An introduction to Python for Scientific Computation

By Edward Smith

17th March 2017

Aims for today

- Using Python to read files and plot.
- Designing a basic Graphical User Interface.
- Unit testing frameworks and version control.
- Running parameter studies by calling executables repeatedly with subprocess.
- Other libraries and how to wrap your own code from Fortran, C++, etc

Overview

- Review of the last two weeks, testing, modules and version control. Introduction to figures for publication and curve fitting (~30mins)
- Hands on session + break (~30 min)
- Classes, objects and inheritance. Overview of other Python modules (~30 min)
- Hands on Session + break (~30 min)

What we have covered so far

- How to use the command prompt and run scripts.py
- A range of data types and mentioned that everything is an object.

```
a = 3.141592653589          # Float
i = 3                       # Integer
s = "some string"           # String
l = [1,2,3]                 # List, note square brackets tuple if ()
d = {"red":4, "blue":5}     # Dictonary
x = np.array([1,2,3])       # Numpy array
```

- Show how to use them in other constructs including conditionals (if statements) iterators (for loops) and functions (def name)
- Introduce external libraries numpy and matplotlib for scientific computing

Running Python

- Open up a command prompt in python (or ipython for more debugging info) and use like a calculator

```
a = 3.141592653589          # Float
i = 3                       # Integer
b = a*i**2                  # Some calculations
```

- Write a script in notepad, save with extension *.py and run using python or ipython from a terminal (opened with cmd in windows). Note you need to be in the same directory, check directory with pwd and change with cd command.

```
cd ./path/to/location/of/save/script/

python script.py

ipython script.py -i     (For interactive ipython session on
error)
```

Strings

- String manipulations

```
s = "some string"
t = s + " with more"     Out: "some string with more"
s*3   Out: "some stringsome stringsome string"
s[3]                            Out: e
s[0:4]                          Out: some
s.title()                       Out: 'Some String'
s.capitalize()                  Out: "Some string"
s.split(" ")                    Out: ["some", "string"]
s.find("o")                     Out: 1
t = s.replace("some", "a")      Out: t="a string"
```

- In ipython, use tab to check what functions (methods) are avaliable

Lists and iterators

- A way of storing data. We can make lists of any type

```
l = [1,2,3,4]

m = ["another string", 3, 3.141592653589793, [5,6]]
```

- Iterators – loop through the contents of a list

```
for item in m:
    print(type(item), " with value ", item)
OUT: (<type 'str'>, ' with value ', 'another string')
     (<type 'int'>, ' with value ', 3)
     (<type 'float'>, ' with value ', 3.141592653589793)
     (<type 'list'>, ' with value ', [5, 6])
```

- To add one to every element we could use

```
for i in range(len(l)):

    l[i] = l[i] + 1
```

Note: will not work:

```
for i in l:
    i = i + 1
```

List comprehension

l = [i+1 for i in l]

Dictionaries

- Dictonaries for more complex data storage

item

```
d = {"strings" : ["red", "blue"],
     "integers": 6,
     "floats": [5.0, 7.5]}
```

key Value

```
e.items()

e.keys()
e.values()
```

- Access elements using strings

```
d["strings"]     out: ["red", "blue"]
```

- Elements can also be accessed using key iterators

```
for key in d:
    print(key, d[key])
```

Conditionals

- Allow logical tests

```
#Example of an if statement
if a > b:
    print(a)
else:
    print(a, b)

if type(a) is int:
    a = a + b
else:
    print("Error – a is type ", type(a))
```

Logical test to determine which branch of the code is run

```
if a < b:
    out = a
elif a == b:
    c = a * b
    out = c
else:
    out = b
```

Indent determine scope
4 spaces here

Numpy arrays

- Numpy – The basis for all other numerical packages to allow arrays instead of lists (implemented in c so more efficient)

```
import numpy as np
x = np.array([[1,2,3],[4,5,6],[7,8,9]])
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

x = x + 1
array([[ 2,  3,  4],
       [ 5,  6,  7],
       [ 8,  9, 10]])
```

