Scientific Python Tutorial
Scientific Python

Yann Tambouret

Scientific Computing and Visualization
Information Services & Technology
Boston University
111 Cummington St.
yannpaul@bu.edu

October, 2012
This tutorial is for someone with basic python experience.

First:

1. matplotlib - plotting library, like Matlab
2. numpy - fast arrays manipulation
3. scipy - math methods galore.
4. ipython - better interactive python

Then I will cover a few intermediate details about python programming in general.
Plotting in scripts

- **matplotlib** is a package that has many modules, **pyplot** is the main driver.
- **matplotlib** is designed for both interactive and script-based use.
matplotlib is a package that has many modules, `pyplot` is the main driver.

matplotlib is designed for both interactive and script-based use.

A Figure can contain many Axes, which contain many plots.

`pyplot` creates a default Figure and Axes if you just want to get to plotting things quickly.
Basic Example

```python
from matplotlib import pyplot
pyplot.plot([0, 2, 4, 8, 16, 32], "o")
pyplot.ylabel("Value")
pyplot.xlabel("Time")
pyplot.title("Test plot")
pyplot.show()
```
Basic Plot - Results

Test plot

Value

Time

0 1 2 3 4 5

0 5 10 15 20 25 30 35
We read in the data

def get_data():
    data = file("world_population.txt", "r").readlines()
    dates = []
    populations = []
    for point in data:
        date, population = point.split()
        dates.append(date)
        populations.append(population)
    return dates, populations
We read in the data

For each line ('point') we need to separate the date from population
We read in the data

For each line ('point') we need to separate the date from population

`split()` splits text at any whitespace, by default.
World Population - Getting Help

How to read online docs

1. Google “matplotlib” and pick first choice
2. "Quick Search" (on right side) for "pyplot.plot"
3. Select result entitled "matplotlib.pyplot.plot"

```python
import matplotlib.pyplot as plt
plt.plot(x, y, 'bo')
```

Plot lines and/or markers to the Axes. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

- `plot(x, y)`
- `plot(x, y, 'bo')`
- `plot(y, 'r+')`

If *x* and/or *y* is 2-dimensional, then the corresponding columns will be plotted.

An arbitrary number of *x*, *y*, *fmt* groups can be specified, as in:

```python
a.plot(x1, y1, 'g-', x2, y2, 'g-')
```

Return value is a list of lines that were added.

The following format string characters are accepted to control the line style or marker:

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>solid line style</td>
</tr>
<tr>
<td>--</td>
<td>dashed line style</td>
</tr>
<tr>
<td>----</td>
<td>dash-dot line style</td>
</tr>
<tr>
<td>.</td>
<td>dotted line style</td>
</tr>
<tr>
<td>o</td>
<td>point marker</td>
</tr>
<tr>
<td>s</td>
<td>pixel marker</td>
</tr>
<tr>
<td>^</td>
<td>circle marker</td>
</tr>
<tr>
<td>v</td>
<td>triangle_down marker</td>
</tr>
<tr>
<td>&lt;</td>
<td>triangle_up marker</td>
</tr>
<tr>
<td>&gt;</td>
<td>triangle_left marker</td>
</tr>
</tbody>
</table>
Your Assignment

```python
13 def plot_world_pop(dates, populations):
    pass

16 dates, populations = get_data()
17 plot_world_pop(dates, populations)
18 pyplot.show()
```

1. Make a plot of population (y) vs dates (x)
2. Title: “World population over time”
3. Y-axis: ”World population in millions”
4. X-axis: ”Year”
5. Set the plot type to a Magenta, downward-triangle
```
from matplotlib import pyplot

def get_data():
    # each line: year, men, women
    data = file("life_expectancies_usa.txt", "r").readlines()
    dates = []
    men = []
    women = []
    return dates, men, women

def plot_expectancies(dates, men, women):
    pass

dates, men, women = get_data()
plot_expectancies(dates, men, women)
pyplot.show()
```
1. Use `split(',')` to split strings at commas
2. Add a label to each plot (look at documentation)
3. Label Axes and give a title.
4. Call plot 2x to plot two lines
5. Add a legend: `pyplot.legend`
   - “Quick Search” again
   - search for ”`pyplot.legend””
Life Expectancy - Results

