Python Scientific lecture notes

Release 2010

EuroScipy tutorial team

Editors: Emmanuelle Gouillart, Gaël Varoquaux

http://scipy-lectures.github.com

February 17, 2011

Contents

1 Getting started with Python for science **1**
1.1 Scientific computing: why Python? . . . 1
1.1.1 The scientist's needs . . . 1
1.1.2 Specifications . . . 1
1.1.3 Existing solutions . . . 1
1.2 Building blocks of scientific computing with Python . . . 2
1.3 A (very short) introduction to Python . . . 5
1.3.1 First steps . . . 6
1.3.2 Basic types . . . 6
1.3.3 Assignment operator . . . 12
1.3.4 Control Flow . . . 13
1.3.5 Defining functions . . . 17
1.3.6 Reusing code: scripts and modules . . . 22
1.3.7 Input and Output . . . 28
1.3.8 Standard Library . . . 29
1.3.9 Exceptions handling in Python . . . 33
1.3.10 Object-oriented programming (OOP) . . . 36
1.4 NumPy: creating and manipulating numerical data . . . 36
1.4.1 Creating NumPy data arrays . . . 37
1.4.2 Graphical data representation : matplotlib and Mayavi . . . 38
1.4.3 Indexing . . . 40
1.4.4 Slicing . . . 41
1.4.5 Manipulating the shape of arrays . . . 43
1.4.6 Exercises : some simple array creations . . . 44
1.4.7 Real data: read/write arrays from/to files . . . 44
1.4.8 Simple mathematical and statistical operations on arrays . . . 47
1.4.9 Fancy indexing . . . 49
1.4.10 Broadcasting . . . 51
1.4.11 Synthesis exercises: framing Lena . . . 54
1.5 Getting help and finding documentation . . . 55
1.6 Matplotlib . . . 58
1.6.1 Introduction . . . 58
1.6.2 IPython . . . 58
1.6.3 pylab . . . 58
1.6.4 Simple Plots . . . 58
1.6.5 Properties . . . 60
1.6.6 Text . . . 62

1.6.7 Ticks . . . 63
1.6.8 Figures, Subplots, and Axes . . . 65
1.6.9 Other Types of Plots . . . 67
1.6.10 The Class Library . . . 73
1.7 Scipy : high-level scientific computing . . . 75
1.7.1 Scipy builds upon Numpy . . . 75
1.7.2 File input/output: scipy.io . . . 76
1.7.3 Signal processing: scipy.signal . . . 76
1.7.4 Special functions: scipy.special . . . 77
1.7.5 Statistics and random numbers: scipy.stats . . . 77
1.7.6 Linear algebra operations: scipy.linalg . . . 79
1.7.7 Numerical integration: scipy.integrate . . . 80
1.7.8 Fast Fourier transforms: scipy.fftpack . . . 82
1.7.9 Interpolation: scipy.interpolate . . . 84
1.7.10 Optimization and fit: scipy.optimize . . . 85
1.7.11 Image processing: scipy.ndimage . . . 87
1.7.12 Summary exercices on scientific computing . . . 91

2 Advanced topics **104**
2.1 Advanced Numpy . . . 104
2.1.1 Abstract . . . 104
2.1.2 Advanced Numpy . . . 104
2.1.3 Life of ndarray . . . 105
2.1.4 Universal functions . . . 116
2.1.5 Interoperability features . . . 125
2.1.6 Siblings: chararray, maskedarray, matrix . . . 133
2.1.7 Summary . . . 134
2.1.8 Hit list of the future for Numpy core . . . 134
2.1.9 Contributing to Numpy/Scipy . . . 134
2.1.10 Python 2 and 3, single code base . . . 137
2.2 Sparse Matrices in SciPy . . . 138
2.2.1 Introduction . . . 138
2.2.2 Storage Schemes . . . 140
2.2.3 Linear System Solvers . . . 152
2.2.4 Other Interesting Packages . . . 157
2.3 Sympy : Symbolic Mathematics in Python . . . 157
2.3.1 Objectives . . . 157
2.3.2 What is SymPy? . . . 158
2.3.3 First Steps with SymPy . . . 158
2.3.4 Algebraic manipulations . . . 159
2.3.5 Calculus . . . 160
2.3.6 Equation solving . . . 161
2.3.7 Linear Algebra . . . 162
2.4 3D plotting with Mayavi . . . 163
2.4.1 A simple example . . . 163
2.4.2 3D plotting functions . . . 164
2.4.3 Figures and decorations . . . 167
2.4.4 Interaction . . . 170

Bibliography **171**

Index **172**

CHAPTER 1

Getting started with Python for science

1.1 Scientific computing: why Python?

authors Fernando Perez, Emmanuelle Gouillart

1.1.1 The scientist's needs

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

1.1.2 Specifications

- Rich collection of already existing **bricks** corresponding to classical numerical methods or basic actions: we don't want to re-program the plotting of a curve, a Fourier transform or a fitting algorithm. Don't reinvent the wheel!
- Easy to learn: computer science 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.

1.1.3 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.

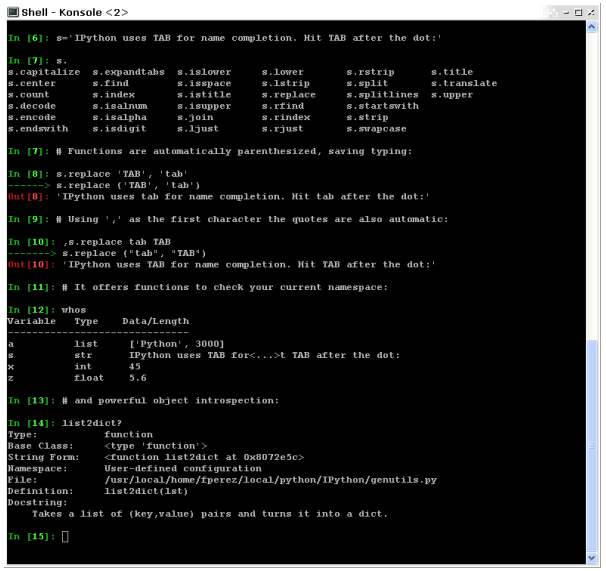
1.2 Building blocks of scientific computing with Python

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

  http://ipython.scipy.org/moin/




```
In [1]: a = 2

In [2]: print "hello"
hello

In [3]: %run my_script.py
```

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

```
>>> 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:

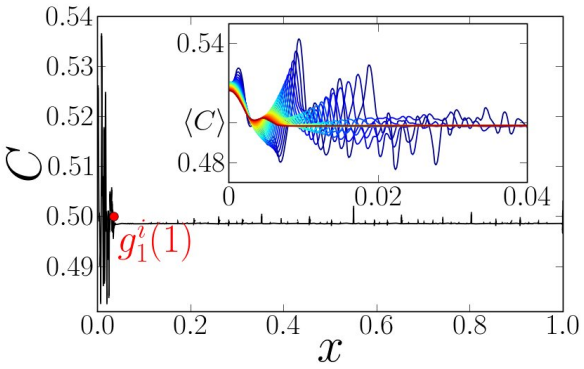
```
>>> import numpy as np
>>> import scipy
>>> t = np.arange(10)
```

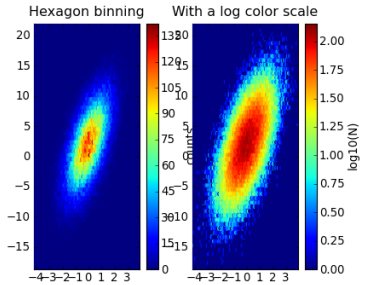
```
>>> t
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> signal = t**2 + 2*t + 2+ 1.e-2*np.random.random(10)
>>> scipy.polyfit(t, signal, 2)
array([ 1.00001151,  1.99920674,  2.00902748])
```

http://www.scipy.org/

- **Matplotlib** : 2-D visualization, "publication-ready" plots

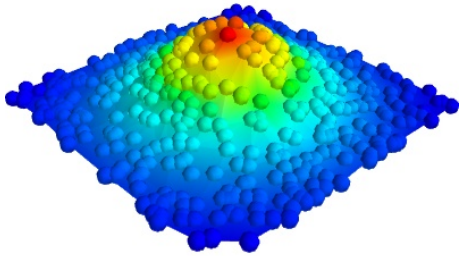
  http://matplotlib.sourceforge.net/





- **Mayavi** : 3-D visualization

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



- and many others.

1.3 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/.

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.

1.3.1 First steps

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

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

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

Once you have started the interpreter, type

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

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

To get yourself started, type the following stack of instructions

```
>>> a = 3
>>> 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:

```
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.

1.3.2 Basic types

Numerical types

Integer variables:

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

floats

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

complex (a native type in Python!)


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

and booleans:

```
>>> 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
```

Containers

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

Lists

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

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

- Indexing: accessing individual objects contained in the list:

  ```
  >>> l[2]
  3
  ```

  Counting from the end with negative indices:

  ```
  >>> 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

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

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

Slicing syntax: *l[start:stop:stride]*

All slicing parameters are optional:

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

Lists are *mutable* objects and can be modified:

```
>>> 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:

```
>>> l = [3, 2, 'hello']
>>> l
[3, 2, 'hello']
>>> l[1], l[2]
(2, 'hello')
```


As the elements of a list can be of any type and size, accessing the i *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 istems. With Numpy arrays, accessing the i'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:

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

Reverse *l*:

```
>>> r = l[::-1]
>>> r
[5, 4, 3, 2, 1]
```

Concatenate and repeat lists:

```
>>> r + l
[5, 4, 3, 2, 1, 1, 2, 3, 4, 5]
>>> 2 * r
[5, 4, 3, 2, 1, 5, 4, 3, 2, 1]
```

Sort r (in-place):

```
>>> r.sort()
>>> r
[1, 2, 3, 4, 5]
```

Note: Methods and Object-Oriented Programming

The notation `r.method()` (`r.sort()`, `r.append(3)`, `l.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.
r.__add__           r.__iadd__          r.__setattr__
r.__class__         r.__imul__          r.__setitem__
r.__contains__      r.__init__          r.__setslice__
r.__delattr__       r.__iter__          r.__sizeof__
r.__delitem__       r.__le__            r.__str__
r.__delslice__      r.__len__           r.__subclasshook__
r.__doc__           r.__lt__            r.append
r.__eq__            r.__mul__           r.count
r.__format__        r.__ne__            r.extend
r.__ge__            r.__new__           r.index
```

```
r.__getattribute__  r.__reduce__        r.insert
r.__getitem__       r.__reduce_ex__     r.pop
r.__getslice__      r.__repr__          r.remove
r.__gt__            r.__reversed__      r.reverse
r.__hash__          r.__rmul__          r.sort
```

Strings

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

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

```
In [1]: 'Hi, what's up?'
------------------------------------------------------------
   File "<ipython console>", line 1
     'Hi, what's up?'
               ^
SyntaxError: invalid syntax
```

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:

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

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

Slicing:

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

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.

```
In [53]: a = "hello, world!"
In [54]: a[2] = 'z'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)

/home/gouillar/travail/sgr/2009/talks/dakar_python/cours/gael/essai/source/<ipython
```



```
console> in <module>()

TypeError: 'str' object does not support item assignment
In [55]: a.replace('l', 'z', 1)
Out[55]: 'hezlo, world!'
In [56]: a.replace('l', 'z')
Out[56]: 'hezzo, worzd!'
```

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:

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

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

Dictionnaries

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

```
>>> 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]
>>> 'francis' in tel
True
```

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

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

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

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

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

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.

```
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
```

1.3.3 Assignment operator

Python library reference says:

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

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

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

Things to note:

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

```
In [1]: a = [1, 2, 3]
In [2]: b = a
In [3]: a
Out[3]: [1, 2, 3]
In [4]: b
Out[4]: [1, 2, 3]
In [5]: a is b
Out[5]: True
In [6]: b[1] = 'hi!'
In [7]: a
Out[7]: [1, 'hi!', 3]
```

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

```
In [1]: a = [1, 2, 3]
In [3]: a
Out[3]: [1, 2, 3]
In [4]: a = ['a', 'b', 'c'] # Creates another object.
In [5]: a
Out[5]: ['a', 'b', 'c']
In [6]: id(a)
Out[6]: 138641676
In [7]: a[:] = [1, 2, 3] # Modifies object in place.
In [8]: a
Out[8]: [1, 2, 3]
In [9]: id(a)
Out[9]: 138641676 # Same as in Out[6], yours will differ...
```

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

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

1.3.4 Control Flow

Controls the order in which the code is executed.

if/elif/else

```
In [1]: if 2**2 == 4:
   ...:     print('Obvious!')
   ...:
Obvious!
```

Blocks are delimited by indentation

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

```
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.

for/range

Iterating with an index:

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

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

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

while/break/continue

Typical C-style while loop (Mandelbrot problem):

```
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.:

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

Conditional Expressions

- *if object*

Evaluates to True:

- any non-zero value
- any sequence with a length > 0

Evaluates to False:

- any zero value
- any empty sequence

- *a == b*

  Tests equality, with logics:

  ```
  In [19]: 1 == 1.
  Out[19]: True
  ```

- *a is b*

  Tests identity: both objects are the same

  ```
  In [20]: 1 is 1.
  Out[20]: False

  In [21]: a = 1

  In [22]: b = 1

  In [23]: a is b
  Out[23]: True
  ```

- *a in b*

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

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

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

Advanced iteration

Iterate over any *sequence*

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

  ```
  In [11]: vowels = 'aeiouy'

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

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

```
Hello
how
are
you?
```

Few languages (in particular, languages for scienfic 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:

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

- But Python provides **enumerate** for this:

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

Looping over a dictionary

Use **iteritems**:

```
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
```

List Comprehensions

```
In [16]: [i**2 for i in range(4)]
Out[16]: [0, 1, 4, 9]
```


Exercise

Compute the decimals of Pi using the Wallis formula:

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

1.3.5 Defining functions

Function definition

```
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.

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.

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()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/<ipython console> in <module>()

TypeError: double_it() takes exactly 1 argument (0 given)
```

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.

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 refereence 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:

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

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

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

Global variables

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

```
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:

```
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:

```
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
```

Variable number of parameters

Special forms of parameters:

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

```
In [35]: def variable_args(*args, **kwargs):
   ....:     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}
```

Docstrings

Documention about what the function does and it's parameters. General convention:

```
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.

    Extended summary which can contain multiple paragraphs.
```

Note: Docstring guidelines

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

Also, the Numpy and Scipy modules have defined a precised standard for documenting scientific functions, that you may want to follow for your own functions, with a `Parameters` section, an `Examples` section, etc. See http://projects.scipy.org/numpy/wiki/CodingStyleGuidelines#docstring-standard and http://projects.scipy.org/numpy/browser/trunk/doc/example.py#L37

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}
```

Methods

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

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`

1.3.6 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)).

Scripts

Let us first write a **script**, that is a file with a sequence of instructions that are executed each time the script is called.

Instructions may be e.g. copied-and-pasted from the interpreter (but take care to respect indentation rules!). The extension for Python files is **.py**. Write or copy-and-paste the following lines in a file called **test.py**

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

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

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

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

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

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

```
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*.

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 vise-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.

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*:

```
" A demo module. "

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

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


c = 2
d = 2
```

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)
```

'__main__' and module loading

File *demo2.py*:

```
import sys

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

print sys.argv

if __name__ == '__main__':
    print_a()
```

Importing it:

```
In [11]: import demo2
b

In [12]: import demo2
```

Running it:

```
In [13]: %run demo2
b
a
```

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

```
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.
- modify the environment variable **PYTHONPATH** to include the directories containing the user-defined modules. On Linux/Unix, add the following line to a file read by the shell at startup (e.g. /etc/profile, .profile)

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

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

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

```
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 for more information about modules.

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]
cluster/        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/
integrate/      odr/         special/
interpolate/    optimize/    stats/
sd-2116 /usr/lib/python2.6/dist-packages/scipy $ 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@
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.
```

Good practices

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**

1.3.7 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:


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

To read from a file

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

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

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

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

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

Iterating over a file

```
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.

1.3.8 Standard Library

Note: Reference document for this section:

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

`os` module: operating system functionality

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

Directory and file manipulation

Current directory:

```
In [17]: os.getcwd()
Out[17]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
```

List a directory:

```
In [31]: os.listdir(os.curdir)
Out[31]:
['.index.rst.swo',
 '.python_language.rst.swp',
 '.view_array.py.swp',
 '_static',
 '_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.


```
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

```
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.

```
In [10]: for dirpath, dirnames, filenames in os.walk(os.curdir):
   ....:     for fp in filenames:
   ....:         print os.path.abspath(fp)
   ....:
   ....:
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/.index.rst.swo
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/.view_array.py.swp
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/basic_types.rst
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/conf.py
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/control_flow.rst
...
```

Environment variables:

```
In [9]: import os

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

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'

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'
```

`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.

`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']
```

`sys` module: system-specific information

System-specific information related to the Python interpreter.

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


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

In [118]: sys.version
Out[118]: '2.5.2 (r252:60911, Feb 22 2008, 07:57:53) \n
          [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:

```
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:

```
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',
 ...
```

`pickle`: easy persistence

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

```
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`.

path_site

1.3.9 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.

Exceptions

Exceptions are raised by errors in 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.

Catching exceptions

try/except

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

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

try/finally

```
In [10]: try:
   ....:     x = int(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

```
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
```

Raising exceptions

- Capturing and reraising an exception:

```
In [15]: 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]: filter_name('Gaël')
OK, Gaël
Out[16]: 'Ga\xc3\xabl'

In [17]: filter_name('Stéfan')
---------------------------------------------------------------------------
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 2: ordinal not in ran
```

- Exceptions to pass messages between parts of the code:

```
In [17]: def achilles_arrow(x):
   ....:     if abs(x - 1) < 1e-3:
   ....:         raise StopIteration
   ....:     x = 1 - (1-x)/2.
   ....:     return x
   ....:

In [18]: x = 0

In [19]: while True:
   ....:     try:
   ....:         x = achilles_arrow(x)
   ....:     except StopIteration:
   ....:         break
   ....:
   ....:
```

```
In [20]: x
Out[20]: 0.9990234375
```

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

1.3.10 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:

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

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

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

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

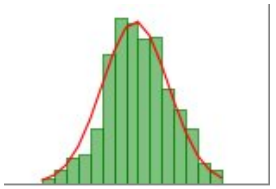
The MasterStudent class inherited from the Student attributes and methods.

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

1.4 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`

1.4.1 Creating NumPy data arrays

A small introductory example:

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

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

- Evenly spaced values:

```
>>> 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:

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

- Constructors for common arrays:

```
>>> 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.]])
```

1.4.2 Graphical data representation : matplotlib and Mayavi

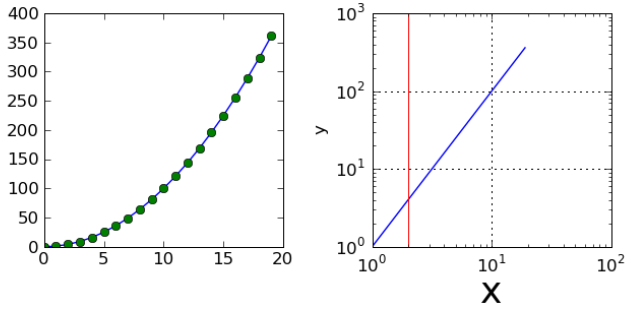
Now that we have our first data arrays, we are going to visualize them. **Matplotlib** is a 2D plotting package. We can import its functions as below:

```
>>> import pylab
>>> # or
>>> from pylab import * # imports everything in the namespace
```

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`.

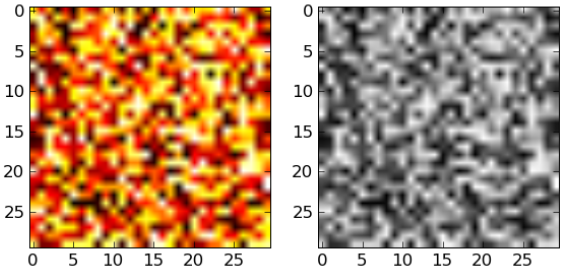
1D curve plotting

```
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 0x9e954ac>
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
```



There are many other features in matplotlib: color choice, marker size, latex font, inclusions within figures, histograms, etc.

To go further :

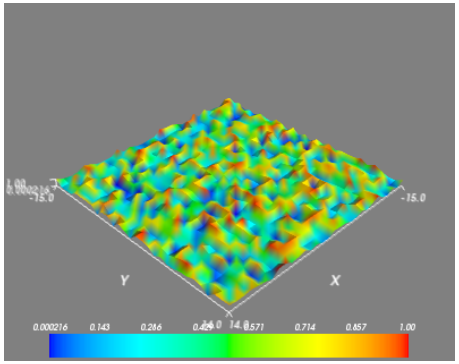
- matplotlib documentation http://matplotlib.sourceforge.net/contents.html
- an example gallery with corresponding sourcecode http://matplotlib.sourceforge.net/gallery.html

3D plotting

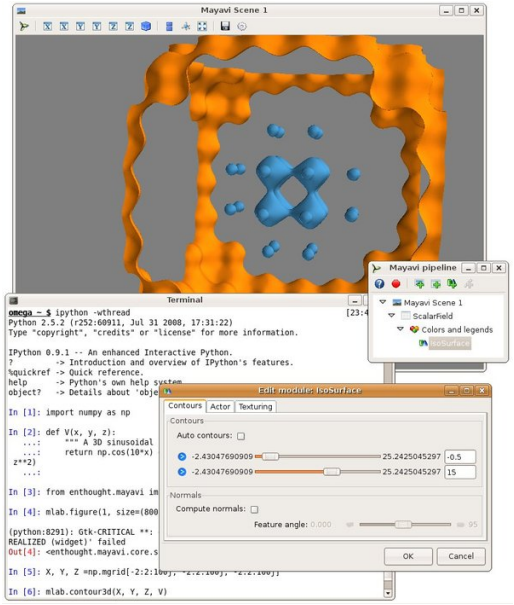
For 3D visualization, we use another package: **Mayavi**. A quick example: start with **relaunching iPython** with these options: **ipython -pylab -wthread**

```
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.



For more information on Mayavi : http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/index.html

1.4.3 Indexing

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

```
>>> 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:

```
>>> 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
>>> a[2,1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  0,  0,  0],
       [ 0, 10,  2,  0,  0],
       [ 0,  0,  0,  3,  0],
       [ 0,  0,  0,  0,  4]])
>>> a[1]
array([0, 1, 0, 0, 0])
```

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.

