://matplotlib.sourceforge.net/
Mayavi: 3-D visualization

http://code.enthought.com/projects/mayavi/

and many others.
A (very short) introduction to Python

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 http://docs.python.org/tutorial. Dedicated books are also available, such as http://diveintopython.org/.

3.1 First steps

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

- an *interpreted* (as opposed to *compiled*) language. Contrary to e.g. C or Fortran, one does not compile Python code before executing it. In addition, Python can be used interactively: many Python interpreters are available, from which commands and scripts can be executed.

- a free software released under an open-source license: Python can be used and distributed free of charge, even for building commercial software.

- multi-platform: Python is available for all major operating systems, Windows, Linux/Unix, MacOS X, most likely your mobile phone OS, etc.

- a very readable language with clear non-verbose syntax

- a language for which a large variety of high-quality packages are available for various applications, from web frameworks to scientific computing.

- a language very easy to interface with other languages, in particular C and C++.
• Some other features of the language are illustrated just below. For example, Python is an object-oriented language, with dynamic typing (an object’s type can change during the course of a program).

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

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 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

```python
>>> 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

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

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

```c
int a;
a = 3;
```

In addition, the type of an object may change. b was first an integer, but it became a string when it was assigned the value hello. Operations on integers (b=2*a) are coded natively in the Python standard library, and so are some operations on strings such as additions and multiplications, which amount respectively to concatenation and repetition.
A bag of Ipython tricks

- Several Linux shell commands work in Ipython, such as `ls`, `pwd`, `cd`, etc.
- To get help about objects, functions, etc., type `help object`. Just type `help()` to get started.
- Use tab-completion as much as possible: while typing the beginning of an object’s name (variable, function, module), press the Tab key and Ipython will complete the expression to match available names. If many names are possible, a list of names is displayed.
- **History**: press the up (resp. down) arrow to go through all previous (resp. next) instructions starting with the expression on the left of the cursor (put the cursor at the beginning of the line to go through all previous commands)
- You may log your session by using the Ipython “magic command” `%logstart`. Your instructions will be saved in a file, that you can execute as a script in a different session.

```python
In [1]: %logstart commandes.log
Activating auto-logging. Current session state plus future input saved.
Filename : commandes.log
Mode : backup
Output logging : False
Raw input log : False
Timestamping : False
State : active
```

3.2 Basic types

3.2.1 Numerical types

Integer variables:

```python
>>> 1 + 1
2
>>> a = 4
```

floats

```python
>>> c = 2.1
```

complex (a native type in Python!)

```python
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

and booleans:

```python
>>> 3 > 4
False
>>> test = (3 > 4)
>>> test
False
>>> 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
```

**Warning:** Integer division

```
>>> 3/2  
1
```

**Trick:** use floats:

```
>>> 3/2.  
1.5
```

```
>>> a = 3  
>>> b = 2  
>>> a/b  
1
>>> a/float(b)  
1.5
```

- Scalar types: int, float, complex, bool:

```
>>> type(1)  
<type 'int'>
>>> type(1.)  
<type 'float'>
>>> type(1. + 0j )  
<type 'complex'>
```

```
>>> a = 3  
>>> type(a)  
<type 'int'>
```

- Type conversion:

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

### 3.2.2 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
• Indexing: accessing individual objects contained in the list:

```python
>>> l
[1, 2, 3, 4, 5]
>>> type(l)
<type 'list'>
```

Counting from the end with negative indices:

```python
>>> l[-1]
5
>>> l[-2]
4
```

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

• Slicing: obtaining sublists of regularly-spaced elements

```python
>>> 1
[1, 2, 3, 4, 5]
>>> 1[2:4]
[3, 4]
```

**Warning:** Note that \(l[^{\text{start}}:^{\text{stop}}]\) contains the elements with indices \(i\) such as \(\text{start} \leq i < \text{stop}\) (\(i\) ranging from \(\text{start}\) to \(\text{stop}-1\)). Therefore, \(l[^{\text{start}}:^{\text{stop}}]\) has \((\text{stop}-\text{start})\) elements.

**Slicing syntax:** \(l[^{\text{start}}:^{\text{stop}}:^{\text{stride}}]\)

All slicing parameters are optional:

```python
>>> 1[3:]
[4, 5]
>>> 1[:3]
[1, 2, 3]
>>> 1[:2]
[1, 3, 5]
```

Lists are *mutable* objects and can be modified:

```python
>>> l[0] = 28
>>> l
[28, 2, 3, 4, 5]
>>> l[2:4] = [3, 8]
>>> l
[28, 2, 3, 8, 5]
```

**Note:** The elements of a list may have different types:

```python
>>> 1 = [3, 2, 'hello']
>>> 1
[3, 2, 'hello']
>>> 1[1], 1[2]
(2, 'hello')
```
As the elements of a list can be of any type and size, accessing the \( i \text{th} \) element of a list has a complexity \( O(i) \). For collections of numerical data that all have the same type, it is more efficient to use the `array` type provided by the `Numpy` module, which is a sequence of regularly-spaced chunks of memory containing fixed-sized data items. With Numpy arrays, accessing the \( i \text{th} \) element has a complexity of \( O(1) \) because the elements are regularly spaced in memory.

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

Add and remove elements:

```python
>>> l = [1, 2, 3, 4, 5]
```n
```python
>>> l.append(6)
>>> l
[1, 2, 3, 4, 5, 6]
```n
```python
>>> l.pop()
6
>>> l
[1, 2, 3, 4, 5]
```n
```python
>>> l.extend([6, 7])  # extend l, in-place
```n
```python
>>> l
[1, 2, 3, 4, 5, 6, 7]
```n
```python
>>> l = l[:-2]
```n
```python
>>> l
[1, 2, 3, 4, 5]
```n
Reverse \( l \):

```python
>>> r = l[::-1]
```n
```python
>>> r
[5, 4, 3, 2, 1]
```n
Concatenate and repeat lists:

```python
>>> r + l
[5, 4, 3, 2, 1, 1, 2, 3, 4, 5]
```n
```python
>>> 2 * r
[5, 4, 3, 2, 1, 5, 4, 3, 2, 1]
```n
Sort \( r \) (in-place):

```python
>>> r.sort()
```n
```python
>>> r
[1, 2, 3, 4, 5]
```n
Note: Methods and Object-Oriented Programming

The notation \( r \text{.method()} \) (\( r \text{.sort()}, \ r \text{.append}(3), \ r \text{.pop()} \)) is our first example of object-oriented programming (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.

Note: Discovering methods:

In IPython: tab-completion (press tab)

```
In [28]: r
___add___          ___iadd___          ___setattr___
___class___        ___imul___          ___setitem___
```
Strings

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

- `s = 'Hello, how are you?'`
- `s = "Hi, what’s up"
- `s = '''Hello,
 how are you'''`
- `s = """Hi,
 what’s up?'"'

The newline character is \n, and the tab characted is \t.

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

Indexing:

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

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

Slicing:

```python
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,
'lo o'
```
Accents and special characters can also be handled in Unicode strings (see
http://docs.python.org/tutorial/introduction.html#unicode-strings).

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

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.

**Note:** Python offers advanced possibilities for manipulating strings, looking for patterns or formatting. Due to lack of time this topic is not addressed here, but the interested reader is referred to
http://docs.python.org/library/stdtypes.html#string-methods and http://docs.python.org/library/string.html#new-string-formatting

- **String substitution:**

```python
>>> 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string')
'An integer: 1; a float: 0.100000; another string: string'
```

```python
>>> i = 102
>>> filename = 'processing_of_dataset_%03d.txt'%i
>>> filename
'processing_of_dataset_102.txt'
```

### Dictionaries

A dictionary is basically a hash table that maps keys to values. It is therefore an unordered container:

```python
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'sebastian': 5578, 'francis': 5915, 'emmanuelle': 5752}
>>> tel['sebastian']
5578
>>> tel.keys()  
['sebastian', 'francis', 'emmanuelle']
>>> tel.values()  
[5578, 5915, 5752]
```
This is a very convenient data container in order to store values associated to a name (a string for a date, a name, etc.). See http://docs.python.org/tutorial/datastructures.html#dictionaries for more information.

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

```python
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
dict({'a': 1, 3: 'hello', 'b': 2})
```

More container types

- **Tuples**

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

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

- **Sets:** non ordered, unique items:

```python
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
set(['a', 'c', 'b'])
>>> s.difference(('a', 'b'))
set(['c'])
```

### 3.3 Control Flow

Controls the order in which the code is executed.

#### 3.3.1 if/elif/else

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.

```python
In [1]: if 2**2 == 4:
   ...:     print('Obvious!')
   ...:
Obvious!
```
In [2]: a = 10
In [3]: if a == 1:
   ...:     print(1)
   ...: elif a == 2:
   ...:     print(2)
   ...: else:
   ...:     print('A lot')
   ...
A lot

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

### 3.3.2 for/range

Iterating with an index:

```python
In [4]: for i in range(4):
   ...:     print(i)
   ...
0
1
2
3
```

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

```python
In [5]: for word in ('cool', 'powerful', 'readable'):
   ...:     print('Python is %s' % word)
   ...
Python is cool
Python is powerful
Python is readable
```

### 3.3.3 while/break/continue

Typical C-style while loop (Mandelbrot problem):

```python
In [6]: z = 1 + 1j
In [7]: while abs(z) < 100:
   ...:     z = z**2 + 1
   ...
In [8]: z
Out[8]: (-134+352j)
```

More advanced features

break out of enclosing for/while loop:
In [9]: z = 1 + 1j

In [10]: while abs(z) < 100:
    ....:     if z.imag == 0:
    ....:         break
    ....:     z = z**2 + 1
    ....:
    ....:

continue the next iteration of a loop.

```python
code
>>> a = [1, 0, 2, 4]
>>> for element in a:
...     if element == 0:
...         continue
...     print 1. / element
...1.0
0.5
0.25
```
• *a in b*

For any collection *b*: *b* contains *a*

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

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

### 3.3.5 Advanced iteration

**Iterate over any sequence**

- You can iterate over any sequence (string, list, dictionary, file, ...)

```python
In [11]: vowels = 'aeiouy'
In [12]: for i in 'powerful':
       ....:   if i in vowels:
       ....:     print(i),
       ....:
       ....:
       o e u
```

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

Few languages (in particular, languages for scientific computing) allow to loop over anything but integers/indices. With Python it is possible to loop exactly over the objects of interest without bothering with indices you often don’t care about.

**Warning:** Not safe to modify the sequence you are iterating over.

**Keeping track of enumeration number**

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

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

```python
In [13]: for i in range(0, len(words)):
       ....:   print(i, words[i])
       ....:
       ....:
```
0 cool
1 powerful
2 readable

- But Python provides \texttt{enumerate} for this:

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

\subsection*{Looping over a dictionary}

Use \texttt{iteritems}:

\begin{verbatim}
In [15]: d = {'a': 1, 'b':1.2, 'c':1j}
In [15]: for key, val in d.iteritems():
....:     print('Key: %s has value: %s' % (key, val))
....:
Key: a has value: 1
Key: c has value: 1j
Key: b has value: 1.2
\end{verbatim}

\subsection*{3.3.6 List Comprehensions}

\begin{verbatim}
In [16]: [i**2 for i in range(4)]
Out[16]: [0, 1, 4, 9]
\end{verbatim}

\section*{Exercise}

Compute the decimals of Pi using the Wallis formula:

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

\section*{Note: Good practices}

- Indentation: no choice!
Indenting is compulsory in Python. Every commands block following a colon bears an additional indentation level with respect to the previous line with a colon. One must therefore indent after `def f()`: or `while:`. At the end of such logical blocks, one decreases the indentation depth (and re-increases it if a new block is entered, etc.)

Strict respect of indentation is the price to pay for getting rid of `{` or `;` characters that delineate logical blocks in other languages. Improper indentation leads to errors such as

```
---------------------------
IndentationError: unexpected indent (test.py, line 2)
```

All this indentation business can be a bit confusing in the beginning. However, with the clear indentation, and in the absence of extra characters, the resulting code is very nice to read compared to other languages.

- **Indentation depth:**

  Inside your text editor, you may choose to indent with any positive number of spaces (1, 2, 3, 4, ...). However, it is considered good practice to **indent with 4 spaces**. You may configure your editor to map the Tab key to a 4-space indentation. In Python(x,y), the editor Scite 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 lines can be broken with the `\` character

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

**Spaces**

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

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

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

- **Use meaningful object names**

  Self-explaining names improve greatly the readability of a code.

### 3.4 Defining functions

#### 3.4.1 Function definition

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

**Warning:** Function blocks must be indented as other control-flow blocks.
3.4.2 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
```

**Note:** By default, functions return `None`.

**Note:** Note the syntax to define a function:
- the `def` keyword;
- is followed by the function’s `name`, then
- the arguments of the function are given between brackets followed by a colon.
- the function body;
- and `return object` for optionally returning values.

3.4.3 Parameters

Mandatory parameters (positional arguments)

```
In [81]: def double_it(x):
    ...:     return x * 2
    ...:

In [82]: double_it(3)
Out[82]: 6

In [83]: double_it()
```

Optional parameters (keyword or named arguments)

```
In [84]: def double_it(x=2):
    ...:     return x * 2
    ...:

In [85]: double_it()
Out[85]: 4

In [86]: double_it(3)
Out[86]: 6
```

Keyword arguments allow you to specify default values.
Warning: Default values are evaluated when the function is defined, not when it is called.

In [124]: bigx = 10

In [125]: def double_it(x=bigx):
       .....:     return x * 2
       .....:

In [126]: bigx = 1e9  # Now really big

In [128]: double_it()
Out[128]: 20

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.

3.4.4 Passed 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 whether your variables are going to be modified or not. Fortunately, there exist clear rules.

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

```python
>>> def try_to_modify(x, y, z):
    x = 23
    y.append(42)
    z = [99]  # new reference
    print(x)
    print(y)
    print(z)

>>> a = 77  # immutable variable
>>> b = [99]  # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
```

Functions have a local variable table. Called a local namespace.

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

### 3.4.5 Global variables

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

```python
In [114]: x = 5

In [115]: def addx(y):
       ....:     return x + y
       ....:

In [116]: addx(10)
Out[116]: 15
```

But these “global” variables cannot be modified within the function, unless declared `global` in the function.

This doesn’t work:

```python
In [117]: def setx(y):
       ....:     x = y
       ....:     print(’x is %d’ % x)
       ....:
       ....:

In [118]: setx(10)
x is 10
In [120]: x
Out[120]: 5
```
This works:

```python
In [121]: def setx(y):
    .....:     global x
    .....:     x = y
    .....:     print ('x is %d' % x)
    .....:

In [122]: setx(10)
x is 10
In [123]: x
Out[123]: 10
```

### 3.4.6 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

```python
In [35]: def variable_args(*args, **kwargs):
    .....:     print ('args is', args)
    .....:     print ('kwargs is', kwargs)

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

### 3.4.7 Docstrings

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

```python
In [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: function
Base Class: <type 'function'>
String Form: <function funcname at 0xeaa0f0>
Namespace: Interactive
File: /Users/cburns/src/scipy2009/.../<ipython console>
Definition: funcname(params)
Docstring:
    Concise one-line sentence describing the function.
```

3.4. Defining functions
3.4.8 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)
args is ('three',)
kwargs is {'y': 2, 'x': 1}
```

3.4.9 Methods

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

3.4.10 Exercices

**Exercice: Quicksort**

Implement the quicksort algorithm, as defined by wikipedia:

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

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

- \( u_0 = 1; u_1 = 1 \)
- \( u_{(n+2)} = u_{(n+1)} + u_n \)

3.5 Reusing code: scripts and modules

For now, we have typed all instructions in the interpreter. For longer sets of instructions we need to change tack 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)).

3.5.1 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

```python
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?
```

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:

```
epsilon:~/sandbox$ python test.py
Hello
how
are
you?
```
Standalone scripts may also take command-line arguments

In *file.py*:

```python
import sys
print sys.argv
```

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

**Note:** Don’t implement option parsing yourself. Use modules such as `optparse`.

### 3.5.2 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',
       'functions.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
```

**Warning:**

```
from os import *
```

**Do not do it.**

- Makes the code harder to read and understand: where do symbols come from?
- Makes it impossible to guess the functionality by the context and the name (hint: *os.name* is the name of the OS), and to profit usefully from tab completion.
- Restricts the variable names you can use: *os.name* might override *name*, or vice-versa.
- 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:
>>> import numpy as np  # data arrays
>>> np.linspace(0, 10, 6)
array([ 0., 2., 4., 6., 8., 10.])
>>> import scipy  # scientific computing

In Python(x,y) software, Ipython(x,y) execute the following imports at startup:

>>> import numpy
>>> import numpy as np
>>> from pylab import *
>>> import scipy
and it is not necessary to re-import these modules.

