Introduction to Python for Science

http://www.pyHPC.org

Introduction to Python

Objectives

1. You will understand how scripting languages fit into the toolbox of a computational scientist.
2. You will see why Python is a powerful choice
3. You will get a taste of Python for actual scientific computing

Sources

- Scipy Lectures **http://scipy-lectures.github.io**
- Software Carpentry **https://github.com/swcarpentry/boot-camps**

See also

- Scipy Conference Tutorials: **http://conference.scipy.org/scipy2013/tutorials_schedule.php**

Why Python?

The scientist's needs

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

Specifications

- Existing **bricks** corresponding to classical numerical methods or basic actions
- Easy to learn
- Easy to communicate with collaborators, students, customers, to make the code live within a lab or a company
- Efficient code that executes quickly...
- A single environment/language for everything

Existing solutions

Which solutions do scientists use to work?

- Compiled languages: C, C++, Fortran
- Matlab
- Other scripting languages: Scilab, Octave, Igor, R, IDL, etc.

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

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

Scripting languages: Matlab

- Advantages:
  - Very rich collection of libraries with numerous algorithms, for many different domains. Fast execution because these libraries are often written in a compiled language.
  - Pleasant development environment: comprehensive and well organized help, integrated editor, etc.
  - Commercial support is available.
- Drawbacks:
  - Base language is quite poor and can become restrictive for advanced users.
  - Not free.

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

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

What about Python?

- Advantages:
  - Very rich scientific computing libraries (a bit less than Matlab, though)
  - Well thought out language, allowing to write very readable and well structured code: we "code what we think".
  - Many libraries for other tasks than scientific computing (web server management, serial port 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 software or toolboxes.

Tour of Scripts

Let's look at a few simple examples.

Hello Scientific World

```
import math
r = math.pi / 2.0
s = math.sin(r)
print "Hello world, sin(%f)=%f" % (r,s)
```

Running in the shell:

```
$ python examples/intro/01_hello_world.py
Hello world, sin(1.570796)=1.000000
```

Input / Output

```
import math
import os

data_dir = os.path.join(os.path.dirname(__file__), "..", "data")
infile = "numbers"
outfile = "f_numbers"

f = open(os.path.join(data_dir, infile), 'r')
g = open(os.path.join(data_dir, outfile), 'w')

def func(y):
    if y >= 0.0:
        return y**5.0*math.exp(-y)
    else:
        return 0.0

for line in f:
    line = line.split()
    x, y = float(line[0]), float(line[1])
    g.write("%g %12.5e\n" % (x,func(y)))

f.close(); g.close()
```

Input / Ouput Continued

```
$ cat examples/data/numbers
1 2
3 4
5 6
7 8
9 10
$ python examples/intro/02_write_numbers.py
Read from examples/intro/../data/numbers
Wrote to examples/intro/../data/f_numbers
$ cat examples/data/f_numbers
1  4.33073e+00
3  1.87552e+01
5  1.92748e+01
7  1.09924e+01
9  4.53999e+00
```

System commands

```
 #!/usr/bin/env python
import sys,os
cmd = 'date'
output = os.popen(cmd)
lines = output.readlines()
fail = output.close()

if fail: print 'You do not have the date command'; sys.exit()

for line in lines:
    line = line.split()
    print "The current time is %s on %s %s, %s" %  (line[3],line[2],line[1],li
ne[-1])
```

```
$ ./examples/intro/03_call_sys_commands.py
The current time is 11:50:25 on 24 Aug, 2013
```

Regular Expressions

```
@Book{Langtangen2011,
  author =    {Hans Petter Langtangen},
  title =     {A Primer on Scientific Programming with Python},
  publisher = {Springer},
  year =      {2011}
}
@Book{Langtangen2010,
  author =    {Hans Petter Langtangen},
  title =     {Python Scripting for Computational Science},
  publisher = {Springer},
  year =      {2010}
}
```

Regular Expressions Continued

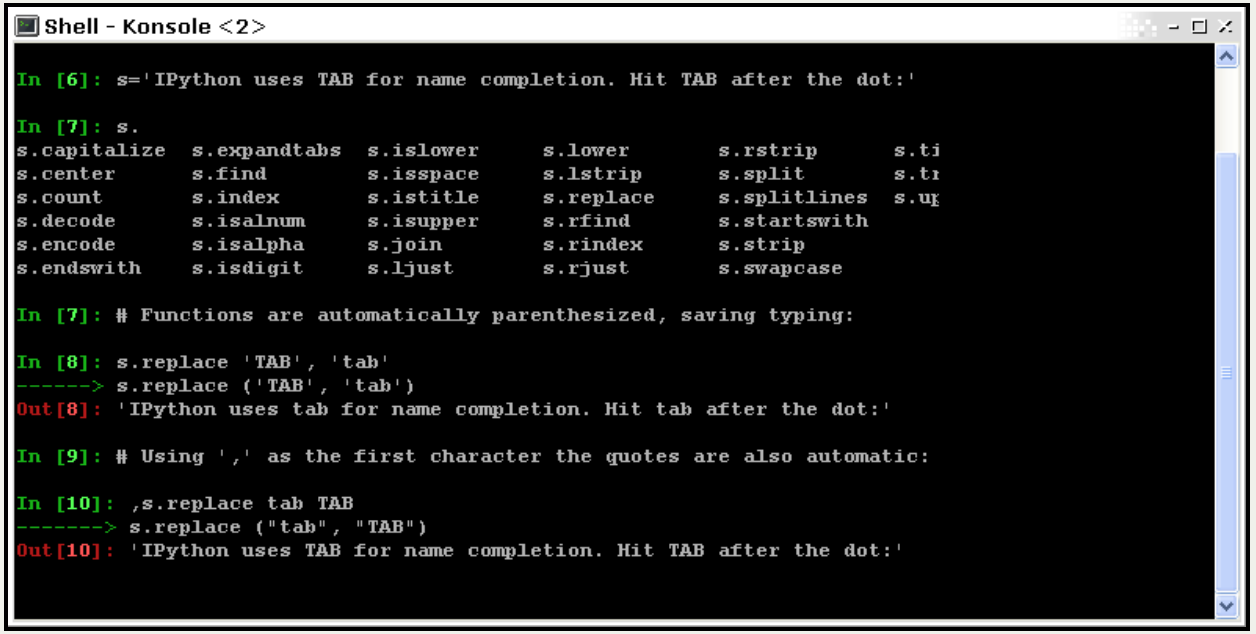
```
import os
import re

data_dir = os.path.join(os.path.dirname(__file__), "..", "data")
infile = os.path.join(data_dir, "python.bib")

pattern1 = "@Book{(.*),"
pattern2 = "\s+title\s+=\s+{(.*)},"

print "Reading from", infile
for line in file(infile):
    match = re.search(pattern1, line)
    if match:
        print "Found a book with the tag '%s'" % match.group(1)

    match = re.search(pattern2, line)
    if match:
        print "The title is '%s'" % match.group(1)
```

```
$ python examples/intro/04_regular_expressions.py
Reading from examples/intro/../data/python.bib
Found a book with the tag 'Langtangen2011'
The title is 'A Primer on Scientific Programming with Python'
Found a book with the tag 'Langtangen2010'
The title is 'Python Scripting for Computational Science'
```

Basic Building Blocks

Scientific Python is un-bundled, so users need some basic building blocks.