1.4.4 Slicing

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

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

Note that the last index is not included!:

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

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

```
>>> sl = slice(1, 9, 2)
>>> a = np.arange(10)
>>> b = 2*a + 1
>>> a, b
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19]))
>>> a[sl], b[sl]
(array([1, 3, 5, 7]), array([ 3,  7, 11, 15]))
```

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

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

Of course, it works with multidimensional arrays:

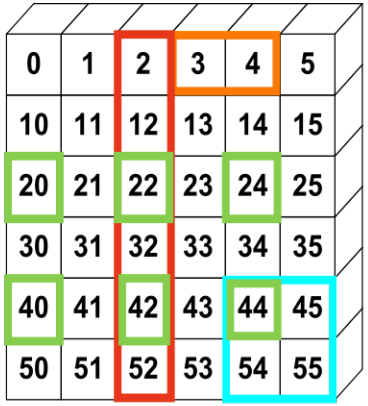
```
>>> a = np.eye(5)
>>> a
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
>>> a[2:4,:3] #3rd and 4th lines, 3 first columns
array([[ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

All elements specified by a slice can be easily modified:

```
>>> 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...

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

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

>>> a[:,2]
array([2,22,52])

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



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*:

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



```
>>> b
array([12,  2,  4,  6,  8])
>>> a # a a été modifié aussi !
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.

1.4.5 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:

```
>>> 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`:

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

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

```
>>> 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:

```
>>> 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`:

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

```
>>> b
array([[0, 1],
       [2, 3],
       [0, 0]])
```

A large array can be tiled with a smaller one:

```
>>> a = np.arange(4).reshape((2,2))
>>> a
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]])
```

1.4.6 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

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

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

1.4.7 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:

- 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:

```
>>> 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

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

- Writing a text file (in ASCII):

```
>>> np.savetxt('data_a.txt', a)
```

- Writing a binary file (`.npy` extension, recommended format)

```
>>> np.save('data_a.npy', a)
```

Loading a data array from a file

- Reading from a text file:

```
>>> b = np.loadtxt('data_a.txt')
```

- Reading from a binary file:

```
>>> c = np.load('data_a.npy')
```

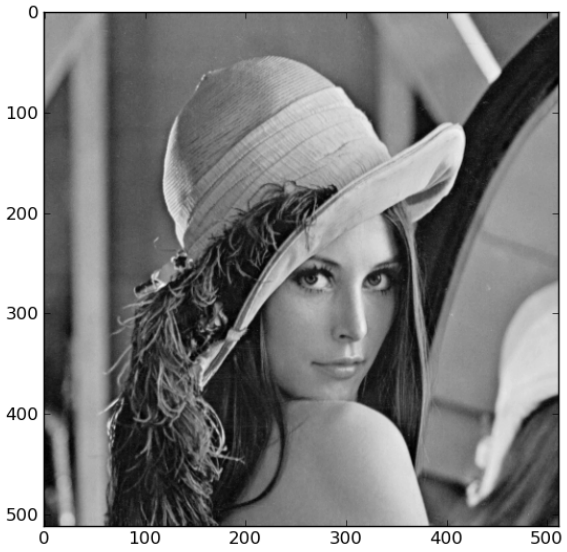
To read matlab data files

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

Opening and saving images: imsave and imread

```
>>> 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:

```
>>> 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:

```
>>> import glob
>>> filelist = glob.glob('line*')
>>> filelist
['line_0', 'line_1', 'line_2', 'line_3', 'line_4', 'line_5', 'line_6', 'line_7', 'line_8', 'line_
```



```
>>> # Note that the line is not always sorted
>>> filelist.sort()
>>> l2 = np.loadtxt(filelist[2])
```

Note: arrays can also be created from Excel/Calc files, HDF5 files, etc. (but with additional modules not described here: xlrd, pytables, etc.).

1.4.8 Simple mathematical and statistical operations on arrays

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

```
>>> 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:

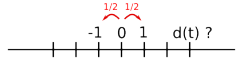
```
>>> a = np.array([[1, 3], [9, 6]])
>>> a
array([[1, 3],
       [9, 6]])
>>> a.mean(axis=0) # the array contains the mean of each column
array([ 5. ,  4.5])
>>> a.mean(axis=1) # the array contains the mean of each line
array([ 2. ,  7.5])
```

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

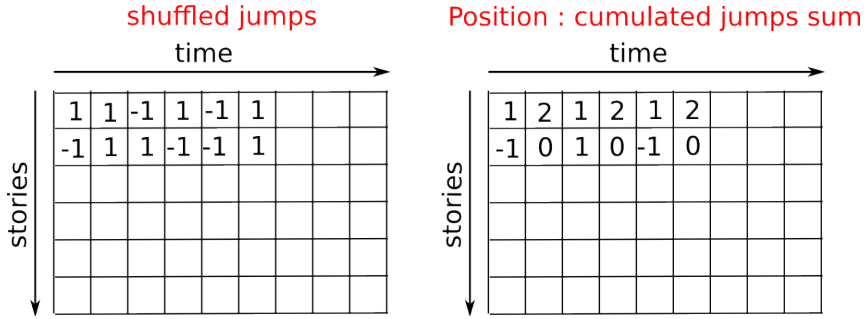
Note: Arithmetic operations on arrays correspond to operations on each individual element. In particular, the multiplication is not a matrix multiplication (**unlike Matlab**)! The matrix multiplication is provided by `np.dot`:

```
>>> 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

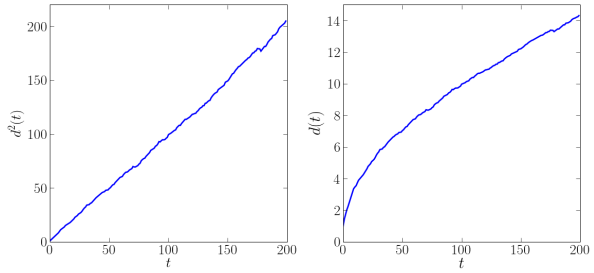


What is the typical distance from the origin of a random walker after `t` left or right jumps?




```
>>> nreal = 1000 # number of walks
>>> tmax = 200 # time during which we follow the walker
>>> # We randomly choose all the steps 1 or -1 of the walk
>>> walk = 2 * ( np.random.random_integers(0, 1, (nreal,tmax)) - 0.5 )
>>> np.unique(walk) # Verification : all steps are 1 or -1
array([-1.,  1.])
>>> # We build the walks by summing steps along the time
>>> cumwalk = np.cumsum(walk, axis=1) # axis = 1 : dimension of time
>>> sq_distance = cumwalk**2
>>> # We get the mean in the axis of the steps
>>> mean_sq_distance = np.mean(sq_distance, axis=0)
```

```
In [39]: figure()
In [40]: plot(mean_sq_distance)
In [41]: figure()
In [42]: plot(np.sqrt(mean_sq_distance))
```



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 visulalization)
10. Create an array that contains the organism where the highest women's percentage is found for the different years.

Answers *stat_recherche*

1.4.9 Fancy indexing

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

Masks

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
>>> (a%3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> mask = (a%3 == 0)
>>> extract_from_a = a[mask] #one could directly write a[a%3==0]
>>> extract_from_a # extract a sub-array with the mask
array([ 3,  0,  9,  6,  0, 12])
```

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

```
>>> 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:

```
>>> 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

```
>>> a = np.arange(10)
>>> a[::2] +=3 #to avoid having always the same np.arange(10)...
>>> a
array([ 3,  1,  5,  3,  7,  5,  9,  7, 11,  9])
>>> a[[2, 5, 1, 8]] # or a[np.array([2, 5, 1, 8])]
array([ 5,  5,  1, 11])
```

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

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

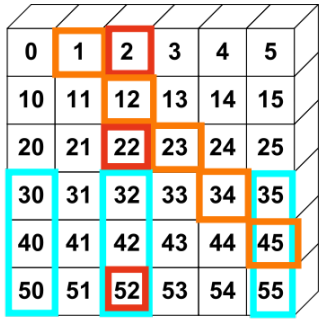
New values can be assigned with this kind of indexing:

```
>>> a[[9, 7]] = -10
>>> a
array([  3,   1,   5,   3,   7,   5,   9, -10,  11, -10])
>>> a[[2, 3, 2, 4, 2]] +=1
>>> a
array([  3,   1,   6,   4,   8,   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]])
>>> b = np.arange(10)

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

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

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

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

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



Exercise

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

1. Create a `sup30` array of the same size than `data` with a value of 1 if the value of `data` is greater than 30%, 0 otherwise.

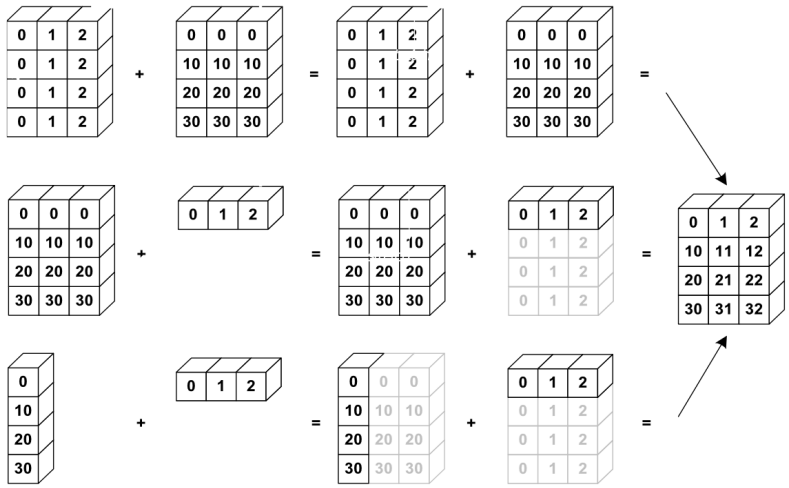
2. Create an array containing the organisme having the greatest percentage of women of each year.

Answers *stat_recherche*

1.4.10 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:

```
>>> 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!:

```
>>> 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]])
```

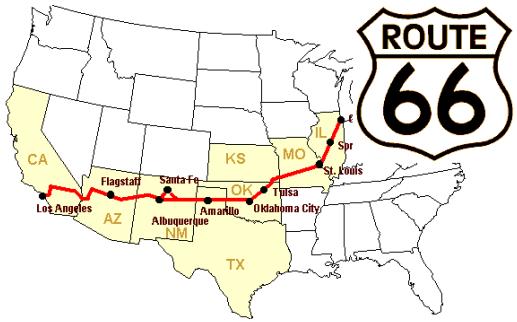
We can even use fancy indexing and broadcasting at the same time. Take again the same example as above::

```
>>> 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, Albucquerque, Flagstaff and Los Angeles.

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



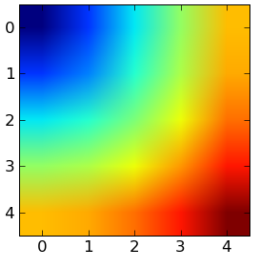
Warning: Good practices
In the previous example, we can note some good (and bad) practices:

- 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:

```
>>> 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).



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

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

```
>>> 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]])
```

1.4.11 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:

```
>>> 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.

```
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.

```
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`.

```
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`

```
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:

```
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:

```
>>> 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* (page 55)) !!
- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more functions of numpy allowing to handle array operations.

1.5 Getting help and finding documentation

author Emmanuelle Gouillart

Rather than knowing all functions in Numpy and Scipy, it is important to find rapidly information throughout the documentation and the available help. Here are some ways to get information:

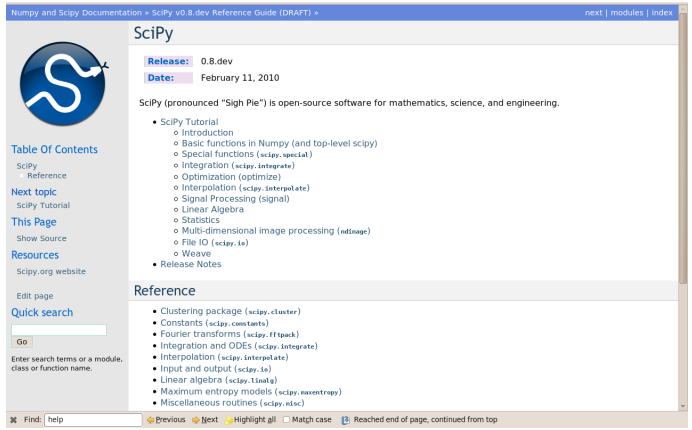
- In Ipython, `help function` opens the docstring of the function. Only type the beginning of the function's name and use tab completion to display the matching functions.

```
In [204]: help np.v
np.vander     np.vdot       np.version    np.void0      np.vstack
np.var        np.vectorize  np.void       np.vsplit

In [204]: help np.vander
```

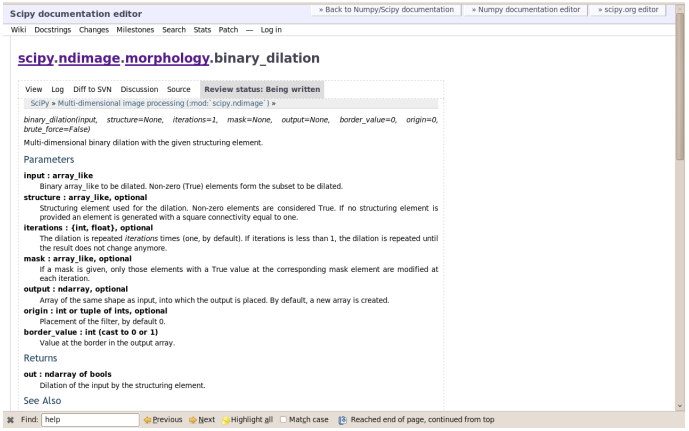
In Ipython it is not possible to open a separated window for help and documentation; however one can always open a second `Ipython` shell just to display help and docstrings...

- Numpy's and Scipy's documentations can be browsed online on http://docs.scipy.org/doc. The `search` button is quite useful inside the reference documentation of the two packages (http://docs.scipy.org/doc/numpy/reference/ and http://docs.scipy.org/doc/scipy/reference/).

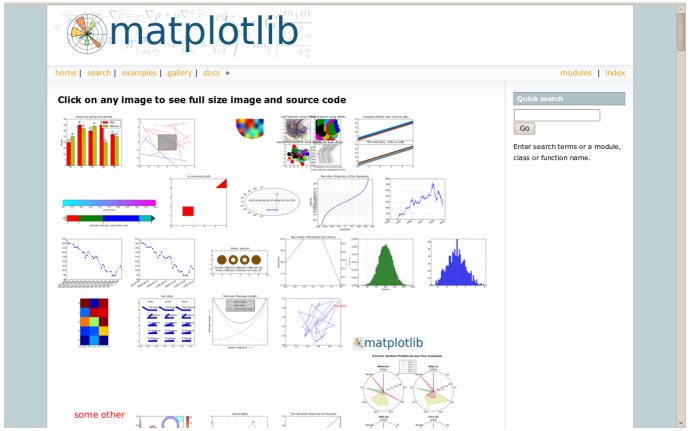
  Tutorials on various topics as well as the complete API with all docstrings are found on this website.



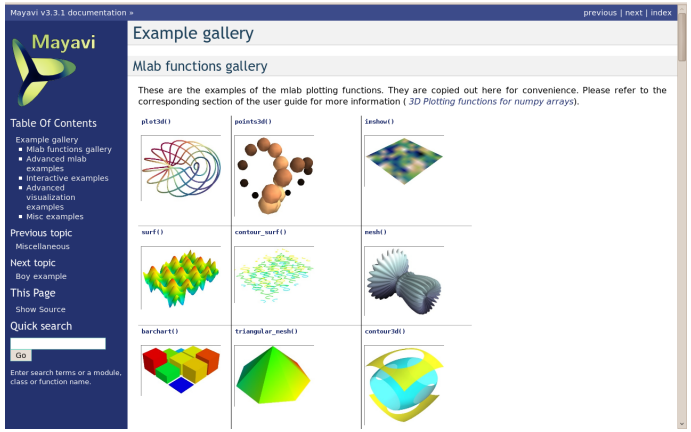
- Numpy's and Scipy's documentation is enriched and updated on a regular basis by users on a wiki http://docs.scipy.org/numpy/. As a result, some docstrings are clearer or more detailed on the wiki, and you may want to read directly the documentation on the wiki instead of the official documentation 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.



- Mayavi's website http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/ also has a very nice gallery of examples http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/auto/examples.html in which one can browse for different visualization solutions.



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.

```
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.

1.6 Matplotlib

author Mike Müller

1.6.1 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.

1.6.2 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.

1.6.3 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.

1.6.4 Simple Plots

Let's start an interactive session:

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

This brings us to the IPython prompt:

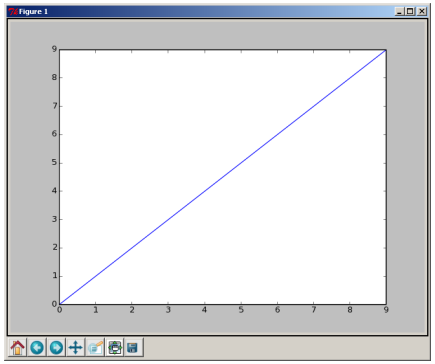
```
IPython 0.8.1 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

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:

```
In [6]: my_plot = gca()
```

and to our line we plotted, which is the first in the plot:

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

Now we can set properties using `set_something` methods:

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

or the `setp` function:

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

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

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

We can also add several lines to one plot:

```
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:

```
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:

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

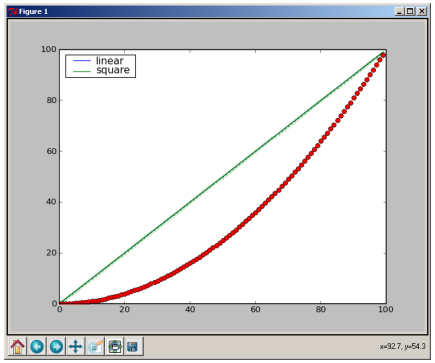
and print it anew providing new line styles (a green dotted line with crosses for the linear and a red dashed line with circles for the square graph):

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

Now we add the legend at the upper left corner:

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

The result looks like this:



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.

1.6.5 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:

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

There are many line styles that can be specified with symbols:

Symbol	Description
-	solid line
–	dashed line
-.	dash-dot line
:	dotted line
.	points
,	pixels
o	circle symbols
^	triangle up symbols
v	triangle down symbols
<	triangle left symbols
>	triangle right symbols
s	square symbols
+	plus symbols
x	cross symbols
D	diamond symbols
d	thin diamond symbols
1	tripod down symbols
2	tripod up symbols
3	tripod left symbols
4	tripod right symbols
h	hexagon symbols
H	rotated hexagon symbols
p	pentagon symbols
|	vertical line symbols
_	horizontal line symbols
steps	use gnuplot style 'steps' # kwarg only

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:

Abbreviation	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

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

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.

1.6.6 Text

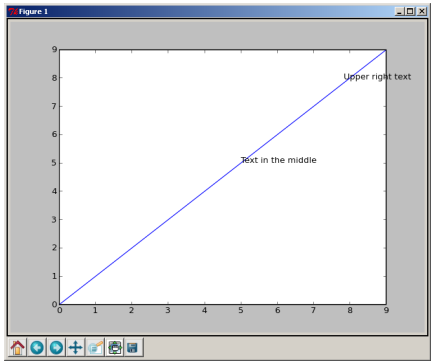
We've already used some commands to add text to our figure: `xlabel` `ylabel`, and `title`.

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

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

`figtext` uses figure coordinates form 0 to 1:

```
In [4]: t2 = figtext(0.8, 0.8, 'Upper right text')
```



`matplotlib` supports TeX mathematical expression. So `r'$\pi$'` will show up as:

$$\pi$$

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


```
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:

argument	coordinate system
figure points	points from the lower left corner of the figure
figure pixels	pixels from the lower left corner of the figure
figure fraction	0,0 is lower left of figure and 1,1 is upper, right
axes points	points from lower left corner of axes
axes pixels	pixels from lower left corner of axes
axes fraction	0,1 is lower left of axes and 1,1 is upper right
data	use the axes data coordinate system

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:

```
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'})
```

Exercise

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

1.6.7 Ticks

Where and What

Well formated 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 formated independently from each other. Per default minor ticks are not shown, i.e. there is only an empty list for them because it is as `NullLocator` (see below).