Import module numpy and name np
Similar to:
- c++ #include
- Fortran use
- R source()
- java import (I think...)
- MATLAB adding code to path

A Plot of Two Axes with Labels

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2*np.pi, 20)
y = np.sin(x)
z = np.cos(x)
fig, ax = plt.subplots(2,1)
ax[0].plot(x, y, lw=3., c='r')
ax[1].plot(x, z, '--bs', alpha=0.5)
ax[1].set_xlabel("$x$", fontsize=24)
ax[0].set_ylabel("$\sin(x)$", fontsize=24)
ax[1].set_ylabel("$\cos(x)$", fontsize=24)
plt.show()
```

2 Plots, returns fig handle and list of axes handles

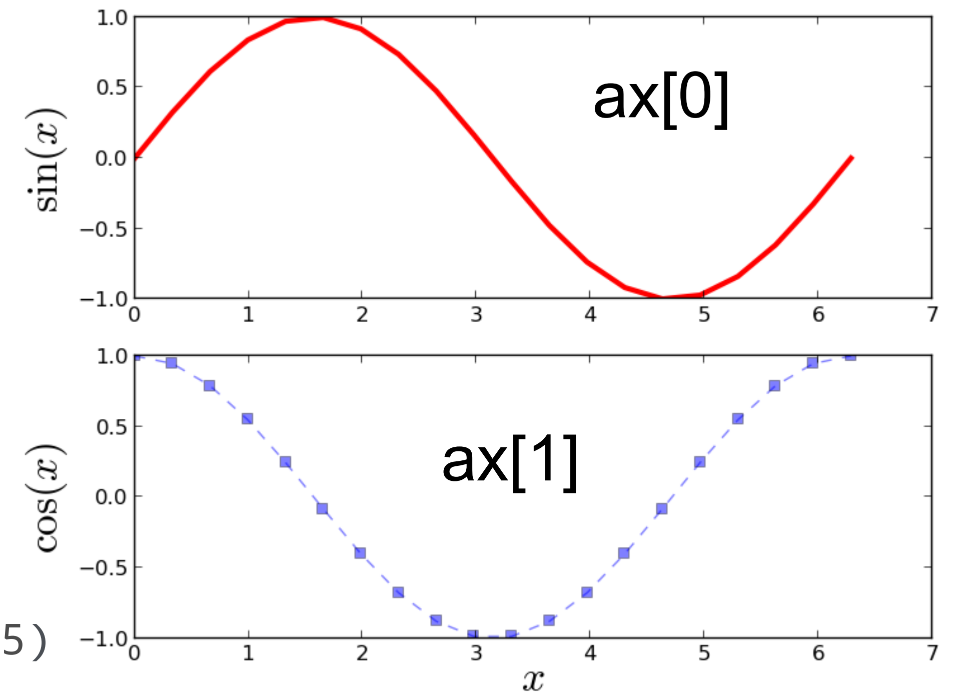
line width and colour arguments

MATLAB syntax and transparency argument alpha

Latex syntax due to dollar signs

A Plot of Two Axes with Labels

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2*np.pi, 20)
y = np.sin(x)
z = np.cos(x)
fig, ax = plt.subplots(2,1)
ax[0].plot(x, y, lw=3., c='r')
ax[1].plot(x, z, '--bs', alpha=0.5)
ax[1].set_xlabel("$x$", fontsize=24)
ax[0].set_ylabel("$\sin(x)$", fontsize=24)
ax[1].set_ylabel("$\cos(x)$", fontsize=24)
plt.show()
```



Functions

```python
#Define a variable
a = 5.0
```

Comment

Function name

Tell Python you are defining a function

Name of input variable to the function

```python
#Define Function

def square(input):
    "calculate square"
    output = input*input
    return output
```

Document function here "text" for one line or """ multi-line verbose and descriptive text """

Level of indent determines what is inside the function definition. Variables defined (scope) exists only inside function. Ideally 4 spaces and avoid tabs. See PEP 8