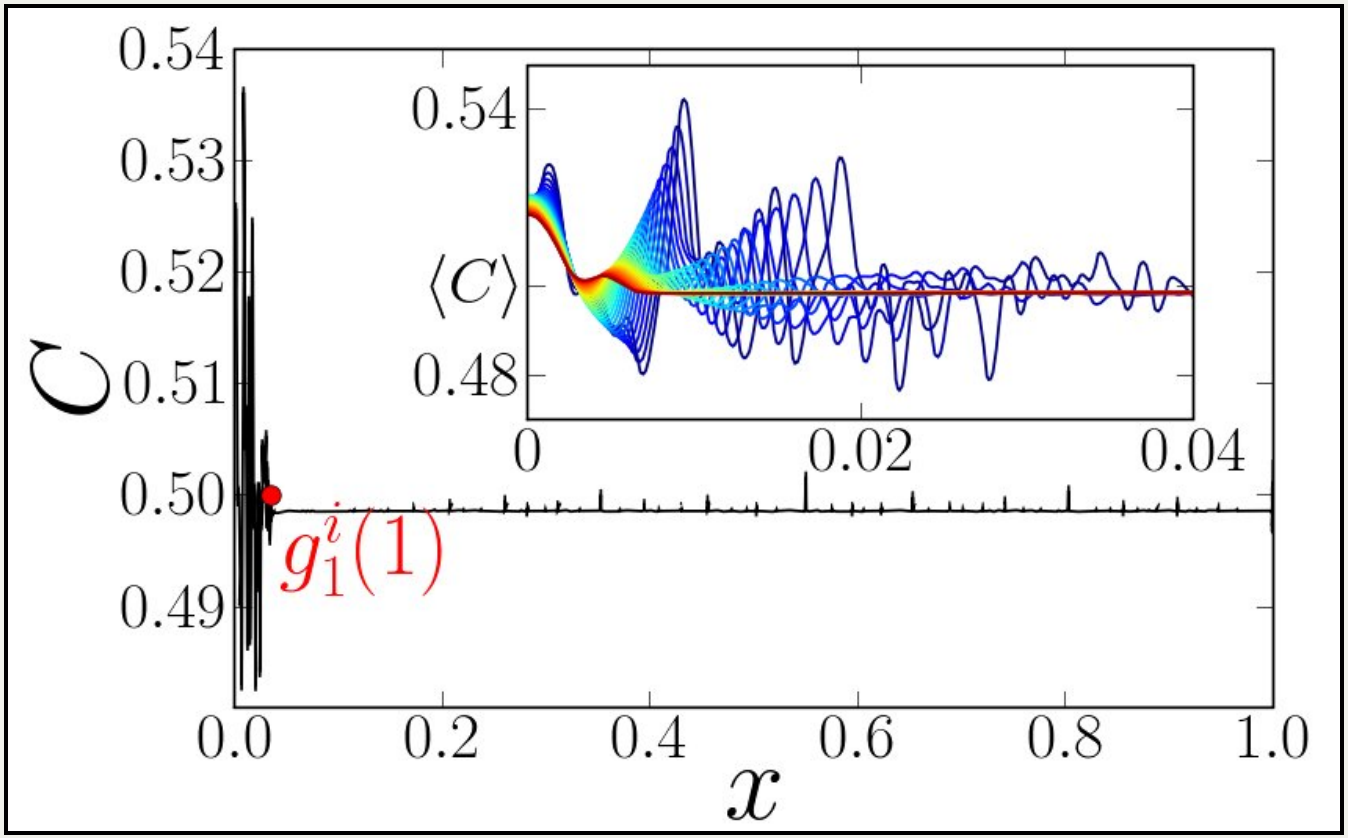
- **Python**, a generic and modern computing language
  - Python language: data types (`string`, `int`), flow control, data collections (lists, dictionaries), patterns, etc.
  - Modules of the standard library.
  - A large number of specialized modules or applications written in Python: web protocols,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



- **IPython**, an advanced **Python shell** http://ipython.org/

- **Numpy** : provides powerful **numerical arrays** objects, and routines to manipulate them. **http://www.numpy.org/**

- **Scipy** : high-level data processing routines. Optimization, regression, interpolation, etc **http://www.scipy.org**/



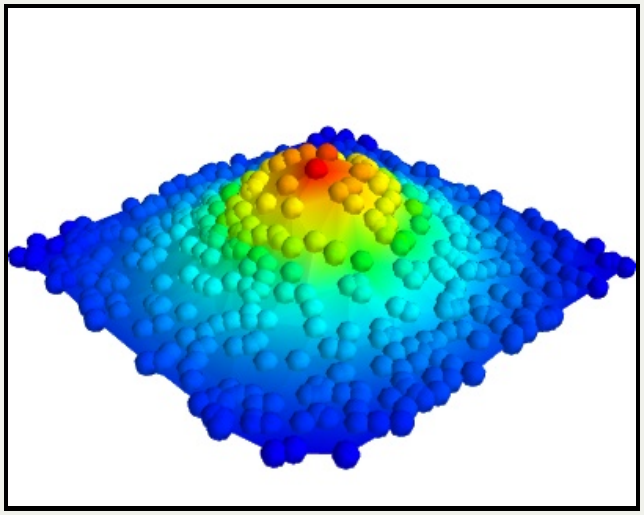
- **Matplotlib** : 2-D visualization, "publication-ready" plots
http://matplotlib.org/

- **SymPy**: Symbolic computing inside python **http://sympy.org**

- **Pandas**: Data manipulation **http://pandas.pydata.org**

- **PyTables**: Very large on node files and operations
  http://www.pytables.org/

- **Cython**: Compile Typed Python to C and call C functions
  http://cython.org/



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

HPC Building Blocks

Libraries from the HPC Community using python.

Parallelism

- mpi4py
- Ipython.parallel
- pypar
- pyUPC

Data

- pyh5
- Blaze
- gain (numpy + global arrays)
- Odin

Linear Algebra

- elemental
- pestc4py
- pyTrilinos

Applications

- gpaw
- galaxy
- FEniCS
- PyClaw
- Yt / Aztro
- Visit
- Flash

Speed ups

- Numba
- Copperhead
- Parakeet
- Pythran

The interactive workflow: IPython and a text editor

- Not a single "blessed" environment
- IPython provides many interactive elements missing in base interpreter
  - tab completion
  - documentation in a pager
  - notebook interface for literate style
  - simple parallel features

Command line interaction

Start `ipython`:

```
$ ipython
Python 2.7.5 (default, Aug  2 2013, 22:27:50)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print("Hello World")
Hello World
```

Getting help

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

```
In [6]: list?
Type:        type
String Form:<type 'list'>
Namespace:   Python builtin
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

View *python* source with **??** operator:

```
In [13]: os.path.join??
Type:       function
String Form:<function join at 0x10198eed8>
File:       /usr/local/Cellar/python/2.7.5/Frameworks/Python.framework/Version
s/2.7/lib/python2.7/posixpath.py
Definition: os.path.join(a, *p)
Source:
def join(a, *p):
    """Join two or more pathname components, inserting '/' as needed.
    If any component is an absolute path, all previous path components
    will be discarded.  An empty last part will result in a path that
    ends with a separator."""
    path = a
    for b in p:
        if b.startswith('/'):
            path = b
        elif path == '' or path.endswith('/'):
            path +=  b
        else:
            path += '/' + b
    return path
```

IPython Tips and Tricks

Ipython also contains many *magic* functions for iterating on your algorithm:

- `%run` - run file as if it were a script
- `%timeit` - times a single expression
- `%debug` - debug the last traceback

`%run` - run file as if it were a script

```
In [14]: %run examples/intro/04_regular_expressions.py
Reading from examples/intro/../data/python.bib
Found a book with the tag 'Langtangen2011'
The title is 'A Primer on Scientific Programming with Python'
Found a book with the tag 'Langtangen2010'
The title is 'Python Scripting for Computational Science'
In [15]: infile
Out[15]: u'examples/intro/../data/python.bib'
```

`%timeit` - times a single expression

```
In [16]: %timeit range(100)
1000000 loops, best of 3: 1.2 us per loop
In [17]: %timeit sum(xrange(100))
100000 loops, best of 3: 1.54 us per loop
In [18]: %timeit sum(range(100))
100000 loops, best of 3: 2.65 us per loop
In [19]: def loop_sum():
   ....:     sum = 0
   ....:     for i in xrange(100):
   ....:         sum += i
   ....:     return sum
   ....:

In [20]: %timeit loop_sum()
100000 loops, best of 3: 6.72 us per loop
```

%debug - debug the last traceback

```
In [17]: %run examples/intro/05_debug.py
IndexError                                Traceback (most recent call last)
<snip>
In [18]: %debug
> /Users/aterrel/Dropbox/Documents/Teaching/pyhpc-tutorial/examples/intro/05_d
ebug.py(3)<module>()
      1 l = range(10)
      2
----> 3 l[10] = 5