### 3.5.3 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`:

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 `test_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
In [3]: demo.print_b()
b
```

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

**Introspection**

```
In [4]: demo ?
Type: module
Base Class: <type 'module'>
String Form: <module 'demo' from 'demo.py'>
Namespace: Interactive
File: /home/varoquau/Projects/Python_talks/scipy_2009_tutorial/source/demo.py
Docstring:
   A demo module.

In [5]: who
demo

In [6]: whos
Variable  Type  Data/Info
```
demo module <module 'demo' from 'demo.py'>

In [7]: dir(demo)
Out[7]:
['__builtins__',
'__doc__',
'__file__',
'__name__',
'__package__',
'c',
'd',
'print_a',
'print_b']

In [8]: demo.
demo.__builtins__ demo.__init__ demo.__str__
demo.__class__ demo.__name__ demo.__subclasshook__
demo.__delattr__ demo.__new__ demo.c
demo.__dict__ demo.__package__ demo.d
demo.__doc__ demo.__reduce__ demo.print_a
demo.__file__ demo.__reduce_ex__ demo.print_b
demo.__format__ demo.__repr__ demo.py
demo.__getattribute__ demo.__setattr__ demo.pyc
demo.__hash__ demo.__sizeof__

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()
a

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

In [10]: reload(demo)

3.5.4 '__main__' and module loading

File demo2.py:

Importing it:
3.5.5 Scripts or modules? How to organize your code

Note: Rule of thumb

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

Note: How to import a module from a remote directory?

Many solutions exist, depending mainly on your operating system. When the import mymodule statement is executed, the module mymodule is searched in a given list of directories. This list includes a list of installation-dependent default path (e.g., /usr/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

```python
In [1]: import sys

In [2]: sys.path
Out[2]: ['','
'/usr/bin',
'/usr/local/include/enthought.traits-1.1.0',
'/usr/lib/python2.6',
'/usr/lib/python2.6/plat-linux2',
'/usr/lib/python2.6/lib-tk',
'/usr/lib/python2.6/lib-old',
'/usr/lib/python2.6/lib-dynload',
'/usr/lib/python2.6/dist-packages',
'/usr/lib/pymodules/python2.6',
'/usr/lib/pymodules/python2.6/gtk-2.0',
'/usr/lib/python2.6/dist-packages/wx-2.8-gtk2-unicode',
'/usr/local/lib/python2.6/dist-packages',
'/usr/lib/python2.6/dist-packages',
'/usr/lib/pymodules/python2.6/IPython/Extensions',
'u'/home/gouillar/.ipython']
```

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. '/usr/local/lib/python2.6/dist-packages'). You may use symbolic links (on Linux) to keep the code somewhere else.

3.5. Reusing code: scripts and modules
• 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)

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


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

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

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

See [http://docs.python.org/tutorial/modules.html](http://docs.python.org/tutorial/modules.html) for more information about modules.

### 3.5.6 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.

```
sd-2116 /usr/lib/python2.6/dist-packages/scipy $ ls
[17:07]
ccluster/   io/        README.txt  stsci/
__config__.py__ LATEST.txt setup.py __svn_version__.py__
__config__.pyc lib/ setup.pyc __svn_version__.pyc
constants/  linalg/    setupscons.py THANKS.txt
fftpack/    linsolve/  setupscons.pyc TOCHANGE.txt
__init__.py  maxentropy/ signal/ version.py
__init__.pyc misc/  sparse/  version.pyc
INSTALL.txt  ndimage/  spatial/  weave/
interpolate/  odr/  special/
interpolate/  optimize/  stats/
```

```
sd-2116 /usr/lib/python2.6/dist-packages/scipy ndimage $ cd ndimage
[17:07]
sd-2116 /usr/lib/python2.6/dist-packages/scipy/ndimage $ ls
[17:07]
doccer.py  fourier.pyc  interpolation.py  morphology.pyc  setup.pyc
doccer.pyc  info.py  interpolation.pyc  _nd_image.so
setupscons.py@
setupscons.py
setupscons.py
filters.py  info.pyc  measurements.py  _ni_support.py
setupscons.pyc
filters.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`

In [4]: scipy.version.version
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>
Namespace: Interactive
File: /usr/lib/python2.6/dist-packages/scipy/ndimage/morphology.py
Definition: morphology.binary_dilation(input, structure=None, iterations=1, mask=None, output=None, border_value=0, origin=0, 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 to one. The dilation operation is repeated iterations times. If iterations is less than 1, the dilation is repeated until the result does not change anymore. If a mask is given, only those elements with a true value at the corresponding mask element are modified at each iteration.

3.6 Input and Output

To be exhaustive, here are some informations 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:

```python
>>> f = open('workfile', 'w')  # opens the workfile file
>>> type(f)
<type 'file'>
>>> f.write('This is a test
and another test')
>>> f.close()
```

To read from a file

```python
In [1]: f = open('workfile', 'r')
In [2]: s = f.read()
In [3]: print(s)
This is a test
and another test
In [4]: f.close()
```
3.6.1 Iterating over a file

```python
In [6]: f = open('workfile', 'r')
In [7]: for line in f:
    ...:     print line
    ...:
    ...:
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: `r+`
- Binary mode: `b`
  - Note: Use for binary files, especially on Windows.

3.7 Standard Library

Note: Reference document for this section:
- Python Essential Reference, David Beazley, Addison-Wesley Professional

3.7.1 os module: operating system functionality

“A portable way of using operating system dependent functionality.”

Directory and file manipulation

Current directory:

```python
In [17]: os.getcwd()
```

List a directory:
In [31]: os.listdir(os.curdir)
Out[31]: ['.index.rst.swo',
     '.python_language.rst.swp',
     '.view_array.py.swp',
     '__static__',
     '__templates__',
     '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)
Out[46]: True
In [47]: os.remove('junk.txt')
In [48]: 'junk.txt' in os.listdir(os.curdir)
Out[48]: False

```

os.path: path manipulations

os.path provides common operations on pathnames.
```
```python
In [70]: fp = open('junk.txt', 'w')
In [71]: fp.close()
In [72]: a = os.path.abspath('junk.txt')
In [73]: a
Out[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)
Out[78]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
In [79]: os.path.basename(a)
Out[79]: 'junk.txt'
In [80]: os.path.splitext(os.path.basename(a))
Out[80]: ('junk', '.txt')
In [84]: os.path.exists('junk.txt')
Out[84]: True
In [86]: os.path.isfile('junk.txt')
Out[86]: True
In [87]: os.path.isdir('junk.txt')
Out[87]: False
In [88]: os.path.expanduser('~/local')
Out[88]: '/Users/cburns/local'
In [92]: os.path.join(os.path.expanduser('~'), 'local', 'bin')
Out[92]: '/Users/cburns/local/bin'
```

### Running an external command

```bash
In [8]: os.system('ls *')
conf.py debug_file.py demo2.py~ demo.py demo.pyc my_file.py~
conf.py~ demo2.py demo2.pyc demo.py~ my_file.py pi_wallis_image.py
```

### Walking a directory

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

```python
In [10]: for dirpath, dirnames, filenames in os.walk(os.curdir):
    ....:     for fp in filenames:
    ....:         print os.path.abspath(fp)
    ....:
    ....:
```

34 Chapter 3. A (very short) introduction to Python
Environment variables:

```python
In [9]: import os
In [11]: os.environ.keys()
Out[11]:
["/_", 
'FSLDIR', 
'TERM_PROGRAM_VERSION',
'FSLREMOTECALL',
'USER',
'HOME',
'PATH',
'PS1',
'SHELL',
'EDITOR',
'WORKON_HOME',
'PYTHONPATH',
...
```

```python
In [12]: os.environ['PYTHONPATH']
Out[12]: '.:/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'  
```

```python
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/lib/python2.5'  
```

### 3.7.2 `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.

### 3.7.3 `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')`
Out[19]: `['holy_grail.txt', 'junk.txt', 'newfile.txt']`

### 3.7.4 `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:

  ```python
  In [117]: `sys.platform`
  Out[117]: `darwin`

  In [118]: `sys.version`
  Out[118]: `2.5.2 (r252:60911, Feb 22 2008, 07:57:53) 
  [GCC 4.0.1 (Apple Computer, Inc. build 5363)]`

  In [119]: `sys.prefix`
  Out[119]: `'/Library/Frameworks/Python.framework/Versions/2.5'`
  ```

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

  ```python
  In [100]: `sys.argv`
  Out[100]: `['/Users/cburns/local/bin/ipython']`
  ```

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

```python
In [121]: `sys.path`
Out[121]: `['',
  '/Users/cburns/local/bin',
  '/Users/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',
  '/Users/cburns/local/lib/python2.5/site-packages/virtualenv-1.2-py2.5.egg',
  ...
```

### 3.7.5 `pickle`: easy persistence

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

```python
In [1]: `import pickle`

In [2]: `l = [1, None, 'Stan']`

In [3]: `pickle.dump(l, file('test.pkl', 'w'))`

In [4]: `pickle.load(file('test.pkl'))`
Out[4]: `[1, None, 'Stan']`
```
Exercise
Write a program to search your PYTHONPATH for the module site.py.

The PYTHONPATH Search Solution

3.8 Exceptions handling in Python

It is highly unlikely that you haven’t yet raised Exceptions if you have typed all the previous commands of the tutorial. For example, you may have raised an exception if you entered a command with a typo.

Exceptions are raised by different kinds of errors arising when executing Python code. In you own code, you may also catch errors, or define custom error types.

3.8.1 Exceptions

Exceptions are raised by errors in Python:

```python
In [1]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError: integer division or modulo by zero

In [2]: 1 + 'e'
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for +: 'int' and 'str'

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

In [4]: d[3]
---------------------------------------------------------------------------
KeyError: 3

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

In [6]: l[4]
---------------------------------------------------------------------------
IndexError: list index out of range

In [7]: l.foobar
---------------------------------------------------------------------------
AttributeError: 'list' object has no attribute 'foobar'
```

Different types of exceptions for different errors.

3.8.2 Catching exceptions

try/except

```python
In [8]: while True:
    ....:     try:
    ....:         x = int(raw_input(’Please enter a number: ‘))
```

3.8. Exceptions handling in Python
try/finally

```python
In [10]: try:
    ....:     x = int(raw_input('Please enter a number: '))
    ....: finally:
    ....:     print('Thank you for your input')
    ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError: invalid literal for int() with base 10: 'a'
```

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

Easier to ask for forgiveness than for permission

```python
In [11]: def print_sorted(collection):
    ....:     try:
    ....:         collection.sort()
    ....:     except AttributeError:
    ....:         pass
    ....:     print(collection)
    ....:

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

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

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

3.8.3 Raising exceptions

- Capturing and reraising an exception:
In [15]: ```python
def filter_name(name):
    try:
        name = name.encode('ascii')
    except UnicodeError, e:
        if name == 'Gaël':
            print('OK, Gaël')
        else:
            raise e
    return name
```  
In [16]: ```python
call(filter_name('Gaël'))
OK, Gaël
Out[16]: 'Gaël'
```  
In [17]: ```python
call(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]: ```python
def achilles_arrow(x):
    if abs(x - 1) < 1e-3:
        raise StopIteration
    x = 1 - (1-x)/2.
    return x
```  
In [18]: ```python
x = 0
```  
In [19]: ```python
while True:
    try:
        x = achilles_arrow(x)
    except StopIteration:
        break
```  
In [20]: ```python
x
Out[20]: 0.9990234375
```  
Use exceptions to notify certain conditions are met (e.g. StopIteration) or not (e.g. custom error raising)

### 3.9 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 functions (methods) and variables (attributes), we will be able to use:

```python
>>> class Student(object):
...     def __init__(self, name):
```
In the previous example, the Student class has \texttt{\_\_init\_}, \texttt{set\_age} and \texttt{set\_major} methods. Its attributes are \texttt{name}, \texttt{age} and \texttt{major}. We can call these methods and attributes with the following notation: \texttt{classinstance.method} or \texttt{classinstance.attribute}. The \texttt{\_\_init\_} constructor is a special method we call with: \texttt{MyClass(init parameters if any)}.

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

\begin{verbatim}
>>> class MasterStudent(Student):
...     internship = 'mandatory, from March to June'
...
>>> james = MasterStudent('james')
>>> james.internship
'mandatory, from March to June'
>>> james.set_age(23)
>>> james.age
23
\end{verbatim}

The \texttt{MasterStudent} class inherited from the \texttt{Student} attributes and methods.

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

**authors** Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux

The array: the basic tool for scientific computing

Frequent manipulation of **discrete sorted datasets**:  
- discretized time of an experiment/simulation  
- signal recorded by a measurement device  
- pixels of an image, ...

The **NumPy** module allows to  
- create such datasets in one shot  
- realize batch operations on data arrays (no loops on their items)

Data arrays := `numpy.ndarray`

4.1 Creating NumPy data arrays

A small introductory example:

```python
>>> import numpy as np
>>> a = np.array([0, 1, 2])
>>> a
array([0, 1, 2])
>>> print a
[0 1 2]
```
In practice, we rarely enter items one by one...

