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

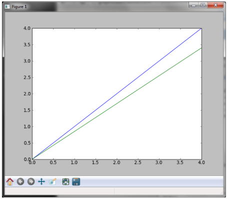
- This tutorial is for someone with basic python experience.
- First:
  1. matplotlib - plotting library, like Matlab
  2. numpy - fast arrays manipulation
  3. scipy - math methods galore.
  4. ipython - better interactive python
- Then I will cover a few intermediate details about python programming 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.

Plotting in scripts

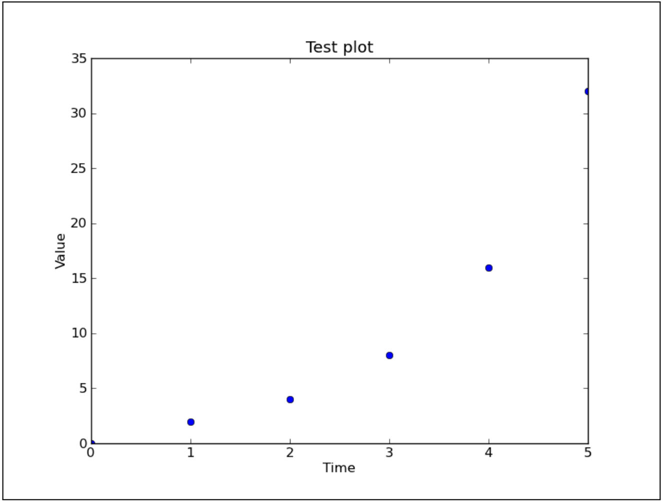
- `matplotlib` is a package that has many modules, `pyplot` is the main driver.
- `matplotlib` is designed for both interactive and script-based use.
- a `Figure` can contain many `Axes`, which contain many plots.
- `pyplot` creates a default `Figure` and `Axes` if you just want get to plotting things quickly.



Basic Example

```
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



World Population - Practice Part 1

Explaining what's there

practice/world_population.py

```
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
```

1. We read in the data

World Population - Practice Part 1

Explaining what's there

practice/world_population.py

```python
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
```

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

World Population - Practice Part 1

Explaining what's there

practice/world_population.py

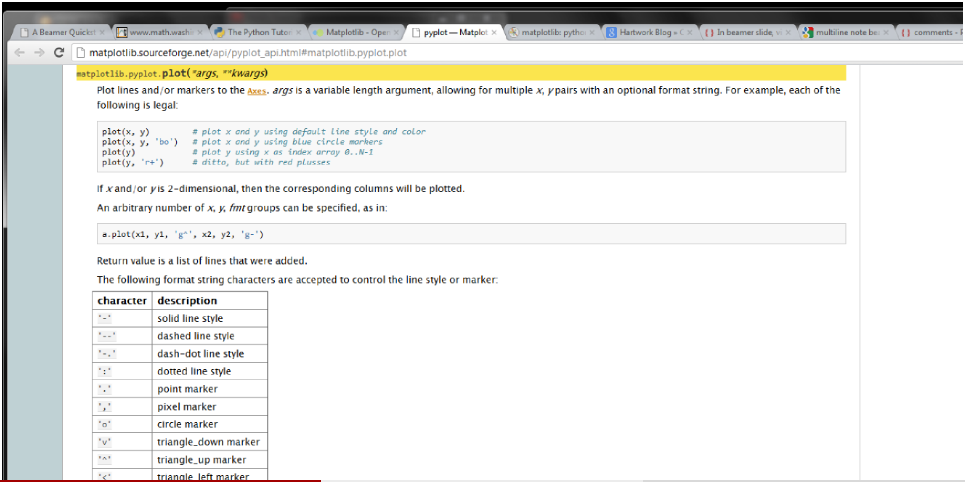
```python
def get_data():
    data = file("world_population.txt", "r").readlines()
    dates = []
    populations = []
    for point in data:
        date, population = point.split()
        dates.append(date)
        populations.append(population)
    return dates, populations
```

1. We read in the data
2. For each line ('point') we need to separate the date from population
3. `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”



matplotlib.pyplot.plot(*args, **kwargs)

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 and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

If *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:

```
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:

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker

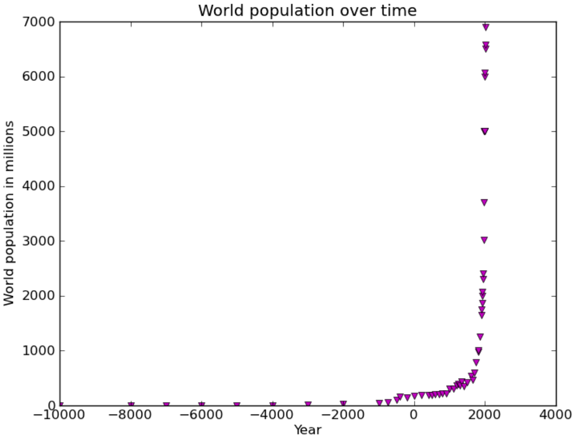
World Population - Practice Part 2

Your Assignment

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

dates, populations = get_data()
plot_world_pop(dates, populations)
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

World Population - Final



Life Expectancy - Practice Part 1

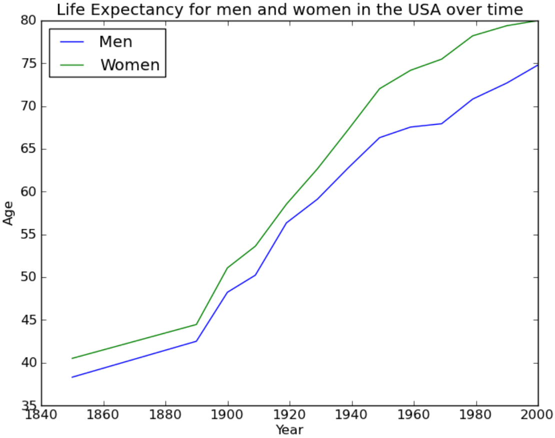
practice/life_expectacies_usa.py

```
from matplotlib import pyplot

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

def plot_expectancies(dates, men, women):
    pass

dates, men, women = get_data()
plot_expectancies(dates, men, women)
pyplot.show()
```

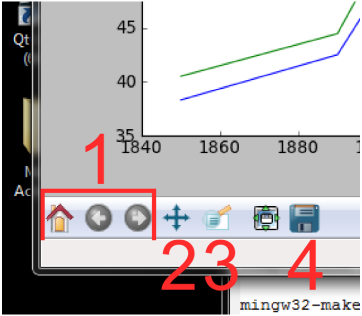
Life Expectancy - Practice Part 2

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

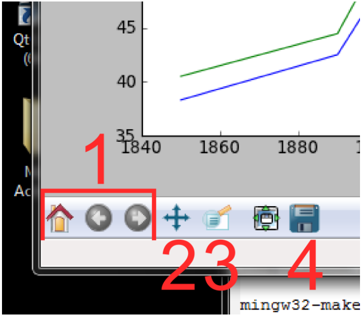


Plotting Interactively



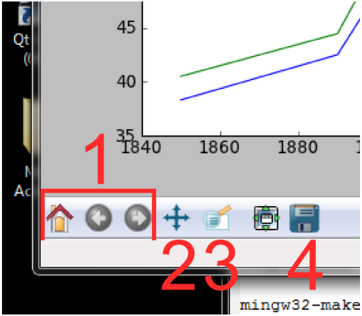
1. **Home** - **Backward** - **Foward** - Control edit history

Plotting Interactively



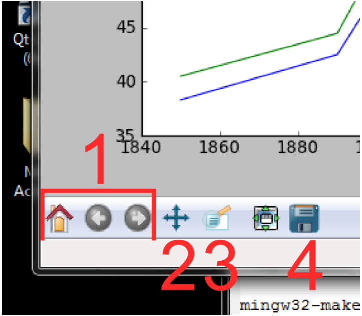
1. **Home** - **Backward** - **Foward** - Control edit history
2. **Pan** - **Zoom** - Left click + drag shifts center, right click + drag changes zoom

Plotting Interactively



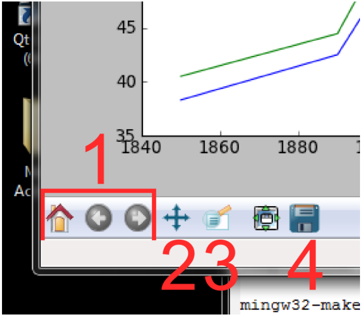
1. **Home** - **Backward** - **Foward** - Control edit history
2. **Pan** - **Zoom** - Left click + drag shifts center, right click + drag changes zoom
3. **Zoom** - Select zoom region

Plotting Interactively



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

Plotting Interactively



1. **Home - Backward - Foward** - 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.