ipdb> print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
ipdb> print(len(l))
10
ipdb> print(l[9])
9
```

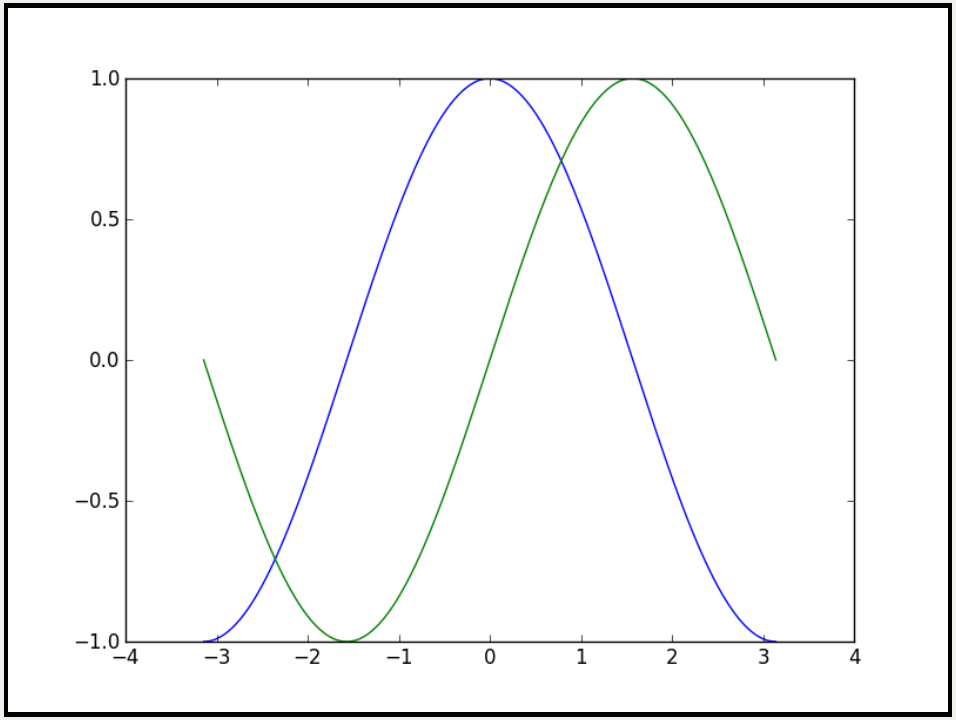
Other useful magic functions are:

- `%cd` to change the current directory (see `alias` for other mapped shell commands).
- `!<command>` to use shell commands.
- `%cpaste` allows you to paste code, especially code from websites which has been prefixed with the standard python prompt (e.g. `>>>`) or with an ipython prompt, (e.g. `in [3]`):
- `%history` to see your history (or save your session)
- To explore use *tab completion*

Matplotlib - Plotting

- Publication worthy plotting
- Focus on 2D with some 3D and animation support
- Can hit many different backend
- See large gallery at: **http://matplotlib.org/gallery.html**

Simple plot

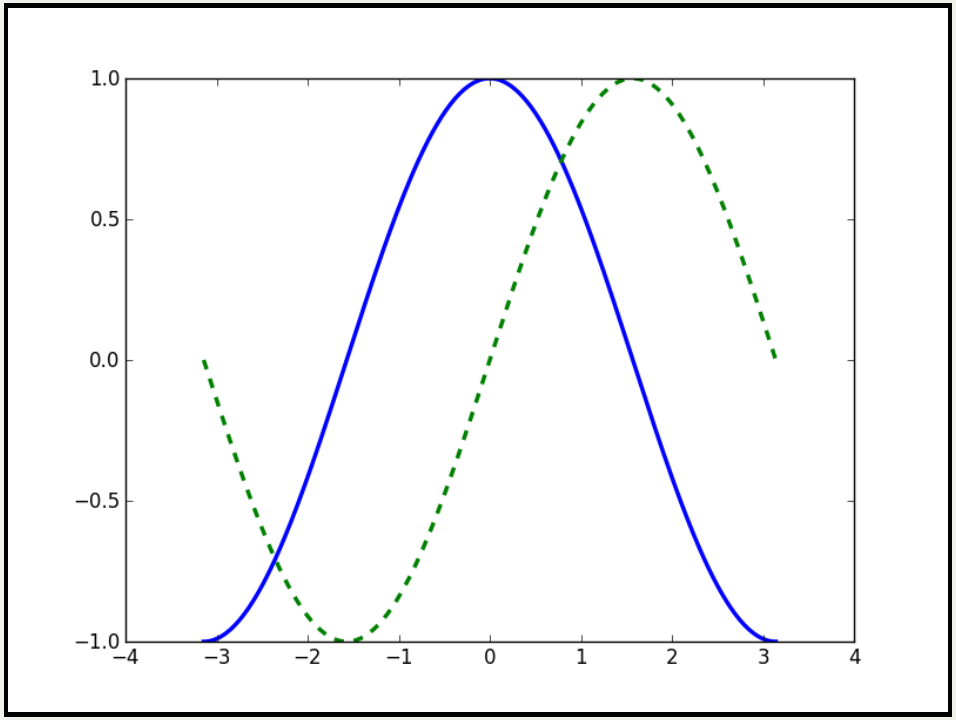



```
import pylab as pl
import numpy as np

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

pl.plot(X, C)
pl.plot(X, S)
pl.show()
```

Customize Everything

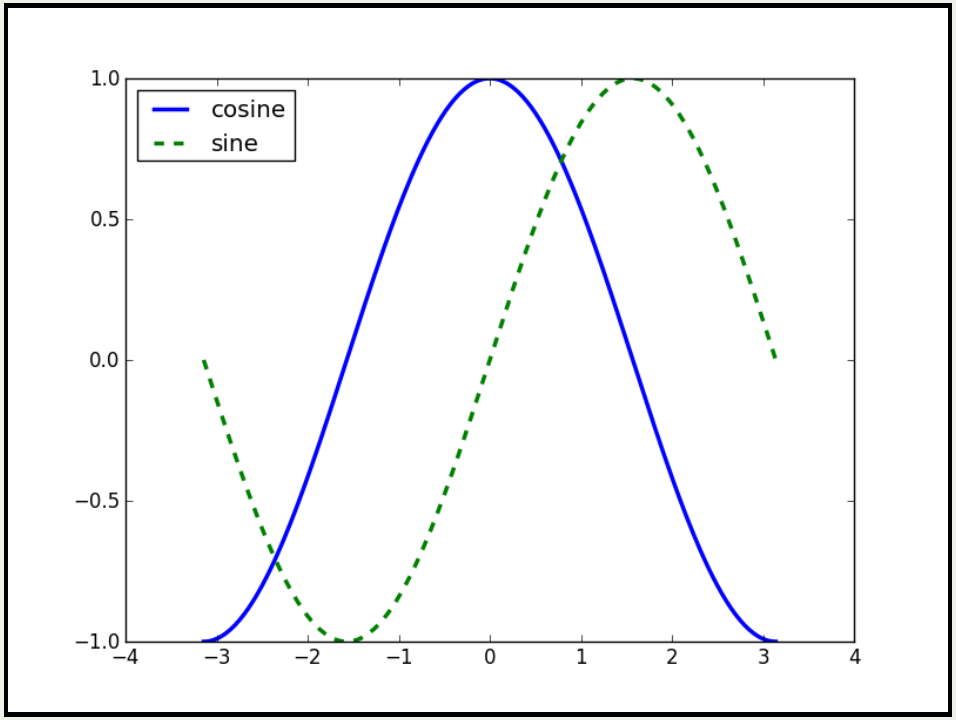
```
 # Create a figure of size 8x6 points, 80 dots per inch
pl.figure(figsize=(8, 6), dpi=80)
 # Create a new subplot from a grid of 1x1
pl.subplot(1, 1, 1)
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)
 # Plot cosine with a blue continuous line of width 1 (pixels)
pl.plot(X, C, color="blue", linewidth=1.0, linestyle="-")
 # Plot sine with a green continuous line of width 1 (pixels)
pl.plot(X, S, color="green", linewidth=1.0, linestyle="-") # Set x limits
pl.xlim(-4.0, 4.0) # Set x ticks
pl.xticks(np.linspace(-4, 4, 9, endpoint=True)) # Set y limits
pl.ylim(-1.0, 1.0) # Set y ticks
pl.yticks(np.linspace(-1, 1, 5, endpoint=True)) # Save figure using 72 dots pe
r inch
 # savefig("exercice_2.png", dpi=72)
 # Show result on screen
pl.show()
```

Change linestyles



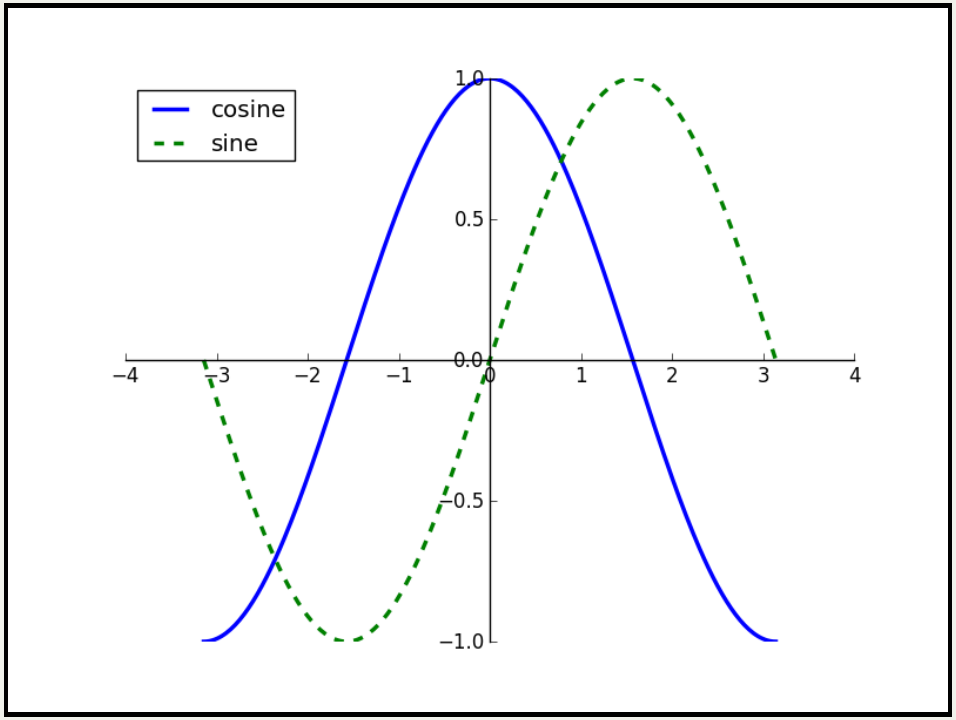