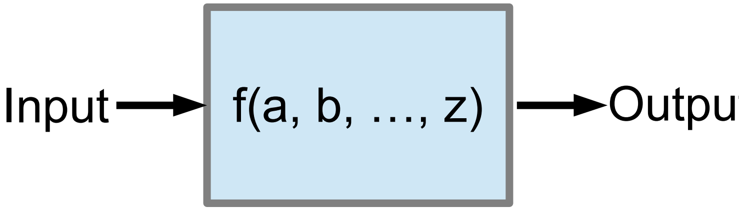
Operation on input variable

Value to return from function

```python
#We call the function like this
square(a) Out: 25.0
```

Examples of Functions

- take some inputs
- perform some operation
- return outputs




```
def divide(a, b):
    output = a/b
    return output
```

```
def do_nothing(a, b):
    a+b
```

```
def get_27():
    return 27

#Call using
get_27()
```

```
def redundant(a, b):
    return b
```

```
def line(m, x, c=3):
    y = m*x + c
    return y
```

Optional variable. Given a value if not specified

```
def quadratic(a, b, c):
    "Solve: y = ax² + bx + c"
    D = b**2 + 4*a*c
    sol1 = (-b + D**0.5)/(2*a)
    sol2 = (-b – D**0.5)/(2*a)
    return sol1, sol2
```

Curve Fitting with Scipy

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

x = np.linspace(0, 4., 30)
y = x + (2.*(np.random.random(30)-.5))
plt.plot(x, y, 'ob')

def linear(x, m, c):
    "Define line function"
    return m*x + c

params, cov = curve_fit(linear, x, y)
yf = linear(x, params[0], params[1])
plt.plot(x, yf, 'r-')

plt.show()
```

Function from scipy. Takes function handle for the fit you want with x and y data. It returns fit parameters (here m and c) as a list with 2 elements and the covariance (for goodness of fits, etc)

We use params (m and c) with the linear function to plot the fit

A GUI with a Slider

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.widgets as mw

#Setup initial plot of sine function
x = np.linspace(0, 2*np.pi, 200)
l, = plt.plot(x, np.sin(x))

#Adjust figure to make room for slider
plt.subplots_adjust(bottom=0.15)
axslide = plt.axes([0.15, 0.05, 0.75, 0.03])
s = mw.Slider(axslide, 'A value', 0., 5.)

#Define function
def update(A):
    l.set_ydata(np.sin(A*x))
    plt.draw()

#Bind update function to change in slider
s.on_changed(update)
plt.show()
```

Adjust figure to make room for the slider and add a new axis axslide for the slider to go on

Define a function to change figure based on slider value. Here this updates the plot data and redraws the plot

Bind function update to slider change

Functional Interface

- Functions are like a contract with the user, here we take in the file name and return the data from the file

```
#Iterate through the files
for f in files:
    #read the data
    data = read_file(f)
```

TAKES A FILENAME AND RETURNS ITS CONTENTS

- We aim to design them so for a given input we get back the expected output

```
def square(a):
    return a**2
```

TAKES A NUMBER AND RETURNS ITS SQUARE

Unit Testing in Python

```
import unittest

def square(a):

    pass

def cube(a):

    pass

class test_square(unittest.TestCase):

    def test_square(self):

        assert square(2.) == 4.

    def test_cube(self):

        assert cube(2.) == 8.

unittest.main()
```