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

Array Creation

A 2D array:

```
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

Array Creation

- `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

Array Basics - Results

```
>>> 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

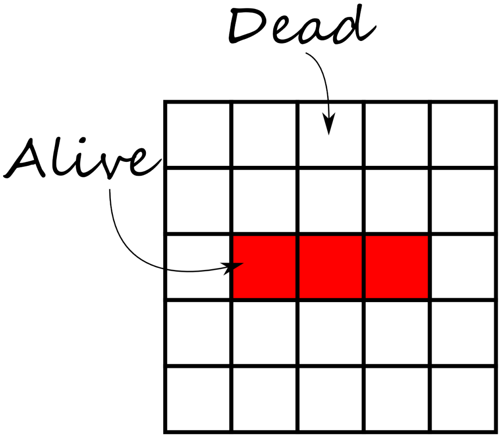
```
x = np.array([0, np.pi/4, np.pi/2])
np.sin(x)
np.dot([2, 2, 2], [2, 2, 2])
```

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

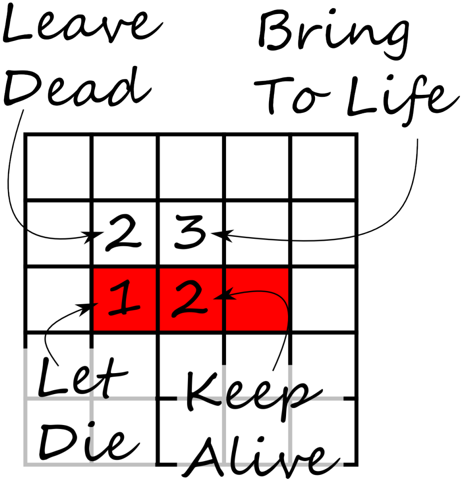


Calculating Neighbors

Neighbors
Number

0	0	0	0	0
1	2	3	2	1
1	1	2	1	1
1	2	3	2	1
0	0	0	0	0

Use the Neighbors to Update



A Practice Problem

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:

```
a = np.array(((1,2),(3,4),(5,6))) # make a 3x2
print a.shape[0] # prints 3
print a.shape[1] # prints 2
```

Conway's Game of Life

`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

`np.where(b ==0)` returns a series of indicies where it's true.
`np.where(b ==0, -1, b)` to replace 0's with -1's.

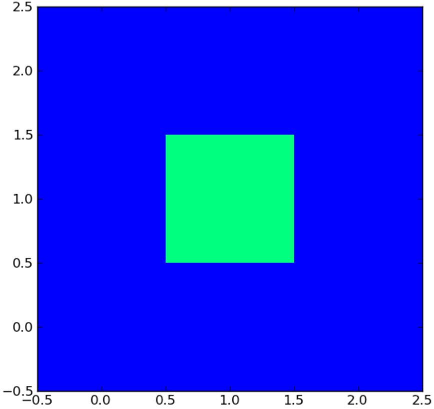
```
>>> b
array([[ 0.,  0.,  0.],
       [ 2.,  2.,  2.],
       [ 0.,  3.,  0.]])
>>> np.where(b == 0)
(array([0, 0, 0, 2, 2]), array([0, 1, 2, 0, 2]))
>>> np.where(b == 0, -1, b)
array([[-1., -1., -1.],
       [ 2.,  2.,  2.],
       [-1.,  3., -1.]])
```

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

Matplotlib Imshow

```python
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')
```