```
pl.figure(figsize=(10, 6), dpi=80)
pl.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
pl.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
```

Adding legend



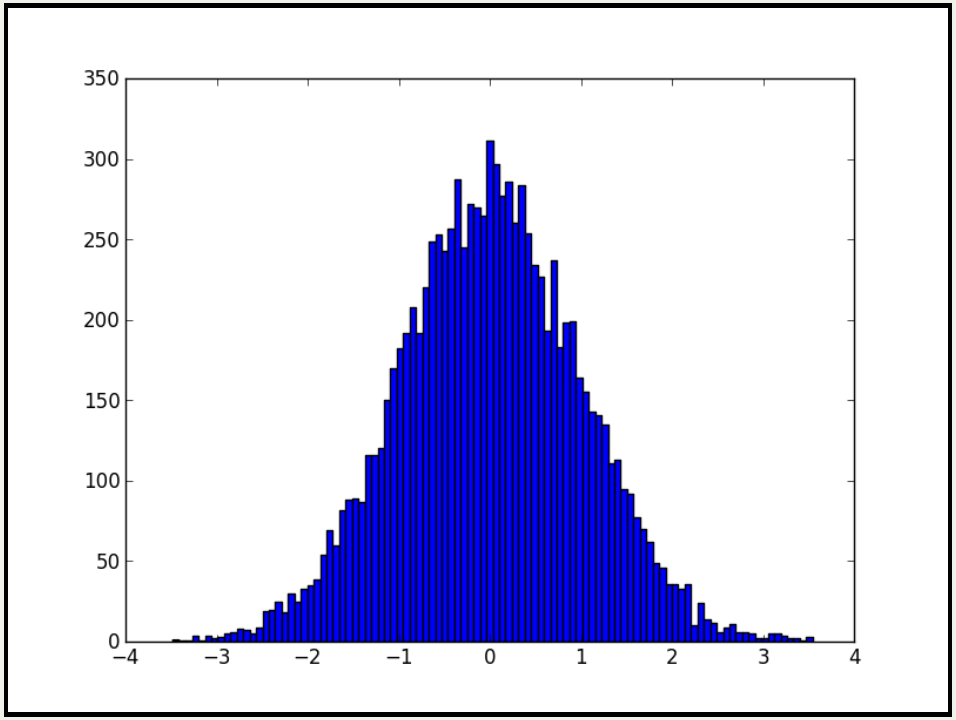

```
pl.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
pl.plot(X, S, color="green", linewidth=2.5, linestyle="--", label="sine") # Se
t x limits
pl.legend(loc='upper left')
```

Moving splines



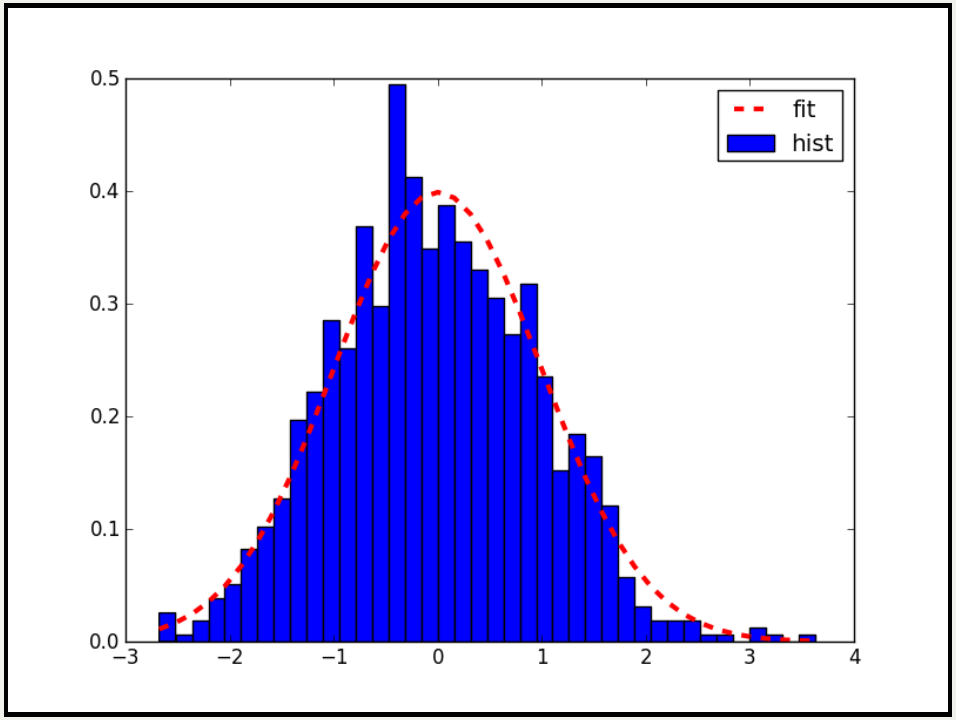

```
ax = pl.gca() # gca stands for 'get current axis'
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
```

Histograms