Tick Locators

There are several locators for different kind of requirements:

Class	Description
NullLocator	no ticks
IndexLocator	locator for index plots (e.g. where `x = range(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	choose a MultipleLocator and dynamically reassign

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`:

Class	Description
MinuteLocator	locate minutes
HourLocator	locate hours
DayLocator	locate specified days of the month
WeekdayLocator	locate days of the week, e.g. MO, TU
MonthLocator	locate months, e.g. 10 for October
YearLocator	locate years that are multiples of base
RRuleLocator	locate using a matplotlib.dates.rrule

Tick Formatters

Similarly to locators, there are formatters:

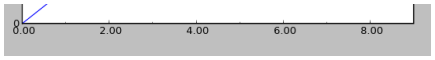
Class	Description
NullFormatter	no labels on the ticks
FixedFormatter	set the strings manually for the labels
FuncFormatter	user defined function sets the labels
FormatStrFormatter	use a sprintf format string
IndexFormatter	cycle through fixed strings by tick position
ScalarFormatter	default formatter for scalars; autopick the fmt string
LogFormatter	formatter for log axes
DateFormatter	use an strftime string to format the date

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`:

```
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:



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.

1.6.8 Figures, Subplots, and Axes

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.

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:

Argument	Default	Description
`num`	`1`	number of figure
`figsize`	`figure.figsize`	figure size in in inches (width, height)
`dpi`	`figure.dpi`	resolution in dots per inch
`facecolor`	`figure.facecolor`	color of the drawing background
`edgecolor`	`figure.edgecolor`	color of edge around the drawing background
`frameon`	`True`	draw figure frame or not

The defaults can be specified in the resource file and will be used most of the time. Only the number of the figure is frequently changed.

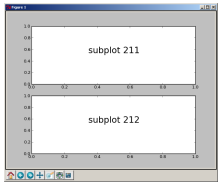
When you work with the GUI you can close a figure by clicking on the x in the upper right corner. But you can close a figure programmatically by calling `close`. Depending on the argument it closes (1) the current figure (no argument), (2) a specific figure (figure number or figure instance as argument), or (3) all figures (`all` as argument).

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

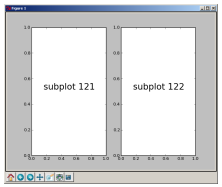
Subplots

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.

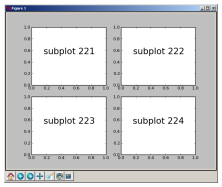
A plot with two rows and one column is created with `subplot(211)` and `subplot(212)`. The result looks like this:



If you want two plots side by side, you create one row and two columns with `subplot(121)` and `subplot(112)`. 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:



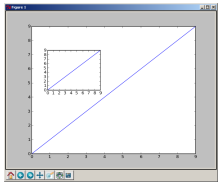
Frequently, you don't want all subplots to have ticks or labels. You can set the `xticklabels` or the `yticklabels` to an empty list (`[]`). Every subplot defines the methods `'is_first_row, is_first_col, is_last_row, is_last_col`. These can help to set ticks and labels only for the outer pots.

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`:

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

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

In [24]: plot(x)
```

The result looks like this:



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.

1.6.9 Other Types of Plots

Many More

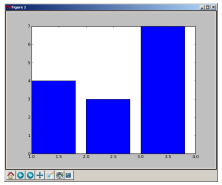
So far we have used only line plots. `matplotlib` offers many more types of plots. We will have a brief look at some of them. All functions have many optional arguments that are not shown here.

Bar Charts

The function `bar` creates a new bar chart:

```
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:



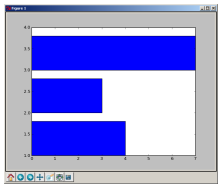
The default column width is 0.8. It can be changed with common methods by setting `width`. As it can be `color` and `bottom`, we can also set an error bar with `yerr` or `xerr`.

Horizontal Bar Charts

The function `barh` creates an vertical bar chart. Using the same data:

```
barh([1, 2, 3], [4, 3, 7])
```

We get:



Broken Horizontal Bar Charts

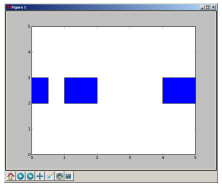
We can also have discontinuous vertical bars with `broken_barh`. We specify start and width of the range in y-direction and all start-width pairs in x-direction:

```
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:

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

and get this:



Box and Whisker Plots

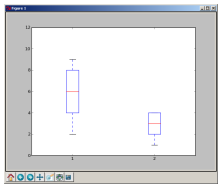
We can draw box and whisker plots:

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

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

```
ax = gca()
ax.set_ylim(0, 12)
draw()
```

Our plot looks like this:



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 `whis*(75%-25%)` of the data.

Contour Plots

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

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

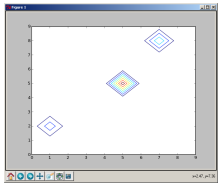
and also a 2D array for z:

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


Now we can make a simple contour plot:

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

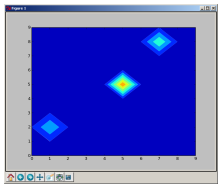
Our plot looks like this:



We can also fill the area. We just use numbers form 0 to 9 for the values `v`:

```
v = x
contourf(x, x, z, v)
```

Now our plot area is filled:



Histograms

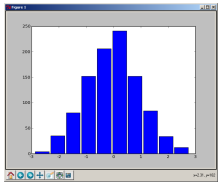
We can make histograms. Let's get some normally distributed random numbers from `numpy`:

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

Now we make a simple histogram:

```
hist(r_numbers)
```

With 100 numbers our figure looks pretty good:



Loglog Plots

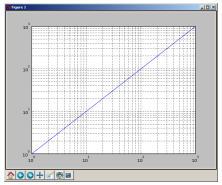
Plots with logarithmic scales are easy:

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

We set the mayor and minor grid:

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

Now we have loglog plot:



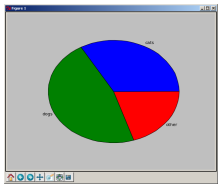
If we want only one axis with a logarithmic scale we can use `semilogx` or `semilogy`.

Pie Charts

Pie charts can also be created with a few lines:

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

The result looks as expected:



Polar Plots

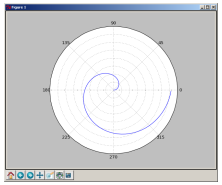
Polar plots are also possible. Let's define our `r` from 0 to 360 and our `theta` from 0 to 360 degrees. We need to convert them to radians:

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

Now plot in polar coordinates:

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

We get a nice spiral:



Arrow Plots

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

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

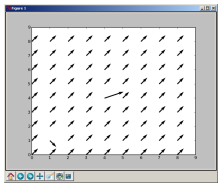
The x and y components of the arrows are specified as 2D arrays:

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

Now we can plot the arrows:

```
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:



Scatter Plots

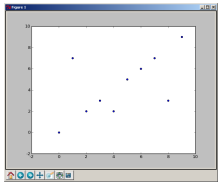
Scatter plots print x vs. y diagrams. We define `x` and `y` and make some point in `y` random:

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

Now make a scatter plot:

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

The three different values for `y` break out of the line:



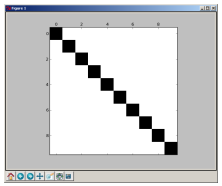
Sparsity Pattern Plots

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:

```
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:

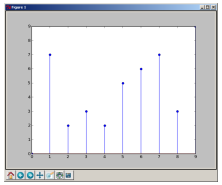
```
spy(i)
```



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):

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



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
dates
[datetime.datetime(2000, 1, 1, 0, 0),
 datetime.datetime(2000, 1, 16, 0, 0),
 datetime.datetime(2000, 1, 31, 0, 0),
 datetime.datetime(2000, 2, 15, 0, 0),
 datetime.datetime(2000, 3, 1, 0, 0),
 datetime.datetime(2000, 3, 16, 0, 0),
 datetime.datetime(2000, 3, 31, 0, 0),
 datetime.datetime(2000, 4, 15, 0, 0),
```



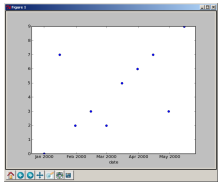
```
 datetime.datetime(2000, 4, 30, 0, 0),
 datetime.datetime(2000, 5, 15, 0, 0)]
```

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

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

Now we can plot the dates at the x axis:

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



1.6.10 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.

The Figure Class

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

```
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:

```
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.

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.

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.

Example

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

```
#file matplotlib/oo.py

from matplotlib.figure import Figure          #1

figsize = (8, 5)                              #2
fig = Figure(figsize=figsize)                 #3
ax = fig.add_subplot(111)                     #4
line = ax.plot(range(10))[0]                  #5
ax.set_title('Plotted with OO interface')     #6
ax.set_xlabel('measured')
ax.set_ylabel('calculated')
ax.grid(True)                                 #7
line.set_marker('o')                          #8

from matplotlib.backends.backend_agg import FigureCanvasAgg #9
canvas = FigureCanvasAgg(fig)                 #10
canvas.print_figure("oo.png", dpi=80)         #11


import Tkinter as Tk                          #12
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg #13

root = Tk.Tk()                                #13
canvas2 = FigureCanvasTkAgg(fig, master=root) #14
canvas2.show()                                #15
canvas2.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1) #16
Tk.mainloop()                                 #17

from matplotlib import _pylab_helpers         #18
import pylab                                  #19

pylab_fig = pylab.figure(1, figsize=figsize)  #20
figManager = _pylab_helpers.Gcf.get_active()  #21
figManager.canvas.figure = fig                #22
pylab.show()                                  #23
```

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.

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.

1.7 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 submodules and functions in scipy would be very boring, we concentrate instead on a few examples to give a general idea of how to use `scipy` for scientific computing.

To begin with

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

scipy is mainly composed of task-specific sub-modules:

cluster	Vector quantization / Kmeans
fftpack	Fourier transform
integrate	Integration routines
interpolate	Interpolation
io	Data input and output
linalg	Linear algebra routines
maxentropy	Routines for fitting maximum entropy models
ndimage	n-dimensional image package
odr	Orthogonal distance regression
optimize	Optimization
signal	Signal processing
sparse	Sparse matrices
spatial	Spatial data structures and algorithms
special	Any special mathematical functions
stats	Statistics

1.7.1 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:

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

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

```
>>> 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 *`.

1.7.2 File input/output: `scipy.io`

- Loading and saving matlab files:

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

See also:

- Load text files:

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

- Clever loading of text/csv files:

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

- Fast an efficient, but numpy-specific, binary format:

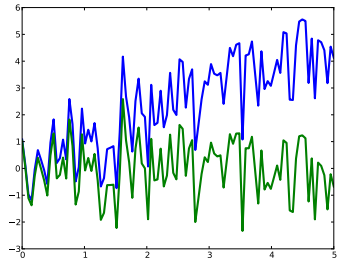
```
np.save/np.load
```

1.7.3 Signal processing: `scipy.signal`

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

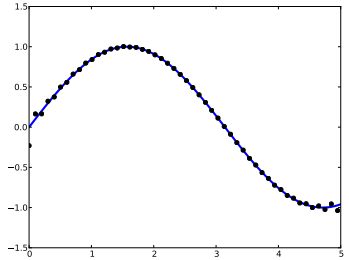
- Detrend: remove linear trend from signal:

```
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.


```
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')
```



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.

1.7.4 Special functions: `scipy.special`

Special functions are transcendal 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.ellipj* for the Jacobian elliptic function, ...)
- Gamma function: *special.gamma*, alos note *special.gammaln* which will give the log of Gamma to a higher numerical precision.
- Erf, the area under a Gaussian curve: *special.erf*

1.7.5 Statistics and random numbers: `scipy.stats`

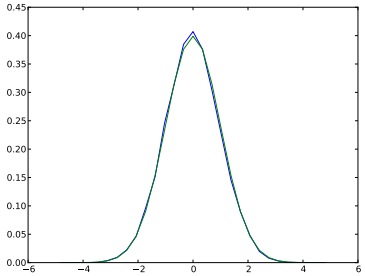
The module *scipy.stats* contains statistical tools and probabilistic description of random processes. Random number generators for various random process can be found in *numpy.random*.

Histogram and probability density function

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

```
>>> a = np.random.normal(size=1000)
>>> bins = np.arange(-4, 5)
>>> bins
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> histogram = np.histogram(a, bins=bins, normed=True)[0]
>>> bins = 0.5*(bins[1:] + bins[:-1])
>>> 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)
```

```
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:

```
>>> loc, std = stats.norm.fit(a)
>>> loc
0.003738964114102075
>>> std
0.97450996668871193
```

Percentiles

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

```
>>> np.median(a)
0.0071645570292782519
```

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

```
>>> stats.scoreatpercentile(a, 50)
0.0071645570292782519
```

Similarly, we can calculate the percentile 90:

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

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

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:

```
>>> 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.

1.7.6 Linear algebra operations: `scipy.linalg`

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

```
>>> from scipy import linalg
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> linalg.det(arr)
-2.0
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.det(arr)
0.0
>>> 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:

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

```
>>> 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`:

```
>>> 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):

```
>>> arr = np.arange(12).reshape((3, 4)) + 1
>>> uarr, spec, vharr = linalg.svd(arr)
```

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`.

1.7.7 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:

```
``dy/dt = rhs(y1, y2, .., t0,...)``
```

As an introduction, let us solve the ODE `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:

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

The trajectory will now be computed:

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

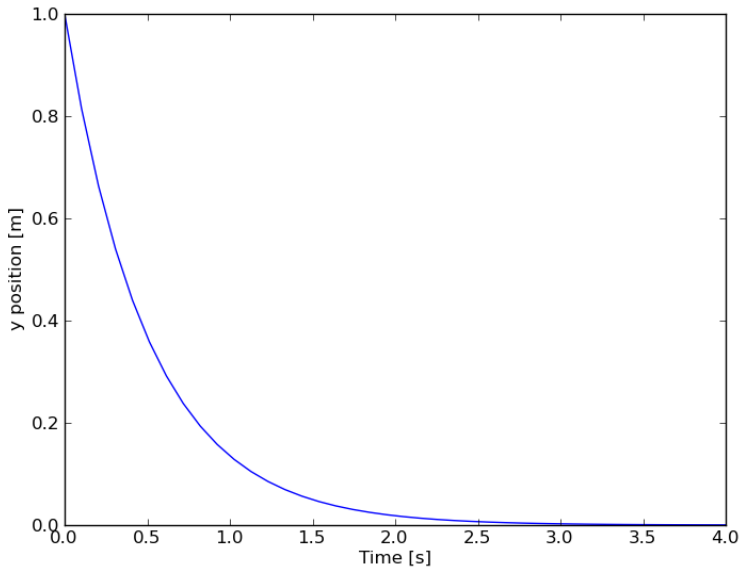
Thus the derivative function has been called more than 40 times:

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

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

```
>>> 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 eps wo y' + wo^2 y = 0` with `wo^2 = k/m` being `k` the spring constant, `m` the mass and `eps=c/(2 m wo)` with `c` the damping coefficient. For a computing example, the parameters will be:

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

so the system will be underdamped because:

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

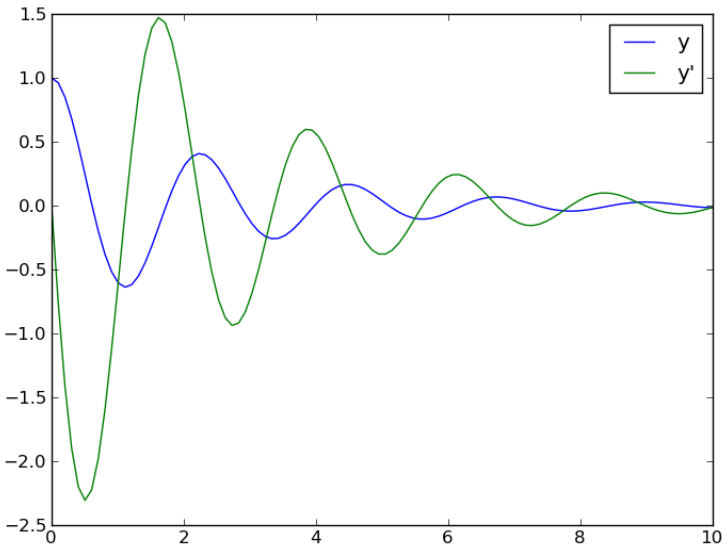
For the `odeint` solver the 2nd order equation needs to be transformed in a system of two first-order equations for the vector `Y=(y, y')`. It will be convenient to define `nu = 2 eps wo = c / m` and `om = wo^2 = k/m`:

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

Thus the function will calculate the velocity and acceleration by:

```
>>> 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.

1.7.8 Fast Fourier transforms: `scipy.fftpack`

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

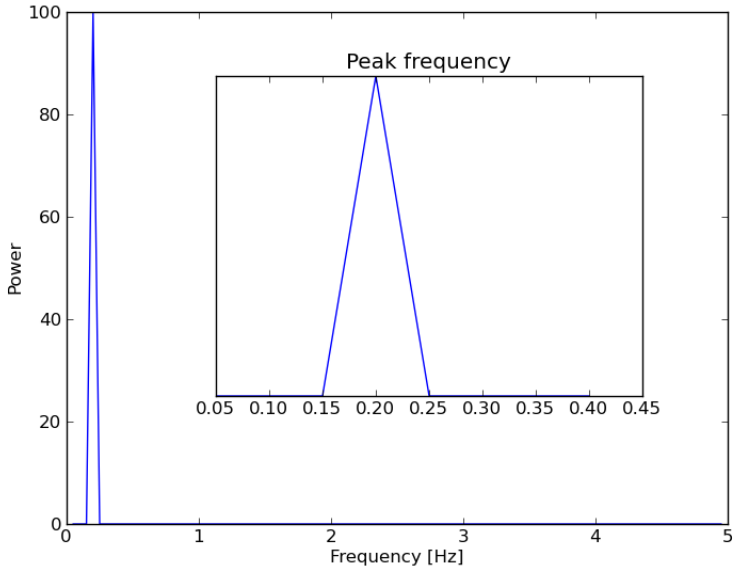
```
>>> time_step = 0.1
>>> period = 5.
>>> time_vec = np.arange(0, 20, time_step)
>>> sig = np.sin(2 * np.pi / period * time_vec) + \
...       np.cos(10 * np.pi * time_vec)
```

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:

```
>>> 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:

```
>>> pidxs = np.where(sample_freq > 0)
>>> freqs = sample_freq[pidxs]
>>> power = np.abs(sig_fft)[pidxs]
```

Thus the signal frequency can be found by:

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

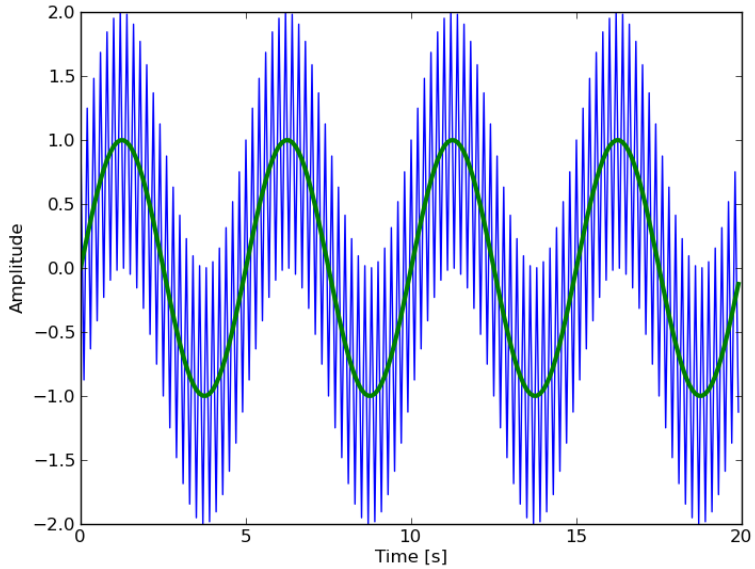
Now only the main signal component will be extracted from the Fourier transform:

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

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

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

The result is shown on the Matplotlib figure generated by the fftpack-illustration.py script.



1.7.9 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:

```
>>> 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 built a linear interpolation function:

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

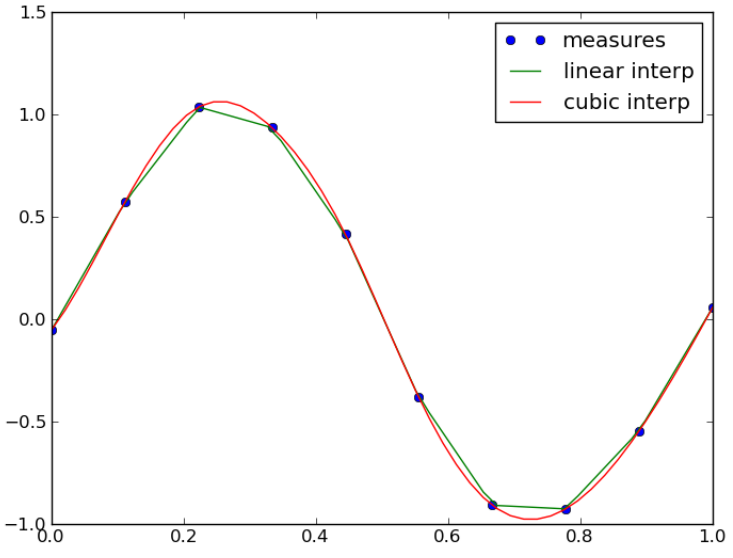
Then the `linear_interp` instance needs to be evaluated on time of interest:

```
>>> computed_time = np.linspace(0, 1, 50)
>>> linear_results = linear_interp(computed_time)
```

A cubic interpolation can also be selected by providing the `kind` optional keyword argument:

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

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



`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 exercice on **'Maximum wind speed prediction at the Sprogø station'_** for a more advance spline interpolation example.

1.7.10 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.

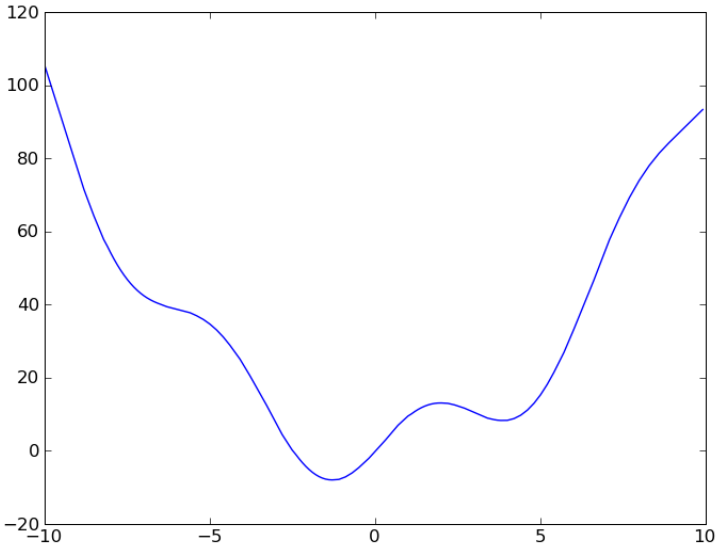
Example: Minimizing a scalar function using different algorithms

Let's define the following function:

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

and plot it:

```
>>> 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.

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:

```
>>> 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 opimtization.