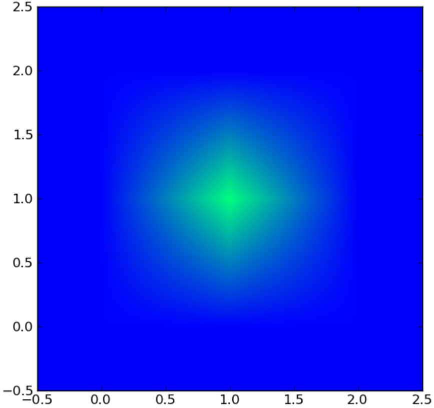
Nearest Neighbors



Matplotlib Imshow

```
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

http://docs.scipy.org/doc/scipy/reference/tutorial/index.html

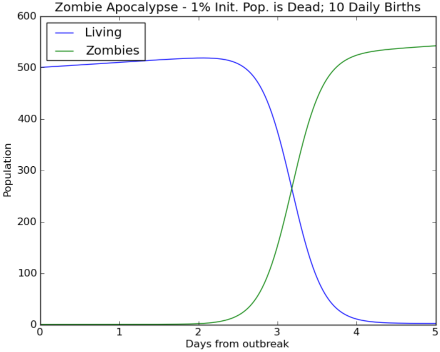
- Numpy
- Integration
- Optimization
- Interpolation
- FFT
- Linear Algebra
- Statistics
- And much more....

Constants

```
>>> 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")[:3]
>>> for val in masses:
...     print "'{}' 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



Zombie Apocalypse - ODEINT

enerated/scipy.integrate.odeint.html

on » SciPy v0.11.dev Reference Guide (DRAFT) » Integration and ODEs (`scipy.integrate`) » previous | next | modules | modules | index

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*) [source]

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:

```
dy/dt = func(y,t0,...)
```

where y can be a vector.

Parameters :

func : callable(y, t0, ...)
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

Zombie Apocalypse - ODEINT

Look at `examples\zombie.py`

```
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
```

ipython - plotting

Amazing interactive python.

Using the '-pylab' flag, you can get plotting for free

```
% ipython -pylab
...
In [1]: plot(range(10))
```

ipython - Magic Functions

Amazing interactive python.
It provids “Magic” functions:

- `cd` - like unix change directory
- `ls` - list directory
- `timeit` - time execution of statement
- and so on ...

ipython - Logging

Amazing interactive python.
You can log an interactive session:

```
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, us the **run** command:

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

ipython - Misc.

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`

Simple Parallelized Python

`multiprocessing` is a module that allows you to easily start other processes. They do **not** share memory, but multiprocess helps them communicate info/data.

Pool

`multiprocessing.Pool` creates a pool of processes that you give tasks to perform. See in

examples/mp_vowels_pool.py

```
    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.

```
    map_results = task_pool.map(count_vowels, words)
```

The `map` method is blocking. The code progresses when all tasks are complete.

apply_async

`task_pool.apply_async` takes a function and a sequence of arguments and applies that function to that argument.

```
    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

`task_pool.map` just returns a list of results

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

But `task_pool.apply_async` returns an object from which you can `get` results.

```
    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`

- `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!

```
def step1():
    pass
step1() # no error!
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

```
name=None
if name is None:
    name = 'Johnny'
```

"__main__"

- Every module has a name, which is stored in `__name__`
- The script/console that you are running is called `"__main__"`.

"__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...

```
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

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

Returning Values

- A function can return multiple values.
- Formally this creates a **Tuple** - an *immutable* list

Returning Values

- 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.

```
def pows(val):
    return val, val*val, val*val*val
two, four, eight = pows(2)
ones = pows(1)
print ones[0], ones[1], ones[2] # 1 1 1
```

‘‘string’’.format()

- Strings can be paired with values to control printing.

```
>>> "Hi there {}!".format('Yann')
'Hi there Yann!'
>>> "Coords: {}, {};".format(-1, 2)
'Coords: -1, 2;'
>>> "{1}, the cost is {0:.2f}".format(1125.747025, 'Y
'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

```
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

csv reader and writer

- 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

```
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

```
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.