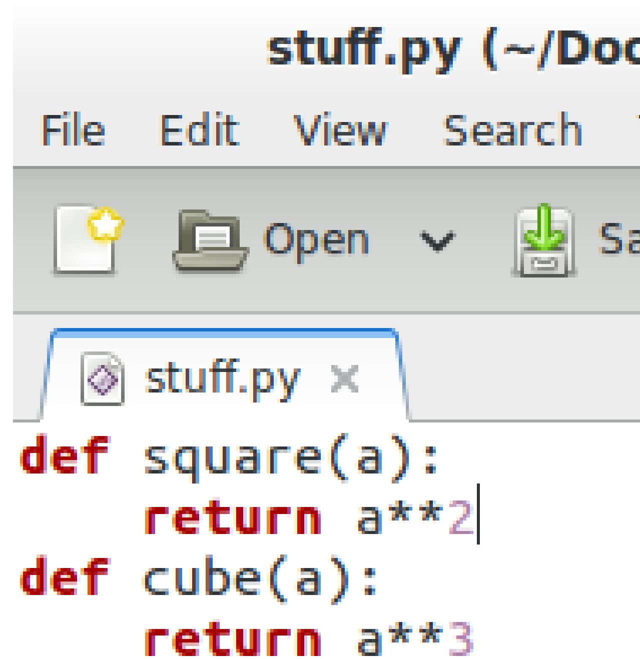
Function initial empty and written to satisfy required functionality

Assert raises an error if the logical statement is not true

Making a Module

- Simply copy code to a new file, for example `stuff.py`. Any script or Python session running in the same folder can import this,

```
import stuff

stuff.square(4.0)

stuff.cube(4.0)
```



- Module code should be functions and classes ONLY. Code to test/run can be included using the following:

```
if __name__ == "__main__":

    print(square(2.0), cube(2.0))

    unittest.main()
```

Version Control

- Once you have some code, put it into a code repository
  - Backup in case you lose you computer
  - Access to code from home, work and anywhere else.
  - Allows you to keep a clear history of code changes
  - Only reasonable option when working together on a code
- Three main repositories are git, mercurial and subversion.
- Most common is git, a steep learing curve and helps the maintainer more than the developer (in my opinion). Mercurial may be better... Subversion is often disregarded due to centralised model.
- Range of free services for hosting, including https://github.com/ (your code has to be open-source, although there is talk of an Imperial subscription) or http://bitbucket.com/ (allows 3 private repositories)

Version Control

- Once you have some code, put it into a code repository
  - git clone http://www.github/repo/loc ./out Clone directory to out
  - git log Check history of commits
  - git diff Check changes made by user since last
  - git pull Get latest changes from origin (fetch+merge)
  - git add Add changes to staging area
  - git commit -m ”Log message” Commit changes with message
  - git push Push changes to origin
  - git branch Create a branch of the code
- Takes a bit of getting used to but even basic usage will greatly improve your code and keep all versions up to date

Automated Testing

- Travis CI – you write a script and your tests are run automatically. If your latest change breaks the test, you will get an email

```
os: linux
language: python
python:
    - 2.7
before_install:
    - sh ./make/travis/travis-install-numpy-matplotlib.sh
install:
    - make
script:
    - make test-all
after_success:
     - echo "Success"
```

- Or setup your own local solution using Python scripts
- **Start your test suite now!** It will improve your software development

Hands on session 1 – Tutors

- Isaac and Edu

- Ask the person next to you – there is a wide range of programming experience in this room and things are only obvious if you've done them before!

Hands-On Session 1

1) Use test driven development (i.e. write the tests first) to design functions to give the square, cube and an arbitary power N for a number a.
2) Save these functions to number.py, put the tests inside an if statement: if __name__ == "__main__". In a new script/session import with import number as num and try to call the functions.
3) Create x=np.linspace(0., 100.,1000) and plot x^2 and x^3 using functions from the number module on seperate axes using `plt.subplots(2,1)`. Label the x and y axis, change line colour, markers and width.
4) Run the slider example and adapt to plot $\sin(Ax^2)$ using function from number, num.square, with the value of A specified by the slider value.
5) Fit an appropriate line to

```
x = np.linspace(0, 2*np.pi, 100)
y = y = np.sin(x) + (2.*(np.random.random(100)-.5))
```

<u>Advanced</u>

6) Develop a slider example with both sine and cosine on the plot updated by slider. Adapt this to add a new slider for a second coefficient B for cos(Bx).