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:

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


This approach take 20 ms on our computer.

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

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

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

```
>>> # 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 *Non linear least squares curve fitting: application to point extraction in topographical lidar data* (page 95) for a more advanced example.

1.7.11 Image processing: `scipy.ndimage`

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

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

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

Geometrical transformations on images

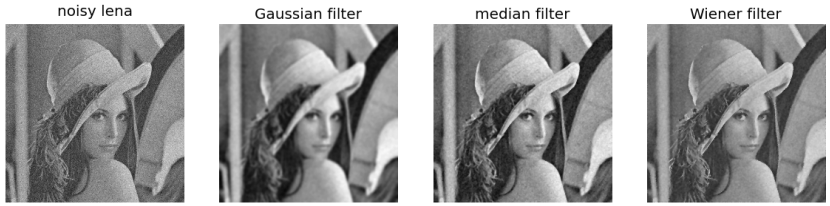
Changing orientation, resolution, ..

```
>>> 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)
>>> zoomed_lena.shape
(1024, 1024)
```

```
In [35]: subplot(151)
Out[35]: <matplotlib.axes.AxesSubplot object at 0x925f46c>

In [36]: imshow(shifted_lena, cmap=cm.gray)
Out[36]: <matplotlib.image.AxesImage object at 0x9593f6c>

In [37]: axis('off')
Out[37]: (-0.5, 511.5, 511.5, -0.5)

In [39]: # etc.
```

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))
```



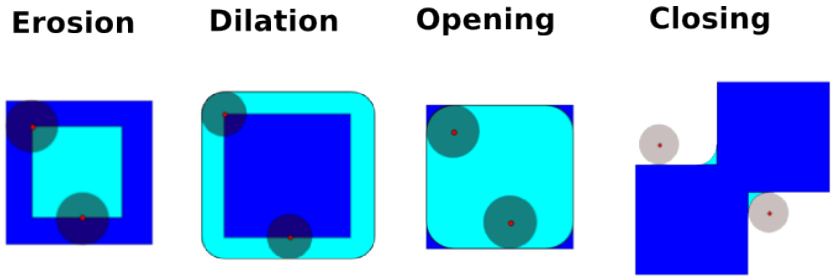
And many other filters in `scipy.ndimage.filters` and `scipy.signal` can be applied to images

Exercise

Compare histograms for the different filtered images.

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

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



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

- **Erosion**

```
>>> a = np.zeros((7,7), dtype=np.int)
>>> 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
>>> ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

- **Dilation**

```
>>> 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**

```
>>> 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, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> ndimage.binary_opening(a).astype(np.int)
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]])
```

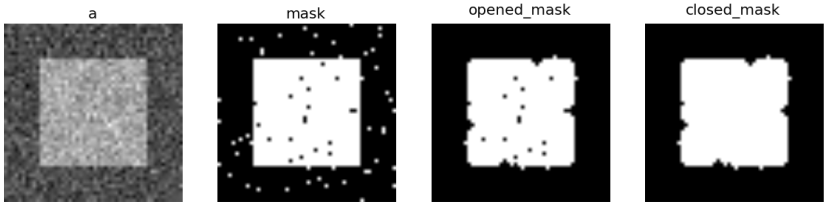
- **Closing:** `ndimage.binary_closing`

Exercise

Check that opening amounts to eroding, then dilating.

An opening operation removes small structures, while a closing operation fills small holes. Such operation can therefore be used to "clean" an image.

```
>>> 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.

```
>>> 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]])
```

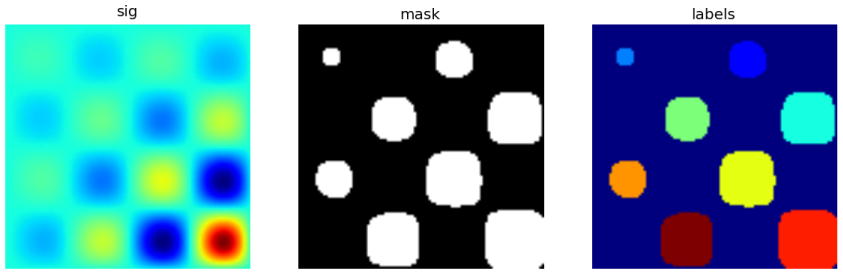
Measurements on images

Let us first generate a nice synthetic binary image.

```
>>> x, y = np.indices((100, 100))
>>> sig = np.sin(2*np.pi*x/50.)*np.sin(2*np.pi*y/50.)*(1+x*y/50.**2)**2
>>> mask = sig > 1
```

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

```
>>> labels, nb = ndimage.label(mask)
>>> nb
8
>>> areas = ndimage.sum(mask, labels, xrange(1, labels.max()+1))
>>> areas
[190.0, 45.0, 424.0, 278.0, 459.0, 190.0, 549.0, 424.0]
>>> maxima = ndimage.maximum(sig, labels, xrange(1, labels.max()+1))
>>> maxima
[1.8023823799830032, 1.1352760475048373, 5.5195407887291426,
2.4961181804217221, 6.7167361922608864, 1.8023823799830032,
16.765472169131161, 5.5195407887291426]
>>> ndimage.find_objects(labels==4)
[(slice(30, 48, None), slice(30, 48, None))]
>>> sl = ndimage.find_objects(labels==4)
>>> imshow(sig[sl[0]])
```



See the summary exercise on *Image processing application: counting bubbles and unmolten grains* (page 99) for a more advanced example.

1.7.12 Summary exercices on scientific computing

The summary exercices 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 exercices.

Maximum wind speed prediction at the Sprogø station

The exercice 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 probabilites

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:

```
>>> 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:

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

and they are assumed to fit the given wind speeds:

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

Prediction with UnivariateSpline

In this section the quantile function will be estimated by using the `UnivariateSpline` class which can represent a spline from points. The default behavior is to build a spline of degree 3 and points can have different weights according to their reliability. Variantes 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:

```
>>> 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 probabilties:

```
>>> nprob = np.linspace(0, 1, 1e2)
>>> fitted_max_speeds = quantile_func(nprob)
```

2%

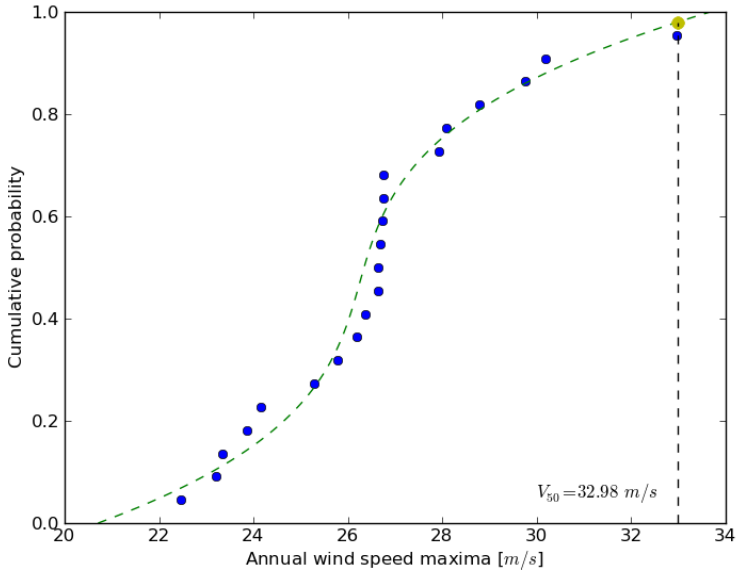
In the current model, the maximum wind speed occuring every 50 years is defined as the upper 2% quantile. As a result, the cumulative probability value will be:

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

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

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

The results are now gathered on a Matplotlib figure.

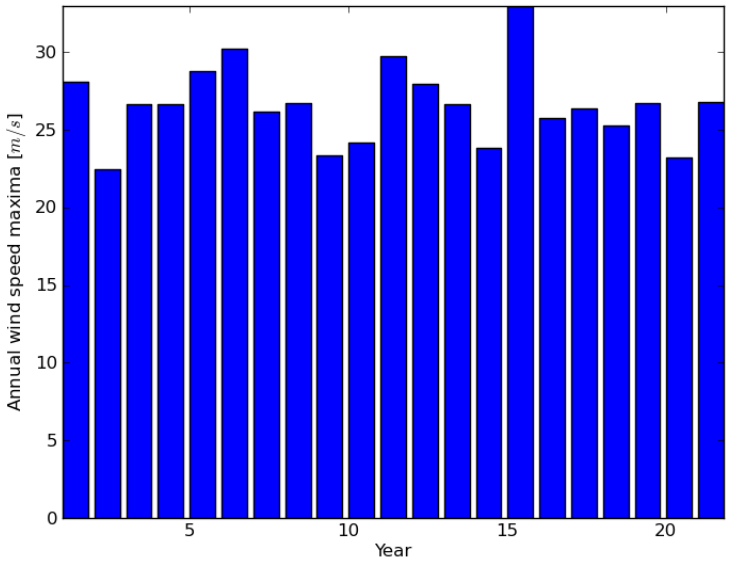


All those steps have been gathered in the script cumulative-wind-speed-prediction.py.

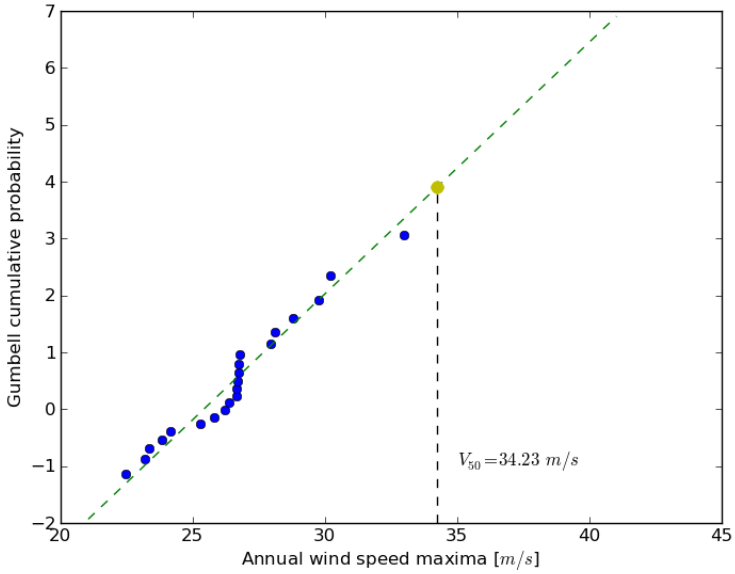
Exercice with the Gumbell distribution

The interested readers are now invited to make an exercice 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 exercice 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 occuring every 50 years.

Once done, you may compare your code with a solution example available in the script gumbell-wind-speed-prediction.py.

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 ang go directly to *Loading and visualization* (page 96).

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 sytem 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] (page 171) 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 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.

[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 DRAIX.

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

Therefore, we use the `scipy.optimize` module to fit a waveform to one or a sum of Gaussian functions.

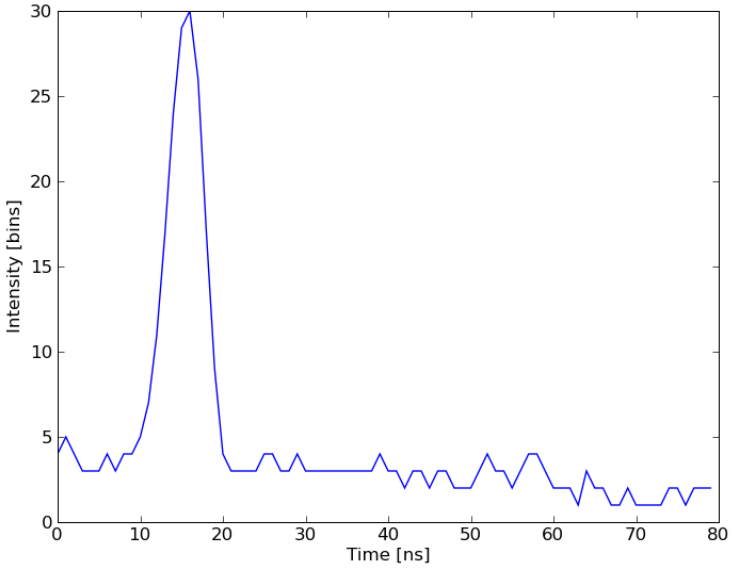
Loading and visualization

Load the first waveform using:

```
>>> import numpy as np
>>> waveform_1 = np.load('data/waveform_1.npy')
```

and visualize it:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(len(waveform_1))
>>> plt.plot(t, waveform_1)
>>> plt.show()
```



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:

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

where

- `coeffs[0]` is B (noise)
- `coeffs[1]` is A (amplitude)
- `coeffs[2]` is μ (center)
- `coeffs[3]` is σ (width)

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

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

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

```
>>> def residuals(coeffs, y, t):
...     return y - model(t, coeffs)
```

So let's get our solution by calling `scipy.optimize.leastsq` with the following arguments:

- the function to minimize
- an initial solution
- the additional arguments to pass to the function

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

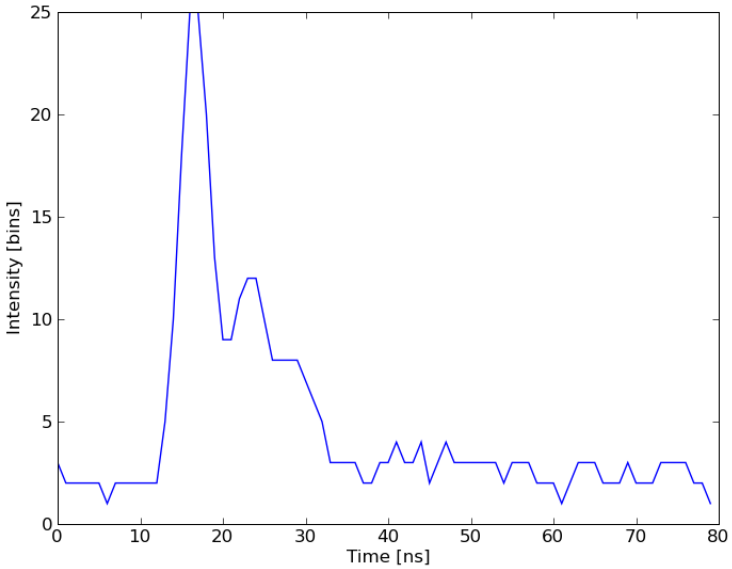
And visualize the solution:

```
>>> 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.



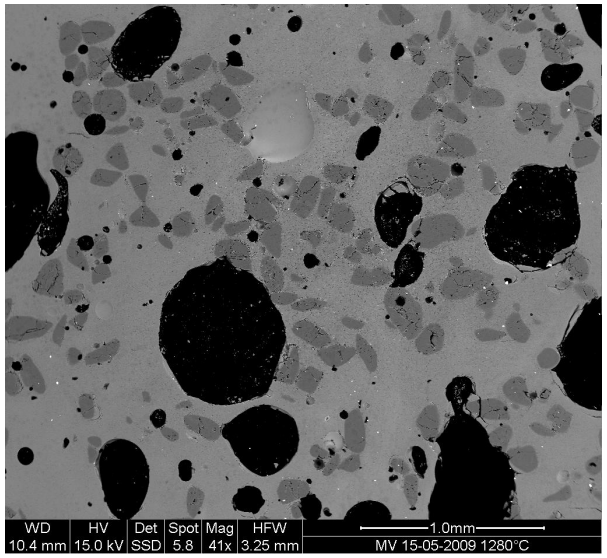
- In some cases, writing an explicit function to compute the Jacobian is faster than letting `leastsq` estimate it numerically. Create a function to compute the Jacobian of the residuals and use it as an input for `leastsq`.
- When we want to detect very small peaks in the signal, or when the initial 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:

```
>>> 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.

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.

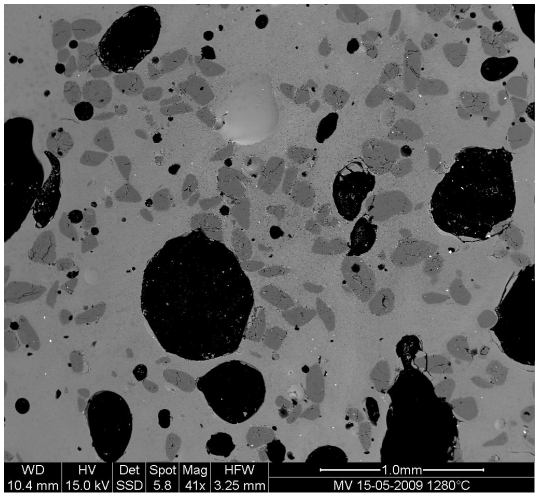
6. Use mathematical morphology to clean the different phases.

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

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



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

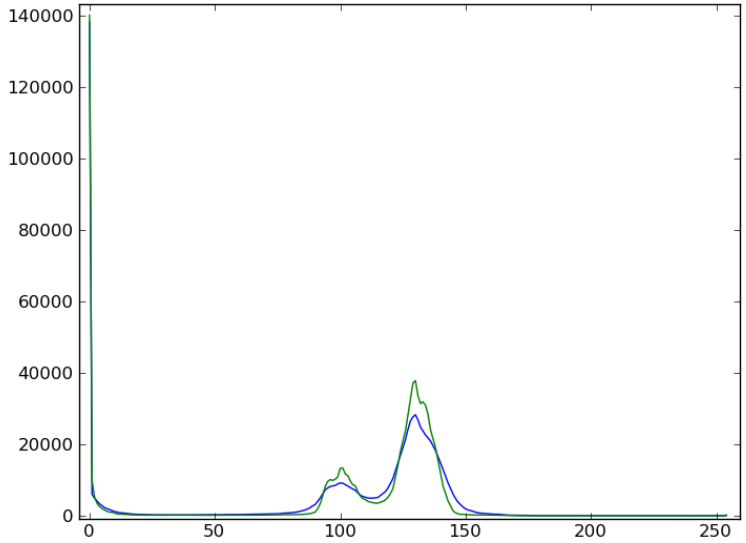
```
>>> dat = imread('MV_HFV_012.jpg')
```

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

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

3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.

```
>>> filtdat = ndimage.median_filter(dat, size=(7,7))
>>> hi_dat = np.histogram(dat, bins=np.arange(256))
>>> hi_filtdat = np.histogram(filtdat, bins=np.arange(256))
```

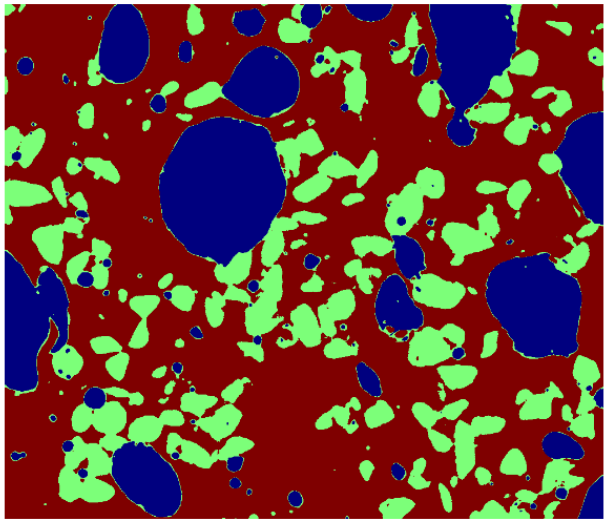



4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option (homework): write a function that determines automatically the thresholds from the minima of the histogram.

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

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

```
>>> phases = void.astype(np.int) + 2*glass.astype(np.int) +\
        3*sand.astype(np.int)
```

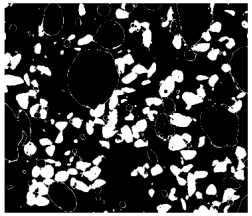
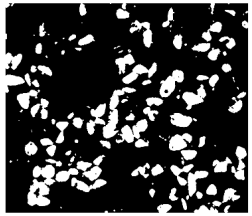
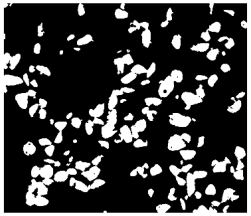


6. Use mathematical morphology to clean the different phases.

```
>>> sand_op = ndimage.binary_opening(sand, iterations=2)
```

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

```
>>> sand_labels, sand_nb = ndimage.label(sand_op)
>>> sand_areas = np.array(ndimage.sum(sand_op, sand_labels,\
...     np.arange(sand_labels.max()+1)))
>>> mask = sand_areas>100
>>> remove_small_sand = mask[sand_labels.ravel()].reshape(sand_labels.shape)
```

8. Compute the mean size of bubbles.

```
>>> 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)
```

CHAPTER 2

Advanced topics

2.1 Advanced Numpy

2.1.1 Abstract

Numpy is at the base of Python's scientific stack of tools. Its purpose is simple: implementing efficient operations on many items in a block of memory. Understanding how it works in detail helps in making efficient use of its flexibility, taking useful shortcuts, and in building new work based on it.

This tutorial aims to cover:

- Anatomy of Numpy arrays, and its consequences. Tips and tricks.
- Universal functions: what, why, and what to do if you want a new one.
- Integration with other tools: Numpy offers several ways to wrap any data in an ndarray, without unnecessary copies.
- Recently added features, and what's in them for me: PEP 3118 buffers, generalized ufuncs, ...