Life Expectancy for men and women in the USA over time

Men  
Women

Age

Year

1840 1860 1880 1900 1920 1940 1960 1980 2000
Plotting Interactively

1. Home - Backward - Forward - Control edit history

2. Pan - Zoom
   - Left click + drag shifts center, right click + drag changes zoom

3. Zoom
   - Select zoom region

4. Save
   - `pyplot.savefig('filename')` is an alternative to `pyplot.show()` when you are using `pyplot` non-interactively.
Plotting Interactively

1. **Home - Backward - Forward - Control edit history**

2. **Pan - Zoom** - Left click + drag shifts center, right click + drag changes zoom

```python
pyplot.savefig('filename')
```
is an alternative to `pyplot.show()` when you are using pyplot non-interactively.

Yann - yannpaul@bu.edu (SCV)

Scientific Python

October 2012
Plotting Interactively

1. **Home - Backward - Forward - Control edit history**
2. **Pan - Zoom** - Left click + drag shifts center, right click + drag changes zoom
3. **Zoom** - Select zoom region

Save 
`pyplot.savefig('filename')` is an alternative to `pyplot.show()` when you are using `pyplot` non-interactively.

Yann - yannpaul@bu.edu (SCV)
Plotting Interactively

1. **Home - Backward - Forward - Control edit history**
2. **Pan - Zoom** - Left click + drag shifts center, right click + drag changes zoom
3. **Zoom** - Select zoom region
4. **Save**

`pyplot.savefig('filename')` is an alternative to `pyplot.show()` when you are using `pyplot` non-interactively.

Yann - yannpaul@bu.edu (SCV)
Plotting Interactively

1. **Home - Backward - Forward - Control edit history**
2. **Pan - Zoom** - Left click + drag shifts center, right click + drag changes zoom
3. **Zoom** - Select zoom region
4. **Save**

`pyplot.savefig('filename')` is an alternative to `pyplot.show()` when you are using `pyplot` non-interactively.

Yann - yannpaul@bu.edu (SCV)

Scientific Python

October 2012 13 / 76
Matplotlib Resources

- **Possibilities:**
  - matplotlib.org/gallery.html
  - http://matplotlib.org/basemap/users/examples.html

- **Guide/Tutorial:**
  - matplotlib.org/users/index.html

- **Questions:**
  - stackoverflow.com
  - Or contact us: help@scv.bu.edu

- **Source for tutorial:**
  -openhatch.org/wiki/Matplotlib
Array Creation

- By convention: `import numpy as np`
- `x = np.array([1,2,3], int); x` is a Numpy array
- `x` is like a list, but certain operations are much faster
A 2D array:

```python
A = np.array(((1, 0, 0),
              (0, 0, -1),
              (0, 1, 0)))
```
Array Creation

- `np.ones(5)` makes array([ 1., 1., 1., 1., 1.])
- `np.zeros` same thing for zeroes
np.ones(5) makes array([ 1., 1., 1., 1., 1.])

np.zeros same thing for zeroes

np.zeros((2,2)) makes array([[0., 0.], [0., 0.]])
that is a 2D, 2x2 array of zeros
Array Creation

- `np.arange`, numpy’s version of built-in `range` method

- `np.linspace` is similar to `np.arange` but you pass the number of elements, *not* step size
```python
>>> import numpy as np
>>> x = np.array(((1, 2, 3), (4, 5, 6)))
>>> x.size # total number of elements
6
>>> x.ndim # number of dimensions
2
>>> x.shape # number of elements in each dimension
(2L, 3L)
>>> x[1,2] # first index is the rows, then the column
6
>>> x[1] # give me '1' row
array([[4, 5, 6]])
>>> x[1][2]
6
>>> x.dtype
dtype('int32')
```
Basic Operations