```
import pylab as pl

pl.plot(pylab.randn(10000), 100)
pl.show()
```

Add a fit line and legend




```
n, bins, patches = pl.hist(pl.randn(1000), 40, normed=1)
l, = pl.plot(bins, pl.normpdf(bins, 0.0, 1.0), 'r--', label='fit', linewidth=3
)
legend([l, patches[0]], ['fit', 'hist'])
```

Other types of plots include:

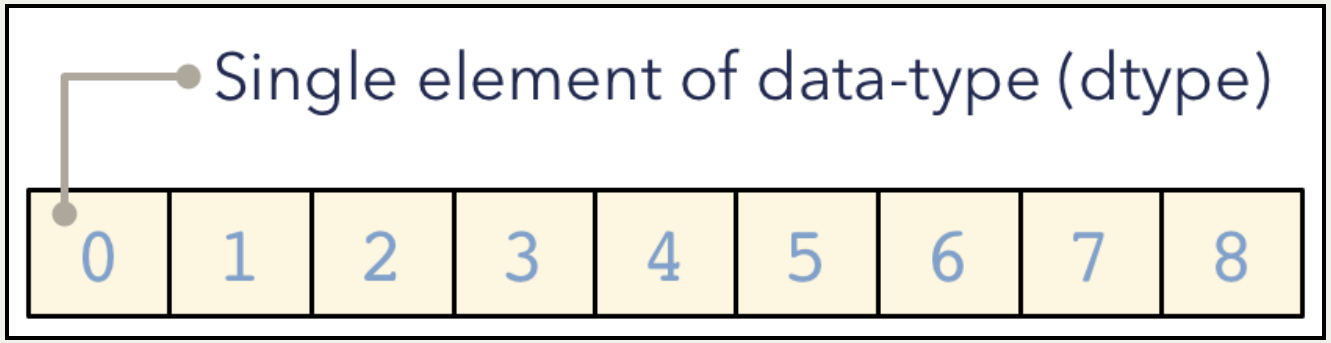
- Scatter
- Bar
- Pie
- Sankey
- Images
- Quivers
- Multiplots
- Polar
- 3D

Other elements to know about:

- Ticks: control how the ticks look
- Annotations: Add visual elements to your plot
- Axes: draw plots on top of themselves
- Backends: draw for different rendering engines

NumPy

Python library that provides multi-dimensional arrays, tables, and matrices for Python



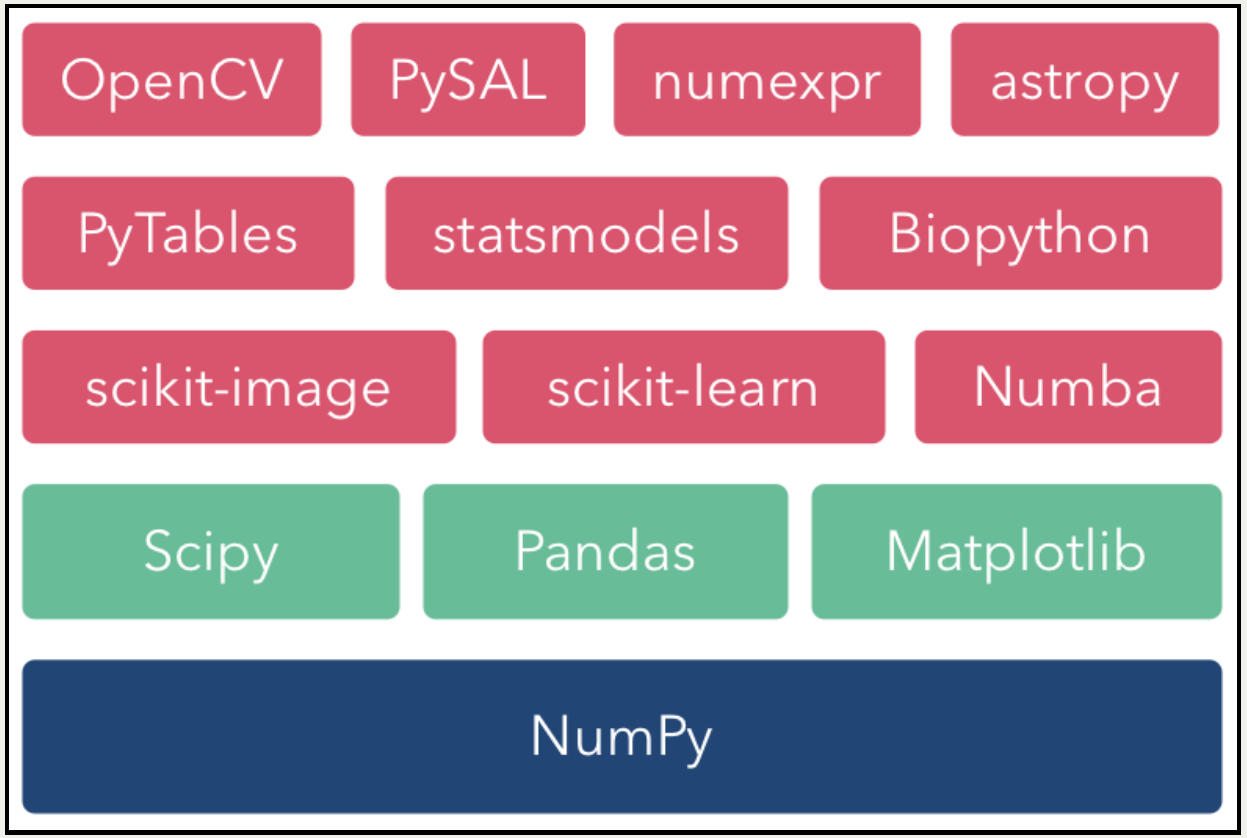
- Contiguous or strided arrays
- Homogeneous (but types can be algebraic)
  - Arrays of records and nested records
- Fast routines for array operations (C, ATLAS, MKL)

NumPy's Many Uses

- Image and signal processing
- Linear algebra
- Data transformation and query
- Time series analysis
- Statistical analysis
- Many more!

NumPy is the foundation of the Python scientific stack

NumPy Ecosystem



Basic array tour

```
In [18]: a = np.array([0,1,2,3,4,5], dtype=int)

In [19]: a
Out[19]: array([0, 1, 2, 3, 4, 5])

In [20]: a[1:3]
Out[20]: array([1, 2])

In [21]: a.ndim
Out[21]: 1

In [22]: a.shape
Out[22]: (6,)
```

Simple 2D

```
In [24]: b = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=float)

In [25]: b.ndim
Out[25]: 2

In [26]: b.shape
Out[26]: (3, 3)

In [27]: b[1:3,1:3]
Out[27]:
array([[4., 5.],
       [7., 8.]])

In [31]: b[..., 1:3]
Out[31]:
array([[1., 2.],
       [4., 5.],
       [7., 8.]])
```

But I thought arrays were just a pointer?

```
In [47]: c = np.arange(9)

In [48]: c.data
Out[48]: <read-write buffer for 0x7f923b47d3f0, size 72, offset 0 at 0x1077829
30>

In [49]: c.strides
Out[49]: (8,)

In [50]: c.shape
Out[50]: (9,)

In [51]: d = c.reshape((3,3))

In [52]: d.data
Out[52]: <read-write buffer for 0x7f923b51b470, size 72, offset 0 at 0x1077829
b0>

In [53]: d.strides
Out[53]: (24, 8)

In [54]: d.shape
Out[54]: (3, 3)
```

Common arrays

```
In [66]: print np.arange(10) # Like range [0, 1, ..., 9]
[0 1 2 3 4 5 6 7 8 9]

In [67]: print np.arange(1,9, 2) # [1, 3, 5, 7]
[1 3 5 7]

In [68]: print np.linspace(0, 1, 6) # A linear space of [0,1] with 6 pts
[ 0.   0.2  0.4  0.6  0.8  1. ]

In [69]: print np.linspace(0, 1, 6, endpoint=False) # [0,1[
[ 0.          0.16666667  0.33333333  0.5         0.66666667  0.83333333]
```

Common arrays

```
In [70]: print np.ones((3,3)) # 3 X 3 2D array of 1's
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

In [71]: print np.eye(3)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

In [72]: print np.diag(np.arange(4))
[[0 0 0 0]
 [0 1 0 0]
 [0 0 2 0]
 [0 0 0 3]]

In [73]: print np.random.rand(4) # Uniform distribution
[ 0.39259348  0.84921539  0.70292474  0.10054081]

In [74]: print np.random.randn(4) # Gaussian distribution
[ 0.53405047 -3.12422252  0.19564584  0.217296  ]
```

Fast operations

Just like matlab, vectorized operations are much faster in NumPy

```
In [9]: %timeit [x + 1 for x in xrange(100000)]
100 loops, best of 3: 8.38 ms per loop

In [10]: %timeit np.arange(100000) + 1
10000 loops, best of 3: 153 us per loop

In [11]: a = range(100)
In [12]: b = range(100)
In [13]: %timeit [a[i]*b[i] for i in xrange(len(a))]
100000 loops, best of 3: 19.8 us per loop

In [15]: c = np.arange(100)
In [16]: d = np.arange(100)
In [17]: %timeit c*d
100000 loops, best of 3: 2.25 us per loop
```

Scalar and aggregate operations

```
In [90]: b = np.arange(10)

In [91]: b * 2 + 1
Out[91]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])

In [93]: np.max(b)
Out[93]: 9

In [94]: np.sin(b)
Out[94]:
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

Dot products

```
In [79]: a = np.arange(3)

In [80]: b = np.arange(9, dtype=float).reshape((3,3))

In [81]: c = np.arange(9, dtype=float).reshape((3,3))

In [82]: np.dot(b, a)
Out[82]: array([  5.,  14.,  23.])

In [83]: np.dot(b,c)
Out[83]:
array([[  15.,   18.,   21.],
       [  42.,   54.,   66.],
       [  69.,   90.,  111.]])
```

Other pieces of NumPy to be aware of:

```
In [84]: np.linalg?

In [85]: np.random?

In [86]: np.fft?
```

Scipy : high-level scientific computing

Set of useful modules for all parts of scientific computing.

ster
nstants
back
egrate
erpolate

alg
mage

- odr
- optimize
- signal
- sparse
- spatial
- special
- stats

Import

In general, you can grab SciPy functionality via

```
import scipy
```

SciPy Constants

A plethora of important, fundamental constants can be found in **scipy.constants**. NOTE: However, this module is not automatically included when you **import scipy**. Still, some very basic pieces of information are given as module attributes.

The following, for example:

```
  import scipy.constants
  import math

  print("SciPy thinks that pi = %.16f"%scipy.constants.pi)
  print("While math thinks that pi = %.16f"%math.pi)
  print("SciPy also thinks that the speed of light is c = %.1F"%scipy.constant
s.c)
```

will return

```
  >>> SciPy thinks that pi = 3.1415926535897931
  >>> While math thinks that pi = 3.1415926535897931
  >>> SciPy also thinks that the speed of light is c = 299792458.0
```

However, the real value of SciPy Constants is its enormous physical constant database. These are of the form: **scipy.constants.physical_constants[name] = (value, units, uncertainty)**.