2.1.2 Advanced Numpy

July 8, @ EuroScipy 2010

Pauli Virtanen

Prerequisites

- Numpy (>= 1.2; preferably newer...)
- Cython (>= 0.12, for the Ufunc example)
- PIL (I'll use it in a couple of examples)
- Source codes:
  - http://pav.iki.fi/tmp/advnumpy-ex.zip (updated recently)

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


Contents

- Advanced Numpy (page 104)
- Life of ndarray (page 105)
- Universal functions (page 116)
- Interoperability features (page 125)
- Siblings: `chararray`, `maskedarray`, `matrix` (page 133)
- Summary (page 134)
- Hit list of the future for Numpy core (page 134)
- Contributing to Numpy/Scipy (page 134)
- Python 2 and 3, single code base (page 137)

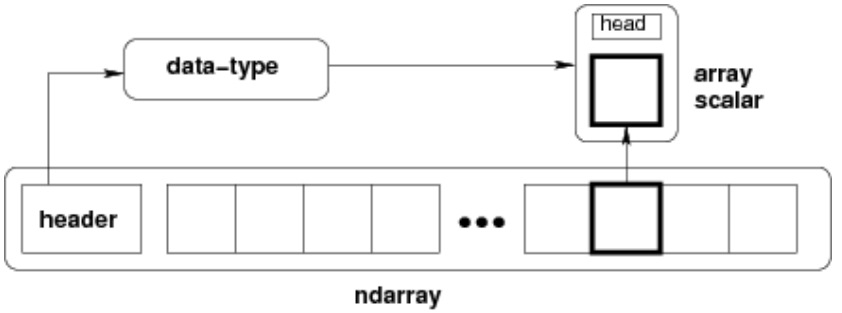
2.1.3 Life of ndarray

It's...

ndarray =

block of memory + indexing scheme + data type descriptor

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




```
typedef struct PyArrayObject {
        PyObject_HEAD

        /* Block of memory */
        char *data;

        /* Data type descriptor */
        PyArray_Descr *descr;

        /* Indexing scheme */
        int nd;
        npy_intp *dimensions;
        npy_intp *strides;

        /* Other stuff */
        PyObject *base;
        int flags;
        PyObject *weakreflist;
} PyArrayObject;
```

Block of memory

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int32)
>>> x.data
<read-write buffer for 0xa37bfd8, size 16, offset 0 at 0xa4eabe0>
>>> str(x.data)
'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00\x04\x00\x00\x00'
```

Memory address of the data:

```
>>> x.__array_interface__['data'][0]
159755776
```

Reminder: two `ndarrays` may share the same memory:

```
>>> x = np.array([1,2,3,4])
>>> y = x[:-1]
>>> x[0] = 9
>>> y
array([9, 2, 3])
```

Memory does not need to be owned by an `ndarray`:

```
>>> x = '1234'
>>> y = np.frombuffer(x, dtype=np.int8)
>>> y.data
<read-only buffer for 0xa588ba8, size 4, offset 0 at 0xa55cd60>
>>> y.base is x
True
```

```
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : False
  ALIGNED : True
  UPDATEIFCOPY : False
```

The `owndata` and `writeable` flags indicate status of the memory block.

Data types

The descriptor

`dtype` describes a single item in the array:

type	**scalar type** of the data, one of: int8, int16, float64, *et al.* (fixed size) str, unicode, void (flexible size)
itemsize	**size** of the data block
byteorder	**byte order**: big-endian > / little-endian < / not applicable |
fields	sub-dtypes, if it's a **structured data type**
shape	shape of the array, if it's a **sub-array**

```
>>> np.dtype(int).type
<type 'numpy.int32'>
>>> np.dtype(int).itemsize
4
>>> np.dtype(int).byteorder
'='
```


Example: reading `.wav` files

The `.wav` file header:

chunk_id	"RIFF"
chunk_size	4-byte unsigned little-endian integer
format	"WAVE"
fmt_id	"fmt "
fmt_size	4-byte unsigned little-endian integer
audio_fmt	2-byte unsigned little-endian integer
num_channels	2-byte unsigned little-endian integer
sample_rate	4-byte unsigned little-endian integer
byte_rate	4-byte unsigned little-endian integer
block_align	2-byte unsigned little-endian integer
bits_per_sample	2-byte unsigned little-endian integer
data_id	"data"
data_size	4-byte unsigned little-endian integer

- 44-byte block of raw data (in the beginning of the file)
- ... followed by `data_size` bytes of actual sound data.

The `.wav` file header as a Numpy *structured* data type:

```
>>> wav_header_dtype = np.dtype([
     ("chunk_id", (str, 4)),    # flexible-sized scalar type, item size 4
     ("chunk_size", "<u4"),     # little-endian unsigned 32-bit integer
     ("format", "S4"),          # 4-byte string
     ("fmt_id", "S4"),
     ("fmt_size", "<u4"),
     ("audio_fmt", "<u2"),      #
     ("num_channels", "<u2"),   # .. more of the same ...
     ("sample_rate", "<u4"),    #
     ("byte_rate", "<u4"),
     ("block_align", "<u2"),
     ("bits_per_sample", "<u2"),
     ("data_id", ("S1", (2, 2))), # sub-array, just for fun!
     ("data_size", "u4"),
     #
     # the sound data itself cannot be represented here:
     # it does not have a fixed size
])
```

See Also:

wavreader.py

```
>>> wav_header_dtype['format']
dtype('|S4')
>>> wav_header_dtype.fields
<dictproxy object at 0x85e9704>
>>> wav_header_dtype.fields['format']
(dtype('|S4'), 8)
```

- The first element is the sub-dtype in the structured data, corresponding to the name `format`
- The second one is its offset (in bytes) from the beginning of the item

Note: Mini-exercise, make a "sparse" dtype by using offsets, and only some of the fields:

```
>>> wav_header_dtype = np.dtype(dict(
    names=['format', 'sample_rate', 'data_id'],
    offsets=[offset_1, offset_2, offset_3], # counted from start of structure in bytes
    formats=list of dtypes for each of the fields,
))
```

and use that to read the sample rate, and `data_id` (as sub-array).

```
>>> f = open('test.wav', 'r')
>>> wav_header = np.fromfile(f, dtype=wav_header_dtype, count=1)
>>> f.close()
>>> print(wav_header)
[ ('RIFF', 17402L, 'WAVE', 'fmt ', 16L, 1, 1, 16000L, 32000L, 2, 16,
  [['d', 'a'], ['t', 'a']], 17366L)]
>>> wav_header['sample_rate']
array([16000], dtype=uint32)
```

Let's try accessing the sub-array:

```
>>> wav_header['data_id']
array([[['d', 'a'],
        ['t', 'a']]],
      dtype='|S1')
>>> wav_header.shape
(1,)
>>> wav_header['data_id'].shape
(1, 2, 2)
```

When accessing sub-arrays, the dimensions get added to the end!

Note: There are existing modules such as `wavfile`, `audiolab`, etc. for loading sound data...

Casting and re-interpretation/views

casting

- on assignment
- on array construction
- on arithmetic
- etc.
- and manually: `.astype(dtype)`

data re-interpretation

- manually: `.view(dtype)`

Casting

- Casting in arithmetic, in nutshell:
  - only type (not value!) of operands matters
  - largest "safe" type able to represent both is picked
  - scalars can "lose" to arrays in some situations
- Casting in general copies data

```
>>> x = np.array([1, 2, 3, 4], dtype=np.float)
>>> x
array([ 1.,  2.,  3.,  4.])
>>> y = x.astype(np.int8)
>>> y
array([1, 2, 3, 4], dtype=int8)
>>> y + 1
array([2, 3, 4, 5], dtype=int8)
>>> y + 256
```



```
array([1, 2, 3, 4], dtype=int8)
>>> y + 256.0
array([ 257.,  258.,  259.,  260.])
>>> y + np.array([256], dtype=np.int32)
array([258, 259, 260, 261])
```

- Casting on setitem: dtype of the array is not changed on item assignment

```
>>> y[:] = y + 1.5
>>> y
array([2, 3, 4, 5], dtype=int8)
```

Note: Exact rules: see documentation: http://docs.scipy.org/doc/numpy/reference/ufuncs.html#casting-rules

Re-interpretation / viewing

- Data block in memory (4 bytes)

0x01	0x02	0x03	0x04

  - 4 of uint8, OR,
  - 4 of int8, OR,
  - 2 of int16, OR,
  - 1 of int32, OR,
  - 1 of float32, OR,
  - ...

  How to switch from one to another?

1. Switch the dtype:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.uint8)
>>> x.dtype = "<i2"
>>> x
array([ 513, 1027], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
```

0x01	0x02	0x03	0x04

   Note: little-endian: least significant byte is on the *left* in memory

2. Create a new view:

```
>>> y = x.view("<i4")
>>> y
array([67305985])
>>> 0x04030201
67305985
```

0x01	0x02	0x03	0x04

Note:

- `.view()` makes *views*, does not copy (or alter) the memory block
- only changes the dtype (and adjusts array shape)

```
>>> x[1] = 5
>>> y
array([328193])
>>> y.base is x
True
```

Mini-exercise: data re-interpretation

See Also:

view-colors.py

You have RGBA data in an array

```
>>> x = np.zeros((10, 10, 4), dtype=np.int8)
>>> x[:,:,0] = 1
>>> x[:,:,1] = 2
>>> x[:,:,2] = 3
>>> x[:,:,3] = 4
```

where the last three dimensions are the R, B, and G, and alpha channels.

How to make a (10, 10) structured array with field names 'r', 'g', 'b', 'a' without copying data?

```
>>> y = ...
```

```
>>> assert (y['r'] == 1).all()
>>> assert (y['g'] == 2).all()
>>> assert (y['b'] == 3).all()
>>> assert (y['a'] == 4).all()
```

Solution

```
>>> y = x.view([('r', 'i1'),
                ('g', 'i1'),
                ('b', 'i1'),
                ('a', 'i1')]
               )[:,:,0]
```

Warning: Another array taking exactly 4 bytes of memory:

```
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
>>> x = y.copy()
>>> x
array([[1, 2],
       [3, 4]], dtype=uint8)
>>> y
array([[1, 2],
       [3, 4]], dtype=uint8)
>>> x.view(np.int16)
array([[ 513],
       [1027]], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
>>> y.view(np.int16)
array([[ 769, 1026]], dtype=int16)
```

- What happened?
- ... we need to look into what `x[0,1]` actually means

```
>>> 0x0301, 0x0402
(769, 1026)
```


Indexing scheme: strides

Main point

The question

```
>>> x = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]], dtype=np.int8)
>>> str(x.data)
'\x01\x02\x03\x04\x05\x06\x07\x08\t'
```

At which byte in `x.data` does the item `x[1,2]` begin?

The answer (in Numpy)

- **strides**: the number of bytes to jump to find the next element
- 1 stride per dimension

```
>>> x.strides
(3, 1)
>>> byte_offset = 3*1 + 1*2   # to find x[1,2]
>>> x.data[byte_offset]
'\x06'
>>> x[1,2]
6
```

- simple, **flexible**

C and Fortran order

```
>>> x = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]], dtype=np.int16, order='C')
>>> x.strides
(6, 2)
>>> str(x.data)
'\x01\x00\x02\x00\x03\x00\x04\x00\x05\x00\x06\x00\x07\x00\x08\x00\t\x00'
```

- Need to jump 6 bytes to find the next row
- Need to jump 2 bytes to find the next column

```
>>> y = np.array(x, order='F')
>>> y.strides
(2, 6)
>>> str(y.data)
'\x01\x00\x04\x00\x07\x00\x02\x00\x05\x00\x08\x00\x03\x00\x06\x00\t\x00'
```

- Need to jump 2 bytes to find the next row
- Need to jump 6 bytes to find the next column
- Similarly to higher dimensions:
  - C: last dimensions vary fastest (= smaller strides)
  - F: first dimensions vary fastest

$$\text{shape} = (d_1, d_2, ..., d_n)$$
$$\text{strides} = (s_1, s_2, ..., s_n)$$
$$s_j^C = d_{j+1} d_{j+2} ... d_n \times \text{itemsize}$$
$$s_j^F = d_1 d_2 ... d_{j-1} \times \text{itemsize}$$

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

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

Transposition does not affect the memory layout of the data, only strides

```
>>> x.strides
(2, 1)
>>> y.strides
(1, 2)
```

```
>>> str(x.data)
'\x01\x02\x03\x04'
>>> str(y.data)
'\x01\x03\x02\x04'
```

- the results are different when interpreted as 2 of int16
- `.copy()` creates new arrays in the C order (by default)

Slicing with integers

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

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

```
>>> y = x[2:]
>>> y.__array_interface__['data'][0] - x.__array_interface__['data'][0]
8
```

```
>>> x = np.zeros((10, 10, 10), dtype=np.float)
>>> x.strides
(800, 80, 8)
>>> x[::2,::3,::4].strides
(1600, 240, 32)
```

Example: fake dimensions with strides

Stride manipulation

```
>>> from numpy.lib.stride_tricks import as_strided
>>> help(as_strided)
as_strided(x, shape=None, strides=None)
   Make an ndarray from the given array with the given shape and strides
```


Warning: `as_strided` does **not** check that you stay inside the memory block bounds...

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> as_strided(x, strides=(2*2,), shape=(2,))
array([1, 3], dtype=int16)
>>> x[::2]
array([1, 3], dtype=int16)
```

See Also:

stride-fakedims.py

Exercise

```
array([1, 2, 3, 4], dtype=np.int8)

-> array([[1, 2, 3, 4],
          [1, 2, 3, 4],
          [1, 2, 3, 4]], dtype=np.int8)
```

using only `as_strided`.:

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

Spoiler

Stride can also be *0*:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int8)
>>> y = as_strided(x, strides=(0, 1), shape=(3, 4))
>>> y
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int8)
>>> y.base.base is x
True
```

Broadcasting

- Doing something useful with it: outer product of `[1, 2, 3, 4]` and `[5, 6, 7]`

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> x2 = as_strided(x, strides=(0, 1*2), shape=(3, 4))
>>> x2
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int16)
```

```
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> y2 = as_strided(y, strides=(1*2, 0), shape=(3, 4))
>>> y2
array([[5, 5, 5, 5],
       [6, 6, 6, 6],
       [7, 7, 7, 7]], dtype=int16)
```

```
>>> x2 * y2
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

... seems somehow familiar ...

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> x[np.newaxis,:] * y[:,np.newaxis]
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

- Internally, array **broadcasting** is indeed implemented using 0-strides.

More tricks: diagonals

See Also:

stride-diagonals.py

Challenge

Pick diagonal entries of the matrix: (assume C memory order)

```
>>> x = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]], dtype=np.int32)
```

```
>>> x_diag = as_strided(x, shape=(3,), strides=(???,))
```

Pick the first super-diagonal entries `[2, 6]`.

And the sub-diagonals?

(**Hint to the last two: slicing first moves the point where striding** starts from.)

Solution

Pick diagonals:

```
>>> x_diag = as_strided(x, shape=(3,), strides=((3+1)*x.itemsize,))
>>> x_diag
array([1, 5, 9])
```

Slice first, to adjust the data pointer:

```
>>> as_strided(x[0,1:], shape=(2,), strides=((3+1)*x.itemsize,))
array([2, 6])
```

```
>>> as_strided(x[1:,0], shape=(2,), strides=((3+1)*x.itemsize,))
array([4, 8])
```

Note:

```
>>> y = np.diag(x, k=1)
>>> y
array([2, 6])
```

However,

```
>>> y.flags.owndata
True
```

It makes a copy?! Room for improvement... (bug #xxx)

See Also:

stride-diagonals.py

Challenge

Compute the tensor trace

```
>>> x = np.arange(5*5*5*5).reshape(5,5,5,5)
>>> s = 0
>>> for i in xrange(5):
>>>     for j in xrange(5):
>>>         s += x[j,i,j,i]
```

by striding, and using `sum()` on the result.

```
>>> y = as_strided(x, shape=(5, 5), strides=(TODO, TODO))
>>> s2 = ...
>>> assert s == s2
```

Solution

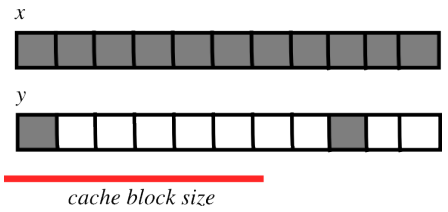
```
>>> y = as_strided(x, shape=(5, 5), strides=((5*5*5+5)*x.itemsize,
                                             (5*5+1)*x.itemsize))
>>> s2 = y.sum()
```

CPU cache effects

Memory layout can affect performance:

```
In [1]: x = np.zeros((20000,))

In [2]: y = np.zeros((20000*67,))[::67]

In [3]: x.shape, y.shape
((20000,), (20000,))

In [4]: %timeit x.sum()
100000 loops, best of 3: 0.180 ms per loop

In [5]: %timeit y.sum()
100000 loops, best of 3: 2.34 ms per loop

In [6]: x.strides, y.strides
((8,), (536,))
```

Smaller strides are faster?



- CPU pulls data from main memory to its cache in blocks
- If many array items consecutively operated on fit in a single block (small stride):
  - ⇒ fewer transfers needed
  - ⇒ faster

See Also:

Much more about this in the next session (Francesc Alted) today!

Example: inplace operations (caveat emptor)

- Sometimes,

```
>>> a -= b
```

  is not the same as

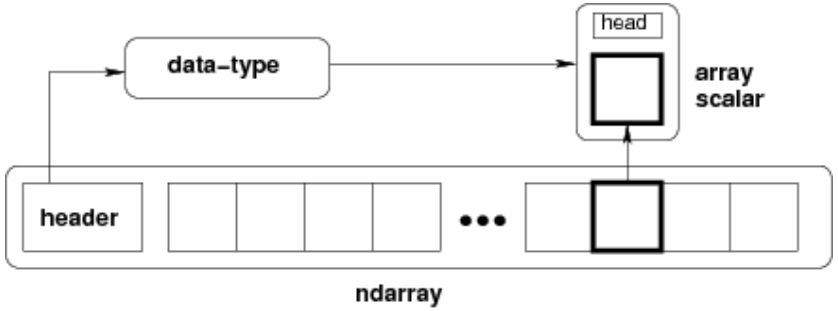
```
>>> a -= b.copy()
```

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x -= x.transpose()
>>> x
array([[ 0, -1],
       [ 4,  0]])
```

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

- `x` and `x.transpose()` share data
- `x -= x.transpose()` modifies the data element-by-element...
- because `x` and `x.transpose()` have different striding, modified data re-appears on the RHS

Findings in dissection



- *memory block*: may be shared, `.base`, `.data`
- *data type descriptor*: structured data, sub-arrays, byte order, casting, viewing, `.astype()`, `.view()`
- *strided indexing*: strides, C/F-order, slicing w/ integers, `as_strided`, broadcasting, stride tricks, `diag`, CPU cache coherence

2.1.4 Universal functions

What they are?

- Ufunc performs and elementwise operation on all elements of an array.

  Examples:


```
np.add, np.subtract, scipy.special.*, ...
```

- Automatically support: broadcasting, casting, ...
- The author of an ufunc only has to supply the elementwise operation, Numpy takes care of the rest.
- The elementwise operation needs to be implemented in C (or, e.g., Cython)

Parts of an Ufunc

1. Provided by user

```
void ufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    /*
     * int8 output = elementwise_function(int8 input_1, int8 input_2)
     *
     * This function must compute the ufunc for many values at once,
     * in the way shown below.
     */
    char *input_1 = (char*)args[0];
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];
    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        *output = elementwise_function(*input_1, *input_2);
        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

2. The Numpy part, built by

```
char types[3]

types[0] = NPY_BYTE   /* type of first input arg */
types[1] = NPY_BYTE   /* type of second input arg */
types[2] = NPY_BYTE   /* type of third input arg */

PyObject *python_ufunc = PyUFunc_FromFuncAndData(
    ufunc_loop,
    NULL,
    types,
    1, /* ntypes */
    2, /* num_inputs */
    1, /* num_outputs */
    identity_element,
    name,
    docstring,
    unused)
```

- A ufunc can also support multiple different input-output type combinations.

... can be made slightly easier

3. `ufunc_loop` is of very generic form, and Numpy provides pre-made ones

PyUfunc_f_f	float elementwise_func(float input_1)
PyUfunc_ff_f	float elementwise_func(float input_1, float input_2)
PyUfunc_d_d	double elementwise_func(double input_1)
PyUfunc_dd_d	double elementwise_func(double input_1, double input_2)
PyUfunc_D_D	elementwise_func(npy_cdouble *input, npy_cdouble* output)
PyUfunc_DD_D	elementwise_func(npy_cdouble *in1, npy_cdouble *in2, npy_cdouble* out)

- Only `elementwise_func` needs to be supplied
- ... except when your elementwise function is not in one of the above forms

Exercise: building an ufunc from scratch

The Mandelbrot fractal is defined by the iteration

$$z \leftarrow z^2 + c$$

where $c = x + iy$ is a complex number. This iteration is repeated – if z stays finite no matter how long the iteration runs, c belongs to the Mandelbrot set.

- Make ufunc called `mandel(z0, c)` that computes:

```
z = z0
for k in range(iterations):
    z = z*z + c
```

  say, 100 iterations or until `z.real**2 + z.imag**2 > 1000`. Use it to determine which *c* are in the Mandelbrot set.