argparse – example

Look over `examples\argparse_ex.py`

```
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 startring 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 $\nabla^2 u = 0$

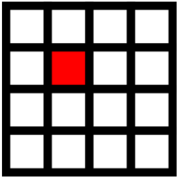
Solve Laplace's Equation 1a

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

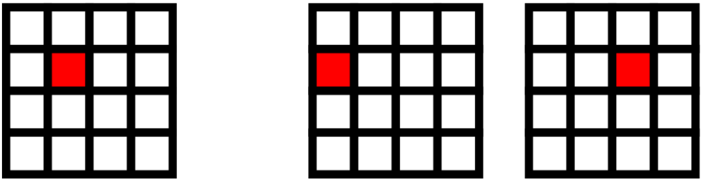
$$U^{n+1}_{j,l} = 1/4\ ($$



Solve Laplace's Equation 1b

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

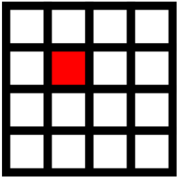
$$U_{j,l}^{n+1} = 1/4 \left(U_{j-1,l}^{n} + U_{j+1,l}^{n} + \ldots \right.$$

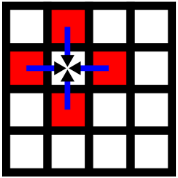


Solve Laplace's Equation 1c

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

$$U_{j,l}^{n+1} = 1/4 \left(\quad \ldots \quad \right)$$





Solve Laplace's Equation 1d

- Solve $\nabla^2 u = 0$
- Solve iteratively, each time changing u with following equation:
  $u_{j,l}^{n+1} = 1/4(u_{j+1,l}^{n} + u_{j-1,l}^{n+1} + u_{j,l+1}^{n} + u_{j,l-1}^{n+1})$
- 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`

Solve Laplace's Equation 2

First task:

```
def pure_python_step(u):
    '''Pure python implementation of Gauss-Siedel method. Performs
    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
```

Slicing - 1

```
x = np.array((1, 2, 3, 4, 5, 6))
x[2]
x[2:5]
x[2:-1]
x[:5]
x[:5:2] = 10
x
```

Slicing - 2

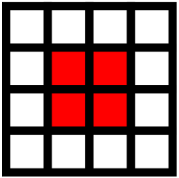
```
>>> x = np.array((1, 2, 3, 4, 5, 6))
>>> x[2]
3
>>> x[2:5]
array([3, 4, 5])
>>> x[2:-1]
array([3, 4, 5])
>>> x[:5]
array([1, 2, 3, 4, 5])
>>> x[:5:2] = 10
>>> x
array([10,  2, 10,  4, 10,  6])
```

Array Copying

```
>>> 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()
```

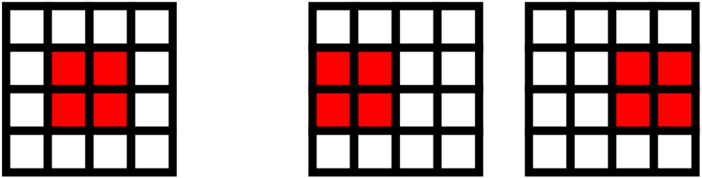
Solve Laplace's Equation 3a

$$U_{j,l}^{n+1} = 1/4\ ($$



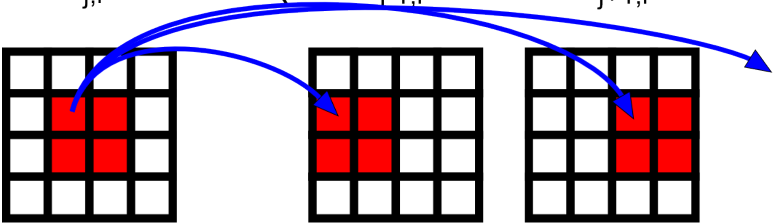
Solve Laplace's Equation 3b

$$U_{j,l}^{n+1} = 1/4\ (\ U_{j-1,l}^{n} + U_{j+1,l}^{n} + \ldots$$



Solve Laplace's Equation 3c

$$U^{n+1}_{j,l} = 1/4\ (\quad U^{n}_{j-1,l} \quad + \quad U^{n}_{j+1,l} \quad + \ \ldots .$$



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
    # as u[1,1]
    # then
    # utmp = u[0:-2, 1:-1] makes a new array that leads to a shift
    # 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)
```

Solve Laplace's Equation 4a

Third task:

```
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 max
    #     perform single iteration using stepper method
    # return rms_error
    return rms_err
```

Solve Laplace's Equation 4b

```python
def time_method(stepper):
    '''Time how long a particular stepper takes to solve an ideal
    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_py
    print "Numpy method takes {:.3f} seconds".format(numpy_time)
```

Solve Laplace's Results

```
Pure python method takes 3.624 seconds
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