Classes in Python

- A number class which includes methods to get square and cube

```
class Number():
    def __init__(self, a):
        self.a = a
    def square(self):
        return self.a**2
    def cube(self):
        return self.a**3
n = Number(4.5)
n.square()        #Out: 20.25
n.cube()          #Out: 91.125
```

Python provides the following syntax for a constructor, a function which MUST be called when creating an instance of a class
Called automatically when we instantiate

Classes in Python

- A person class could include name, age and method say their name

```
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_name(self):
        print("Hello, I'm "
              + self.name)
```

Python provides the following syntax for a constructor, a function which MUST be called when creating an instance of a class

Classes in Python

- A person class could include name, age and method say their name

```
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_name(self):
        print("Hello, I'm "
              + self.name)
bob_jones = Person('Bob Jones', 24)
jane_bones = Person('Jane Bones', 32)
bob_jones.say_name()
jane_bones.say_name()
```

Python provides the following syntax for a constructor, a function which MUST be called when creating an instance of a class

Called automatically when we instantiate

Classes for Postprocessing

- We can use classes with the data reading functions

```
class postproc():
    def __init__(self, foldername, headername, filename):
        self.foldername = foldername
        self.headername = headername
        self.filename = filename
    def get_header(self):
        f = open(self.foldername+self.headername)
        ...
    def get_files(self):
        ...
```

Classes for Postprocessing

- We can use classes with the data reading functions

```
class postproc():
    def __init__(self, foldername, headername, filename):
        self.foldername = foldername
        self.headername = headername
        self.filename = filename
        self.header = self.read_header()
        self.files = self.get_files()
    def get_header(self):
        f = open(self.foldername+self.headername)
        ...
    def get_files(self):
```

Classes for Postprocessing

- We can then use this as follows to get and plot data

```
pp = postproc(foldername='./binary/',
              headername='header',
              filename = 'filename')
for f in pp.files:
    data = pp.get_file(f)
    field = data.reshape(pp.header['Nx'],pp.header['Nz'])
    plt.imshow(field)
    plt.colorbar()
    plt.show()
```

Classes in Python

- A person can train in a particular area and gain specialist skills

```
class Person():
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_name(self):
        print("Hello, I'm "
              + self.name)
```

```
class Scientist(Person):
    def do_science(self):
        print(self.name +
              'is researching')
```

```
class Artist(Person):
    def do_art(self):
        print(self.name +
              'is painting')
```

Classes in Python

- A person can train in a particular area and gain specialist skills

```
class Person():

    def __init__(self, name, age):

        self.name = name

        self.age = age

    def say_name(self):

        print("Hello, I'm "

               + self.name)
```

```
class Scientist(Person):
    def do_science(self):
        print(self.name +
              'is researching')
```

```
class Artist(Person):
    def do_art(self):
        print(self.name +
              'is painting')
```

```
bob_jones = Scientist('Bob Jones', 24)

jane_bones = Artist('Jane Bones', 32)

bob_jones.do_science()

jane_bones.do_art()
```

A Hierarchy of Classes for Postprocessing

```
class postproc():
        …
    def read_file(self, filename):
        raise NotImplemented
        …
```

The base class defines the constructor, get_files, etc but does not specify how to read_files as this is unique to each data type

A Hierarchy of Classes for Postprocessing

```
class postproc():
        …
    def read_file(self, filename):
        raise NotImplemented
        …
```

The base class defines the constructor, get_files, etc but does not specify how to read_files as this is unique to each data type

```
#Binary IS A type of postproc reader
class postproc_binary(postproc):
    def read_file(self, filename):
        return np.fromfile(open(filename,'rb'), dtype='d')
class postproc_column(postproc):
    def read_file(self, filename):
        return np.genfromtxt(filename)
```

Inherit and only need to define read_file to customise for each data type

A Hierarchy of Classes for Postprocessing

```
class postproc_text(postproc):
    def read_header(self):
        f = open(self.foldername + self.headername)
        filestr = f.read()
        indx = filestr.find("FoamFile")
        header = {}
        for l in filestr[indx:].split("\n")[2:]:
            if l is "}":
                break
            key, value = l.strip(";").split()
            #As before...
    def read_file(self, filename):
        f = open(filename)
        filestr = f.read()
        indx = filestr.find("internalField")
        return np.array(filestr[indx:].split("\n")[3:-3], dtype="d")
```