- **Evenly spaced values:**

  ```python
  >>> import numpy as np
  >>> a = np.arange(10) # de 0 a n-1
  >>> a
  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  >>> b = np.arange(1., 9., 2) # syntax : start, end, step
  >>> b
  array([1., 3., 5., 7.])
  ```

  or by specifying the number of points:

  ```python
  >>> c = np.linspace(0, 1, 6)
  >>> c
  array([0. , 0.2, 0.4, 0.6, 0.8, 1.0])
  >>> d = np.linspace(0, 1, 5, endpoint=False)
  >>> d
  array([0.0, 0.2, 0.4, 0.6, 0.8])
  ```

- **Constructors for common arrays:**

  ```python
  >>> a = np.ones((3,3))
  >>> a
  array([[1., 1., 1.],
          [1., 1., 1.],
          [1., 1., 1.]])
  >>> a.dtype
  dtype('float64')
  >>> b = np.ones(5, dtype=np.int)
  >>> b
  array([1, 1, 1, 1, 1])
  >>> c = np.zeros((2,2))
  >>> c
  array([[0., 0.],
          [0., 0.]])
  >>> d = np.eye(3)
  >>> d
  array([[1., 0., 0.],
          [0., 1., 0.],
          [0., 0., 1.]])
  ```

----

### 4.2 Graphical data representation: matplotlib and Mayavi

Now that we have our first data arrays, we are going to visualize them. **Matplotlib** is a 2D plotting package. We can import its functions as below:
If you launched Ipython with python(x,y), or with ipython -pylab (under Linux), all the functions/objects of pylab are already imported, without needing from pylab import *. In the remainder of this tutorial, we assume you have already run from pylab import * or ipython -pylab: as a consequence, we won’t write pylab.function() but directly function.

1D curve plotting

In [6]: a = np.arange(20)
In [7]: plot(a, a**2) # line plot
Out[7]: [<matplotlib.lines.Line2D object at 0x95abd0c>]
In [8]: plot(a, a**2, ’o’) # dotted plot
Out[8]: [<matplotlib.lines.Line2D object at 0x95b1c8c>]
In [9]: clf() # clear figure
In [10]: loglog(a, a**2)
Out[10]: [<matplotlib.lines.Line2D object at 0x95abf6c>]
In [11]: xlabel(’x’) # a bit too small
Out[11]: <matplotlib.text.Text object at 0x98923ec>
In [12]: xlabel(’x’, fontsize=26) # bigger
Out[12]: <matplotlib.text.Text object at 0x98923ec>
In [13]: ylabel(’y’)
Out[13]: <matplotlib.text.Text object at 0x9892b8c>
In [14]: grid()
In [15]: axvline(2)
Out[15]: <matplotlib.lines.Line2D object at 0x9b633cc>

2D arrays (such as images)

In [48]: # 30x30 array with random floats btw 0 and 1
In [49]: image = np.random.rand(30,30)
In [50]: imshow(image)
Out[50]: <matplotlib.image.AxesImage object at 0xe954ac>
In [51]: gray()
In [52]: hot()
In [53]: imshow(image, cmap=cm.gray)
Out[53]: <matplotlib.image.AxesImage object at 0xa23972c>
In [54]: axis(’off’) # we remove ticks and labels

4.2. Graphical data representation : matplotlib and Mayavi
There are many other features in matplotlib: color choice, marker size, latex font, inclusions within figures, histograms, etc.

To go further:
- matplotlib documentation http://matplotlib.sourceforge.net/contents.html
- an example gallery with corresponding sourcecode http://matplotlib.sourceforge.net/gallery.html

**3D plotting**

For 3D visualization, we use another package: Mayavi. A quick example: start with relaunching iPython with these options: `ipython -pylab -wthread`

```python
In [59]: from enthought.mayavi import mlab
In [60]: mlab.figure()
get fences failed: -1
param: 6, val: 0
Out[60]: <enthought.mayavi.core.scene.Scene object at 0xcb2677c>
In [61]: mlab.surf(image)
Out[61]: <enthought.mayavi.modules.surface.Surface object at 0xd0862fc>
In [62]: mlab.axes()
Out[62]: <enthought.mayavi.modules.axes.Axes object at 0xd07892c>
```

The mayavi/mlab window that opens is interactive: by clicking on the left mouse button you can rotate the image, zoom with the mouse wheel, etc.
4.3 Indexing

The items of an array can be accessed the same way as other Python sequences (list, tuple):

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

Warning! Indexes begin at 0, like other Python sequences (and C/C++). In Fortran or Matlab, indexes begin with 1.

For multidimensional arrays, indexes are tuples of integers:

```python
>>> a = np.diag(np.arange(5))
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4]])
>>> a[1,1]
1
```

Note that:

- In 2D, the first dimension corresponds to lines, the second to columns.
- for an array a with more than one dimension, ‘a[0]’ is interpreted by taking all elements in the unspecified dimensions.

### 4.4 Slicing

Like indexing, it’s similar to Python sequences slicing:

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

Note that the last index is not included!

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

`start:end:step` is a slice object which represents the set of indexes range(start, end, step). A slice can be explicitly created:

```python
>>> sl = slice(1, 9, 2)
>>> a[sl]
array([1, 3, 5, 7])
```

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

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

```python
>>> a[::2]
array([0, 2, 4, 6, 8])
```

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

Of course, it works with multidimensional arrays:
```python
>>> a = np.eye(5)
>>> a
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
>>> a[2:4,3] #3rd and 4th lines, 3 first columns
array([[ 0.,  0.,  1.],
        [ 0.,  0.,  0.]])
```

All elements specified by a slice can be easily modified:

```python
>>> a[3:,3] = 4
>>> a
array([[ 4.,  4.,  4.,  0.,  0.],
       [ 4.,  4.,  4.,  0.,  0.],
       [ 4.,  4.,  4.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

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

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

```plaintext
>>> a[4:4:]
array([[44, 45],
       [54, 55]])
```

```plaintext
>>> a[::2]
array([0, 2, 4, 6, 8])
```

```python
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b[0] = 12
>>> b
array([12, 2, 4, 6, 8, 10, 11, 13, 14, 15])
```

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

```python
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]  # b is a view on the first half of a
>>> b
array([0, 2, 4, 6, 8])
>>> b[0] = 12
>>> a
array([12, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
This behaviour can be surprising at first sight... but it allows to save a lot of memory.

## 4.5 Manipulating the shape of arrays

The shape of an array can be retrieved with the `ndarray.shape` method which returns a tuple with the dimensions of the array:

```python
>>> a = np.arange(10)
>>> a.shape
(10,)
>>> b = np.ones((3,4))
>>> b.shape
(3, 4)
>>> b.shape[0]  # the shape tuple elements can be accessed
3
>>> # an other way of doing the same
>>> np.shape(b)
(3, 4)
```

Moreover, the length of the first dimension can be queried with `np.alen` (by analogy with `len` for a list) and the total number of elements with `ndarray.size`:

```python
>>> np.alen(b)
3
>>> b.size
12
```

Several NumPy functions allow to create an array with a different shape, from another array:

```python
>>> a = np.arange(36)
>>> b = a.reshape((6, 6))
>>> b
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

`ndarray.reshape` returns a view, not a copy:

```python
>>> b[0,0] = 10
>>> a
array([10,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35])
```

An array with a different number of elements can also be created with `ndarray.resize`:  

```python
>>> a = np.arange(36)
>>> a.resize((4,2))
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])

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

A large array can be tiled with a smaller one:

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

>>> np.tile(a, (2,3))
array([[0, 1, 0, 1, 0, 1],
       [2, 3, 2, 3, 2, 3],
       [0, 1, 0, 1, 0, 1],
       [2, 3, 2, 3, 2, 3]])

4.6 Exercises: some simple array creations

By using miscellaneous constructors, indexing, slicing, and simple operations (+/-/x/), large arrays with various patterns can be created.

Example: create this array:

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
[10 11 12 13  0]
[15 16 17 18 19]
[20 21 22 23 24]]
```

Solution

```python
>>> a = np.arange(25).reshape((5,5))
>>> a[2, 4] = 0
```

Exercises: Create the following array with the simplest solution:

```
[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  2.  1.]
 [ 1.  6.  1.  1.  1.]]
```

```
[[0  0  0  0  0]
 [2  0  0  0  0]
[0  3  0  0  0]
[0  0  4  0  0]]
```
4.7 Real data: read/write arrays from/to files

Often, our experiments or simulations write some results in files. These results must then be loaded in Python as NumPy arrays to be able to manipulate them. We also need to save some arrays into files.

Going to the right folder

To move in a folder hierarchy:

1. use the iPython commands: cd, pwd, tab-completion.

In [1]: mkdir python_scripts
In [2]: cd python_scripts/
/home/gouillar/python_scripts
In [3]: pwd
Out[3]: ’/home/gouillar/python_scripts’
In [4]: ls
In [5]: np.savetxt(’integers.txt’, np.arange(10))
In [6]: ls
integers.txt

- os (system routines) and os.path (path management) modules:

```python
>>> import os, os.path
>>> current_dir = os.getcwd()
>>> current_dir
’/home/gouillar/sandbox’
>>> data_dir = os.path.join(current_dir, ’data’)
>>> data_dir
’/home/gouillar/sandbox/data’
>>> if not(os.path.exists(data_dir)):
... os.mkdir(’data’)
... print ”created ’data’ folder”
...
>>> os.chdir(data_dir) # or in Ipython : cd data
```

IPython can actually be used like a shell, thanks to its integrated features and the os module.

Writing a data array in a file

```python
>>> a = np.arange(100)
>>> a = a.reshape((10, 10))
```

- Writing a text file (in ASCII):

```python
>>> np.savetxt(’data_a.txt’, a)
```

- Writing a binary file (.npy extension, recommended format)
Loading a data array from a file

- Reading from a text file:
  ```python
  >>> b = np.loadtxt('data_a.txt')
  ```

- Reading from a binary file:
  ```python
  >>> c = np.load('data_a.npy')
  ```

To read matlab data files

`scipy.io.loadmat`: the matlab structure of a .mat file is stored as a dictionary.

Opening and saving images: `imsave` and `imread`

```python
>>> import scipy
>>> from pylab import imread, imsave, savefig
>>> lena = scipy.lena()
>>> imsave('lena.png', lena, cmap=cm.gray)
>>> lena_reloaded = imread('lena.png')
>>> imshow(lena_reloaded, cmap=gray)
<matplotlib.image.AxesImage object at 0x989e14c>
>>> savefig('lena_figure.png')
```
Selecting a file from a list

Each line of `a` will be saved in a different file:

```python
>>> for i, l in enumerate(a):
...    print i, l
...    np.savetxt('line_'+str(i), l)
...
0 [ 0  1  2  3  4  5  6  7  8  9]
1 [10 11 12 13 14 15 16 17 18 19]
2 [20 21 22 23 24 25 26 27 28 29]
3 [30 31 32 33 34 35 36 37 38 39]
4 [40 41 42 43 44 45 46 47 48 49]
5 [50 51 52 53 54 55 56 57 58 59]
6 [60 61 62 63 64 65 66 67 68 69]
7 [70 71 72 73 74 75 76 77 78 79]
8 [80 81 82 83 84 85 86 87 88 89]
9 [90 91 92 93 94 95 96 97 98 99]
```

To get a list of all files beginning with `line`, we use the `glob` module which matches all paths corresponding to a pattern. Example:

```python
>>> import glob
>>> filelist = glob.glob('line*')
>>> filelist
```
Note: arrays can also be created from Excel/Calc files, HDF5 files, etc. (but with additional modules not described here: xIrd, pytables, etc.).

### 4.8 Simple mathematical and statistical operations on arrays

Some operations on arrays are natively available in NumPy (and are generally very efficient):

```python
>>> a = np.arange(10)
>>> a.min()  # or np.min(a)
0
>>> a.max()  # or np.max(a)
9
>>> a.sum()  # or np.sum(a)
45
```

Operations can also be run along an axis, instead of on all elements:

```python
>>> a = np.array([[1, 3], [9, 6]])
>>> a
array([[1, 3],
       [9, 6]])
>>> a.mean(axis=0)  # the array contains the mean of each column
array([ 5., 4.5])
>>> a.mean(axis=1)  # the array contains the mean of each line
array([ 2., 7.5])
```

Many other operations are available. We will discover some of them in this course.

**Note:** Arithmetic operations on arrays correspond to operations on each individual element. In particular, the multiplication is not a matrix multiplication (unlike Matlab)! The matrix multiplication is provided by `np.dot`:

```python
>>> a = np.ones((2,2))
>>> a*a
array([[ 1., 1.],
       [ 1., 1.]])
>>> np.dot(a,a)
array([[ 2., 2.],
       [ 2., 2.]])
```

**Example**: diffusion simulation using a random walk algorithm

![Random Walk Diagram]

What is the typical distance from the origin of a random walker after \( t \) left or right jumps?
We find again that the distance grows like the square root of the time!

Exercise: statistics on the number of women in French research (INSEE data)

1. Get the following files `organisms.txt` and `women_percentage.txt` in the data directory.

2. Create a data array by opening the `women_percentage.txt` file with `np.loadtxt`. What is the shape of this array?

3. Columns correspond to year 2006 to 2001. Create a `years` array with integers corresponding to these years.
4. The different lines correspond to the research organisms whose names are stored in the `organisms.txt` file. Create a `organisms` array by opening this file. Beware that `np.loadtxt` creates float arrays by default, and it must be specified to use strings instead: 

```
organisms = np.loadtxt('organisms.txt', dtype=str)
```

5. Check that the number of lines of `data` equals the number of lines of the organisms.
6. What is the maximal percentage of women in all organisms, for all years taken together?
7. Create an array with the temporal mean of the percentage of women for each organism? (i.e. the mean of `data` along axis 1).
8. Which organism had the highest percentage of women in 2004? (hint: `np.argmax`)
9. Create a histogram of the percentage of women the different organisms in 2006 (hint: `np.histogram`, then `matplotlib` bar or plot for visualization)
10. Create an array that contains the organism where the highest women’s percentage is found for the different years.

**Answers** `stat_recherche`

### 4.9 Fancy indexing

Numpy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**). This method is called **fancy indexing**.

**Masks**

```python
>>> np.random.seed(3)
>>> a = np.random.randint(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,  True, False,
       True, False, True, False,  True, False, False], dtype=bool)
>>> mask = (a%3 == 0)
>>> extract_from_a = a[mask] # one could directly write a[a%3==0]
>>> extract_from_a
array([ 3,  0,  9,  6,  0, 12])
```

Extracting a sub-array using a mask produces a copy of this sub-array, not a view:

```python
>>> extract_from_a = -1
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
```

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

```python
>>> a[mask] = 0
>>> a
array([10,  0,  8,  0, 19, 10, 11,  0, 10,  0,  0, 20,  0,  7, 14])
```

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

```
>>> a[[2, 3, 2, 4, 2]]
array([5, 3, 5, 7, 5])
```

New values can be assigned with this kind of indexing:

```
>>> a[[9, 7]] = -10
>>> a
array([ 3, 1, 5, 3, 7, 5, 9, -10, 11, -10])
```

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]])
>>> a[idx]
array([[3, 4],
[9, 7]])
```

```python
>>> i = np.array([ [0,1], ...
[1,2] ]
>>> j = np.array([ [2,1], ...
[3,3] ]
>>> a[i,j]
array([[ 2, 5],
[ 7, 11]])
```
Exercise

Let’s take the same statistics about the percentage of women in the research (data and organisms arrays)

1. Create a \texttt{sup30} array of the same size than data with a value of 1 if the value of data is greater than 30%, 0 otherwise.
2. Create an array containing the organisme having the greatest percentage of women of each year.