For example, the mass of an alpha particle is:

```
>>> scipy.constants.physical_constants["alpha particle mass"]
>>> (6.6446564999999997e-27, 'kg', 1.1e-33)
```

How can you tell what the key is for this function? My favorite way is with the scipy.constants.find() method.

```
scipy.constants.find("light")
```

gives :

```
['speed of light in vacuum']
```

But buyer beware! Let's look at the speed of light again.

```
 >>> print("c = %s"%str(scipy.constants.physical_constants["speed of light in
vacuum"]))
 >>> c = (299792458.0, 'm s^-1', 0.0)
```

The uncertainty in c should not be zero! Right? Wrong. The **meter** is in fact set by the speed of light in a vacuum. So there is, by definition, no error. However, this has not always been the case. Moreover, any actual determination of the **c** or the meter has a measurement uncertainty, but SciPy does not acknowledge it.

BASIC LESSON: As always, pay attention.

Check
http://docs.scipy.org/doc/scipy/reference/constants.html for a complete constants listing.

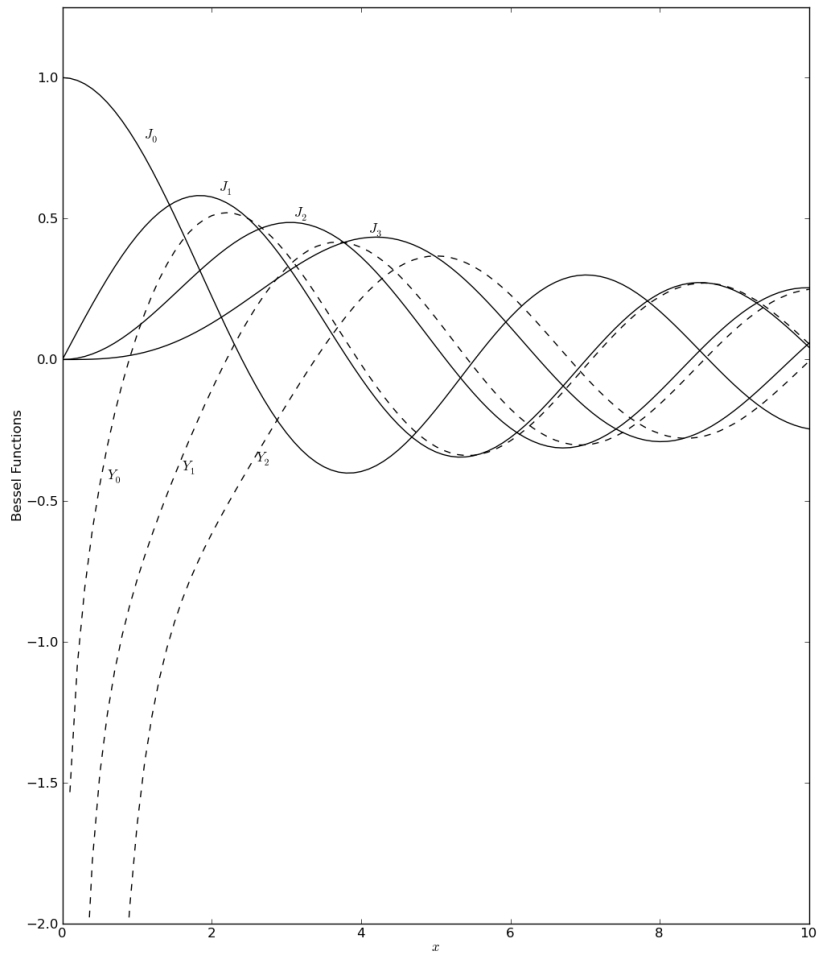
The above code is reproduced concisely in the constants.py file found in the SciPy directory of your PyTrieste repository.

SciPy Special Functions

Code that numerically approximates common (and some not-so-common) special functions can be found in **scipy.special**. Here you can find things like error functions, gamma functions, Legendre polynomials, etc. But as a example let's focus on my favorites: the Bessel functions.

```
from scipy.special import *
from pylab import *

x = arange(0.0, 10.1, 0.1)

for n in range(4):
  j = jn(n, x)
  plot(x, j, 'k-')
  text(x[10*(n+1)+1], j[10*(n+1)], r'$J_%r$'%n)

for n in range(3):
  y = yn(n, x) plot(x, y, 'k--')
  text(x[10*(n)+6], y[10*(n)+5], r'$Y_%r$'%n)

axis([0, 10, -2, 1.25])
xlabel(r'$x$')
ylabel("Bessel Functions")

show()
```

These 20-ish lines of code should produce :



Note that the figure that was created here is a reproduction of Figure 6.5.1 in "Numerical Recipes" by W. H. Press, et al... (http://www.nr.com/).

Check out http://docs.scipy.org/doc/scipy/reference/special.html for a complete listing of special functions.

SciPy Integration

Tools used to calculate numerical, definite integrals may be found in the **integrate** module. There are two basic ways you can integrate in SciPy:

1. Integrate a function,
2. Integrate piecewise data.

First, let's deal with integration of functions. Recall that in Python, functions are also objects. Therefore you can pass functions as arguments to other functions! Just make sure that the function that you want to integrate returns a float, or at the very least, an object that has a ***_float_*()** method.

Integration Example: 1D

The simplest way to compute a functions definite integral is via the **quad(...)** function. The script

```
import scipy.integrate #For kicks, let's also grab
import scipy.special
import numpy
def CrazyFunc(x):
  return (scipy.special.i1(x) - 1)**3

print("Try integrating CrazyFunc on the range [-5, 10]...")

val, err = scipy.integrate.quad(CrazyFunc, -5, 10)

print("A Crazy Function integrates to %.8E"%val)
print("And with insanely low error of %.8E"%err)
```

will return

```
>>> Try integrating CrazyFunc on the range [-5, 10]...
>>> A Crazy Function integrates to 6.65625226E+09
>>> And with insanely low error of 3.21172897E-03
```

Integration Example: Infinite Limits

You can also use **scipy.integrate.Inf** for infinity in the limits of integration. For example, try integrating e\^x on `[-inf, 0]`:

```
>>> print("(val, err) = " +
    str( scipy.integrate.quad(scipy.exp, -scipy.integrate.Inf, 0.0) ))
```

will return :

```
>>> (val, err) = (1.0000000000000002, 5.8426067429060041e-11)
```

Integration Example: 2D

Two dimensional integrations follow similarly to the 1D case. However, now we need to use the **dblquad(f(y,x), ...)** function instead of simply **quad(f(x), ...)**. Because a picture is worth 10\^3 words, SciPy computes the right-hand side of the following equation:

$$\int\int_F f(y,x)\,dA = \int_a^b \int_{g(x)}^{h(x)} f(y,x)\,dy\,dx$$

More information on the justification of this integral may be found at http://en.wikipedia.org/wiki/Order_of_integration_(calculus). For example, let's try to integrate the surface area of a unit sphere:

```
def dA_Sphere(phi, theta):
  return  scipy.sin(phi)

print("Integrate the surface area of the unit sphere...")
val, err = scipy.integrate.dblquad(dA_Sphere, 0.0, 2.0*scipy.pi,
  lambda theta: 0.0,
  lambda theta: scipy.pi )
print("val = %.8F"%val)
print("err = %.8E"%err)
```

This will return :

```
>>> Integrate the surface area of the unit sphere...
>>> val = 12.56637061
>>> err = 1.39514740E-13
```

There are a couple of subtleties here. First is that the function you are integrating over is defined as **f(y,x)** and NOT the more standard **f(x,y)**. Moreover, while **x**'s limits of integration are given directly **[0, 2*pi*]**,***y****'s limits have to be functions**[g(x), h(x)]** **(given by the 'lambdas' here). This method of doing double integrals allows for**y* *to have a more complicated edge in*x* than a simple point. This is great for some functions but a little annoying for simple integrations. In any event, the above integral computes the surface are of a unit sphere to be *4pi** to within floating point error.

Integration Example: 3D

Three dimensional integration is more similar to 2D than 1D. Once again, we define our function variables in reverse order, **f(z, y, x)**, and integrate using **tplquad(f(z, y, x))**. Moreover, **z** has limits of integration defined by surfaces give as functions **[q(x,y), r(x,y)]**. Thus, **tplquad(...)** integrates the right-hand side of :

$$\int\int\int_F f(z,y,x)\,dV = \int_a^b \int_{g(x)}^{h(x)} \int_{q(x,y)}^{r(x,y)} f(z,y,x)\,dz\,dy\,dx$$

To continue with the previous example, let's try integrating the volume of a sphere. Take the radius here to be 3.5.

```
def dV_Sphere(phi, theta, r):
  return r * r * dA_Sphere(phi, theta)

print("Integrate the volume of a sphere with r=3.5...")
val, err = scipy.integrate.tplquad(dV_Sphere, 0.0, 3.5, lambda r: 0.0,
  lambda r: 2.0*scipy.pi, lambda x, y: 0.0, lambda x, y: scipy.pi)
print("val = %.8F"%val)
print("err = %.8E"%err)
```

will return:

```
>>> Integrate the volume of a sphere with r=3.5...
>>> val = 179.59438003
>>> err = 1.99389816E-12
```

A simple hand calculation verifies this result.