Text is a little more complex.... We need to redefine read_header as well

A Hierarchy of Classes for Postprocessing

- We can now plot any format of data

```
import postproclib as ppl

ds= "binary"

if ds is "text":

    pp = ppl.postproc_text(ds+'/', 'filename00000', 'filename')

elif ds is "column":

    pp = ppl.postproc_column(ds+'/', 'header', 'filename')

elif ds is "binary":

    pp = ppl.postproc_binary(ds+'/', 'header', 'filename')

print("Datasource is " + ds)

for i in range(pp.get_Nrecs()):

    f = pp.read_field(i)

    plt.imshow(f)

    plt.colorbar()

    plt.show()
```

Interface is the same for all objects so the plot code does not need to be changed

A GUI using Postproc Library with a Slider

```
import matplotlib.pyplot as plt
from matplotlib.widgets import
Slider
import postprocmod as ppl
#function which loads new
#record based on input
def update(i):
    print("record = ", int(i))
    field = pp.read_field(int(i))
    cm.set_array(field.ravel())
    plt.draw()
#Get postproc object and plot
initrec = 0
pp = ppl.postproc_binary(
'./binary/','header', 'filename')
field = pp.read_field(initrec)
cm = plt.pcolormesh(field)
plt.axis("tight")
```

```
#Adjust figure to make room
for slider and add an axis
plt.subplots_adjust(bottom=0.2)
axslide = plt.axes(
    [0.15, 0.1, 0.75, 0.03])

#Bind update function
#to change in slider
s = Slider(axslide, 'Record',
       0, pp.get_Nrecs()-0.1,
       valinit=initrec)
s.on_changed(update)
plt.show()
```

Other libraries

- Graphical User Interfaces (GUI) e.g. Tkinter, wxpython, pyGTK, pyQT
- Multi-threading and parallel e.g. Subprocess, threading, mpi4py
- Image and video manipulation e.g. mayavi, pyCV, PIL, blender plugin
- Machine learning e.g. Scikit-learn, Pybrain
- Build system e.g. scons, make using os/system
- Differential equations solvers e.g. FEniCS, Firedrake
- Databasing and file storage e.g. pickle, h5py, pysqlite, vtk
- Web and networking e.g. HTTPLib2, twisted, django, flask
- Web scraping – e.g. scrapy, beautiful soup
- Any many others, e.g. PyGame, maps, audio, cryptography, etc, etc
- Wrappers/Glue for accelerated code e.g. HOOMD, PyFR (CUDA)
- It is also possible to roll your own using ctype or f90wrap

Save Data in Python's own format using pickle

- Import and save data from Python in any python format

```
import pickle
a = 4.
s = "test"
l = [2,3,4]
d = {"stuff":2}
pickle.dump([a, s, l, d],open('./out.p','w'))
```

- Then in a different script or session of Python we can load any of these types in the right format

```
import pickle
a, s, l, d = pickle.load(open('./out.p','r'))
```

Reading files in other popular formats HDF5 or vtk

- Reading open-source HDF5 format (large binary data, self documenting) using python package h5py

```
import h5py
f = h5py.File(fpath,'r')
data = f[u'data'].items()[0][1]
```