Answers \texttt{stat_recherche}

\subsection*{4.10 Broadcasting}

Basic operations on numpy arrays (addition, etc.) are done element by element, thus work on arrays of the same size. Nevertheless, it’s 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:
which gives the following in Ipython:

```python
>>> a = np.arange(0, 40, 10)
>>> b = np.arange(0, 3)
>>> a = a.reshape((4,1))  # a must be changed into a vertical array
>>> a + b
array([[ 0, 1, 2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

We actually already used broadcasting without knowing it!

```python
>>> a = np.arange(20).reshape((4,5))
>>> a
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> a[0] = 1  # we assign an array of dimension 0 to an array of dimension 1
>>> a[:3] = np.arange(1,6)
>>> a
array([[ 1, 2, 3, 4, 5],
       [ 1, 2, 3, 4, 5],
       [ 1, 2, 3, 4, 5],
       [15, 16, 17, 18, 19]])
```
We can even use fancy indexing and broadcasting at the same time. Take again the same example as above::

```python
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array([[ 0, 1],
                 [ 1, 2]])
>>> a[i, 2] # same as a[i, 2*np.ones((2,2), dtype=int)]
array([[ 2,  6],
       [ 6, 10]])
```

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.

**Example**: 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.

```python
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544, ... 1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:,np.newaxis])
>>> distance_array
array([[ 0, 198, 303, 736, 871, 1175, 1475, 1544, 1913, 2448],
       [198,  0, 105, 433, 568, 872, 1172, 1241, 1610, 2145],
       [736, 433,   0, 135, 439, 739, 1080, 1312, 1712, 2250],
       [871, 568, 135,  0, 304, 604, 973, 1273, 1642, 2177],
       [1175, 872, 439, 304,  0, 300, 369, 738, 1273, 1712],
       [1475, 1172, 739, 604, 300,  0, 69, 438, 973, 1577],
       [1544, 1241, 808, 673, 369,  69,  0, 369, 904, 1577],
       [1913, 1715, 1610, 1177, 1042, 738, 438, 369,  0, 535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973, 904, 535,  0]])
```

**Warning**: Good practices

In the previous example, we can note some good (and bad) practices:

- **Give explicit variable names** (no need of a comment to explain what is in the variable)
- **Put spaces after commas, around =, etc.** 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** and the **Docstring Conventions** page (to manage help strings).
- **Except some rare cases, write variable names and comments in english.**
A lot of grid-based or network-based problems can also use broadcasting. For instance, if we want to compute the distance from the origin of points on a 10x10 grid, we can do:

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

The values of the distance array can be represented in colour, thanks to the `pylab.imshow` function (syntax: `pylab.imshow(distance)`. See help for other options).

![Distance array represented in colour](image)

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

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

So, `np.ogrid` is very useful as soon as we have to handle computations on a network. 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:

```python
>>> x, y = np.mgrid[0:4, 0:4]
>>> x
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])
>>> y
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])
```
4.11 Synthesis exercises: framing Lena

Let’s do some manipulations on numpy arrays by starting with the famous image of Lena (http://www.cs.cmu.edu/~chuck/lennapg/). scipy provides a 2D array of this image with the scipy.lena function:

```python
>>> import scipy
>>> lena = scipy.lena()
```

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.

```python
In [3]: import pylab
In [4]: lena = scipy.lena()
In [5]: pylab.imshow(lena)
```

- Lena is then displayed in false colors. A colormap must be specified for her to be displayed in grey.

```python
In [6]: pylab.imshow(lena, pl.cm.gray)
In [7]: # ou
In [8]: gray()
```

- Create an array of the image with a narrower centring: for example, remove 30 pixels from all the borders of the image. To check the result, display this new array with imshow.

```python
In [9]: crop_lena = lena[30:-30,30:-30]
```

- We will now frame Lena’s face with a black locket. For this, we need to
  - create a mask corresponding to the pixels we want to be black. The mask is defined by this condition
    
    $$(y-256)^2 + (x-256)^2 > 230^2$$

```python
In [15]: y, x = np.ogrid[0:512,0:512] # x and y indices of pixels
In [16]: y.shape, x.shape
Out[16]: ((512, 1), (1, 512))
In [17]: centerx, centery = (256, 256) # center of the image
In [18]: mask = ((y-centery)**2 + (x-centerx)**2) > 230**2
```
then

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

```python
In [19]: lena[mask]=0
In [20]: imshow(lena)
Out[20]: <matplotlib.image.AxesImage object at 0xa36534c>
```

- Subsidiary question: copy all instructions of this exercise in a script called `lena_locket.py` then execute this script in iPython with `%run lena_locket.py`.

**Conclusion: what do you need to know about numpy arrays to start?**

- 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. Change the shape of the array using `reshape`.
- Obtain a subset of the elements of an array and/or modify their values with masks:

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

- Know miscellaneous operations on arrays, like finding the mean or max (`array.max()`, `array.mean()`). No need to retain everything, but have the reflex to search in the documentation (see *Getting help and finding documentation*)!!
- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more functions of numpy allowing to handle array operations.
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.

```python
In [204]: help np.v
np.vander  np.vdot  np.version  np.void0  np.vstack
np.var     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; however one can always open a second Ipython shell just to display help and docstrings...


  Tutorials on various topics as well as the complete API with all docstrings are found on this website.
Numpy’s and Scipy’s documentation is enriched and updated on a regular basis by users on a wiki http://docs.scipy.org/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’s cookbook http://www.scipy.org/Cookbook gives recipes on many common problems frequently encountered, such as fitting data points, solving ODE, etc.

Matplotlib’s website http://matplotlib.sourceforge.net/ 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 well:

- 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.

```python
In [3]: import numpy as np
In [4]: %psearch np.diag*
np.diag	np.diagflat	np.diagonal
```
• `numpy.lookfor` looks for keywords inside the docstrings of specified modules.

```
In [45]: numpy.lookfor('convolution')
Search results for 'convolution'
-----------------------------
numpy.convolve
   Returns the discrete, linear convolution of two one-dimensional
   sequences.
numpy.bartlett
   Return the Bartlett window.
numpy.correlate
   Discrete, linear correlation of two 1-dimensional sequences.
In [46]: numpy.lookfor('remove', module='os')
Search results for 'remove'
---------------------------
os.remove
   remove(path)
os.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-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.
6.1 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. We also look at the class library which is provided with an object-oriented interface.

6.2 IPython

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. When we start it with the command line argument -pylab, it allows interactive matplotlib sessions that has Matlab/Mathematica-like functionality.

6.3 pylab

pylab provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab(TM). Therefore, the majority of plotting commands in pylab has Matlab(TM) analogs with similar arguments. Important commands are explained with interactive examples.

6.4 Simple Plots

Let’s start an interactive session:

```
$python ipython.py -pylab
```

This brings us to the IPython prompt:
IPython 0.8.1 -- An enhanced Interactive Python.
?   -> Introduction to IPython’s features.
%magic -> Information about IPython’s ‘magic’ % functions.
help  -> Python’s own help system.
object? -> Details about ‘object’. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type ‘help(pylab)’.

In [1]:

Now we can make our first, really simple plot:

In [1]: plot(range(10))
Out[1]: [<matplotlib.lines.Line2D instance at 0x01AA26E8>]

In [2]:

The numbers form 0 through 9 are plotted:

Now we can interactively add features to or plot:

In [2]: xlabel(‘measured’)
Out[2]: <matplotlib.text.Text instance at 0x01A9D210>

In [3]: ylabel(‘calculated’)
Out[3]: <matplotlib.text.Text instance at 0x01A9D918>

In [4]: title(‘Measured vs. calculated’)
Out[4]: <matplotlib.text.Text instance at 0x01A9DF80>

In [5]: grid(True)
In [6]:

We get a reference to our plot:
and to our line we plotted, which is the first in the plot:

```python
In [7]: line = my_plot.lines[0]
```

Now we can set properties using `set_something` methods:

```python
In [8]: line.set_marker('o')
```

or the `setp` function:

```python
In [9]: setp(line, color='g')
Out[9]: [None]
```

To apply the new properties we need to redraw the screen:

```python
In [10]: draw()
```

We can also add several lines to one plot:

```python
In [1]: x = arange(100)
In [2]: linear = arange(100)
In [3]: square = [v * v for v in arange(0, 10, 0.1)]
In [4]: lines = plot(x, linear, x, square)
```

Let’s add a legend:

```python
In [5]: legend(('linear', 'square'))
Out[5]: <matplotlib.legend.Legend instance at 0x01BBC170>
```

This does not look particularly nice. We would rather like to have it at the left. So we clean the old graph:

```python
In [6]: clf()
```

and print it anew providing new line styles (a green dotted line with crosses for the linear and a red dashed line with circles for the square graph):

```python
In [7]: lines = plot(x, linear, 'g:+', x, square, 'r--o')
```

Now we add the legend at the upper left corner:

```python
In [8]: l = legend(('linear', 'square'), loc='upper left')
```

The result looks like this:
6.4.1 Exercises

1. Plot a simple graph of a sinus function in the range 0 to 3 with a step size of 0.01.
2. Make the line red. Add diamond-shaped markers with size of 5.
3. Add a legend and a grid to the plot.

6.5 Properties

So far we have used properties for the lines. There are three possibilities to set them:

1) as keyword arguments at creation time: `plot(x, linear, 'g:+', x, square, 'r--o').`
2. with the function `setp(setp(line, color='g')).`
3. using the `set_something` methods: `line.set_marker('o')`

Lines have several properties as shown in the following table:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>alpha transparency on 0-1 scale</td>
</tr>
<tr>
<td>antialiased</td>
<td>True or False - use antialiased rendering</td>
</tr>
<tr>
<td>color</td>
<td>matplotlib color arg</td>
</tr>
<tr>
<td>data_clipping</td>
<td>whether to use numeric to clip data</td>
</tr>
<tr>
<td>label</td>
<td>string optionally used for legend</td>
</tr>
<tr>
<td>linestyle</td>
<td>one of - : -. -</td>
</tr>
<tr>
<td>linewidth</td>
<td>float, the line width in points</td>
</tr>
<tr>
<td>marker</td>
<td>one of + , o , s , v , x , &lt;, etc</td>
</tr>
<tr>
<td>markeredgewidth</td>
<td>line width around the marker symbol</td>
</tr>
<tr>
<td>markeredgecolor</td>
<td>edge color if a marker is used</td>
</tr>
<tr>
<td>markerfacecolor</td>
<td>face color if a marker is used</td>
</tr>
<tr>
<td>markersize</td>
<td>size of the marker in points</td>
</tr>
</tbody>
</table>

There are many line styles that can be specified with symbols:
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>solid line</td>
</tr>
<tr>
<td>–</td>
<td>dashed line</td>
</tr>
<tr>
<td>-.</td>
<td>dash-dot line</td>
</tr>
<tr>
<td>:</td>
<td>dotted line</td>
</tr>
<tr>
<td>.</td>
<td>points</td>
</tr>
<tr>
<td>,</td>
<td>pixels</td>
</tr>
<tr>
<td>o</td>
<td>circle symbols</td>
</tr>
<tr>
<td>^</td>
<td>triangle up symbols</td>
</tr>
<tr>
<td>v</td>
<td>triangle down symbols</td>
</tr>
<tr>
<td>&lt;</td>
<td>triangle left symbols</td>
</tr>
<tr>
<td>&gt;</td>
<td>triangle right symbols</td>
</tr>
<tr>
<td>s</td>
<td>square symbols</td>
</tr>
<tr>
<td>+</td>
<td>plus symbols</td>
</tr>
<tr>
<td>x</td>
<td>cross symbols</td>
</tr>
<tr>
<td>D</td>
<td>diamond symbols</td>
</tr>
<tr>
<td>d</td>
<td>thin diamond symbols</td>
</tr>
<tr>
<td>1</td>
<td>tripod down symbols</td>
</tr>
<tr>
<td>2</td>
<td>tripod up symbols</td>
</tr>
<tr>
<td>3</td>
<td>tripod left symbols</td>
</tr>
<tr>
<td>4</td>
<td>tripod right symbols</td>
</tr>
<tr>
<td>h</td>
<td>hexagon symbols</td>
</tr>
<tr>
<td>H</td>
<td>rotated hexagon symbols</td>
</tr>
<tr>
<td>p</td>
<td>pentagon symbols</td>
</tr>
<tr>
<td>l</td>
<td>vertical line symbols</td>
</tr>
<tr>
<td>_</td>
<td>horizontal line symbols</td>
</tr>
<tr>
<td>steps</td>
<td>use gnuplot style ‘steps’ # kwarg only</td>
</tr>
</tbody>
</table>

Colors can be given in many ways: one-letter abbreviations, gray scale intensity from 0 to 1, RGB in hex and tuple format as well as any legal html color name.

The one-letter abbreviations are very handy for quick work. With following you can get quite a few things done:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>blue</td>
</tr>
<tr>
<td>g</td>
<td>green</td>
</tr>
<tr>
<td>r</td>
<td>red</td>
</tr>
<tr>
<td>c</td>
<td>cyan</td>
</tr>
<tr>
<td>m</td>
<td>magenta</td>
</tr>
<tr>
<td>y</td>
<td>yellow</td>
</tr>
<tr>
<td>k</td>
<td>black</td>
</tr>
<tr>
<td>w</td>
<td>white</td>
</tr>
</tbody>
</table>

Other objects also have properties. The following table list the text properties:
### Property | Value
---|---
alpha | alpha transparency on 0-1 scale
color | matplotlib color arg
family | set the font family, eg sans-serif, cursive, fantasy
fontangle | the font slant, one of normal, italic, oblique
horizontalalignment | left, right or center
multialignment | left, right or center only for multiline strings
name | font name, eg, Sans, Courier, Helvetica
position | x,y location
variant | font variant, eg normal, small-caps
rotation | angle in degrees for rotated text
size | fontsize in points, eg, 8, 10, 12
style | font style, one of normal, italic, oblique
text | set the text string itself
verticalalignment | top, bottom or center
weight | font weight, e.g. normal, bold, heavy, light

### 6.5.1 Exercise

1. Apply different line styles to a plot. Change line color and thickness as well as the size and the kind of the marker. Experiment with different styles.

### 6.6 Text

We’ve already used some commands to add text to our figure: xlabel ylabel, and title.

There are two functions to put text at a defined position. `text` adds the text with data coordinates:

```python
In [2]: plot(arange(10))
In [3]: t1 = text(5, 5, 'Text in the middle')
```

`figtext` uses figure coordinates form 0 to 1:

```python
In [4]: t2 = figtext(0.8, 0.8, 'Upper right text')
```
matplotlib supports TeX mathematical expression. So $\pi$ will show up as:

$$\pi$$

If you want to get more control over where the text goes, you use annotations:

```python
In [4]: ax.annotate('Here is something special', xy = (1, 1))
```

We will write the text at the position (1, 1) in terms of data. There are many optional arguments that help to customize the position of the text. The arguments `textcoords` and `xycoords` specifies what x and y mean:

<table>
<thead>
<tr>
<th>argument</th>
<th>coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure points</td>
<td>points from the lower left corner of the figure</td>
</tr>
<tr>
<td>figure pixels</td>
<td>pixels from the lower left corner of the figure</td>
</tr>
<tr>
<td>figure fraction</td>
<td>0,0 is lower left of figure and 1,1 is upper, right</td>
</tr>
<tr>
<td>axes points</td>
<td>points from lower left corner of axes</td>
</tr>
<tr>
<td>axes pixels</td>
<td>pixels from lower left corner of axes</td>
</tr>
<tr>
<td>axes fraction</td>
<td>0,1 is lower left of axes and 1,1 is upper right</td>
</tr>
<tr>
<td>data</td>
<td>use the axes data coordinate system</td>
</tr>
</tbody>
</table>

If we do not supply `xycoords`, the text will be written at `xy`.

Furthermore, we can use an arrow whose appearance can also be described in detail:

```python
In [14]: plot(arange(10))
Out[14]: [<matplotlib.lines.Line2D instance at 0x01BB15D0>]

In [15]: ax = gca()

In [16]: ax.annotate('Here is something special', xy = (2, 1), xytext=(1,5))
Out[16]: <matplotlib.text.Annotation instance at 0x01BB1648>

In [17]: ax.annotate('Here is something special', xy = (2, 1), xytext=(1,5),
.....: arrowprops={'facecolor': 'r'})
```

### 6.6.1 Exercise

1. Annotate a line at two places with text. Use green and red arrows and align it according to `figure points` and `data`.

### 6.7 Ticks

#### 6.7.1 Where and What

Well formatted ticks are an important part of publishing-ready figures. matplotlib provides a totally configurable system for ticks. There are tick locators to specify where ticks should appear and tick formatters to make ticks look like the way 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).

#### 6.7.2 Tick Locators

There are several locators for different kind of requirements:
NullLocator | no ticks
IndexLocator | locator for index plots (e.g. where \( x = \text{range}(\text{len}(y)) \))
LinearLocator | evenly spaced ticks from min to max
LogLocator | logarithmically ticks from min to max
MultipleLocator | ticks and range are a multiple of base; either integer or float
AutoLocator |

All of these locators derive from the base class `matplotlib.ticker.Locator`. You can make your own locator deriving from it.

Handling dates as ticks can be especially tricky. Therefore, `matplotlib` provides special locators in `matplotlib.dates`:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinuteLocator</td>
<td>locate minutes</td>
</tr>
<tr>
<td>HourLocator</td>
<td>locate hours</td>
</tr>
<tr>
<td>DayLocator</td>
<td>locate specified days of the month</td>
</tr>
<tr>
<td>WeekdayLocator</td>
<td>locate days of the week, e.g. MO, TU</td>
</tr>
<tr>
<td>MonthLocator</td>
<td>locate months, e.g. 10 for October</td>
</tr>
<tr>
<td>YearLocator</td>
<td>locate years that are multiples of base</td>
</tr>
<tr>
<td>RRueLocator</td>
<td>locate using a <code>matplotlib.dates.rrule</code></td>
</tr>
</tbody>
</table>

### 6.7.3 Tick Formatters

Similarly to locators, there are formatters:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NullFormatter</td>
<td>no labels on the ticks</td>
</tr>
<tr>
<td>FixedFormatter</td>
<td>set the strings manually for the labels</td>
</tr>
<tr>
<td>FuncFormatter</td>
<td>user defined function sets the labels</td>
</tr>
<tr>
<td>FormatStrFormatter</td>
<td>use a <code>sprintf</code> format string</td>
</tr>
<tr>
<td>IndexFormatter</td>
<td>cycle through fixed strings by tick position</td>
</tr>
<tr>
<td>ScalarFormatter</td>
<td>default formatter for scalars; autopick the fmt string</td>
</tr>
<tr>
<td>LogFormatter</td>
<td>formatter for log axes</td>
</tr>
<tr>
<td>DateFormatter</td>
<td>use an <code>strtime</code> string to format the date</td>
</tr>
</tbody>
</table>

All of these formatters derive from the base class `matplotlib.ticker.Formatter`. You can make your own formatter deriving from it.

Now we set our major locator to 2 and the minor locator to 1. We also format the numbers as decimals using the `FormatStrFormatter`:

```python
In [5]: major_locator = MultipleLocator(2)
In [6]: major_formatter = FormatStrFormatter('%5.2f')
In [7]: minor_locator = MultipleLocator(1)
In [8]: ax.xaxis.set_major_locator(major_locator)
In [9]: ax.xaxis.set_minor_locator(minor_locator)
In [10]: ax.xaxis.set_major_formatter(major_formatter)
In [10]: draw()
```
After we redraw the figure our x axis should look like this:

![x-axis diagram]

### 6.7.4 Exercises

1. Plot a graph with dates for one year with daily values at the x axis using the built-in module `datetime`.
2. Format the dates in such a way that only the first day of the month is shown.
3. Display the dates with and without the year. Show the month as number and as first three letters of the month name.

### 6.8 Figures, Subplots, and Axes

#### 6.8.1 The Hierarchy

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. A `figure` in `matplotlib` means the whole window in the user interface. Within this `figure` there can be subplots. 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 work with figures and subplots without explicitly calling them. When we call `plot matplotlib` calls `gca()` to get the current axes and `gca` in turn calls `gcf()` to get the current figure. If there is none it calls `figure()` to make one, strictly speaking, to make a `subplot(111)`. Let’s look at the details.

#### 6.8.2 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 determine how the figure looks like:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>1</td>
<td>number of figure</td>
</tr>
<tr>
<td>figsize</td>
<td>figure.figsize</td>
<td>figure size in in inches (width, height)</td>
</tr>
<tr>
<td>dpi</td>
<td>figure.dpi</td>
<td>resolution in dots per inch</td>
</tr>
<tr>
<td>facecolor</td>
<td>figure.facecolor</td>
<td>color of the drawing background</td>
</tr>
<tr>
<td>edgecolor</td>
<td>figure.edgecolor</td>
<td>color of edge around the drawing background</td>
</tr>
<tr>
<td>frameon</td>
<td>True</td>
<td>draw figure frame or not</td>
</tr>
</tbody>
</table>

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.

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).

As with other objects, you can set figure properties also `setp` or with the `set_something` methods.
6.8.3 Subplots

With `subplot` you can arrange plots in regular grid. You need to specify the number of rows and columns and the number of the plot.

A plot with two rows and one column is created with `subplot(211)` and `subplot(212)`. The result looks like this:

If you want two plots side by side, you create one row and two columns with `subplot(121)` and `subplot(122)`. The result looks like this:

You can arrange as many figures as you want. A two-by-two arrangement can be created with `subplot(221), subplot(222), subplot(223), and subplot(224)`. The result looks like this:

Frequently, you don’t want all subplots to have ticks or labels. You can set the `xticklabels` or the `yticklabels` to an empty list (`[]`). Every subplot defines the methods `is_first_row`, `is_first_col`, `is_last_row`, `is_last_col`. These can help to set ticks and labels only for the outer pots.

6.8.4 Axes

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

```python
In [22]: plot(x)
Out [22]: [<matplotlib.lines.Line2D instance at 0x02C9CE90>]
```

```python
In [23]: a = axes([0.2, 0.5, 0.25, 0.25])
```

```python
In [24]: plot(x)
```
The result looks like this:

![Graph](image)

### 6.8.5 Exercises

1. Draw two figures, one 5 by 5, one 10 by 10 inches.
2. Add four subplots to one figure. Add labels and ticks only to the outermost axes.
3. Place a small plot in one bigger plot.

### 6.9 Other Types of Plots

#### 6.9.1 Many More

So far we have used only line plots. *matplotlib* offers many more types of plots. We will have a brief look at some of them. All functions have many optional arguments that are not shown here.

#### 6.9.2 Bar Charts

The function `bar` creates a new bar chart:

```python
bar([1, 2, 3], [4, 3, 7])
```

Now we have three bars starting at 1, 2, and 3 with height of 4, 3, 7 respectively:

![Bar Chart](image)

The default column width is 0.8. It can be changed with common methods by setting `width`. As it can be `color` and `bottom`, we can also set an error bar with `yerr` or `xerr`.

#### 6.9.3 Horizontal Bar Charts

The function `barh` creates an vertical bar chart. Using the same data:
barh([1, 2, 3], [4, 3, 7])

We get:

![Image of bar chart]

### 6.9.4 Broken Horizontal Bar Charts

We can also have discontinuous vertical bars with `broken_barh`. We specify start and width of the range in y-direction and all start-width pairs in x-direction:

```python
yrange = (2, 1)
xranges = ([0, 0.5], [1, 1], [4, 1])
broken_barh(xranges, yrange)
```

We changes the extension of the y-axis to make plot look nicer:

```python
ax = gca()
ax.set_ylim(0, 5)
draw()
```

and get this:

![Image of broken bar chart]

### 6.9.5 Box and Whisker Plots

We can draw box and whisker plots:

```python
boxplot((arange(2, 10), arange(1, 5)))
```

We want to have the whiskers well within the plot and therefore increase the y axis:

```python
ax = gca()
ax.set_ylim(0, 12)
draw()
```
Our plot looks like this:

![Plot](image)

The range of the whiskers can be determined with the argument `whis`, which defaults to 1.5. The range of the whiskers is between the most extreme data point within $\text{whis} \times (75\%-25\%)$ of the data.

### 6.9.6 Contour Plots

We can also do contour plots. We define arrays for the x and y coordinates:

```python
x = y = arange(10)
```

and also a 2D array for `z`:

```python
z = ones((10, 10))
z[5,5] = 7
z[2,1] = 3
z[8,7] = 4
z
```

```python
array([[ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 3., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 7., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]]
```

Now we can make a simple contour plot:

```python
contour(x, x, z)
```

Our plot looks like this:

![Contour Plot](image)

We can also fill the area. We just use numbers form 0 to 9 for the values `v`:
\[ v = x \]
\[
\text{contourf}(x, x, z, v)
\]

Now our plot area is filled:

6.9.7 Histograms

We can make histograms. Let’s get some normally distributed random numbers from `numpy`:

```python
import numpy as N
r_numbers = N.random.normal(size=1000)
```

Now we make a simple histogram:

```python
hist(r_numbers)
```

With 100 numbers our figure looks pretty good:

6.9.8 Loglog Plots

Plots with logarithmic scales are easy:

```python
loglog(arange(1000))
```

We set the major and minor grid:

```python
grid(True)
grid(True, which='minor')
```

Now we have loglog plot:
If we want only one axis with a logarithmic scale we can use `semilogx` or `semilogy`.

### 6.9.9 Pie Charts

Pie charts can also be created with a few lines:

```python
data = [500, 700, 300]
labels = ['cats', 'dogs', 'other']
pie(data, labels=labels)
```

The result looks as expected:

![Pie Chart Example](image)

### 6.9.10 Polar Plots

Polar plots are also possible. Let’s define our `r` from 0 to 360 and our `theta` from 0 to 360 degrees. We need to convert them to radians:

```python
r = arange(360)
theta = r / (180/pi)
```

Now plot in polar coordinates:

```python
polar(theta, r)
```

We get a nice spiral:

![Polar Plot Example](image)
6.9.11 Arrow Plots

Plotting arrows in 2D plane can be achieved with `quiver`. We define the x and y coordinates of the arrow shafts:

```python
x = y = arange(10)
```

The x and y components of the arrows are specified as 2D arrays:

```python
u = ones((10, 10))
v = ones((10, 10))
u[4, 4] = 3
v[1, 1] = -1
```

Now we can plot the arrows:

```python
quiver(x, y, u, v)
```

All arrows point to the upper right, except two. The one at the location (4, 4) has 3 units in x-direction and the other at location (1, 1) has -1 unit in y direction:

6.9.12 Scatter Plots

Scatter plots print x vs. y diagrams. We define x and y and make some point in y random:

```python
x = arange(10)
y = arange(10)
y[1] = 7
y[4] = 2
y[8] = 3
```

Now make a scatter plot:

```python
scatter(x, y)
```

The three different values for y break out of the line:
6.9.13 Sparsity Pattern Plots

Working with sparse matrices, it is often of interest as how the matrix looks like in terms of sparsity. We take an identity matrix as an example:

```python
i = identity(10)
i
array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
```

Now we look at it more visually:

```python
spy(i)
```

![Sparsity Pattern Plot](image)

6.9.14 Stem Plots

Stem plots vertical lines at the given x location up to the specified y location. Let’s reuse $x$ and $y$ from our scatter (see above):

```python
stem(x, y)
```

![Stem Plot](image)

6.9.15 Date Plots

There is a function for creating date plots. Let’s define 10 dates starting at January 1st 2000 at 15.day intervals:
import datetime
d1 = datetime.datetime(2000, 1, 1)
delta = datetime.timedelta(15)
dates = [d1 + x * delta for x in range(1, 12)]

We reuse our data from the scatter plot (see above):

```python
y = array([0, 7, 2, 3, 2, 5, 6, 7, 3, 9])
```

Now we can plot the dates at the x axis:

```python
plot_date(dates, y)
```

### 6.10 The Class Library

So far we have used the `pylab` interface only. As the name suggests it is just wrapper around the class library. All `pylabb` commands can be invoked via the class library using an object-oriented approach.

#### 6.10.1 The Figure Class

The class `Figure` lives in the module `matplotlib.figure`. Its constructor takes these arguments:

```python
figsize=None, dpi=None, facecolor=None, edgecolor=None,
linewidth=1.0, frameon=True, subplotpars=None
```

Comparing this with the arguments of `figure` in `pylab` shows significant overlap:

```python
num=None, figsize=None, dpi=None, facecolor=None,
edgecolor=None, frameon=True
```
Figure provides lots of methods, many of them have equivalents in `pylab`. The methods `add_axes` and `add_subplot` are called if new axes or subplot are created with `axes` or `subplot` in `pylab`. Also the method `gca` maps directly to `pylab` as do `legend`, `text` and many others.

There are also several `set_something` method such as `set_facecolor` or `set_edgecolor` that will be called through `pylab` to set properties of the figure. Figure also implements `get_something` methods such as `get_axes` or `get_facecolor` to get properties of the figure.

### 6.10.2 The Classes Axes and Subplot

In the class `Axes` we find most of the figure elements such as `Axis`, `Tick`, `Line2D`, or `Text`. It also sets the coordinate system. The class `Subplot` inherits from `Axes` and adds some more functionality to arrange the plots in a grid.

Analogous to `Figure`, it has methods to get and set properties and methods already encountered as functions in `pylab` such as `annotate`. In addition, `Axes` has methods for all types of plots shown in the previous section.

### 6.10.3 Other Classes

Other classes such as `text`, `Legend` or `Ticker` are setup very similarly. They can be understood mostly by comparing to the `pylab` interface.

### 6.10.4 Example

Let’s look at an example for using the object-oriented API:

```python
# file matplotlib/oo.py

from matplotlib.figure import Figure

figsize = (8, 5)
fig = Figure(figsize=figsize)
ax = fig.add_subplot(111)
line = ax.plot(range(10))[0]
ax.set_title('Plotted with OO interface')
ax.set_xlabel('measured')
ax.set_ylabel('calculated')
ax.grid(True)
line.set_marker('o')

from matplotlib.backends.backend_agg import FigureCanvasAgg
canvas = FigureCanvasAgg(fig)
canvas.print_figure("oo.png", dpi=80)

import Tkinter as Tk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

root = Tk.Tk()
canvas2 = FigureCanvasTkAgg(fig, master=root)
canvas2.show()
canvas2.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
Tk.mainloop()
```

---

### 6.10. The Class Library

85
Since we are not in the interactive pylab-mode, we need to import the class `Figure` explicitly (#1).

We set the size of our figure to be 8 by 5 inches (#2). Now we initialize a new figure (#3) and add a subplot to the figure (#4). The 111 says one plot at position 1, 1 just as in MATLAB. We create a new plot with the numbers from 0 to 9 and at the same time get a reference to our line (#5). We can add several things to our plot. So we set a title and labels for the x and y axis (#6).

We also want to see the grid (#7) and would like to have little filled circles as markers (#8).

There are many different backends for rendering our figure. We use the Anti-Grain Geometry toolkit (http://www.antigrain.com) to render our figure. First, we import the backend (#9), then we create a new canvas that renders our figure (#10). We save our figure in a png-file with a resolution of 80 dpi (#11).

We can use several GUI toolkits directly. So we import Tkinter (#12) as well as the corresponding backend (#13). Now we have to do some basic GUI programming work. We make a root object for our GUI (#13) and feed it together with our figure to the backend to get our canvas (14). We call the show method (#15), pack our widget (#16), and call the Tkinter mainloop to start the application (#17). You should see GUI window with the figure on your screen. After closing the screen, the next part, the script, will be executed.

We would like to create a screen display just as we would use `pylab`. Therefore we import a helper (#18) and `pylab` itself (#19). We create a normal figure with `pylab` (#20) and get the corresponding figure manager (#21). Now let’s set our figure we created above to be the current figure (#22) and let `pylab` show the result (#23). The lower part of the figure might be cover by the toolbar. If so, please adjust the `figsize` for `pylab` accordingly.

### 6.10.5 Exercises

1. Use the object-oriented API of matplotlib to create a png-file with a plot of two lines, one linear and square with a legend in it.
Scipy: high-level scientific computing

authors  Adrien Chauve, Andre Espaze, Emmanuelle Gouillart, Gaël Varoquaux

Scipy

The scipy package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc. 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, if is worth checking if the desired data processing is not already implemented in Scipy. As non-professional programmers, scientists often tend to re-invent the wheel, which leads to buggy, non-optimal, difficult-to-share and unmaintainable code. By contrast, Scipy's routines are optimized and tested, and should therefore be used when possible.

Warning: This tutorial is far from an introduction to numerical computing. As enumerating the different sub-modules 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.

To begin with

>>> import numpy as np
>>> import scipy

scipy is mainly composed of task-specific sub-modules:
### 7.1 Scipy builds upon Numpy

Numpy is required for running Scipy but also for using it. The most important type introduced to Python is the N-dimensional array, and it can be seen that Scipy uses the same:

```python
>>> scipy.ndarray is np.ndarray
True
```

Moreover most of the Scipy usual functions are provided by Numpy:

```python
>>> scipy.cos is np.cos
True
```

If you would like to know the objects used from Numpy, have a look at the `scipy.__file__[:-1]` file. On version ‘0.6.0’, the whole Numpy namespace is imported by the line `from numpy import *`.

### 7.2 File input/output: `scipy.io`

- Loading and saving matlab files:

  ```python
  >>> from scipy import io
  >>> struct = io.loadmat('file.mat', struct_as_record=True)
  >>> io.savemat('file.mat', struct)
  ```

See also:

- Load text files:

  ```python
  np.loadtxt/np.savetxt
  ```

- Clever loading of text/csv files:

  ```python
  np.genfromtxt/np.recfromcsv
  ```

- Fast an efficient, but numpy-specific, binary format:
7.3 Signal processing: `scipy.signal`

```python
>>> from scipy import signal
```

- Detrend: remove linear trend from signal:

```python
t = np.linspace(0, 5, 100)
x = t + np.random.normal(size=100)
pl.plot(t, x, linewidth=3)
pl.plot(t, signal.detrend(x), linewidth=3)
```

- Resample: resample a signal to \( n \) points using FFT.

```python
t = np.linspace(0, 5, 100)
x = np.sin(t)
pl.plot(t, x, linewidth=3)
pl.plot(t[::-2], signal.resample(x, 50), 'ko')
```

---

7.3. Signal processing: `scipy.signal` 89
Notice how on the side of the window the resampling is less accurate and has a rippling effect.

- Signal has many window function: `hamming`, `bartlett`, `blackman`...
- Signal has filtering (Gaussian, median filter, Wiener), but we will discuss this in the image paragraph.

### 7.4 Special functions: `scipy.special`

Special functions are transcendental functions. The docstring of the module is well-written and we will not list them. Frequently used ones are:

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

### 7.5 Statistics and random numbers: `scipy.stats`

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

#### 7.5.1 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):

```python
>>> 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)
>>> bins = 0.5*(bins[1:] + bins[:-1])
>>> bins
array([-3.5, -2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5])
>>> from scipy import stats
>>> b = stats.norm.pdf(bins)
```

```python
In [1]: pl.plot(bins, histogram)
In [2]: pl.plot(bins, b)
```
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:

```python
>>> loc, std = stats.norm.fit(a)
>>> loc
0.003738964114102075
>>> std
0.9745099666887119
```

### 7.5.2 Percentiles

The median is the value with half of the observations below, and half above:

```python
>>> np.median(a)
0.007164557029278252
```

It is also called the percentile 50, because 50% of the observation are below it:

```python
>>> stats.scoreatpercentile(a, 50)
0.007164557029278252
```

Similarly, we can calculate the percentile 90:

```python
>>> stats.scoreatpercentile(a, 90)
1.2729556087871292
```

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

### 7.5.3 Statistical tests

A statistical test is a decision indicator. For instance, if we have 2 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:

```python
>>> a = np.random.normal(0, 1, size=100)
>>> b = np.random.normal(1, 1, size=10)
>>> stats.ttest_ind(a, b)
(-2.389876434401887, 0.018586471712806949)
```
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 process 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 mean.

### 7.6 Linear algebra operations: `scipy.linalg`

First, the `linalg` module provides standard linear algebra operations. The `det` function computes the determinant of a square matrix:

```python
>>> from scipy import linalg
>>> arr = np.array([[1, 2],
...                  [3, 4]])
>>> linalg.det(arr)
-2.0
```

```python
>>> arr = np.array([[3, 2],
...                  [6, 4]])
>>> linalg.det(arr)
0.0
```

```python
>>> linalg.det(np.ones((3, 4)))
Traceback (most recent call last):
...
ValueError: expected square matrix
```

The `inv` function computes the inverse of a square matrix:

```python
>>> arr = np.array([[1, 2],
...                  [3, 4]])
>>> iarr = linalg.inv(arr)
>>> iarr
array([-2. , 1. ],
      [ 1.5, -0.5])
```

```python
>>> np.allclose(np.dot(arr, iarr), np.eye(2))
True
```

Note that in case you use the matrix type, the inverse is computed when requesting the $I$ attribute:

```python
>>> ma = np.matrix(arr, copy=False)
>>> np.allclose(ma.I, iarr)
True
```

Finally computing the inverse of a singular matrix (its determinant is zero) will raise `LinAlgError`:

```python
>>> arr = np.array([[3, 2],
...                  [6, 4]])
>>> linalg.inv(arr)
Traceback (most recent call last):
...
LinAlgError: singular matrix
```

More advanced operations are available like singular-value decomposition (SVD):
The resulting array spectrum is:

```
>>> spec
array([ 2.54368356e+01, 1.72261225e+00, 5.14037515e-16])
```

For the recomposition, an alias for manipulating matrix will first be defined:

```
>>> asmat = np.asmatrix

then the steps are:

```
>>> sarr = np.zeros((3, 4))
>>> sarr.put((0, 5, 10), spec)
>>> svd_mat = asmat(uarr) * asmat(sarr) * asmat(vharr)
>>> np.allclose(svd_mat, arr)
True
```

SVD is commonly used in statistics or signal processing. Many other standard decompositions (QR, LU, Cholesky, Schur), as well as solvers for linear systems, are available in scipy.linalg.

### 7.7 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)
>>> np.allclose(res, 1)
True
>>> np.allclose(err, 1 - res)
True
```

Others integration schemes are available with fixed_quad, quadrature, romberg.

scipy.integrate also features routines for Ordinary differential equations (ODE) integration. In particular, `scipy.integrate.odeint` is a general-purpose integrator using LSODA (Livermore solver for ordinary differential equations with automatic method switching for stiff and nonstiff problems), see the ODEPACK Fortran library for more details.

odeint solves first-order ODE systems of the form:

```
```
```
As an introduction, let us solve the ODE \( \frac{dy}{dt} = -2y \) 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 += 1
...   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:

```python
>>> counter = np.zeros((1,), np.uint16)
```

The trajectory will now be computed:

```python
>>> from scipy.integrate import odeint
>>> time_vec = np.linspace(0, 4, 40)
>>> yvec, info = odeint(calc_derivative, 1, time_vec,
...                      args=(counter,), full_output=True)
```

Thus the derivative function has been called more than 40 times:

```python
>>> counter
array([129], dtype=uint16)
```

and the cumulative iterations number for the 10 first convergences can be obtained by:

```python
>>> info['nfe'][:10]
array([31, 35, 43, 49, 53, 57, 59, 63, 65, 69], dtype=int32)
```

The solver requires more iterations at start. The final trajectory is seen on the Matplotlib figure computed with odeint-introduction.py.
Another example with `odeint` will be a damped spring-mass oscillator (2nd order oscillator). The position of a mass attached to a spring obeys the 2nd order ODE \( y'' + 2 \varepsilon \omega_0 y' + \omega_0^2 y = 0 \) with \( \omega_0^2 = k/m \) being \( k \) the spring constant, \( m \) the mass and \( \varepsilon = c/(2 m \omega_0) \) with \( c \) the damping coefficient. For a computing example, the parameters will be:

```python
>>> mass = 0.5 # kg
>>> kspring = 4 # N/m
>>> cviscous = 0.4 # N s/m
```

so the system will be underdamped because:

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

For the `odeint` solver the 2nd order equation needs to be transformed in a system of two first-order equations for the vector \( Y = (y, y') \). It will be convenient to define \( \nu = 2 \varepsilon \omega_0 = c/m \) and \( \omega = \omega_0^2 = k/m \):

```python
>>> nu_coef = cviscous/mass
>>> om_coef = kspring/mass
```

Thus the function will calculate the velocity and acceleration by:

```python
>>> def calc_deri(yvec, time, nuc, omc):
...     return (yvec[1], -nuc * yvec[1] - omc * yvec[0])
...
>>> time_vec = np.linspace(0, 10, 100)
>>> yarr = odeint(calc_deri, (1, 0), time_vec, args=(nu_coef, om_coef))
```

The final position and velocity are shown on a Matplotlib figure built with the `odeint-damped-spring-mass.py` script.
There is no Partial Differential Equations (PDE) solver in scipy. Some PDE packages are written in Python, such as fipy or SfePy.

### 7.8 Fast Fourier transforms: `scipy.fftpack`

The `fftpack` module allows to compute fast Fourier transforms. As an illustration, an input signal may look like:

```python
>>> time_step = 0.1
>>> period = 5.
>>> time_vec = np.arange(0, 20, time_step)
>>> sig = np.sin(2 * np.pi / period * time_vec) + np.cos(10 * np.pi * time_vec)
```

However the observer does not know the signal frequency, only the sampling time step of the signal `sig`. But the signal is supposed to come from a real function so the Fourier transform will be symmetric. The `fftfreq` function will generate the sampling frequencies and `fft` will compute the fast fourier transform:

```python
>>> from scipy import fftpack
>>> sample_freq = fftpack.fftfreq(sig.size, d=time_step)
>>> sig_fft = fftpack.fft(sig)
```

Nevertheless only the positive part will be used for finding the frequency because the resulting power is symmetric:
Thus the signal frequency can be found by:

```python
>>> freq = freqs[power.argmax()]
>>> np.allclose(freq, 1./period)
True
```

Now only the main signal component will be extracted from the Fourier transform:

```python
>>> sig_fft[np.abs(sample_freq) > freq] = 0
```

The resulting signal can be computed by the `ifft` function:

```python
>>> main_sig = fftpack.ifft(sig_fft)
```

The result is shown on the Matplotlib figure generated by the `fftpack-illustration.py` script.
7.9 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 sinus function:

```python
>>> measured_time = np.linspace(0, 1, 10)
>>> noise = (np.random.random(10)*2 - 1) * 1e-1
>>> measures = np.sin(2 * np.pi * measured_time) + noise
```

The `interp1d` class can build a linear interpolation function:

```python
>>> from scipy.interpolate import interp1d
>>> linear_interp = interp1d(measured_time, measures)
```

Then the `linear_interp` instance needs to be evaluated on time of interest:

```python
>>> 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:
```python
>>> cubic_interp = interp1d(measured_time, measures, kind='cubic')
>>> cubic_results = cubic_interp(computed_time)
```

The results are now gathered on a Matplotlib figure generated by the script scipy-interpolation.py.

![Interpolation Figure](image)

scipy.interpolate.interp2d is similar to 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 exercise on ‘Maximum wind speed prediction at the Sprogø station’ for a more advance spline interpolation example.

### 7.10 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.

**Example: Minimizing a scalar function using different algorithms**

Let’s define the following function:

```python
>>> def f(x):
...     return x**2 + 10*np.sin(x)
```
and plot it:

```python
>>> x = 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.

### 7.10.1 Local (convex) optimization

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:

```python
>>> optimize.fmin_bfgs(f, 0)
Optimization terminated successfully.
Current function value: -7.945823
Iterations: 5
Function evaluations: 24
Gradient evaluations: 8
array([-1.30644003])
```

This resolution takes 4.11ms on our computer.
The problem with this approach is that, if the function has local minima (is not convex), the algorithm may find these local minima instead of the global minimum depending on the initial point. If we don’t know the neighborhood of the global minima to choose the initial point, we need to resort to costlier global optimization.

### 7.10.2 Global optimization

To find the global minimum, the simplest algorithm is the brute force algorithm, in which the function is evaluated on each point of a given grid:

```python
>>> from scipy import optimize
>>> grid = (-10, 10, 0.1)
>>> optimize.brute(f, (grid,))
array([-1.30641113])
```

This approach takes 20 ms on our computer.