Integration Example: Trapazoidal

Now, only very rarely will scientists (and even more rarely engineers) will truely 'know' the function that they wish to integrate. Much more often we'll have piecewise data that we wish numerically integrate (ie sum an array y(x), biased by array x). This can be done in SciPy through the **trapz(...)** function.

```
y = range(0, 11)
print("Trapazoidally integrate y = x on [0,10]...")
val = scipy.integrate.trapz(y)
print("val = %F"%val)
```

will return:

```
>>> Trapazoidally integrate y = x on [0,10]...
>>> val = 50.000000
```

The above takes a series of y-values that are implicitly spaced 1-unit apart in x and 'trapazoidally integrates' them. Basically, just a sum.

However, you can use the **x= [0, 3,...]** or **dx = 3** argument keywords to explicitly declare different spacings in x. For example, with y = x\^2:

```
x = numpy.arange(0.0, 20.5, 0.5)
y = x * x
print("Trapazoidally integrate y = x^2 on [0,20] with half steps...")
val = scipy.integrate.trapz(y, x)
print("val = %F"%val)
```

```
>>> Trapazoidally integrate y = x^2 on [0,20] with half steps...
>>> val = 2667.500000
```

```
print("Trapazoidally integrate y = x^2 with dx = 0.5...")
val = scipy.integrate.trapz(y, dx = 0.5)
print("val = %F"%val)
```

```
>>> Trapazoidally integrate y = x^2 with dx = 0.5...
>>> val = 2667.500000
```

Integration Example: Ordinary Differential Equations

Say that you have an ODE of the form **dy/dt = f(y, t)** that you "really" need integrated. Then you, my friend, are in luck! SciPy can do this for you using the **scipy.integrate.odeint** function. This is of the form:

```
  odeint( f, y0, [t0, t1, ...])
```

For example take the decay equation: (y, t) = - lambda * y

We can then try integrating this using a decay constant of 0.2`:

```
  def dDecay(y, t, lam):
    return -lam*y

  vals = scipy.integrate.odeint( lambda y, t: dDecay(y, t, 0.2), 1.0, [0.0, 10
.0] )
  print("If you start with a mass of y(0) = %F"%vals[0][0])
  print("you'll only have y(t= 10) = %F left."%vals[1][0])
```

This will return

```
>>> If you start with a mass of y(0) = 1.000000
>>> you'll only have y(t= 10) = 0.135335 left.
```

Check out http://docs.scipy.org/doc/scipy/reference/integrate.html for more information on integreation.

SciPy Image Tricks

SciPy has the ability to treat 2D & 3D arrays as images.

- convert PIL images
- read in external files as numpy arrays!
- buried within the `miscellaneous` module.

http://en.wikipedia.org/wiki/File:JumpingRabbit.JPG

```
import scipy.misc
img = scipy.misc.imread("image.jpg")
#Note that this really is an array!
print(str(img))
```

```
>>> [[[130 174  27]
>>>   [129 173  24]
>>>   [127 171  22]
>>>   ...,
>>>   [147 192  41]
>>>   [146 190  41]
>>>   [146 190  41]]
>>>
>>>  [[137 177  29]

>>>   [133 175  27]
>>>   [130 173  21]
>>>   ...,
>>>   [147 192  37]
>>>   [149 194  41]
>>>   [149 194  41]]
>>>
>>>  [[141 177  29]
>>>   [137 176  25]
>>>   [130 174  19]
>>>   ...,
>>>   [148 194  34]
>>>   [149 195  35]
```

We can now apply some basic filters...

```
img = scipy.misc.imfilter(img, 'blur')
```

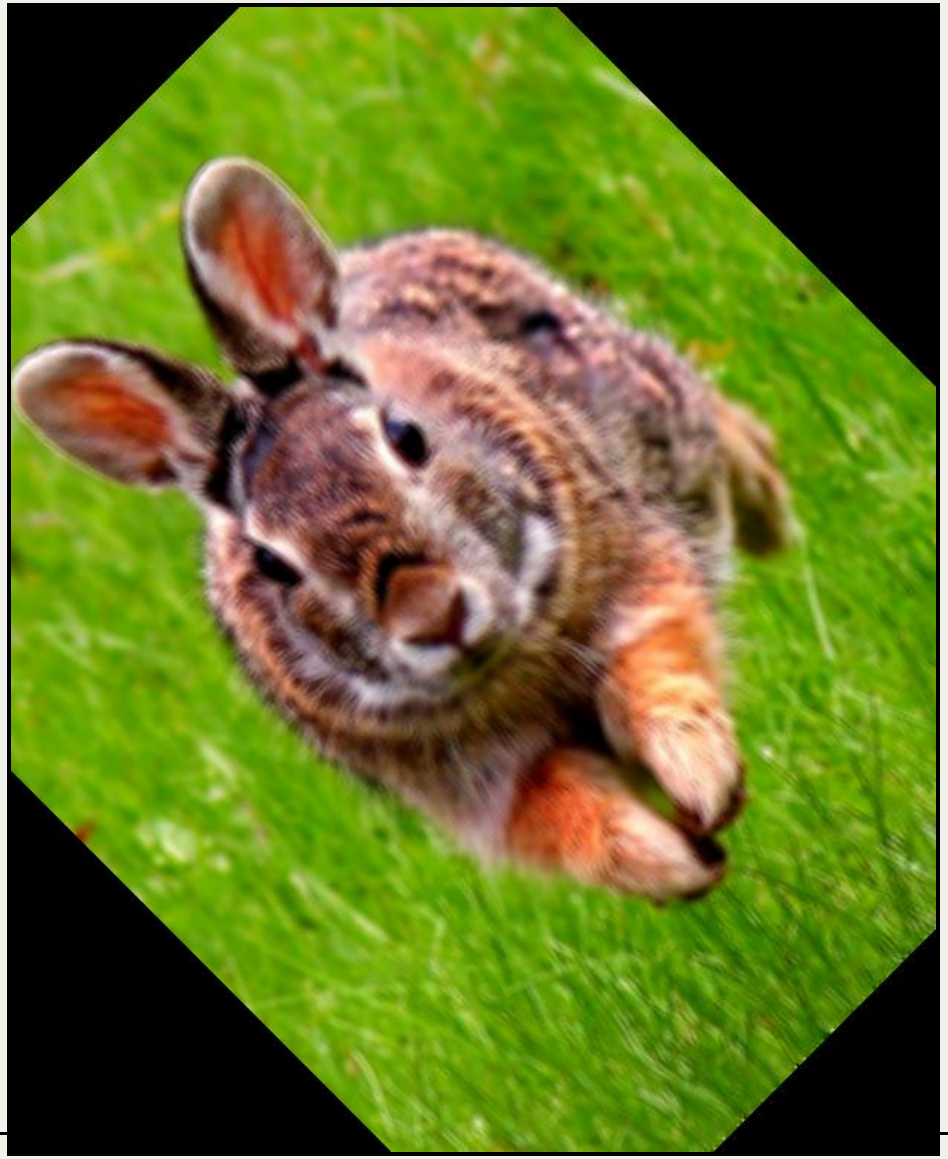
We can even rotate the image, counter-clockwise by degrees.

```
img = scipy.misc.imrotate(img, 45)
```

And then, we can rewrite the array to an image file.

```
scipy.misc.imsave("image1.jpg", img)
```