14 x = np.array([0, np.pi/4, np.pi/2])
15 np.sin(x)
16 np.dot([2, 2, 2], [2, 2, 2])
Basic Operations

```
x = np.array([0, np.pi/4, np.pi/2])
np.sin(x)
np.dot([2, 2, 2], [2, 2, 2])
```

```
>>> x = np.array([0, np.pi/4, np.pi/2])
>>> np.sin(x)
array([ 0. , 0.70710678, 1. ])
>>> np.dot([2, 2, 2], [2, 2, 2])
12
```
It’s a Simple Game

Alive

Dead
## Calculating Neighbors

### Neighbors Number

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Use the Neighbors to Update

Leave Dead Bring To Life

Let 1 2 3 Keep Alive

Die
In the `practice` directory:

1. `conway.py` is a program; try it out.
A Practice Problem

In the `practice` directory:

1. `conway.py` is a program; try it out.
2. `conway_work.py` is an assignment; try to do it.
3. `test_conway.py` is a program to test your work.

**Remember**, the shape attribute:

```python
1 a = np.array(((1,2),(3,4),(5,6))) # make a 3x2
2 print a.shape[0] # prints 3
3 print a.shape[1] # prints 2
```
np.roll(a, 1, axis=0) returns a new array that cycles the values.

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

>>> np.roll(a, 1, axis=0)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

This can be used to make the neighbors function.
Conway’s Game of Life

```python
np.where(b == 0)  # returns a series of indices where it’s true.
np.where(b == 0, -1, b)  # to replace 0’s with -1’s.

>>> b
array([[ 0.,  0.,  0.],
       [ 2.,  2.,  2.],
       [ 0.,  3.,  0.]])

>>> np.where(b == 0)
(array([0, 0, 0, 2, 2]), array([0, 1, 2, 0, 2]))

>>> np.where(b == 0, -1, b)
array([[-1., -1., -1.],
       [ 2.,  2.,  2.],
       [-1.,  3., -1.]])
```

Look up the `np.logical_or` method; this and `np.where` can be used to redefine the `update` function.

Yann - yannpaul@bu.edu (SCV)
```
import matplotlib.pyplot as plt
import numpy as np

a = np.zeros((3, 3))
a[1, 1] = 1
plt.imshow(a,
           interpolation='nearest',
           cmap=plt.winter(),
           origin='lower')
```
Nearest Neighbors
Matplotlib Imshow

```python
plt.imshow(a,
           interpolation='nearest',
           cmap=plt.winter(),
           origin='lower')
plt.show()
```
Default: usually 'bilinear'
Plenty More

- www.scipy.org/Tentative_NumPy_Tutorial
- "Universal Functions" section
- PyTrieste Numpy Tutorial
scipy: Many Useful Scientific Tools


- Numpy
- Integration
- Optimization
- Interpolation
- FFT
- Linear Algebra
- Statistics
- And much more....
>>> from scipy import constants
>>> constants.c # speed of light
299792458.0
>>> # physical constants: value, units, uncertainty
>>> constants.physical_constants["electron mass"]
(9.10938291e-31, 'kg', 4e-38)
>>> # look-up constants, only first 3!
>>> masses = constants.find("mass")[0:3]
>>> for val in masses:
... print "'{0}" is available".format(val)
... 
'Planck mass' is available
'Planck mass energy equivalent in GeV' is available
'alpha particle mass' is available
Zombie Apocalypse - ODEINT

http://www.scipy.org/Cookbook/Zombie_Apocalypse_ODEINT

![Graph depicting the Zombie Apocalypse model](http://www.scipy.org/Cookbook/Zombie_Apocalypse_ODEINT)
scipy.integrate.odeint

`scipy.integrate.odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0)`

Integrate a system of ordinary differential equations.