This simple algorithm becomes very slow as the size of the grid grows, so you should use `optimize.brent` instead for scalar functions:

```python
>>> optimize.brent(f)
-1.3064400120612139
```

To find the local minimum, let’s add some constraints on the variable using `optimize.fminbound`:

```python
>>> # search the minimum only between 0 and 10
>>> optimize.fminbound(f, 0, 10)
array([ 3.83746712])
```

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

See the summary exercise on *Nonlinear least squares curve fitting: application to point extraction in topographical lidar data* for a more advanced example.

### 7.11 Image processing: *scipy.ndimage*

The submodule dedicated to image processing in `scipy` is `scipy.ndimage`.

```python
>>> from scipy import ndimage
```

Image processing routines may be sorted according to the category of processing they perform.

#### 7.11.1 Geometrical transformations on images

Changing orientation, resolution, ..

```python
>>> import scipy
>>> lena = scipy.lena()
>>> shifted_lena = ndimage.shift(lena, (50, 50))
>>> shifted_lena2 = ndimage.shift(lena, (50, 50), mode='nearest')
>>> rotated_lena = ndimage.rotate(lena, 30)
>>> cropped_lena = lena[50:-50, 50:-50]
>>> zoomed_lena = ndimage.zoom(lena, 2)
```
7.11.2 Image filtering

```
>>> lena = scipy.lena()
```

```
>>> import numpy as np
>>> noisy_lena = np.copy(lena)
>>> noisy_lena += lena.std()*0.5*np.random.standard_normal(lena.shape)
>>> blurred_lena = ndimage.gaussian_filter(noisy_lena, sigma=3)
>>> median_lena = ndimage.median_filter(blurred_lena, size=5)
>>> import scipy.signal
>>> wiener_lena = scipy.signal.wiener(blurred_lena, (5,5))
```

And many other filters in `scipy.ndimage.filters` and `scipy.signal` can be applied to images.

**Exercise**

Compare histograms for the different filtered images.
7.11.3 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 theory: the sets to be transformed are the sets of neighboring non-zero-valued pixels. The theory was also 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

```python
>>> el = ndimage.generate_binary_structure(2, 1)
```

```console
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
```

```python
>>> el.astype(np.int)
```

```console
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```

- Erosion

```python
>>> 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]])
>>> ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

#Erosion removes objects smaller than the structure

```python
>>> ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 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]])
```

---

7.11. Image processing: scipy.ndimage
• **Dilation**

```python
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[ 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0.],
       [ 0., 0., 1., 0., 0.],
       [ 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0.]]),
>>> ndimage.binary_dilation(a).astype(a.dtype)
array([[ 0., 0., 0., 0., 0.],
       [ 0., 0., 1., 0., 0.],
       [ 0., 1., 1., 1., 0.],
       [ 0., 0., 1., 0., 0.],
       [ 0., 0., 0., 0., 0.]]))
```

• **Opening**

```python
>>> a = np.zeros((5, 5), dtype=np.int)
>>> a[1:4, 1:4] = 1; a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
>>> ndimage.binary_opening(a, structure=np.ones((3,3))).astype(np.int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> ndimage.binary_opening(a).astype(np.int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

• **Closing**: `ndimage.binary_closing`
An opening operation removes small structures, while a closing operation fills small holes. Such operation can therefore be used to “clean” an image.

```python
>>> a = np.zeros((50, 50))
>>> a[10:-10, 10:-10] = 1
>>> a += 0.25*np.random.standard_normal(a.shape)
>>> mask = a>=0.5
>>> opened_mask = ndimage.binary_opening(mask)
>>> closed_mask = ndimage.binary_closing(opened_mask)
```

Exercise

Check that the area of the reconstructed square is smaller than the area of the initial square. (The opposite would occur if the closing step was performed before the opening).

For gray-valued images, eroding (resp. dilating) amounts to replacing a pixel by the minimal (resp. maximal) value among pixels covered by the structuring element centered on the pixel of interest.

```python
>>> a = np.zeros((7,7), dtype=np.int)
>>> a[1:6, 1:6] = 3
>>> a[4,4] = 2; a[2,3] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 1, 3, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 3, 2, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_erosion(a, size=(3,3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 3, 2, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

7.11.4 Measurements on images

Let us first generate a nice synthetic binary image.
```python
def bins(counts, binsize):
    return (binsize / 2 + binsize * np.arange(1, len(bins) + 1)), counts
```

This returns the bin edges and the counts in each bin.

```python
counts, bins = np.histogram(values, bins)
edges, counts = bins(counts, binsize)
```

Now we look for various information about the objects in the image:

```python
labels, nb = ndimage.label(mask)
nb
8
areas = ndimage.sum(mask, labels, xrange(1, labels.max()+1))
maxima = ndimage.maximum(sig, labels, xrange(1, labels.max()+1))
maxima
[1.8023823799830032, 1.1352760475048373, 5.5195407887291426, 2.4961181804217221, 16.765472169131161, 5.5195407887291426]
ndimage.find_objects(labels==4)
[(slice(30, 48, None), slice(30, 48, None))]
s1 = ndimage.find_objects(labels==4)
imshow(sig[s1[0]])
```

See the summary exercise on *Image processing application: counting bubbles and unmolten grains* for a more advanced example.

### 7.12 Summary exercises on scientific computing

The summary exercises use mainly Numpy, Scipy and Matplotlib. They first aim at providing real life examples on scientific computing with Python. Once the groundwork is introduced, the interested user is invited to try some exercises.

#### 7.12.1 Maximum wind speed prediction at the Sprogø station

The exercise goal is to predict the maximum wind speed occuring every 50 years even if no measure exists for such a period. The available data are only measured over 21 years at the Sprogø meteorological station located in Denmark. First, the statistical steps will be given and then illustrated with functions from the scipy.interpolate module. At the end the interested readers are invited to compute results from raw data and in a slightly different approach.
Statistical approach

The annual maxima are supposed to fit a normal probability density function. However such function is not going to be estimated because it gives a probability from a wind speed maxima. Finding the maximum wind speed occuring every 50 years requires the opposite approach, the result needs to be found from a defined probabilty. That is the quantile function role and the exercice goal will be to find it. In the current model, it is supposed that the maximum wind speed occuring 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 exercice, the cumulative probabilty $p_i$ for a given year $i$ is defined as $p_i = i/(N+1)$ with $N = 21$, the number of measured years. Thus it will be possible to calculate the cumulative probability of every measured wind speed maxima. From those experimental points, the scipy.interpolate module will be very useful for fitting the quantile function. Finally the 50 years maxima is going to be evaluated from the cumulative probability of the 2% quantile.

Computing the cumulative probabilities

The annual wind speeds maxima have already been computed and saved in the numpy format in the file max-speeds.npy, thus they will be loaded by using numpy:

```python
>>> import numpy as np
>>> max_speeds = np.load('data/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:

```python
>>> cprob = (np.arange(years_nb, dtype=np.float32) + 1)/(years_nb + 1)
```

and they are assumed to fit the given wind speeds:

```python
>>> sorted_max_speeds = np.sort(max_speeds)
```

Prediction with UnivariateSpline

In this section the quantile function will be estimated by using the `UnivariateSpline` class which can represent a spline from points. The default behavior is to build a spline of degree 3 and points can have different weights according to their reliability. Variante are `InterpolatedUnivariateSpline` and `LSQUnivariateSpline` on which errors checking is going to change. In case a 2D spline is wanted, the `BivariateSpline` class family is provided. All thoses classes for 1D and 2D splines use the FITPACK Fortran subroutines, that’s why a lower library access is available through the `splrep` and `splev` functions for respectively representing and evaluating a spline. Moreover interpolation functions without the use of FITPACK parameters are also provided for simpler use (see `interp1d`, `interp2d`, `barycentric_interpolate` and so on).

For the Sprogø maxima wind speeds, the `UnivariateSpline` will be used because a spline of degree 3 seems to correctly fit the data:

```python
>>> 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:

```python
>>> nprob = np.linspace(0, 1, 1e2)
>>> fitted_max_speeds = quantile_func(nprob)
```
In the current model, the maximum wind speed occurring every 50 years $V_{50}$ is defined as the upper 2% quantile. As a result, the cumulative probability value will be:

```python
>>> fifty_prob = 1. - 0.02
```

So the storm wind speed occurring every 50 years can be guessed by:

```python
>>> fifty_wind = quantile_func(fifty_prob)
>>> fifty_wind
array([ 32.97989825])
```

The results are now gathered on a Matplotlib figure.

All those steps have been gathered in the script `cumulative-wind-speed-prediction.py`.

**Exercice with the Gumbell distribution**

The interested readers are now invited to make an exercise by using the wind speeds measured over 21 years. The measurement period is around 90 minutes (the original period was around 10 minutes but the file size has been reduced for making the exercise setup easier). The data are stored in numpy format inside the file `sprog-windspeeds.npy`.

- The first step will be to find the annual maxima by using numpy and plot them as a matplotlib bar figure.
• The second step will be to use the Gumbell distribution on cumulative probabilities \( p_i \) defined as \(-\log(-\log(p_i))\) for fitting a linear quantile function (remember that you can define the degree of the UnivariateSpline). Plotting the annual maxima versus the Gumbell distribution should give you the following figure.
• The last step will be to find 34.23 m/s for the maximum wind speed occurring every 50 years. Once done, you may compare your code with a solution example available in the script gumbell-wind-speed-prediction.py.

7.12.2 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 practise 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 them emit a short light impulsion towards a target and record the reflected signal. This signal is then processed to extract the distance between the lidar system and the target.

Topographical lidar systems are such systems embedded in airborne platforms. They measure distances between the platform and the Earth, so as to deliver information on the Earth’s topography (see [Mallet09] for more details).

In this tutorial, the goal is to analyze the waveform recorded by the lidar system ¹. Such a signal contains peaks whose

¹ The data used for this tutorial are part of the demonstration data available for the FullAnalyze software and were kindly provided by the GIS
center and amplitude permit to compute the position and some characteristics of the hit target. When the footprint of
the laser beam is around 1m on the Earth surface, the beam can hit multiple targets during the two-way propagation
(for example the ground and the top of a tree or building). The sum of the contributions of each target hit by the laser
beam then produces a complex signal with multiple peaks, each one containing information about one target.

One state of the art method to extract information from these data is to decompose them in a sum of Gaussian functions
where each function represents the contribution of a target hit by the laser beam.

Therefore, we use the \texttt{scipy.optimize} module to fit a waveform to one or a sum of Gaussian functions.

\textbf{Loading and visualization}

Load the first waveform using:

\begin{verbatim}
>>> import numpy as np
>>> waveform_1 = np.load('data/waveform_1.npy')
\end{verbatim}

and visualize it:

\begin{verbatim}
>>> import matplotlib.pyplot as plt
>>> t = np.arange(len(waveform_1))
>>> plt.plot(t, waveform_1)
>>> plt.show()
\end{verbatim}
As you can notice, this waveform is a 80-bin-length signal with a single peak.

**Fitting a waveform with a simple Gaussian model**

The signal is very simple and can be modelled as a single Gaussian function and an offset corresponding to the background noise. To fit the signal with the function, we must:

- define the model
- propose an initial solution
- call `scipy.optimize.leastsq`

**Model**

A gaussian function defined by

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

can be defined in python by:

```python
>>> def model(t, coeffs):
...     return coeffs[0] + coeffs[1] * np.exp(-((t-coeffs[2])/coeffs[3])**2)
```

where

- `coeffs[0]` is \( B \) (noise)
- `coeffs[1]` is \( A \) (amplitude)
- `coeffs[2]` is \( \mu \) (center)
- `coeffs[3]` is \( \sigma \) (width)

**Initial solution**

An approximative initial solution that we can find from looking at the graph is for instance:

```python
>>> x0 = np.array([3, 30, 15, 1], dtype=float)
```

**Fit**

`scipy.optimize.leastsq` minimizes the sum of squares of the function given as an argument. Basically, the function to minimize is the residuals (the difference between the data and the model):

```python
>>> 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

```python
>>> from scipy.optimize import leastsq
>>> x, flag = leastsq(residuals, x0, args=(waveform_1, t))
>>> print x
[ 2.70363341  27.82020742  15.47924562  3.05636228]
```

And visualize the solution:

```python
>>> plt.plot(t, waveform_1, t, model(t, x))
>>> plt.legend(['waveform', 'model'])
>>> plt.show()
```

Remark: from scipy v0.8 and above, you should rather use `scipy.optimize.curve_fit` which takes the model and the data as arguments, so you don’t need to define the residuals any more.

**Going further**

• Try with a more complex waveform (for instance `data/waveform_2.npy`) that contains three significant peaks. You must adapt the model which is now a sum of Gaussian functions instead of only one Gaussian peak.

In some cases, writing an explicit function to compute the Jacobian is faster than letting `leastsq` estimate it numerically. Create a function to compute the Jacobian of the residuals and use it as an input for `leastsq`. 
• When we want to detect very small peaks in the signal, or when the initial guess is too far from a good solution, the result given by the algorithm is often not satisfying. Adding constraints to the parameters of the model enables to overcome such limitations. An example of a priori knowledge we can add is the sign of our variables (which are all positive).

With the following initial solution:

```python
>>> 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.

### 7.12.3 Image processing application: counting bubbles and unmolten grains

#### Statement of the problem

1. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the docstring of `imshow` to display the image with the “right” orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).

This Scanning Element Microscopy image shows a glass sample (light gray matrix) with some bubbles (on black) and unmolten sand grains (dark gray). We wish to determine the fraction of the sample covered by these three phases, and to estimate the typical size of sand grains and bubbles, their sizes, etc.

2. Crop the image to remove the lower panel with measure information.

3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.
4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option (homework): write a function that determines automatically the thresholds from the minima of the histogram.

5. Display an image in which the three phases are colored with three different colors.

6. Use mathematical morphology to clean the different phases.

7. Attribute labels to all bubbles and sand grains, and remove from the sand mask grains that are smaller than 10 pixels. To do so, use `ndimage.sum` or `np.bincount` to compute the grain sizes.

8. Compute the mean size of bubbles.

**Proposed solution**

**7.12.4 Example of solution for the image processing exercise: unmolten grains in glass**

1. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the docstring of `imshow` to display the image with the “right” orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).

```python
>>> dat = imread('MV_HFV_012.jpg')
```

2. Crop the image to remove the lower panel with measure information.

```python
>>> dat = dat[60:]
```

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.

```python
>>> void = filtdat <= 50
>>> sand = np.logical_and(filtdat>50, filtdat<=114)
>>> glass = filtdat > 114
```

5. Display an image in which the three phases are colored with three different colors.

```python
>>> phases = void.astype(np.int) + 2*glass.astype(np.int) +
  3*sand.astype(np.int)
```
6. Use mathematical morphology to clean the different phases.

```python
>>> 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.