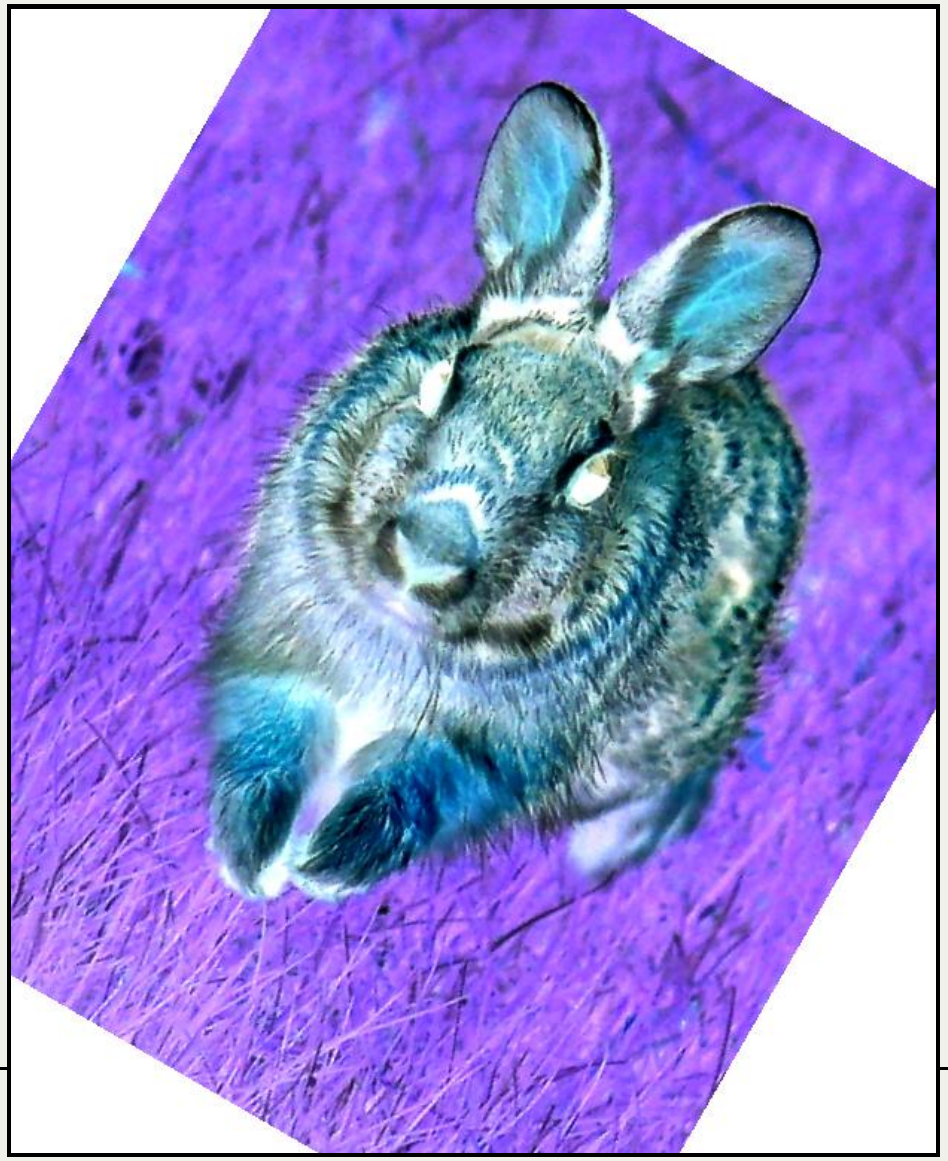
These functions produce the following image:



Because the array takes integer values from 0 - 255, we can easily define our own filters as well! For instance, you could write a two-line function to inverse the image...

```
def InverseImage(imgarr):
   return 255 - imgarr
#Starting fresh we get...
img = scipy.misc.imread("image.jpg")
img = scipy.misc.imrotate(img, 330)
img = InverseImage(img)
scipy.misc.imsave("image2.jpg", img)
```

Having this much fun, the rabbit becomes a twisted shade of its former self!



Check out
http://docs.scipy.org/doc/scipy/reference/misc.html for a

complete listing of associated image functions.

File input/output:

Loading and saving matlab files::

```
>>> from scipy import io as spio
>>> a = np.ones((3, 3))
>>> spio.savemat('file.mat', {'a': a}) # savemat expects a dictionary
>>> data = spio.loadmat('file.mat', struct_as_record=True)
>>> data['a']
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Reading images::

```
>>> from scipy import misc
>>> misc.imread('fname.png')
>>> # Matplotlib also has a similar function
>>> import matplotlib.pyplot as plt
>>> plt.imread('fname.png')
```

See also:

```
* Load text files: :func:`numpy.loadtxt`/:func:`numpy.savetxt`

* Clever loading of text/csv files:
  :func:`numpy.genfromtxt`/:func:`numpy.recfromcsv`

* Fast and efficient, but numpy-specific, binary format:
  :func:`numpy.save`/:func:`numpy.load`
```

Special functions:

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

Linear algebra operations:

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

- The `scipy.linalg.det` function computes the determinant of a square matrix::

```
>>> from scipy import linalg
>>> arr = np.array([[1, 2],
...                 [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 `scipy.linalg.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
```

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, for example singular-value decomposition (SVD)::

```
>>> arr = np.arange(9).reshape((3, 3)) + np.diag([1, 0, 1])
>>> uarr, spec, vharr = linalg.svd(arr)
```

The resulting array spectrum is::

```
>>> spec
array([ 14.88982544,   0.45294236,   0.29654967])
```

The original matrix can be re-composed by matrix multiplication of the outputs of `svd` with `np.dot`::

```
>>> sarr = np.diag(spec)
>>> svd_mat = uarr.dot(sarr).dot(vharr)
>>> np.allclose(svd_mat, arr)
True
```

Fast Fourier transforms:

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

```
>>> time_step = 0.02
>>> period = 5.
>>> time_vec = np.arange(0, 20, time_step)
>>> sig = np.sin(2 * np.pi / period * time_vec) + \
...       0.5 * np.random.randn(time_vec.size)
```

The observer doesn't know the signal frequency, only the sampling time step of the signal `sig`. The signal is supposed to come from a real function so the Fourier transform will be symmetric.

The `scipy.fftpack.fftfreq` function will generate the sampling frequencies and `scipy.fftpack.fft` will compute the fast Fourier transform::

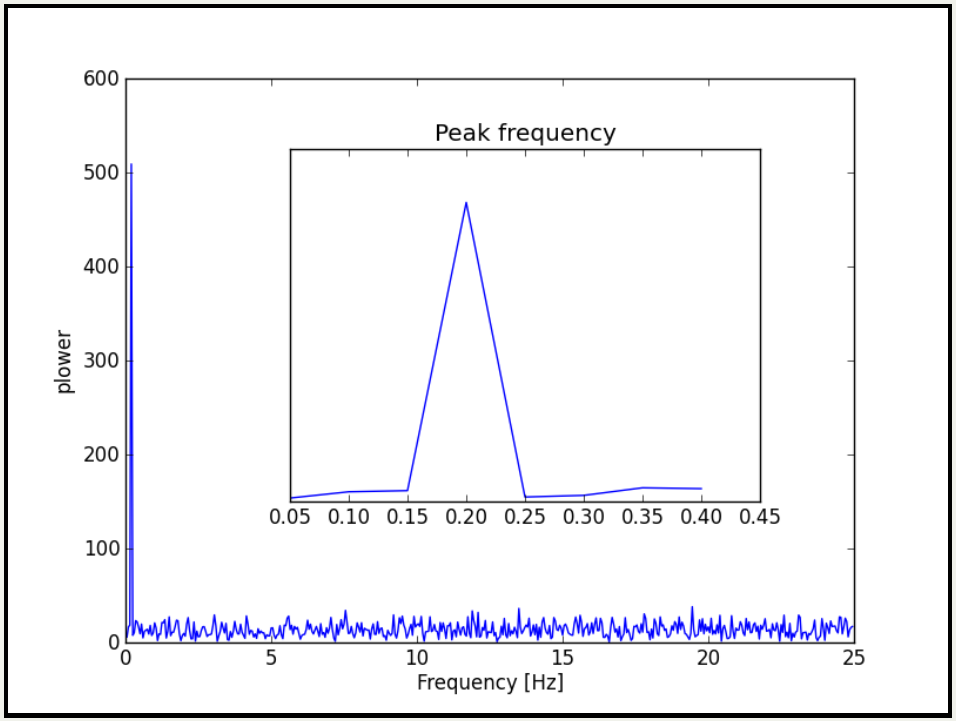
```
>>> from scipy import fftpack
>>> sample_freq = fftpack.fftfreq(sig.size, d=time_step)
>>> sig_fft = fftpack.fft(sig)
```

Because the resulting power is symmetric, only the positive part of the spectrum needs to be used for finding the frequency::

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

The signal frequency can be found by::

```
>>> freq = freqs[power.argmax()]
>>> np.allclose(freq, 1./period)  # check that correct freq is found
True
```



Now the high-frequency noise will be removed from the Fourier transformed signal::

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

The resulting filtered signal can be computed by the `scipy.fftpack.ifft` function::

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

The result can be viewed with::

```
>>> import pylab as plt
>>> plt.figure()
>>> plt.plot(time_vec, sig)
>>> plt.plot(time_vec, main_sig, linewidth=3)
>>> plt.xlabel('Time [s]')
>>> plt.ylabel('Amplitude')
```