Solve a system of ordinary differential equations using lsoda from the FORTRAN library odepack.

Solves the initial value problem for stiff or non-stiff systems of first order ode-s:

\[
\frac{dy}{dt} = \text{func}(y,t0,\ldots)
\]

where \(y\) can be a vector.

**Parameters:**

- `func` : callable(y, t0, \ldots)
  
  Computes the derivative of \(y\) at \(t0\).

- `y0` : array
  
  Initial condition on \(y\) (can be a vector).

- `t` : array
  
  A sequence of time points for which to solve for \(y\). The initial value point should be the first element of this sequence.

- `args` : tuple
Look at examples/zombie.py

```python
def calc_rate(P=0, d=0.0001, B=0.0095, G=0.0001, A=0.0001, S0=500):
    def f(y, t):
        Si = y[0]
        Zi = y[1]
        Ri = y[2]
        # the model equations (see Munz et al. 2009)
        f0 = P - B*Si*Zi - d*Si
        f1 = B*Si*Zi + G*Ri - A*Si*Zi
        f2 = d*Si + A*Si*Zi - G*Ri
        return [f0, f1, f2]
    Z0 = 0  # initial zombie population
    R0 = 0  # initial death population
    y0 = [S0, Z0, R0]  # initial condition vector
    t = np.linspace(0, 5, 1000)  # time grid
    # solve the DEs
    soln = odeint(f, y0, t)
    S = soln[:, 0]
    Z = soln[:, 1]
    R = soln[:, 2]
    return t, S, Z
```

Yann - yannpaul@bu.edu (SCV)  Scientific Python  October 2012  36 / 76
Amazing interactive python.
Using the '-pylab' flag, you can get plotting for free

```python
In [1]: plot(range(10))
```
Amazing interactive python.
It provides “Magic” functions:

- cd - like unix change directory
- ls - list directory
- timeit - time execution of statement
- and so on ...
Amazing interactive python.
You can log an interactive session:

```
1 In [1]: logstart mylogfile.py
```

- Everything you type will be recorded to mylogfile.py
- `logon` and `logoff` turn it on and off
- `logstop` turns it off and saves file

To re-run it in ipython, use the `run` command:

```
1 In [1]: run -i mylogfile.py
```
Amazing interactive python.

- You can use `?` instead of `help()`.
  
  In `[1]: len? and Enter will print help
  In fact you get even more information than the standard `help()`

- Tab-completion

- Start typing a method, and "pop-up" help is shown
  You need a newer version than Katana’s, and
  you need to type `ipython qtconsole`
`multiprocessing` is a module that allows you to easily start other processes. They do **not** share memory, but `multiprocess` helps them communicate info/data.
multiprocessing.Pool creates a pool of processes that you give tasks to perform. See in

examples/mp_vowels_pool.py

```python
nproc = multiprocessing.cpu_count()
task_pool = multiprocessing.Pool(
    processes=nproc,
    initializer=start_proc)

words = "Mary had a little lamb".split()
```
map

task_pool.map takes a function and a list of arguments, and applies that function to each argument, in parallel.

24  map_results = task_pool.map(count_vowels, words)

The map method is blocking. The code progresses when all tasks are complete.
task_pool.apply_async takes a function and a sequence of arguments and applies that function to that argument.

```python
async_results = []
for word in words:
    async_results.append(
        task_pool.apply_async(count_vowels, (word,)))

task_pool.close()  # no more tasks
task_pool.join()   # wait here until all
```

This is not blocking
apply_async, getting results

\[
\text{task\_pool.map \ just returns a list of results}
\]

```python
for word, nvowels in zip(words, map_results):
    print "'{word:s}' has {nvowels:d} vowels".format(word=word,
                                                   nvowels=nvowels)
```