```python
>>> 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.

```python
>>> bubbles_labels, bubbles_nb = ndimage.label(void)
>>> bubbles_areas = np.bincount(bubbles_labels.ravel())[1:]
>>> mean_bubble_size = bubbles_areas.mean()
>>> median_bubble_size = np.median(bubbles_areas)
>>> mean_bubble_size, median_bubble_size
(1699.875, 65.0)
```
8.1 Objectives

At the end of this session you will be able to:

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.

8.2 What is SymPy?

SymPy is a Python library for symbolic mathematics. It aims to become a full featured computer algebra system that can compete directly with commercial alternatives (Mathematica, Maple) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.

8.3 First Steps with SymPy

8.3.1 Using SymPy as a calculator

SymPy defines three numerical types: Real, Rational and Integer.

The Rational class represents a rational number as a pair of two Integers: the numerator and the denominator, so Rational(1,2) represents 1/2, Rational(5,2) 5/2 and so on:
>>> from sympy import *
>>> a = Rational(1,2)

>>> a
1/2

>>> a*2
1

SymPy uses mpmath in the background, which makes it possible to perform computations using arbitrary precision arithmetic. That way, some special constants, like e, pi, oo (Infinity), are treated as symbols and can be evaluated with arbitrary precision:

>>> pi**2
pi**2

>>> pi.evalf()
3.14159265358979

>>> (pi+exp(1)).evalf()
5.859787448204884

as you see, evalf evaluates the expression to a floating-point number.

There is also a class representing mathematical infinity, called oo:

>>> oo > 99999
True

>>> oo + 1
oo

8.3.2 Exercises

1. Calculate $\sqrt{2}$ with 100 decimals.
2. Calculate $1/2 + 1/3$ in rational arithmetic.

8.3.3 Symbols

In contrast to other Computer Algebra Systems, in SymPy you have to declare symbolic variables explicitly:

>>> from sympy import *
>>> x = Symbol('x')
>>> y = Symbol('y')

Then you can manipulate them:

>>> x+y+x-y
2*x

>>> (x+y)**2
(x + y)**2

Symbols can now be manipulated using some of python operators: +, -, * (arithmetic), &, l, ~, >>, << (boolean).
8.4 Algebraic manipulations

SymPy is capable of performing powerful algebraic manipulations. We’ll take a look into some of the most frequently used: expand and simplify.

8.4.1 Expand

Use this to expand an algebraic expression. It will try to denest powers and multiplications:

```
In [23]: expand((x+y)**3)
Out[23]: 3*x*y**2 + 3*y*x**2 + x**3 + y**3
```

Further options can be given in form on keywords:

```
In [28]: expand(x+y, complex=True)
Out[28]: I*im(x) + I*im(y) + re(x) + re(y)
```

```
In [30]: expand(cos(x+y), trig=True)
Out[30]: cos(x)*cos(y) - sin(x)*sin(y)
```

8.4.2 Simplify

Use simplify if you would like to transform an expression into a simpler form:

```
In [19]: simplify((x+x*y)/x)
Out[19]: 1 + y
```

Simplification is a somewhat vague term, and more precise alternatives to simplify exists: powsimp (simplification of exponents), trigsimp (for trigonometrical expressions), logcombine, radsimp, together.

8.4.3 Exercises

1. Calculate the expanded form of \((x+y)^6\).
2. Simplify the trigonometrical expression \(\sin(x) / \cos(x)\)

8.5 Calculus

8.5.1 Limits

Limits are easy to use in sympy, they follow the syntax limit(function, variable, point), so to compute the limit of \(f(x)\) as \(x \to 0\), you would issue limit(f, x, 0):

```
>>> limit(sin(x)/x, x, 0)
1
```

you can also calculate the limit at infinity:
8.5.2 Differentiation

You can differentiate any SymPy expression using `diff(func, var)`. Examples:

```python
>>> diff(sin(x), x)
cos(x)
```

```python
>>> diff(sin(2*x), x)
2*cos(2*x)
```

```python
>>> diff(tan(x), x)
1 + tan(x)**2
```

You can check, that it is correct by:

```python
>>> limit((tan(x+y)-tan(x))/y, y, 0)
1 + tan(x)**2
```

Higher derivatives can be calculated using the `diff(func, var, n)` method:

```python
>>> diff(sin(2*x), x, 1)
2*cos(2*x)
```

```python
>>> diff(sin(2*x), x, 2)
-4*sin(2*x)
```

```python
>>> diff(sin(2*x), x, 3)
-8*cos(2*x)
```

8.5.3 Series expansion

SymPy also knows how to compute the Taylor series of an expression at a point. Use `series(expr, var)`:

```python
>>> series(cos(x), x)
1 - x**2/2 + x**4/24 + O(x**6)
```

```python
>>> series(1/cos(x), x)
1 + x**2/2 + 5*x**4/24 + O(x**6)
```

8.5.4 Exercises

1. Calculate $\lim_{x \to 0} \frac{\sin(x)}{x}$
2. Calculate the derivative of $\log(x)$ for $x$. 
### 8.5.5 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:

```python
tests = integrate(6*x**5, x)
tests = integrate(sin(x), x)
tests = integrate(log(x), x)
tests = integrate(2*x + sinh(x), x)
```

Also special functions are handled easily:

```python
tests = integrate(exp(-x**2)*erf(x), x)
```

It is possible to compute definite integral:

```python
tests = integrate(x**3, (x, -1, 1))
tests = integrate(sin(x), (x, 0, pi/2))
tests = integrate(cos(x), (x, -pi/2, pi/2))
```

Also improper integrals are supported as well:

```python
tests = integrate(exp(-x), (x, 0, oo))
tests = integrate(exp(-x**2), (x, -oo, oo))
```

### 8.5.6 Exercises

### 8.6 Equation solving

SymPy is able to solve algebraic equations, in one and several variables:

```python
In [7]: solve(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 large part of polynomial equations, and is also capable of solving multiple equations with respect to multiple variables giving a tuple as second argument:

```python
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 transcendental equations:
In [9]: 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 + x - x**2)*(1 - x - x**2)

In [12]: factor(f, modulus=5)
Out[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(x & 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(x & ~x)
Out[14]: False

### 8.6.1 Exercises

1. Solve the system of equations $x + y = 2$, $2x + y = 0$

2. Are there boolean values $x$, $y$ that make $(\neg x \lor y) \land (\neg y \lor x)$ true?

### 8.7 Linear Algebra

#### 8.7.1 Matrices

Matrices are created as instances from the Matrix class:

```python
>>> from sympy import Matrix
>>> Matrix([[1,0], [0,1]])
[1, 0]
[0, 1]
```

Unlike a numpy array, you can also put Symbols in it:

```python
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> A = Matrix([[1,x], [y,1]])
>>> A
[1, x]
[y, 1]
>>> A**2
```

124 Chapter 8. Sympy : Symbolic Mathematics in Python
8.7.2 Differential Equations

SymPy is capable of solving (some) Ordinary Differential Equations. sympy.ode.dsolve works like this:

```python
In [4]: f(x).diff(x, x) + f(x)
Out[4]:
  2
   d
------(f(x)) + f(x)
dx dx

In [5]: dsolve(f(x).diff(x, x) + f(x), f(x))
Out[5]: C1*sin(x) + C2*cos(x)
```

Keyword arguments can be given to this function in order to help if find the best possible resolution system. For example, if you know that it is a separable equations, you can use keyword hint='separable' to force dsolve to resolve it as a separable equation.

```python
In [6]: dsolve(sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x), f(x), hint='separable')
Out[6]: -log(1 - sin(f(x))**2)/2 == C1 + log(1 - sin(x)**2)/2
```

8.7.3 Exercises

1. Solve the Bernoulli differential equation x*f(x).diff(x) + f(x) - f(x)**2
2. Solve the same equation using hint='Bernoulli'. What do you observe?
3D plotting with Mayavi

author Gaël Varoquaux

9.1 A simple example

Warning: Start ipython -wthread

```python
import numpy as np
x, y = np.mgrid[-10:10:100j, -10:10:100j]
r = np.sqrt(x**2 + y**2)
z = np.sin(r)/r
from enthought.mayavi import mlab
mlab.surf(z, warp_scale='auto')
mlab.outline()
mlab.axes()
```

np.mgrid[-10:10:100j, -10:10:100j] creates an x,y grid, going from -10 to 10, with 100 steps in each directions.
9.2 3D plotting functions

9.2.1 Points

In [1]: import numpy as np

In [2]: from enthought.mayavi import mlab

In [3]: x, y, z, value = np.random.random((4, 40))

In [4]: mlab.points3d(x, y, z, value)
Out[4]: <enthought.mayavi.modules.glyph.Glyph object at 0xc3c795c>

9.2.2 Lines

In [5]: mlab.clf()

In [6]: t = np.linspace(0, 20, 200)

In [7]: mlab.plot3d(np.sin(t), np.cos(t), 0.1*t, t)
Out[7]: <enthought.mayavi.modules.surface.Surface object at 0xcc3e1dc>
9.2.3 Elevation surface

In [8]: mlab.clf()

In [9]: x, y = np.mgrid[-10:10:100j, -10:10:100j]

In [10]: r = np.sqrt(x**2 + y**2)

In [11]: z = np.sin(r)/r

In [12]: mlab.surf(z, warp_scale='auto')

Out[12]: <enthought.mayavi.modules.surface.Surface object at 0xcdb98fc>
9.2.4 Arbitrary regular mesh

In [13]: mlab.clf()
In [14]: phi, theta = np.mgrid[0:pi:11j, 0:2*pi:11j]
In [15]: x = sin(phi)*cos(theta)
In [16]: y = sin(phi)*sin(theta)
In [17]: z = cos(phi)
In [18]: mlab.mesh(x, y, z)
In [19]: mlab.mesh(x, y, z, representation='wireframe', color=(0, 0, 0))
Out[19]: <enthought.mayavi.modules.surface.Surface object at 0xce1017c>

Note: A surface is defined by points connected to form triangles or polygones. In mlab.func and mlab.mesh, the connectivity is implicitly given by the layout of the arrays. See also mlab.triangular_mesh.

Our data is often more than points and values: it needs some connectivity information

9.2.5 Volumetric data

In [20]: mlab.clf()
In [21]: x, y, z = np.mgrid[-5:5:64j, -5:5:64j, -5:5:64j]
In [22]: values = x*x*0.5 + y*y + z*z*2.0
In [23]: mlab.contour3d(values)
Out[24]: <enthought.mayavi.modules.iso_surface.IsoSurface object at 0xcfe392c>
This function works with a regular orthogonal grid:

### 9.3 Figures and decorations

#### 9.3.1 Figure management

Here is a list of functions useful to control the current figure:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the current figure:</td>
<td><code>mlab.gcf()</code></td>
</tr>
<tr>
<td>Clear the current figure:</td>
<td><code>mlab.clf()</code></td>
</tr>
<tr>
<td>Set the current figure:</td>
<td><code>mlab.figure(1, bgcolor=(1, 1, 1), fgcolor=(0.5, 0.5, 0.5))</code></td>
</tr>
<tr>
<td>Save figure to image file:</td>
<td><code>mlab.savefig('foo.png', size=(300, 300))</code></td>
</tr>
<tr>
<td>Change the view:</td>
<td><code>mlab.view(azimuth=45, elevation=54, distance=1.)</code></td>
</tr>
</tbody>
</table>
9.3.2 Changing plot properties

In general, many properties of the various objects on the figure can be changed. If these visualization are created via \textit{mlab} functions, the easiest way to change them is to use the keyword arguments of these functions, as described in the docstrings.

**Example docstring: \textit{mlab.mesh}**

Plots a surface using grid-spaced data supplied as 2D arrays.

**Function signatures:**

```python
mesh(x, y, z, ...)```

\(x, y, z\) are 2D arrays, all of the same shape, giving the positions of the vertices of the surface. The connectivity between these points is implied by the connectivity on the arrays.

For simple structures (such as orthogonal grids) prefer the \textit{surf} function, as it will create more efficient data structures.

**Keyword arguments:**

- \texttt{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.
- \texttt{colormap} type of colormap to use.
- \texttt{extent} [\texttt{xmin}, \texttt{xmax}, \texttt{ymin}, \texttt{ymax}, \texttt{zmin}, \texttt{zmax}] Default is the \(x, y, z\) arrays extents. Use this to change the extent of the object created.
- \texttt{figure} Figure to populate.
- \texttt{line_width} The width of the lines, if any used. Must be a float. Default: 2.0
- \texttt{mask} boolean mask array to suppress some data points.
- \texttt{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.
- \texttt{mode} the mode of the glyphs. Must be ‘2darrow’ or ‘2dcircle’ or ‘2dcross’ or ‘2ddash’ or ‘2ddiamond’ or ‘2dhooked_arrow’ or ‘2dsquare’ or ‘2dthick_arrow’ or ‘2dthick_cross’ or ‘2dtriangle’ or ‘2dvertex’ or ‘arrow’ or ‘cone’ or ‘cube’ or ‘cylinder’ or ‘point’ or ‘sphere’. Default: sphere
- \texttt{name} the name of the vtk object created.
- \texttt{representation} the representation type used for the surface. Must be ‘surface’ or ‘wireframe’ or ‘points’ or ‘mesh’ or ‘fancymesh’. Default: surface
- \texttt{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
- \texttt{scalars} optional scalar data.
- \texttt{scale_factor} scale factor of the glyphs used to represent the vertices, in fancy_mesh mode. Must be a float. Default: 0.05
- \texttt{scale_mode} the scaling mode for the glyphs (‘vector’, ‘scalar’, or ‘none’).
- \texttt{transparent} make the opacity of the actor depend on the scalar.
- \texttt{tube_radius} radius of the tubes used to represent the lines, in mesh mode. If None, simple lines are used.
- \texttt{tube_sides} number of sides of the tubes used to represent the lines. Must be an integer. Default: 6
- \texttt{vmax} \texttt{vmax} is used to scale the colormap. If None, the max of the data will be used
- \texttt{vmin} \texttt{vmin} is used to scale the colormap. If None, the min of the data will be used

Example:

```python
In [1]: import numpy as np
```
In [2]: r, theta = np.mgrid[0:10, -np.pi:np.pi:10j]

In [3]: x = r*np.cos(theta)

In [4]: y = r*np.sin(theta)

In [5]: z = np.sin(r)/r

In [6]: from enthought.mayavi import mlab

In [7]: mlab.mesh(x, y, z, colormap='gist_earth', extent=[0, 1, 0, 1, 0, 1])
   Out[7]: <enthought.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))
   Out[8]: <enthought.mayavi.modules.surface.Surface object at 0xdd6a71c>

9.3.3 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')
   Out[9]: <tvtk_classes.scalar_bar_actor.ScalarBarActor object at 0xd897f8c>

In [10]: mlab.title('polar mesh')
   Out[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])
   Out[12]: <enthought.mayavi.modules.axes.Axes object at 0xd2e4bcc>
**Warning:** extent: If we specified extents for a plotting object, `mlab.outline` and `mlab.axes` don’t get them by default.

### 9.4 Interaction

The quicket way to create beautiful visualization with Mayavi is probably to interactively tweak the various settings. Click on the ‘Mayavi’ button in the scene, and you can control properties of objects with dialogs.
To 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.
Bibliography