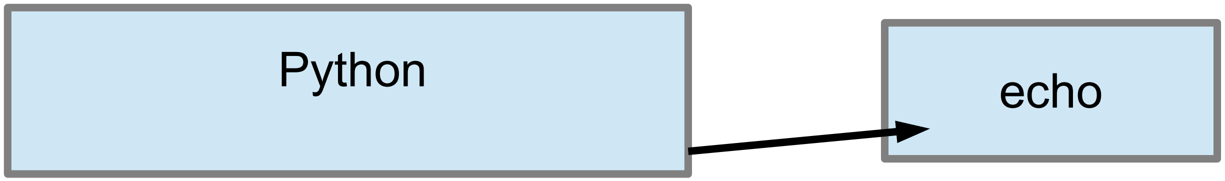
- Another common format is vtk, open-source for 3D graphics visualization but I've had limited success reading: packages like vtk, pyvtk, mayavi/TVTK,

```
import vtk
reader = vtk.vtkUnstructuredGridReader()
reader.SetFileName(filename)
reader.ReadAllVectorsOn()
reader.ReadAllScalarsOn()
reader.Update()
```

Running Jobs Using Subprocess

- Introduction to syntax to run an executable from within Python, in this example echo

```
import subprocess
# Call any executable
sp = subprocess.Popen("echo out",
                      shell = True)
```



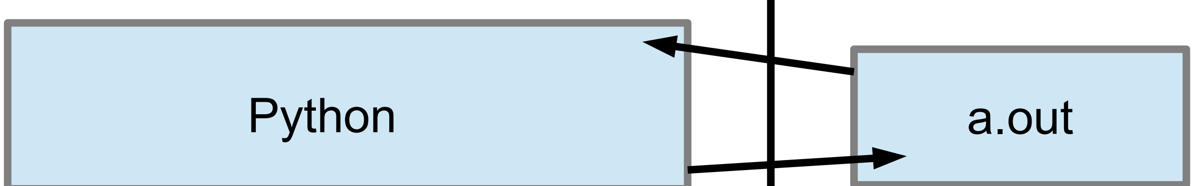
Call external command

Running Jobs Using Subprocess

- Introduction to syntax to run an executable written in c++ and compiled

```python
from subprocess import Popen, PIPE
# Call C++ executable ./a.out
sp = Popen(['./a.out'], shell=True,
          stdout=PIPE, stdin=PIPE)
# test value of 5
value = 5
# Pass to program by standard in
sp.stdin.write(str(value) + '\n')
sp.stdin.flush()
# Get result back from standard out
result = sp.stdout.readline().strip()
print(result)
```

```cpp
//test.cpp code to add
//1 to input and print

#include <iostream>
using namespace std;

int add_one(int i)
{
    return i+1;
}

int main () {

    int i;
    cin >> i;
    i = add_one(i);
    cout << i << "\n";
}
```



a.out compiled using:
g++ test.cpp

Running Jobs Using Subprocess

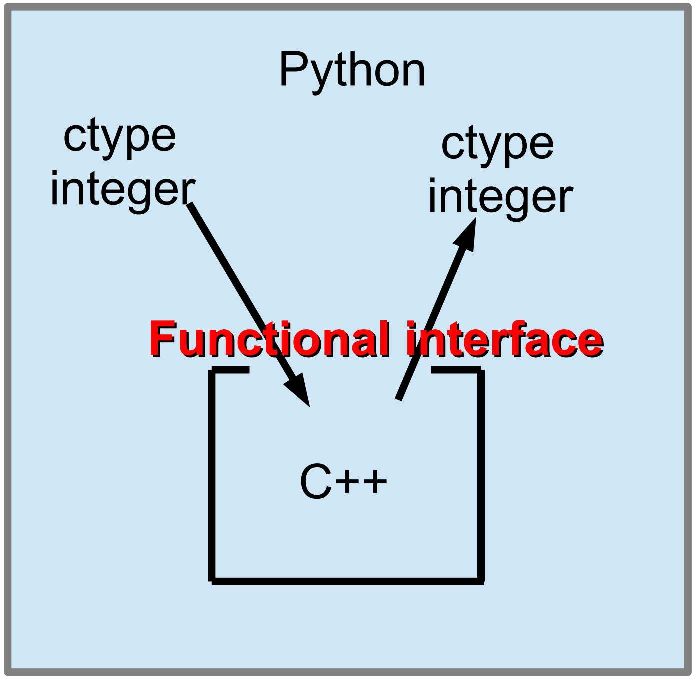
- Use with class to wrap the build, setup, run and postprocess jobs

```
class Run():
    def __init__(self, rundir, srcdir):
        self.rd=rundir
        self.sd=srcdir
    def setup(self):
        #Create a copy of source code and compile
        shutil.copytree(self.sd, self.rd)
        subprocess.Popen("g++ " + self.rd + "test.cpp")
    def run(self):
        self.sp = subprocess.Popen(self.rd + "./a.out")
     def finish(self):
        files = pp.read_files(self.rd)
        for f in files:
            ...
```