But \text{task\_pool.apply\_async} \ returns an object from which you can get results.

```python
for word, result in zip(words, async_results):
    print "'{word:s}' has {nvowels:d} vowels".format(word=word,
                                                   nvowels=result.get())
```
Outline for section 2

1. Scientific Python

2. Beyond the Basics
   - Super Scripting
   - More on Functions
   - String Formatting
   - Files
   - External Programs
   - Command line arguments

3. Solving Laplace’s Equation
pass is python’s way of saying “Keep moving, nothing to see here…”.
**pass**

- **pass** is python’s way of saying “Keep moving, nothing to see here...”.
- It’s used for yet unwritten function.
- If you try to call a function that doesn’t exist, you get an error.
pass

- **pass** is python’s way of saying “Keep moving, nothing to see here...”.
- It’s used for yet unwritten function.
- If you try to call a function that doesn’t exist, you get an error.
- **pass** just creates a function that does nothing.
- Great for planning work!

```python
1  def step1():
2      pass
3  step1() # no error!
4  step2() # error, darn!
```
None is Python’s formal value for “nothing”
- Use this as a default value for a variable,
- or as a return value when things don’t work, and you don’t want a catastrophic error.
None

- None is Python’s formal value for “nothing”
- Use this as a default value for a variable,
- or as a return value when things don’t work, and you don’t want a catastrophic error.
- Test if something is None, to see if you need to handle these special cases

```python
name = None
if name is None:
    name = 'Johnny'
```
Every module has a name, which is stored in `__name__`.
The script/console that you are running is called "__main__".
Every module has a name, which is stored in `__name__`.
The script/console that you are running is called "__main__".
Use this to make a file both a module and a script.

```python
# module greeting.py
def hey_there(name):
    print "Hi!", name, "... How's it going?"

hey_there('Joey')
if __name__ == '__main__':
    hey_there('Timmy')
```

`python greeting.py` → "Hi! Timmy ... How's it going?"
The Docstring

- This is an un-assigned string that is used for documentation.
- It can be at the top of a file, documenting a script or module
- or the first thing inside a function, documenting it.
The Docstring

- This is an un-assigned string that is used for documentation.
- It can be at the top of a file, documenting a script or module
- or the first thing inside a function, documenting it.
- This is what `help()` uses...

```python
def dot_product(v1, v2):
    """Perform the dot product of v1 and v2.

    'v1' is a three element vector.
    'v2' is a three element vector.
    ""
    sum = 0  #.....
```
Keyword Arguments

- Arguments that have default values.
- In the function signature, `keyword=default_value`
- `None` is a good default if you want to make a decision

```python
def hello(name='Joe', repeat=None):
    if repeat is None:
        repeat = len(name)
    print 'Hi!', name*repeat
```

Yann - yannpaul@bu.edu (SCV)
Returning Values

- A function can return multiple values.
- Formally this creates a **Tuple** - an *immutable* list.
A function can return multiple values.
Formally this creates a **Tuple** - an *immutable* list
You can collect the returned values as a Tuple, or as individual values.

```python
1 def pows(val):
2     return val, val*val, val*val*val
3 two, four, eight = pows(2)
4 ones = pows(1)
5 print ones[0], ones[1], ones[2]  # 1 1 1
```
Strings can be paired with values to control printing.

```python
>>> "Hi there {}!".format('Yann')
'Hi there Yann!'

>>> "Coords: {}, {};".format(-1, 2)
'Coords: -1, 2;

>>> "{} , the cost is {:.2f}".format(1125.747025, 'Yann')
'Yann, the cost is 1125.75'
```
Keyword arguments to format

- It’s sometimes tricky to format many values
- You can name some of the format targets

```python
email = ""
Subject: {subject}
Date: {mon:2d}/{day:2d}/{year:2d}
Message: {msg}
""

print email.format(mon=10, year=12, day=31,
subject='Happy Halloween',
msg='Booh')
```
Keyword arguments to format – result