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

See Also:

mandel.pyx, mandelplot.py

```
#
# Fix the parts marked by TODO
#


#
# Compile this file by (Cython >= 0.12 required because of the complex vars)
#
#    cython mandel.pyx
#    python setup.py build_ext -i
#
# and try it out with, in this directory,
#
#    >>> import mandel
#    >>> mandel.mandel(0, 1 + 2j)
#
#

# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    #
    # The Mandelbrot iteration
    #
```



```
    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
    #
    # - Note also that this function receives *pointers* to the data
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    #
    # TODO: write the Mandelbrot iteration for one point here,
    #       as you would write it in Python.
    #
    #       Say, use 100 as the maximum number of iterations, and 1000
    #       as the cutoff for z.real**2 + z.imag**2.
    #

    TODO: mandelbrot iteration should go here

    # Return the answer for this point
    z_out[0] = z


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

cdef extern from "numpy/arrayobject.h":
    void import_array()
    ctypedef int npy_intp
    cdef enum NPY_TYPES:
        NPY_DOUBLE
        NPY_CDOUBLE
        NPY_LONG

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

    # List of pre-defined loop functions

    void PyUFunc_f_f_As_d_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_d_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_f_f(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_g_g(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_F_F_As_D_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_F_F(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_D_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_G_G(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_ff_f_As_dd_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
```

```
    void PyUFunc_ff_f(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_dd_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_gg_g(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_FF_F_As_DD_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_DD_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_FF_F(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
    void PyUFunc_GG_G(char** args, npy_intp* dimensions, npy_intp* steps, void* func)


# Required module initialization
# ------------------------------

import_array()
import_ufunc()


# The actual ufunc declaration
# ----------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

#
# Reminder: some pre-made Ufunc loops:
#
# ================  =======================================================
# ``PyUfunc_f_f``   ``float elementwise_func(float input_1)``
# ``PyUfunc_ff_f``  ``float elementwise_func(float input_1, float input_2)``
# ``PyUfunc_d_d``   ``double elementwise_func(double input_1)``
# ``PyUfunc_dd_d``  ``double elementwise_func(double input_1, double input_2)``
# ``PyUfunc_D_D``   ``elementwise_func(complex_double *input, complex_double* complex_double)``
# ``PyUfunc_DD_D``  ``elementwise_func(complex_double *in1, complex_double *in2, complex_double* (
# ================  =======================================================
#
# The full list is above.
#
#
# Type codes:
#
# NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
# NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
# NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETIME,
# NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID
#

loop_func[0] = ... TODO: suitable PyUFunc_* ...
input_output_types[0] = ... TODO ...
... TODO: fill in rest of input_output_types ...

# This thing is passed as the ``data`` parameter for the generic
# PyUFunc_* loop, to let it know which function it should call.
elementwise_funcs[0] = <void*>mandel_single_point

# Construct the ufunc:

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    TODO, # number of input args
    TODO, # number of output args
```



```
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes z*z + c", # docstring
    0 # unused
    )
```

Reminder: some pre-made Ufunc loops:

PyUfunc_f_f	float elementwise_func(float input_1)
PyUfunc_ff_f	float elementwise_func(float input_1, float input_2)
PyUfunc_d_d	double elementwise_func(double input_1)
PyUfunc_dd_d	double elementwise_func(double input_1, double input_2)
PyUfunc_D_D	elementwise_func(complex_double *input, complex_double* output)
PyUfunc_DD_D	elementwise_func(complex_double *in1, complex_double *in2, complex_double* out)

Type codes:

```
NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETIME,
NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID
```

Solution: building an ufunc from scratch

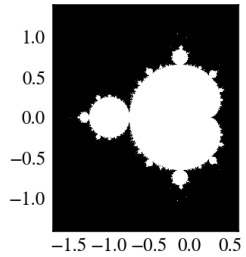
```
# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    #
    # The Mandelbrot iteration
    #

    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
    #
    # - Note also that this function receives *pointers* to the data;
    #   the "traditional" solution to passing complex variables around
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    # Straightforward iteration

    for k in range(100):
        z = z*z + c
        if z.real**2 + z.imag**2 > 1000:
            break

    # Return the answer for this point
```

```
    z_out[0] = z


# Boilerplate Cython definitions
#
# You don't really need to read this part, it just pulls in
# stuff from the Numpy C headers.
# -----------------------------------------------------------

cdef extern from "numpy/arrayobject.h":
    void import_array()
    ctypedef int npy_intp
    cdef enum NPY_TYPES:
        NPY_CDOUBLE

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

    void PyUFunc_DD_D(char**, npy_intp*, npy_intp*, void*)


# Required module initialization
# ------------------------------

import_array()
import_ufunc()


# The actual ufunc declaration
# ----------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

loop_func[0] = PyUFunc_DD_D

input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE

elementwise_funcs[0] = <void*>mandel_single_point

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

```
import numpy as np
import mandel
x = np.linspace(-1.7, 0.6, 1000)
```



```
y = np.linspace(-1.4, 1.4, 1000)
c = x[None,:] + 1j*y[:,None]
z = mandel.mandel(c, c)

import matplotlib.pyplot as plt
plt.imshow(abs(z)**2 < 1000, extent=[-1.7, 0.6, -1.4, 1.4])
plt.gray()
plt.show()
```



Note: Most of the boilerplate could be automated by these Cython modules:

http://wiki.cython.org/MarkLodato/CreatingUfuncs

Several accepted input types

E.g. supporting both single- and double-precision versions

```
cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
   ...

cdef void mandel_single_point_singleprec(float complex *z_in,
                                         float complex *c_in,
                                         float complex *z_out) nogil:
   ...

cdef PyUFuncGenericFunction loop_funcs[2]
cdef char input_output_types[3*2]
cdef void *elementwise_funcs[1*2]

loop_funcs[0] = PyUFunc_DD_D
input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE
elementwise_funcs[0] = <void*>mandel_single_point

loop_funcs[1] = PyUFunc_FF_F
input_output_types[3] = NPY_CFLOAT
input_output_types[4] = NPY_CFLOAT
input_output_types[5] = NPY_CFLOAT
elementwise_funcs[1] = <void*>mandel_single_point_singleprec

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
```

```
    input_output_types,
    2, # number of supported input types   <----------------
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

Generalized ufuncs

ufunc

`output = elementwise_function(input)`

Both `output` and `input` can be a single array element only.

generalized ufunc

`output` and `input` can be arrays with a fixed number of dimensions

For example, matrix trace (sum of diag elements):

```
input shape = (n, n)
output shape = ()      i.e.  scalar

(n, n) -> ()
```

Matrix product:

```
input_1 shape = (m, n)
input_2 shape = (n, p)
output shape  = (m, p)

(m, n), (n, p) -> (m, p)
```

- This is called the *"signature"* of the generalized ufunc
- The dimensions on which the g-ufunc acts, are *"core dimensions"*

Status in Numpy

- g-ufuncs are in Numpy already ...
- new ones can be created with `PyUFunc_FromFuncAndDataAndSignature`
- ... but we don't ship with public g-ufuncs, except for testing, ATM

```
>>> import numpy.core.umath_tests as ut
>>> ut.matrix_multiply.signature
'(m,n),(n,p)->(m,p)'
```

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

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

Generalized ufunc loop

Matrix multiplication `(m,n),(n,p) -> (m,p)`

```
void gufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    char *input_1 = (char*)args[0];  /* these are as previously */
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];

    int input_1_stride_m = steps[3];  /* strides for the core dimensions */
    int input_1_stride_n = steps[4];  /* are added after the non-core */
    int input_2_strides_n = steps[5]; /* steps */
    int input_2_strides_p = steps[6];
    int output_strides_n = steps[7];
    int output_strides_p = steps[8];

    int m = dimension[1]; /* core dimensions are added after */
    int n = dimension[2]; /* the main dimension; order as in */
    int p = dimension[3]; /* signature */

    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        matmul_for_strided_matrices(input_1, input_2, output,
                                    strides for each array...);

        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

2.1.5 Interoperability features

Sharing multidimensional, typed data

Suppose you

1. Write a library than handles (multidimensional) binary data,
2. Want to make it easy to manipulate the data with Numpy, or whatever other library,
3. ... but would **not** like to have Numpy as a dependency.

Currently, 3 solutions:

1. the "old" buffer interface
2. the array interface
3. the "new" buffer interface (**PEP 3118**)

The old buffer protocol

- Only 1-D buffers
- No data type information
- C-level interface; `PyBufferProcs tp_as_buffer` in the type object
- But it's integrated into Python (e.g. strings support it)

Mini-exercise using PIL (Python Imaging Library):

See Also:

pilbuffer.py

```
>>> import Image
>>> x = np.zeros((200, 200, 4), dtype=np.int8)
```

- TODO: RGBA images consist of 32-bit integers whose bytes are [RR,GG,BB,AA].
  - Fill x with opaque red [255,0,0,255]
  - Mangle it to (200,200) 32-bit integer array so that PIL accepts it

```
>>> img = Image.frombuffer("RGBA", (200, 200), data)
>>> img.save('test.png')
```

Q:

Check what happens if x is now modified, and `img` saved again.

The old buffer protocol

```
import numpy as np
import Image

# Let's make a sample image, RGBA format

x = np.zeros((200, 200, 4), dtype=np.int8)

x[:,:,0] = 254 # red
x[:,:,3] = 255 # opaque

data = x.view(np.int32) # Check that you understand why this is OK!

img = Image.frombuffer("RGBA", (200, 200), data)
img.save('test.png')

#
# Modify the original data, and save again.
#
# It turns out that PIL, which knows next to nothing about Numpy,
# happily shares the same data.
#

x[:,:,1] = 254
img.save('test2.png')
```

Array interface protocol

- Multidimensional buffers
- Data type information present
- Numpy-specific approach; slowly deprecated (but not going away)
- Not integrated in Python otherwise

See Also:

Documentation: http://docs.scipy.org/doc/numpy/reference/arrays.interface.html

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x.__array_interface__
{'data': (171694552, False),       # memory address of data, is readonly?
 'descr': [('', '<i4')],           # data type descriptor
 'typestr': '<i4',                 # same, in another form
 'strides': None,                  # strides; or None if in C-order
 'shape': (2, 2),
 'version': 3,
}
```

```
>>> import Image
>>> img = Image.open('test.png')
```

```
>>> img.__array_interface__
{'data': a very long string,
 'shape': (200, 200, 4),
 'typestr': '|u1'}
>>> x = np.asarray(img)
>>> x.shape
(200, 200, 4)
>>> x.dtype
dtype('uint8')
```

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

The new buffer protocol: PEP 3118

- Multidimensional buffers
- Data type information present
- C-level interface; extensions to `tp_as_buffer`
- Integrated into Python (>= 2.6)
- Replaces the "old" buffer interface in Python 3
- Next releases of Numpy (>= 1.5) will also implement it fully
- Demo in Python 3 / Numpy dev version:

```
Python 3.1.2 (r312:79147, Apr 15 2010, 12:35:07)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> np.__version__
'2.0.0.dev8469+15dcfb'
```

- Memoryview object exposes some of the stuff Python-side

```
>>> x = np.array([[1, 2], [3, 4]])
>>> y = memoryview(x)
>>> y.format
'l'
>>> y.itemsize
4
>>> y.ndim
2
>>> y.readonly
False
>>> y.shape
(2, 2)
>>> y.strides
(8, 4)
```

- Roundtrips work

```
>>> z = np.asarray(y)
>>> z
array([[1, 2],
       [3, 4]])
>>> x[0,0] = 9
>>> z
array([[9, 2],
       [3, 4]])
```

- Interoperability with the built-in Python `array` module


```
>>> import array
>>> x = array.array('h', b'1212')    # 2 of int16, but no Numpy!
>>> y = np.asarray(x)
>>> y
array([12849, 12849], dtype=int16)
>>> y.base
<memory at 0xa225acc>
```

PEP 3118 – details

- Suppose: you have a new Python extension type `MyObject` (defined in C)
- ... and want it to interoperable with a (2, 2) array of int32

See Also:

myobject.c, myobject_test.py, setup_myobject.py

```
static PyBufferProcs myobject_as_buffer = {
    (getbufferproc)myobject_getbuffer,
    (releasebufferproc)myobject_releasebuffer,
};

/* .. and stick this to the type object */

static int
myobject_getbuffer(PyObject *obj, Py_buffer *view, int flags)
{
    PyMyObjectObject *self = (PyMyObjectObject*)obj;

    /* Called when something requests that a MyObject-type object
       provides a buffer interface */

    view->buf = self->buffer;
    view->readonly = 0;
    view->format = "i";
    view->shape = malloc(sizeof(Py_ssize_t) * 2);
    view->strides = malloc(sizeof(Py_ssize_t) * 2);;
    view->suboffsets = NULL;

    TODO: Just fill in view->len, view->itemsize, view->strides[0] and 1,
    view->shape[0] and 1, and view->ndim.

    /* Note: if correct interpretation *requires* strides or shape,
       you need to check flags for what was requested, and raise
       appropriate errors.

       The same if the buffer is not readable.
    */

    view->obj = (PyObject*)self;
    Py_INCREF(self);

    return 0;
}

static void
myobject_releasebuffer(PyMemoryViewObject *self, Py_buffer *view)
{
    if (view->shape) {
        free(view->shape);
        view->shape = NULL;
    }
```

```
    if (view->strides) {
        free(view->strides);
        view->strides = NULL;
    }
}
```

```
/*
 * Sample implementation of a custom data type that exposes an array
 * interface. (And does nothing else :)
 *
 * Requires Python >= 3.1
 */

/*
 * Mini-exercises:
 *
 * - make the array strided
 * - change the data type
 *
 */

#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "structmember.h"


typedef struct {
    PyObject_HEAD
    int buffer[4];
} PyMyObjectObject;


static int
myobject_getbuffer(PyObject *obj, Py_buffer *view, int flags)
{
    PyMyObjectObject *self = (PyMyObjectObject*)obj;

    /* Called when something requests that a MyObject-type object
       provides a buffer interface */

    view->buf = self->buffer;
    view->readonly = 0;
    view->format = "i";
    view->len = 4;
    view->itemsize = sizeof(int);
    view->ndim = 2;
    view->shape = malloc(sizeof(Py_ssize_t) * 2);
    view->shape[0] = 2;
    view->shape[1] = 2;
    view->strides = malloc(sizeof(Py_ssize_t) * 2);;
    view->strides[0] = 2*sizeof(int);
    view->strides[1] = sizeof(int);
    view->suboffsets = NULL;

    /* Note: if correct interpretation *requires* strides or shape,
       you need to check flags for what was requested, and raise
       appropriate errors.

       The same if the buffer is not readable.
    */

    view->obj = (PyObject*)self;
    Py_INCREF(self);
```



```
    return 0;
}

static void
myobject_releasebuffer(PyMemoryViewObject *self, Py_buffer *view)
{
    if (view->shape) {
        free(view->shape);
        view->shape = NULL;
    }
    if (view->strides) {
        free(view->strides);
        view->strides = NULL;
    }
}


static PyBufferProcs myobject_as_buffer = {
    (getbufferproc)myobject_getbuffer,
    (releasebufferproc)myobject_releasebuffer,
};

/*
 * Standard stuff follows
 */

PyTypeObject PyMyObject_Type;

static PyObject *
myobject_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
{
    PyMyObjectObject *self;
    static char *kwlist[] = {NULL};
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "", kwlist)) {
        return NULL;
    }
    self = (PyMyObjectObject *)
        PyObject_New(PyMyObjectObject, &PyMyObject_Type);
    self->buffer[0] = 1;
    self->buffer[1] = 2;
    self->buffer[2] = 3;
    self->buffer[3] = 4;
    return (PyObject*)self;
}


PyTypeObject PyMyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "MyObject",
    sizeof(PyMyObjectObject),
    0,                                          /* tp_itemsize */
    /* methods */
    0,                                          /* tp_dealloc */
    0,                                          /* tp_print */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_reserved */
    0,                                          /* tp_repr */
    0,                                          /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    0,                                          /* tp_hash */
```

```
    0,                                          /* tp_call */
    0,                                          /* tp_str */
    0,                                          /* tp_getattro */
    0,                                          /* tp_setattro */
    &myobject_as_buffer,                        /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,                         /* tp_flags */
    0,                                          /* tp_doc */
    0,                                          /* tp_traverse */
    0,                                          /* tp_clear */
    0,                                          /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    0,                                          /* tp_iter */
    0,                                          /* tp_iternext */
    0,                                          /* tp_methods */
    0,                                          /* tp_members */
    0,                                          /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    0,                                          /* tp_dictoffset */
    0,                                          /* tp_init */
    0,                                          /* tp_alloc */
    myobject_new,                               /* tp_new */
    0,                                          /* tp_free */
    0,                                          /* tp_is_gc */
    0,                                          /* tp_bases */
    0,                                          /* tp_mro */
    0,                                          /* tp_cache */
    0,                                          /* tp_subclasses */
    0,                                          /* tp_weaklist */
    0,                                          /* tp_del */
    0,                                          /* tp_version_tag */
};

struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "myobject",
    NULL,
    -1,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

PyObject *PyInit_myobject(void) {
    PyObject *m, *d;

    if (PyType_Ready(&PyMyObject_Type) < 0) {
        return NULL;
    }

    m = PyModule_Create(&moduledef);

    d = PyModule_GetDict(m);

    Py_INCREF(&PyMyObject_Type);
    PyDict_SetItemString(d, "MyObject", (PyObject *)&PyMyObject_Type);

    return m;
}
```


2.1.6 Siblings: `chararray`, `maskedarray`, `matrix`

chararray — vectorized string operations

```
>>> x = np.array(['a', '  bbb', '  ccc']).view(np.chararray)
>>> x.lstrip(' ')
chararray(['a', 'bbb', 'ccc'],
      dtype='|S5')
>>> x.upper()
chararray(['A', '  BBB', '  CCC'],
          dtype='|S5')
```

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

MaskedArray — dealing with (propagation of) missing data

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

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

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

- Masking versions of common functions

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

Note: There's more to them!

recarray — purely convenience

```
>>> arr = np.array([('a', 1), ('b', 2)], dtype=[('x', 'S1'), ('y', int)])
>>> arr2 = arr.view(np.recarray)
>>> arr2.x
chararray(['a', 'b'],
      dtype='|S1')
>>> arr2.y
array([1, 2])
```

matrix — convenience?

- always 2-D
- `*` is the matrix product, not the elementwise one

```
>>> np.matrix([[1, 0], [0, 1]]) * np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
```

2.1.7 Summary

- Anatomy of the ndarray: data, dtype, strides.
- Universal functions: elementwise operations, how to make new ones
- Ndarray subclasses
- Various buffer interfaces for integration with other tools
- Recent additions: PEP 3118, generalized ufuncs

2.1.8 Hit list of the future for Numpy core

- Python 3 – no wait, it already works! (Numpy 1.5, sooner or later)
- PEP 3118 – hey, also that's there! (Numpy 1.5)
- Breaking backwards binary compatibility, we need to clean up a bit (Numpy 2.0, later)
- Refactoring Python out of innards of Numpy (Numpy 2.0 or later, I'd guess)
- Numpy on PyPy? (a GSoC project is working on this currently)
- Memory access pattern optimizations for more efficient CPU cache usage?
- We should really have some public generalized ufuncs (e.g. multiplying many small matrices is often asked for)
- Fixing the inplace operations so that

```
>>> a += b
```

  is guaranteed to produce the same `a` as

```
>>> a = a + b
```

  Preferably without sacrificing too much performance-wise...

2.1.9 Contributing to Numpy/Scipy

Get this tutorial: http://www.euroscipy.org/talk/882

Why

- "There's a bug?"
- "I don't understand what this is supposed to do?"
- "I have this fancy code. Would you like to have it?"
- "I'd like to help! What can I do?"

Reporting bugs

- Bug tracker (prefer **this**)
  - http://projects.scipy.org/numpy
  - http://projects.scipy.org/scipy
  - Click the "Register" link to get an account
- Mailing lists (scipy.org/Mailing_Lists)
  - If you're unsure

  - No replies in a week or so? Just file a bug ticket.

Good bug report

```
Title: numpy.random.permutations fails for non-integer arguments

I'm trying to generate random permutations, using numpy.random.permutations

When calling numpy.random.permutation with non-integer arguments
it fails with a cryptic error message:

>>> np.random.permutation(12)
array([10,  9,  4,  7,  3,  8,  0,  6,  5,  1, 11,  2])
>>> np.random.permutation(long(12))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mtrand.pyx", line 3311, in mtrand.RandomState.permutation
  File "mtrand.pyx", line 3254, in mtrand.RandomState.shuffle
TypeError: len() of unsized object

This also happens with long arguments, and so
np.random.permutation(X.shape[0]) where X is an array fails on 64
bit windows (where shape is a tuple of longs).

It would be great if it could cast to integer or at least raise a
proper error for non-integer types.

I'm using Numpy 1.4.1, built from the official tarball, on Windows
64 with Visual studio 2008, on Python.org 64-bit Python.
```

0. What are you trying to do?
1. **Small code snippet reproducing the bug** (if possible)
   - What actually happens
   - What you'd expect
2. Platform (Windows / Linux / OSX, 32/64 bits, x86/PPC, ...)
3. Version of Numpy/Scipy

```
>>> print numpy.__version__
```

   Check that the following is what you expect

```
>>> print numpy.__file__
```

   In case you have old/broken Numpy installations lying around.

   If unsure, try to remove existing Numpy installations, and reinstall...

Contributing to documentation

1. Documentation editor
   - http://docs.scipy.org/numpy
   - Registration
     - Register an account
     - Subscribe to `scipy-dev` ML (subscribers-only)
     - Problem with mailing lists: you get mail

        * But: **you can turn mail delivery off**
        * "change your subscription options", at the bottom of
          http://mail.scipy.org/mailman/listinfo/scipy-dev
      - Send a mail @ `scipy-dev` mailing list; ask for activation:

```
To: scipy-dev@scipy.org

Hi,

I'd like to edit Numpy/Scipy docstrings. My account is XXXXX

Cheers,
N. N.
```

    - Check the style guide:
      - http://docs.scipy.org/numpy/
      - Don't be intimidated; to fix a small thing, just fix it
    - Edit
2. Edit sources and send patches (as for bugs)
3. Complain on the ML

Contributing features

0. Ask on ML, if unsure where it should go
1. Write a patch, add an enhancement ticket on the bug tracket
2. OR, create a Git branch implementing the feature + add enhancement ticket.
    - Especially for big/invasive additions
    - http://projects.scipy.org/numpy/wiki/GitMirror
    - http://www.spheredev.org/wiki/Git_for_the_lazy

```
# Clone numpy repository
git clone --origin svn http://projects.scipy.org/git/numpy.git numpy
cd numpy

# Create a feature branch
git checkout -b name-of-my-feature-branch  svn/trunk

<edit stuff>

git commit -a
```

    - Create account on http://github.com (or anywhere)
    - Create a new repository @ Github
    - Push your work to github

```
git remote add github git@github:YOURUSERNAME/YOURREPOSITORYNAME.git
git push github name-of-my-feature-branch
```


How to help, in general

- Bug fixes always welcome!
  - What irks you most
  - Browse the tracker
- Documentation work
  - API docs: improvements to docstrings
    - Know some Scipy module well?
  - *User guide*
    - Needs to be done eventually.
    - Want to think? Come up with a Table of Contents
      http://scipy.org/Developer_Zone/UG_Toc
- Ask on communication channels:
  - `numpy-discussion` list
  - `scipy-dev` list
  - or now @ Euroscipy :)

2.1.10 Python 2 and 3, single code base

- The brave new future?
- You can do it!