>>> email = ""
... Subject: {subject}
... Date: {mon:2d}/{day:2d}/{year:2d}
... Message: {msg}
... 
""

>>> print email.format(mon=10, year=12, day=31,
... subject='Happy Halloween',
... msg='Booh')

Subject: Happy Halloween
Date: 10/31/12
Message: Booh

More features at:
http://docs.python.org/library/string.html
For reading and writing comma-separated-values
- `csv.reader` for reading, `csv.writer` for writing
- Dialects option correspond to predefined formats
- `'excel'` for excel output without needing to know the separator and quote characters

```python
reader = csv.reader(file)
for row in reader:
    # row is a list
writer = csv.writer(file)
writer.writerow([1, 2, 3])
```
subprocess – running external programs

```python
import subprocess
output = subprocess.call(['ls', '-1'])
print "output = ", output
```

- subprocess provides many tools
- The most basic is the `call` function.
- It takes a list that is joined and executed.
- `output` just holds the exit code (0 if successful)
- `check_output` is like `call` but 1) returns output and 2) causes an error if program fails
argparse – easy command line arguments

- argparse module is the easiest way.
- You first create a ArgumentParser object
- You define the allowable arguments with the add_argument function
- You can add required or optional arguments
- sys.argv is a list of arguments passed to the program/script.
- Pass this list to parse_args function to process and get an object with your parameters defined.
import sys
import argparse

parser = argparse.ArgumentParser(description='Plot zombie statistics')
parser.add_argument('prefix', help='the prefix for the plot filename')
parser.add_argument('-p', dest='pop', default=500, type=int,
                    help='Set the starting population')
parser.add_argument('-s', dest='show',
                    help='Show the figure', action='store_true')
parser.add_argument('city', help='Plot information for a specific city',
                    nargs='?', default=None)

args = sys.argv[1:]
params = parser.parse_args(args)

print "prefix = ", params.prefix
print "pop = ", params.pop
if params.city is not None:
    print "city = ", params.city
print "show? ",
if params.show:
    print "yes!"
else:
    print "no :-("
Solve Laplace’s Equation 1a

- Solve $\nabla^2 u = 0$
Solve Laplace’s Equation 1a

- Solve $\nabla^2 u = 0$
- Solve iteratively, each time changing $u$ with following equation:
  \[ u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n) \]
- Just an averages of the neighboring points.
Solve Laplace’s Equation 1a

- Solve $\nabla^2 u = 0$
- Solve iteratively, each time changing $u$ with following equation:

$$u_{j,l}^{n+1} = 1/4 (u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n)$$
- Just an averages of the neighboring points.

$$U_{j,l}^{n+1} = 1/4 \left( \right)$$
Solve Laplace’s Equation 1b

- Solve $\nabla^2 u = 0$
- Solve iteratively, each time changing $u$ with following equation:
  \[ u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n) \]
- Just an average of the neighboring points.

\[ U_{j,l}^{n+1} = \frac{1}{4} ( U_{j-1,l}^n + U_{j+1,l}^n + \ldots ) \]
Solve Laplace’s Equation 1c

- Solve $\nabla^2 u = 0$
- Solve iteratively, each time changing $u$ with following equation:
  
  $$u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n)$$
- Just an averages of the neighboring points.

$$U_{j,l}^{n+1} = \frac{1}{4} ( \ldots )$$
Solve Laplace’s Equation 1d

- Solve $\nabla^2 u = 0$
- Solve iteratively, each time changing $u$ with following equation:
  $$u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n)$$
  Just an average of the neighboring points.
- Repeat this calculation until $rms(u^{n+1} - u^n) < \epsilon$, some threshold
- Set some limit on total number of iterations

practice/laplace.py
First task:

def pure_python_step(u):
    ''' Pure python implementation of Gauss-Siedel method. Performs 1
    iteration.'''
    rms_err = 0
    # use for loop to loop through each element in u
    # temporarily store old value to later calculate rms_err
    # update current u using 4 point averaging
    # update running value of rms_err
    # when done looping complete rms_err calculation and return it
    return rms_err
x = np.array((1, 2, 3, 4, 5, 6))
x[2]
x[2:5]
x[2:-1]
x[::5]
x[:5]
x[:5:2] = 10
x
```python
>>> x = np.array((1, 2, 3, 4, 5, 6))
>>> x[2]
3
>>> x[2:5]
array([3, 4, 5])
>>> x[2:-1]
array([3, 4, 5])
>>> x[:5]
array([1, 2, 3, 4, 5])
>>> x[:5:2] = 10
>>> x
array([10, 2, 10, 4, 10, 6])
```
Array Copying

```python
>>> a = np.array([1,2,3,4])
>>> b = a
>>> b is a
True
>>> c = a.view()
>>> c.shape = 2,2
>>> c[1,1]=10
>>> c
array([[ 1,  2],
       [ 3, 10]])
>>> a
array([ 1,  2,  3, 10])
>>> d = a.copy()
```
Solve Laplace’s Equation 3a

\[ U_{j,l}^{n+1} = \frac{1}{4} ( \)
Solve Laplace’s Equation 3b

\[ U_{j,l}^{n+1} = \frac{1}{4} \left( U_{j-1,l}^n + U_{j+1,l}^n \right) + \ldots \]
Solve Laplace’s Equation 3c

\[ U_{j,l}^{n+1} = \frac{1}{4} (U_{j-1,l}^n + U_{j,l}^n + U_{j+1,l}^n) + \ldots \]
Solve Laplace’s Equation 3d

Second task:

```python
def numpy_step(u):
    '''Numpy based Jacobi’s method. Performs one iteration.'''
    # make a copy so that you can calculate the error
    u_old = u.copy()
    # use slicing to shift array
    # utmp = u[1:-1, 1:-1] makes a new array, so that utmp[0,0] is the same as u[1,1]
    # utmp = u[0:-2, 1:-1] makes a new array that leads to a shift of j-1
    # because utmp[0,0] is the same as u[0, 1]
    # use this concept to solve this equation in on line
    # u = 1/4*(u_{j-1,i} + u_{j+1,i} + u_{j, i-1} + u_{j, i+1})
    return calc_err(u, u_old)
```
Third task:

```python
def laplace_driver(u, stepper, maxit=100, err=1e3):
    '''Repeatedly call stepper(u) to solve nabla^2 u = 0 until
rms error < err or maxit number of iterations is reached.'''

    'u'  - a numpy array
    'stepper'  - a function whose sole argument is u

    rms_err = 0
    # take one step with stepper, to define initial rms_err
    # loop until rms_err < err
    # check to see that number of iterations is less than maxit
    # perform single iteration using stepper method
    # return rms_error
    return rms_err
```
def time_method(stepper):
    '''Time how long a particular stepper takes to solve an ideal problem.'''
    u = set_bc(np.zeros((100, 100)))
    start = time.time()
    err = laplace_driver(u, stepper)
    return time.time() - start

if __name__ == '__main__':
    pure_python_time = time_method(pure_python_step)
    numpy_time = time_method(numpy_step)
    print "Pure python method takes {:.3f} seconds".format(pure_python_time)
    print "Numpy method takes {:.3f} seconds".format(numpy_time)
Solve Laplace’s Results

1. Pure python method takes 3.624 seconds
2. Numpy method takes 0.016 seconds
Resources

- **SCV help site**
- **Numpy Tutorial**
- **Scipy tutorial**
- **Scientific Python Tutorial**
- **Another Scientific Python Tutorial**
- **iPython Tutorial**
What I Didn’t Cover

- Exception Handling
- Duck Typing
- Virtualenv
- Pandas
- mpi4py
- And many other useful things