The case of Numpy

- 150 000+ SLOC
- Numpy is complicated (PyString, PyInt, buffer interfaces, ...)!

Porting:

- Python 2 and 3, with a single code base
- Took ca. **2-3 man-weeks**
- `git log --grep="3K" -M -C -p | filterdiff -x '*/mtrand.c' | diffstat | tail -n1`

  `207 files changed, 5626 insertions(+), 2836 deletions(-)`

The case of Scipy

- WIP
- But is much easier: very little string handing etc.

How?

- http://projects.scipy.org/numpy/browser/trunk/doc/Py3K.txt
  (A bit out of date, but illustrative.)
- Python changes:

  - `2to3` on `setup.py build`
  - Compatibility package `numpy.compat.py3k`
  - `str` vs. `bytes`, cyclic imports, `types` module, ...
- C changes:
  - Compatibility header `npy_3kcompat.h`
  - Module & type initialization
  - Buffer interface (provider)
  - Strings: (i) Unicode on 2+3, (ii) UC on 3, Bytes on 2, (iii) Bytes on 2+3
  - Division, comparison
  - I/O
  - PyCObject -> PyCapsule
  - ...

2.2 Sparse Matrices in SciPy

author Robert Cimrman

2.2.1 Introduction

(dense) matrix is:

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

important features:

- **memory allocated once for all items**
  - usually a contiguous chunk, think NumPy ndarray
- *fast* access to individual items (*)

Why Sparse Matrices?

- the memory, that grows like *n***2*
- small example (double precision matrix):

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 1e6, 10)
>>> plt.plot(x, 8.0 * (x**2) / 1e6, lw=5)
>>> plt.xlabel('size n')
>>> plt.ylabel('memory [MB]')
>>> plt.show()
```

Sparse Matrices vs. Sparse Matrix Storage Schemes

- sparse matrix is a matrix, which is *almost empty*
- storing all the zeros is wasteful -> store only nonzero items
- think **compression**

- pros: huge memory savings
- cons: depends on actual storage scheme, (*) usually does not hold

Typical Applications

- **solution of partial differential equations (PDEs)**
  - the *finite element method*
  - mechanical engineering, electrotechnics, physics, ...
- **graph theory**
  - nonzero at *(i, j)* means that node *i* is connected to node *j*
- ...

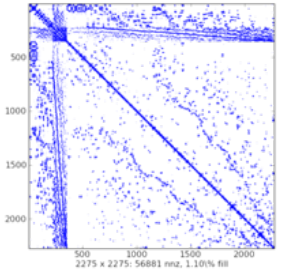
Prerequisites

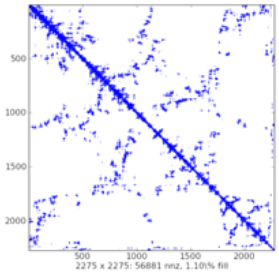
recent versions of

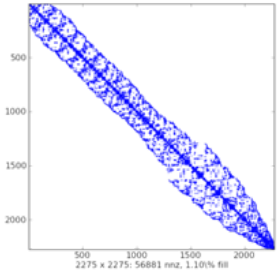
- `numpy`
- `scipy`
- `matplotlib` (optional)
- `ipython` (the enhancements come handy)

Sparsity Structure Visualization

- `spy()` from `matplotlib`
- example plots:







2.2.2 Storage Schemes

- **seven sparse matrix types in scipy.sparse:**
    1. csc_matrix: Compressed Sparse Column format
    2. csr_matrix: Compressed Sparse Row format
    3. bsr_matrix: Block Sparse Row format
    4. lil_matrix: List of Lists format
    5. dok_matrix: Dictionary of Keys format
    6. coo_matrix: COOrdinate format (aka IJV, triplet format)
    7. dia_matrix: DIAgonal format
- each suitable for some tasks
- many employ sparsetools C++ module by Nathan Bell
- assume the following is imported:

```
>>> import numpy as np
>>> import scipy.sparse as sps
>>> import matplotlib.pyplot as plt
```

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

Common Methods

- **all scipy.sparse classes are subclasses of `spmatrix`**
    - **default implementation of arithmetic operations**
        * always converts to CSR
        * subclasses override for efficiency
    - shape, data type set/get
    - nonzero indices
    - format conversion, interaction with NumPy (*toarray()*, *todense()*)
    - ...
- **attributes:**
    - *mtx.A* - same as mtx.toarray()
    - *mtx.T* - transpose (same as mtx.transpose())
    - *mtx.H* - Hermitian (conjugate) transpose
    - *mtx.real* - real part of complex matrix
    - *mtx.imag* - imaginary part of complex matrix
    - *mtx.size* - the number of nonzeros (same as self.getnnz())
    - *mtx.shape* - the number of rows and columns (tuple)
- data usually stored in NumPy arrays

Sparse Matrix Classes

Diagonal Format (DIA)

- very simple scheme
- **diagonals in dense NumPy array of shape *(n_diag, length)***
    - fixed length -> waste space a bit when far from main diagonal
    - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- **offset for each diagonal**
    - 0 is the main diagonal
    - negative offset = below
    - positive offset = above
- fast matrix * vector (sparsetools)
- **fast and easy item-wise operations**
    - manipulate data array directly (fast NumPy machinery)
- **constructor accepts:**
    - dense matrix (array)
    - sparse matrix
    - shape tuple (create empty matrix)
    - *(data, offsets)* tuple
- no slicing, no individual item access

- **use:**
  - rather specialized
  - solving PDEs by finite differences
  - with an iterative solver

Examples

- create some DIA matrices:

```
>>> data = np.array([[1, 2, 3, 4]]).repeat(3, axis=0)
>>> data
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])
>>> offsets = np.array([0, -1, 2])
>>> mtx = sps.dia_matrix((data, offsets), shape=(4, 4))
>>> mtx
<4x4 sparse matrix of type '<type 'numpy.int32'>'
        with 9 stored elements (3 diagonals) in DIAgonal format>
>>> mtx.todense()
matrix([[1, 0, 3, 0],
        [1, 2, 0, 4],
        [0, 2, 3, 0],
        [0, 0, 3, 4]])

>>> data = np.arange(12).reshape((3, 4)) + 1
>>> data
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> mtx = sps.dia_matrix((data, offsets), shape=(4, 4))
>>> mtx.data
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> mtx.offsets
array([ 0, -1,  2])
>>> print mtx
  (0, 0)        1
  (1, 1)        2
  (2, 2)        3
  (3, 3)        4
  (1, 0)        5
  (2, 1)        6
  (3, 2)        7
  (0, 2)        11
  (1, 3)        12
>>> mtx.todense()
matrix([[ 1,  0, 11,  0],
        [ 5,  2,  0, 12],
        [ 0,  6,  3,  0],
        [ 0,  0,  7,  4]])
```

- explanation with a scheme:

```
offset: row

     2:  9
     1:  --10------
     0:  1  . 11  .
    -1:  5  2  . 12
    -2:  .  6  3  .
    -3:  .  .  7  4
         ---------8
```


- matrix-vector multiplication

```
>>> vec = np.ones((4,))
>>> vec
array([ 1.,  1.,  1.,  1.])
>>> mtx * vec
array([ 12.,  19.,   9.,  11.])
>>> mtx.toarray() * vec
array([[  1.,   0.,  11.,   0.],
       [  5.,   2.,   0.,  12.],
       [  0.,   6.,   3.,   0.],
       [  0.,   0.,   7.,   4.]])
```

List of Lists Format (LIL)

- **row-based linked list**
  - each row is a Python list (sorted) of column indices of non-zero elements
  - rows stored in a NumPy array (*dtype=np.object*)
  - non-zero values data stored analogously
- efficient for constructing sparse matrices incrementally
- **constructor accepts:**
  - dense matrix (array)
  - sparse matrix
  - shape tuple (create empty matrix)
- flexible slicing, changing sparsity structure is efficient
- slow arithmetics, slow column slicing due to being row-based
- **use:**
  - when sparsity pattern is not known apriori or changes
  - example: reading a sparse matrix from a text file

Examples

- create an empty LIL matrix:

```
>>> mtx = sps.lil_matrix((4, 5))
```

- prepare random data:

```
>>> from numpy.random import rand
>>> data = np.round(rand(2, 3))
>>> data
array([[ 0.,  0.,  1.],
       [ 0.,  0.,  1.]])
```

- assign the data using fancy indexing:

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

- more slicing and indexing:

```
>>> mtx = sps.lil_matrix([[0, 1, 2, 0], [3, 0, 1, 0], [1, 0, 0, 1]])
>>> mtx.todense()
matrix([[0, 1, 2, 0],
        [3, 0, 1, 0],
        [1, 0, 0, 1]])
>>> print mtx
  (0, 1)        1
  (0, 2)        2
  (1, 0)        3
  (1, 2)        1
  (2, 0)        1
  (2, 3)        1
>>> mtx[:2, :]
<2x4 sparse matrix of type '<type 'numpy.int32'>'
        with 4 stored elements in LInked List format>
>>> mtx[:2, :].todense()
matrix([[0, 1, 2, 0],
        [3, 0, 1, 0]])
>>> mtx[1:2, [0,2]].todense()
matrix([[3, 1]])
>>> mtx.todense()
matrix([[0, 1, 2, 0],
        [3, 0, 1, 0],
        [1, 0, 0, 1]])
```

Dictionary of Keys Format (DOK)

- **subclass of Python dict**
  - keys are *(row, column)* index tuples (no duplicate entries allowed)
  - values are corresponding non-zero values
- efficient for constructing sparse matrices incrementally
- **constructor accepts:**
  - dense matrix (array)
  - sparse matrix
  - shape tuple (create empty matrix)
- efficient O(1) access to individual elements

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

Examples

- create a DOK matrix element by element:

```
>>> mtx = sps.dok_matrix((5, 5), dtype=np.float64)
>>> mtx
<5x5 sparse matrix of type '<type 'numpy.float64'>'
        with 0 stored elements in Dictionary Of Keys format>
>>> for ir in range(5):
>>>     for ic in range(5):
>>>         mtx[ir, ic] = 1.0 * (ir != ic)
>>> mtx
<5x5 sparse matrix of type '<type 'numpy.float64'>'
        with 25 stored elements in Dictionary Of Keys format>
>>> mtx.todense()
matrix([[ 0.,  1.,  1.,  1.,  1.],
        [ 1.,  0.,  1.,  1.,  1.],
        [ 1.,  1.,  0.,  1.,  1.],
        [ 1.,  1.,  1.,  0.,  1.],
        [ 1.,  1.,  1.,  1.,  0.]])
```

- something is wrong, let's start over:

```
>>> mtx = sps.dok_matrix((5, 5), dtype=np.float64)
>>> for ir in range(5):
>>>     for ic in range(5):
>>>         if (ir != ic): mtx[ir, ic] = 1.0
>>> mtx
<5x5 sparse matrix of type '<type 'numpy.float64'>'
        with 20 stored elements in Dictionary Of Keys format>
```

- slicing and indexing:

```
>>> mtx[1, 1]
0.0
>>> mtx[1, 1:3]
<1x2 sparse matrix of type '<type 'numpy.float64'>'
      with 1 stored elements in Dictionary Of Keys format>
>>> mtx[1, 1:3].todense()
matrix([[ 0.,  1.]])
>>> mtx[[2,1], 1:3].todense()
---------------------------------------------------------------
Traceback (most recent call last):
<snip>
NotImplementedError: fancy indexing supported over one axis only
```

Coordinate Format (COO)

- **also known as the 'ijv' or 'triplet' format**
  - three NumPy arrays: *row*, *col*, *data*
  - *data[i]* is value at *(row[i], col[i])* position

  - permits duplicate entries
  - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- fast format for constructing sparse matrices
- **constructor accepts:**
  - dense matrix (array)
  - sparse matrix
  - shape tuple (create empty matrix)
  - *(data, ij)* tuple
- very fast conversion to and from CSR/CSC formats
- fast matrix * vector (sparsetools)
- **fast and easy item-wise operations**
  - manipulate data array directly (fast NumPy machinery)
- no slicing, no arithmetics (directly)
- **use:**
  - facilitates fast conversion among sparse formats
  - when converting to other format (usually CSR or CSC), duplicate entries are summed together
    - facilitates efficient construction of finite element matrices

Examples

- create empty COO matrix:

```
>>> mtx = sps.coo_matrix((3, 4), dtype=np.int8)
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

- create using *(data, ij)* tuple:

```
>>> row = np.array([0, 3, 1, 0])
>>> col = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> mtx = sps.coo_matrix((data, (row, col)), shape=(4, 4))
>>> mtx
<4x4 sparse matrix of type '<type 'numpy.int32'>'
        with 4 stored elements in COOrdinate format>
>>> mtx.todense()
matrix([[4, 0, 9, 0],
        [0, 7, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 5]])
```

- duplicates entries are summed together:

```
>>> row = np.array([0, 0, 1, 3, 1, 0, 0])
>>> col = np.array([0, 2, 1, 3, 1, 0, 0])
>>> data = np.array([1, 1, 1, 1, 1, 1, 1])
>>> mtx = sps.coo_matrix((data, (row, col)), shape=(4, 4))
>>> mtx.todense()
matrix([[3, 0, 1, 0],
        [0, 2, 0, 0],
```



```
        [0, 0, 0, 0],
        [0, 0, 0, 1]])
```

- no slicing...:

```
>>> mtx[2, 3]
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
TypeError: 'coo_matrix' object is unsubscriptable
```

Compressed Sparse Row Format (CSR)

- **row oriented**
  - **three NumPy arrays: *indices*, *indptr*, *data***
    - *indices* is array of column indices
    - *data* is array of corresponding nonzero values
    - *indptr* points to row starts in *indices* and *data*
    - length is *n_row + 1*, last item = number of values = length of both *indices* and *data*
    - nonzero values of the *i*-th row are *data[indptr[i]:indptr[i+1]]* with column indices *indices[indptr[i]:indptr[i+1]]*
    - item *(i, j)* can be accessed as *data[indptr[i]+k]*, where *k* is position of *j* in *indices[indptr[i]:indptr[i+1]]*
  - **subclass of `_cs_matrix` (common CSR/CSC functionality)**
    - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- fast matrix vector products and other arithmetics (sparsetools)
- **constructor accepts:**
  - dense matrix (array)
  - sparse matrix
  - shape tuple (create empty matrix)
  - *(data, ij)* tuple
  - *(data, indices, indptr)* tuple
- efficient row slicing, row-oriented operations
- slow column slicing, expensive changes to the sparsity structure
- **use:**
  - actual computations (most linear solvers support this format)

Examples

- create empty CSR matrix:

```
>>> mtx = sps.csr_matrix((3, 4), dtype=np.int8)
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

- create using *(data, ij)* tuple:

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> mtx = sps.csr_matrix((data, (row, col)), shape=(3, 3))
>>> mtx
<3x3 sparse matrix of type '<type 'numpy.int32'>'
        with 6 stored elements in Compressed Sparse Row format>
>>> mtx.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
>>> mtx.data
array([1, 2, 3, 4, 5, 6])
>>> mtx.indices
array([0, 2, 2, 0, 1, 2])
>>> mtx.indptr
array([0, 2, 3, 6])
```

- create using *(data, indices, indptr)* tuple:

```
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> indptr = np.array([0, 2, 3, 6])
>>> mtx = sps.csr_matrix((data, indices, indptr), shape=(3, 3))
>>> mtx.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```

Compressed Sparse Column Format (CSC)

- **column oriented**
  - **three NumPy arrays: *indices*, *indptr*, *data***
    - *indices* is array of row indices
    - *data* is array of corresponding nonzero values
    - *indptr* points to column starts in *indices* and *data*
    - length is *n_col + 1*, last item = number of values = length of both *indices* and *data*
    - nonzero values of the *i*-th column are *data[indptr[i]:indptr[i+1]]* with row indices *indices[indptr[i]:indptr[i+1]]*
    - item *(i, j)* can be accessed as *data[indptr[j]+k]*, where *k* is position of *i* in *indices[indptr[j]:indptr[j+1]]*
  - **subclass of `_cs_matrix` (common CSR/CSC functionality)**
    - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- fast matrix vector products and other arithmetics (sparsetools)
- **constructor accepts:**
  - dense matrix (array)
  - sparse matrix
  - shape tuple (create empty matrix)
  - *(data, ij)* tuple
  - *(data, indices, indptr)* tuple

- efficient column slicing, column-oriented operations
- slow row slicing, expensive changes to the sparsity structure
- **use:**
  - actual computations (most linear solvers support this format)

Examples

- create empty CSC matrix:

```
>>> mtx = sps.csc_matrix((3, 4), dtype=np.int8)
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

- create using *(data, ij)* tuple:

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> mtx = sps.csc_matrix((data, (row, col)), shape=(3, 3))
>>> mtx
<3x3 sparse matrix of type '<type 'numpy.int32'>'
        with 6 stored elements in Compressed Sparse Column format>
>>> mtx.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
>>> mtx.data
array([1, 4, 5, 2, 3, 6])
>>> mtx.indices
array([0, 2, 2, 0, 1, 2])
>>> mtx.indptr
array([0, 2, 3, 6])
```

- create using *(data, indices, indptr)* tuple:

```
>>> data = np.array([1, 4, 5, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> indptr = np.array([0, 2, 3, 6])
>>> mtx = sps.csc_matrix((data, indices, indptr), shape=(3, 3))
>>> mtx.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```

Block Compressed Row Format (BSR)

- basically a CSR with dense sub-matrices of fixed shape instead of scalar items
  - block size *(R, C)* must evenly divide the shape of the matrix *(M, N)*
  - **three NumPy arrays: *indices*, *indptr*, *data***
    - *indices* is array of column indices for each block
    - *data* is array of corresponding nonzero values of shape *(nnz, R, C)*
    - ...
  - **subclass of `_cs_matrix` (common CSR/CSC functionality)**

    - subclass of `_data_matrix` (sparse matrix classes with *.data* attribute)
- fast matrix vector products and other arithmetics (sparsetools)
- **constructor accepts:**
    - dense matrix (array)
    - sparse matrix
    - shape tuple (create empty matrix)
    - *(data, ij)* tuple
    - *(data, indices, indptr)* tuple
- many arithmetic operations considerably more efficient than CSR for sparse matrices with dense submatrices
- **use:**
    - like CSR
    - vector-valued finite element discretizations

Examples

- create empty BSR matrix with (1, 1) block size (like CSR...):

```
>>> mtx = sps.bsr_matrix((3, 4), dtype=np.int8)
>>> mtx
<3x4 sparse matrix of type '<type 'numpy.int8'>'
        with 0 stored elements (blocksize = 1x1) in Block Sparse Row format>
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

- create empty BSR matrix with (3, 2) block size:

```
>>> mtx = sps.bsr_matrix((3, 4), blocksize=(3, 2), dtype=np.int8)
>>> mtx
<3x4 sparse matrix of type '<type 'numpy.int8'>'
        with 0 stored elements (blocksize = 3x2) in Block Sparse Row format>
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

    - a bug?

- create using *(data, ij)* tuple with (1, 1) block size (like CSR...):

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> mtx = sps.bsr_matrix((data, (row, col)), shape=(3, 3))
>>> mtx
<3x3 sparse matrix of type '<type 'numpy.int32'>'
        with 6 stored elements (blocksize = 1x1) in Block Sparse Row format>
>>> mtx.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
>>> mtx.data
array([[[1]],

       [[2]],
```



```
       [[3]],

       [[4]],

       [[5]],

       [[6]]])
>>> mtx.indices
array([0, 2, 2, 0, 1, 2])
>>> mtx.indptr
array([0, 2, 3, 6])
```

- create using *(data, indices, indptr)* tuple with (2, 2) block size:

```
>>> indptr = np.array([0, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6]).repeat(4).reshape(6, 2, 2)
>>> mtx = sps.bsr_matrix((data, indices, indptr), shape=(6, 6))
>>> mtx.todense()
matrix([[1, 1, 0, 0, 2, 2],
        [1, 1, 0, 0, 2, 2],
        [0, 0, 0, 0, 3, 3],
        [0, 0, 0, 0, 3, 3],
        [4, 4, 5, 5, 6, 6],
        [4, 4, 5, 5, 6, 6]])
>>> data
array([[[1, 1],
        [1, 1]],

       [[2, 2],
        [2, 2]],

       [[3, 3],
        [3, 3]],

       [[4, 4],
        [4, 4]],

       [[5, 5],
        [5, 5]],

       [[6, 6],
        [6, 6]]])
```

Summary

Table 2.1: Summary of storage schemes.

format	matrix * vector	get item	fancy get	set item	fancy set	solvers	note
DIA	sparsetools	.	.	.	.	iterative	has data array, specialized
LIL	via CSR	yes	yes	yes	yes	iterative	arithmetics via CSR, incremental construction
DOK	python	yes	one axis only	yes	yes	iterative	O(1) item access, incremental construction
COO	sparsetools	.	.	.	.	iterative	has data array, facilitates fast conversion
CSR	sparsetools	yes	yes	slow	.	any	has data array, fast row-wise ops
CSC	sparsetools	yes	yes	slow	.	any	has data array, fast column-wise ops
BSR	sparsetools	.	.	.	.	specialized	has data array, specialized

2.2.3 Linear System Solvers

- sparse matrix/eigenvalue problem solvers live in `scipy.sparse.linalg`
- **the submodules:**
  - `dsolve`: direct factorization methods for solving linear systems
  - `isolve`: iterative methods for solving linear systems
  - `eigen`: sparse eigenvalue problem solvers
- all solvers are accessible from:

```
>>> import scipy.sparse.linalg as spla
>>> spla.__all__
['LinearOperator', 'Tester', 'arpack', 'aslinearoperator', 'bicg',
'bicgstab', 'cg', 'cgs', 'csc_matrix', 'csr_matrix', 'dsolve',
'eigen', 'eigen_symmetric', 'factorized', 'gmres', 'interface',
'isolve', 'iterative', 'lgmres', 'linsolve', 'lobpcg', 'lsqr',
'minres', 'np', 'qmr', 'speigs', 'spilu', 'splu', 'spsolve', 'svd',
'test', 'umfpack', 'use_solver', 'utils', 'warnings']
```

Sparse Direct Solvers

- **default solver: SuperLU 4.0**
  - included in SciPy
  - real and complex systems
  - both single and double precision
- **optional: umfpack**
  - real and complex systems
  - double precision only
  - recommended for performance
  - wrappers now live in `scikits.umfpack`
  - check-out the new `scikits.suitesparse` by Nathaniel Smith

Examples

- import the whole module, and see its docstring:

```
>>> import scipy.sparse.linalg.dsolve as dsl
>>> help(dsl)
```

- both superlu and umfpack can be used (if the latter is installed) as follows:
  - prepare a linear system:

```
>>> import numpy as np
>>> import scipy.sparse as sps
>>> mtx = sps.spdiags([[1, 2, 3, 4, 5], [6, 5, 8, 9, 10]], [0, 1], 5, 5)
>>> mtx.todense()
matrix([[ 1,  5,  0,  0,  0],
        [ 0,  2,  8,  0,  0],
        [ 0,  0,  3,  9,  0],
        [ 0,  0,  0,  4, 10],
        [ 0,  0,  0,  0,  5]])
>>> rhs = np.array([1, 2, 3, 4, 5])
```

  - solve as single precision real:

```
>>> mtx1 = mtx.astype(np.float32)
>>> x = dsl.spsolve(mtx1, rhs, use_umfpack=False)
>>> print x
>>> print "Error: ", mtx1 * x - b
```

  - solve as double precision real:

```
>>> mtx2 = mtx.astype(np.float64)
>>> x = dsl.spsolve(mtx2, rhs, use_umfpack=True)
>>> print x
>>> print "Error: ", mtx2 * x - b
```

  - solve as single precision complex:

```
>>> mtx1 = mtx.astype(np.complex64)
>>> x = dsl.spsolve(mtx1, rhs, use_umfpack=False)
>>> print x
>>> print "Error: ", mtx1 * x - b
```

  - solve as double precision complex:

```
>>> mtx2 = mtx.astype(np.complex128)
>>> x = dsl.spsolve(mtx2, rhs, use_umfpack=True)
>>> print x
>>> print "Error: ", mtx2 * x - b
```

```
"""
Construct a 1000x1000 lil_matrix and add some values to it, convert it
to CSR format and solve A x = b for x:and solve a linear system with a
direct solver.
"""
import numpy as np
import scipy.sparse as sps
from matplotlib import pyplot as plt
from scipy.sparse.linalg.dsolve import linsolve

rand = np.random.rand
```

```
mtx = sps.lil_matrix((1000, 1000), dtype=np.float64)
mtx[0, :100] = rand(100)
mtx[1, 100:200] = mtx[0, :100]
mtx.setdiag(rand(1000))

plt.clf()
plt.spy(mtx, marker='.', markersize=2)
plt.show()

mtx = mtx.tocsr()
rhs = rand(1000)

x = linsolve.spsolve(mtx, rhs)

print 'rezidual:', np.linalg.norm(mtx * x - rhs)
```

- `examples/direct_solve.py`

Iterative Solvers

- **the `isolve` module contains the following solvers:**
  - `bicg` (BIConjugate Gradient)
  - `bicgstab` (BIConjugate Gradient STABilized)
  - `cg` (Conjugate Gradient) - symmetric positive definite matrices only
  - `cgs` (Conjugate Gradient Squared)
  - `gmres` (Generalized Minimal RESidual)
  - `minres` (MINimum RESidual)
  - `qmr` (Quasi-Minimal Residual)

Common Parameters

- mandatory:

  A [{sparse matrix, dense matrix, LinearOperator}] The N-by-N matrix of the linear system.

  b [{array, matrix}] Right hand side of the linear system. Has shape (N,) or (N,1).

- optional:

  x0 [{array, matrix}] Starting guess for the solution.

  tol [float] Relative tolerance to achieve before terminating.

  maxiter [integer] Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

  M [{sparse matrix, dense matrix, LinearOperator}] Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

  callback [function] User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

LinearOperator Class

```
from scipy.sparse.linalg.interface import LinearOperator
```

- common interface for performing matrix vector products
- useful abstraction that enables using dense and sparse matrices within the solvers, as well as *matrix-free* solutions
- has *shape* and *matvec()* (+ some optional parameters)
- example:

```
>>> import numpy as np
>>> from scipy.sparse.linalg import LinearOperator
>>> def mv(v):
...     return np.array([2*v[0], 3*v[1]])
...
>>> A = LinearOperator((2, 2), matvec=mv)
>>> A
<2x2 LinearOperator with unspecified dtype>
>>> A.matvec(np.ones(2))
array([ 2.,  3.])
>>> A * np.ones(2)
array([ 2.,  3.])
```

A Few Notes on Preconditioning

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

Eigenvalue Problem Solvers

The `eigen` module

- `arpack` * a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems
- `lobpcg` (Locally Optimal Block Preconditioned Conjugate Gradient Method) * works very well in combination with PyAMG * example by Nathan Bell:

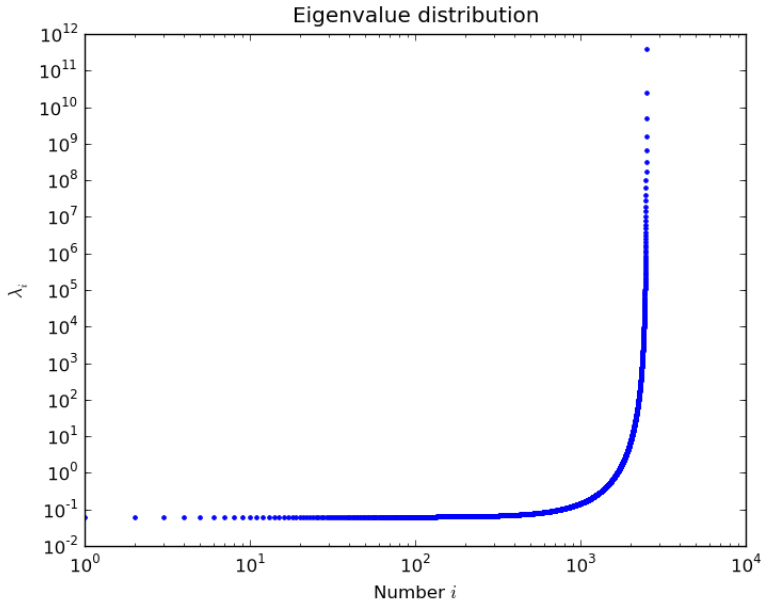
```
"""
Compute eigenvectors and eigenvalues using a preconditioned eigensolver

In this example Smoothed Aggregation (SA) is used to precondition
the LOBPCG eigensolver on a two-dimensional Poisson problem with
Dirichlet boundary conditions.
"""

import scipy
from scipy.sparse.linalg import lobpcg

from pyamg import smoothed_aggregation_solver
from pyamg.gallery import poisson

N = 100
K = 9
A = poisson((N,N), format='csr')
```

```
# create the AMG hierarchy
ml = smoothed_aggregation_solver(A)

# initial approximation to the K eigenvectors
X = scipy.rand(A.shape[0], K)

# preconditioner based on ml
M = ml.aspreconditioner()

# compute eigenvalues and eigenvectors with LOBPCG
W,V = lobpcg(A, X, M=M, tol=1e-8, largest=False)


#plot the eigenvectors
import pylab

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

for i in range(K):
    pylab.subplot(3, 3, i+1)
    pylab.title('Eigenvector %d' % i)
    pylab.pcolor(V[:,i].reshape(N,N))
    pylab.axis('equal')
    pylab.axis('off')
pylab.show()
```

  - `examples/pyamg_with_lobpcg.py`

- example by Nils Wagner:
  - `examples/lobpcg_sakurai.py`
- output:

```
$ python examples/lobpcg_sakurai.py
Results by LOBPCG for n=2500

[ 0.06250083  0.06250028  0.06250007]

Exact eigenvalues

[ 0.06250005  0.0625002   0.06250044]

Elapsed time 7.01
```

2.2.4 Other Interesting Packages

- **PyAMG**
  - algebraic multigrid solvers
  - http://code.google.com/p/pyamg
- **Pysparse**
  - own sparse matrix classes
  - matrix and eigenvalue problem solvers
  - http://pysparse.sourceforge.net/

2.3 Sympy : Symbolic Mathematics in Python

author Fabian Pedregosa

2.3.1 Objectives

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

1. Evaluate expressions with arbitrary precission.
2. Perform algebraic manipulations on symbolic expressions.
3. Perform basic calculus tasks (limits, differentiation and integration) with symbolic expressions.

4. Solve polynomial and trascendental equations.
5. Solve some differential equations.

2.3.2 What is SymPy?

SymPy is a Python library for symbolic mathematics. It aims become a full featured computer algebra system thatn 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.

2.3.3 First Steps with SymPy

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 - precission arithmetic. That way, some special constants, like e, pi, oo (Infinity), are treated as symbols and can be evaluated with arbitrary precission:

```
>>> pi**2
pi**2

>>> pi.evalf()
3.14159265358979

>>> (pi+exp(1)).evalf()
5.85987448204884
```

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

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

```
>>> oo > 99999
True
>>> oo + 1
oo
```

Exercises

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

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), &, |, ~ , >>, << (boolean).

2.3.4 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.

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)
```

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 precises alternatives to simplify exists: powsimp (simplification of exponents), trigsimp (for trigonometrical expressions) , logcombine, radsimp, together.

Exercises

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

2.3.5 Calculus

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 -> 0, you would issue limit(f, x, 0):

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

you can also calculate the limit at infinity:

```
>>> limit(x, x, oo)
oo

>>> limit(1/x, x, oo)
0

>>> limit(x**x, x, 0)
1
```

Differentiation

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

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

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

You can check, that it is correct by:

```
>>> 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:

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

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

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

Series expansion

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

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

Exercises

1. Calculate $\lim x- > 0, sin(x)/x$

2. Calulate the derivative of log(x) for x.

Integration

SymPy has support for indefinite and definite integration of transcendental elementary and special functions via *integrate()* facility, which uses powerful extended Risch-Norman algorithm and some heuristics and pattern matching. You can integrate elementary functions:

```
>>> integrate(6*x**5, x)
x**6
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(log(x), x)
-x + x*log(x)
>>> integrate(2*x + sinh(x), x)
cosh(x) + x**2
```

Also special functions are handled easily:

```
>>> integrate(exp(-x**2)*erf(x), x)
pi**(1/2)*erf(x)**2/4
```

It is possible to compute definite integral:

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

Also improper integrals are supported as well:

```
>>> integrate(exp(-x), (x, 0, oo))
1
>>> integrate(exp(-x**2), (x, -oo, oo))
pi**(1/2)
```

Exercises

2.3.6 Equation solving

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

```
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:

```
In [8]: solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
Out[8]: {y: 1, x: -3}
```

It also has (limited) support for trascendental equations:

```
In [9]: 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
```

Exercises

1. Solve the system of equations x + y = 2, 2*x + y = 0
2. Are there boolean values x, y that make (~x | y) & (~y | x) true?

2.3.7 Linear Algebra

Matrices

Matrices are created as instances from the Matrix class:

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

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

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

>>> A**2
[1 + x*y,     2*x]
[    2*y, 1 + x*y]
```

Differential Equations

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

```
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.

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

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 ?

2.4 3D plotting with Mayavi

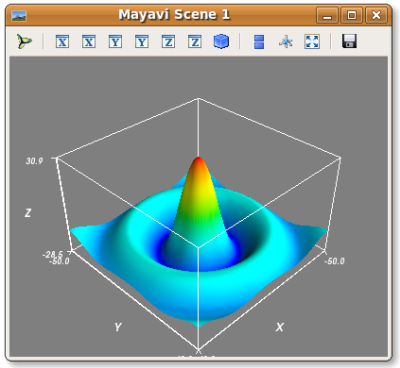
author Gaël Varoquaux

2.4.1 A simple example

Warning: Start *ipython -wthread*

```
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()
```

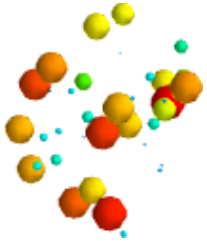
np.mgrid[-10:10:100j, -10:10:100j] creates an x,y grid, going from -10 to 10, with 100 steps in each directions.



2.4.2 3D plotting functions

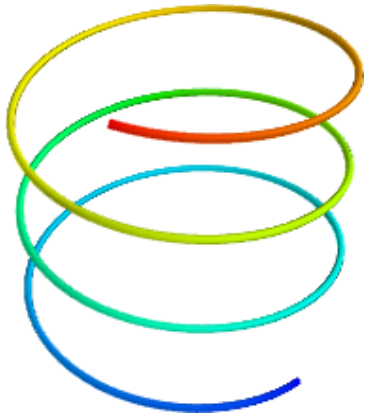
Points

```
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>
```



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>
```

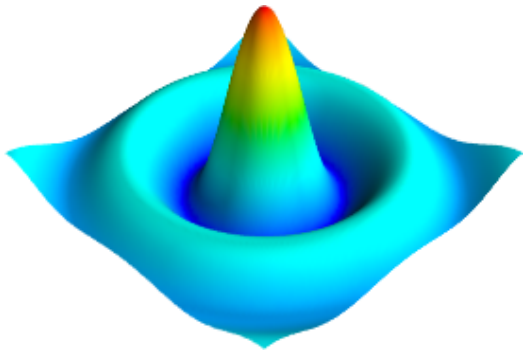


Elevation surface

```
In [8]: mlab.clf()

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

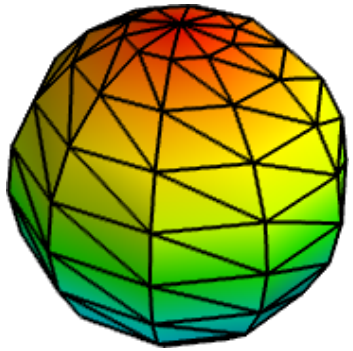


```
In [10]: r = np.sqrt(x**2 + y**2)

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>
```



Arbitrary regular mesh

```
In [13]: mlab.clf()

In [14]: phi, theta = np.mgrid[0:np.pi:11j, 0:2*np.pi:11j]

In [15]: x = np.sin(phi) * np.cos(theta)

In [16]: y = np.sin(phi) * np.sin(theta)

In [17]: z = np.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>
```



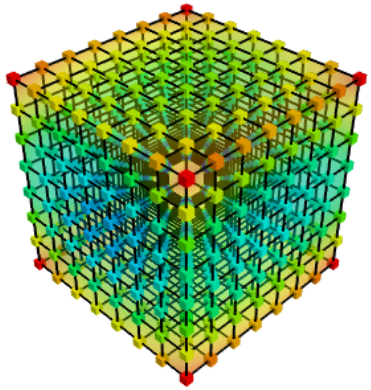
Note: A surface is defined by points **connected** to form triangles or polygones. In *mlab.func* and *mlab.mesh*, the connectivity is implicity given by the layout of the arrays. See also *mlab.triangular_mesh*.

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

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:



2.4.3 Figures and decorations

Figure management

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

Get the current figure:	*mlab.gcf()*
Clear the current figure:	*mlab.clf()*
Set the current figure:	*mlab.figure(1, bgcolor=(1, 1, 1), fgcolor=(0.5, 0.5, 0.5)*
Save figure to image file:	*mlab.savefig('foo.png', size=(300, 300))*
Change the view:	mlab.view(azimuth=45, elevation=54, distance=1.)

Changing plot properties

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

Example docstring: *mlab.mesh*

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

```
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 surf function, as it will create more efficient data structures.
Keyword arguments:

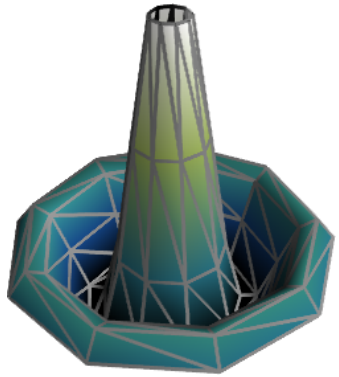
color the color of the vtk object. Overides the colormap, if any, when specified. This is specified as a triplet of float ranging from 0 to 1, eg (1, 1, 1) for white.
colormap type of colormap to use.
extent [xmin, xmax, ymin, ymax, zmin, zmax] Default is the x, y, z arrays extents. Use this to change the extent of the object created.
figure Figure to populate.
line_width The with of the lines, if any used. Must be a float. Default: 2.0
mask boolean mask array to suppress some data points.
mask_points If supplied, only one out of 'mask_points' data point is displayed. This option is usefull to reduce the number of points displayed on large datasets Must be an integer or None.
mode the mode of the glyphs. Must be '2darrow' or '2dcircle' or '2dcross' or '2ddash' or '2ddiamond' or '2dhooked_arrow' or '2dsquare' or '2dthick_arrow' or '2dthick_cross' or '2dtriangle' or '2dvertex' or 'arrow' or 'cone' or 'cube' or 'cylinder' or 'point' or 'sphere'. Default: sphere
name the name of the vtk object created.
representation the representation type used for the surface. Must be 'surface' or 'wireframe' or 'points' or 'mesh' or 'fancymesh'. Default: surface
resolution The resolution of the glyph created. For spheres, for instance, this is the number of divisions along theta and phi. Must be an integer. Default: 8
scalars optional scalar data.
scale_factor scale factor of the glyphs used to represent the vertices, in fancy_mesh mode. Must be a float. Default: 0.05
scale_mode the scaling mode for the glyphs ('vector', 'scalar', or 'none').
transparent make the opacity of the actor depend on the scalar.
tube_radius radius of the tubes used to represent the lines, in mesh mode. If None, simple lines are used.
tube_sides number of sides of the tubes used to represent the lines. Must be an integer. Default: 6
vmax vmax is used to scale the colormap If None, the max of the data will be used
vmin vmin is used to scale the colormap If None, the min of the data will be used

Example:

```
In [1]: import numpy as np

In [2]: r, theta = 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>
```



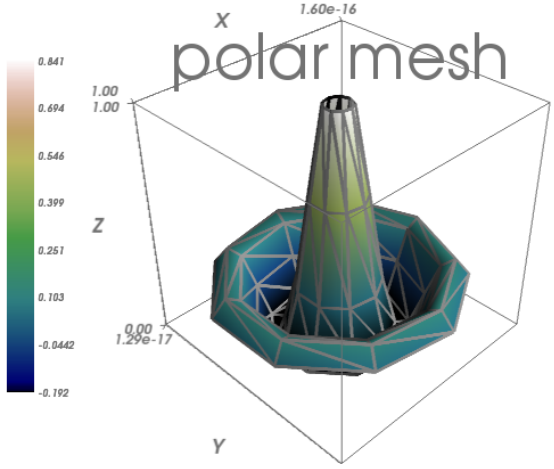
```
In [8]: mlab.mesh(x, y, z, extent=[0, 1, 0, 1, 0, 1],
   ...: representation='wireframe', line_width=1, color=(0.5, 0.5, 0.5))
Out[8]: <enthought.mayavi.modules.surface.Surface object at 0xdd6a71c>
```



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>
```

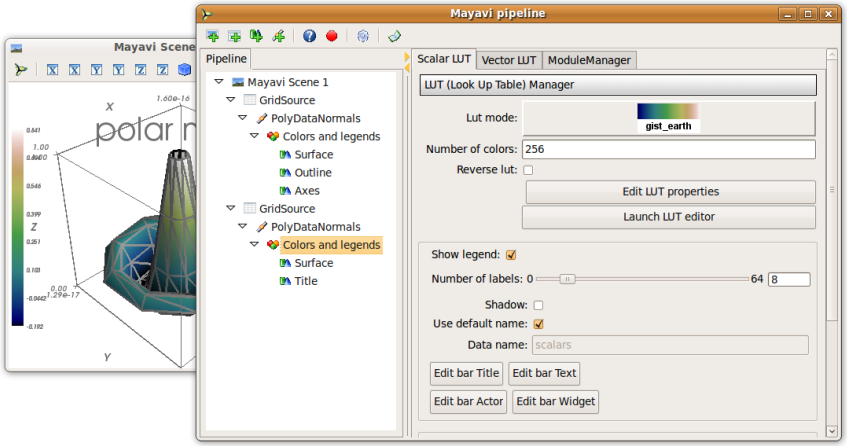


Warning: **extent:** If we specified extents for a plotting object, *mlab.outline' and 'mlab.axes* don't get them by default.

2.4.4 Interaction

The quicket way to create beautiful visualization with Mayavi is probably to interactivly 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

[Mallet09] Mallet, C. and Bretar, F. Full-Waveform Topographic Lidar: State-of-the-Art. *ISPRS Journal of Photogrammetry and Remote Sensing* 64(1), pp.1-16, January 2009 http://dx.doi.org/10.1016/j.isprsjprs.2008.09.007

Index

D

diff, 160, 162
differentiation, 160
dsolve, 162

E

equations
    algebraic, 161
    differential, 162

I

integration, 161

M

Matrix, 162

P

Python Enhancement Proposals
    PEP 3118, 125

S

solve, 161