Discussion of Wrapping Code with ctypes

- Creates an interface for your C or C++ function

```
extern "C" int add_one(int i)
{
    return i+1;

}
```



- Python code to use this function is then

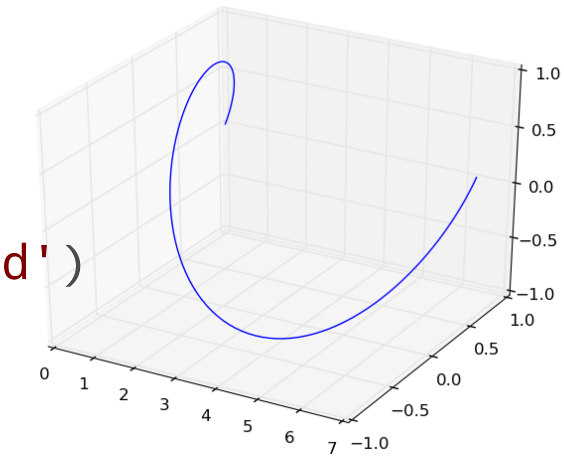
```
import numpy.ctypeslib as ctl
import ctypes

libname = 'testlib.so'
libdir = './'
lib=ctl.load_library(libname, libdir)

py_add_one = lib.add_one
py_add_one.argtypes = [ctypes.c_int]
value = 5
results = py_add_one(value)
print(results)
```

Compile to shared library with: g++ -shared -o testlib.so -fPIC test.cpp

Three dimensional plots in matplotlib and mayavi

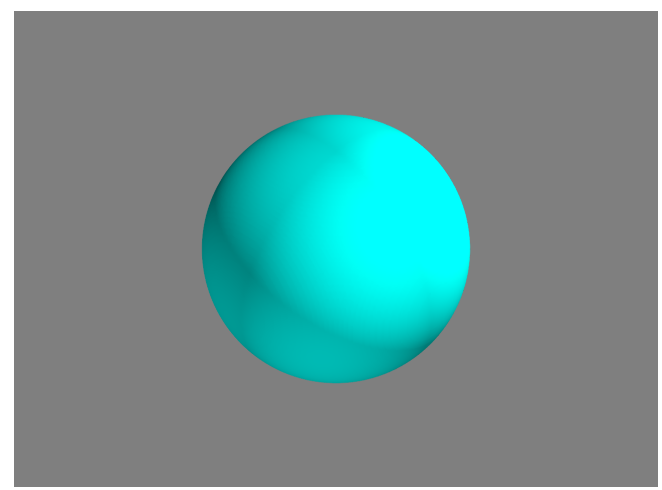
- Some 3D plotting in matplotlib (but limited)

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.linspace(0.,2*np.pi,100)
ax.plot(x, np.cos(x), np.sin(x))
plt.show()
```



- Generate isosurface data using mayavi (better 3D than matplotlib)

```
import numpy as np
import mayavi.mlab as mlab
x = np.linspace(-1., 1., 100)
y = x; z = y
[X,Y,Z] = np.meshgrid(x,y,z)
out1 = mlab.contour3d(X**2+Y**2+Z**2,
                      contours=[0.8])
mlab.show()
```



Three dimensional plots in mayavi

- Generate isosurface data using mayavi from 3D postproc reader

```
import mayavi.mlab as mlab

#3D DATA FIELDS LOADED HERE
# ........................
#3D DATA FIELDS LOADED HERE

for i in range(minrec,maxrec):
    field = pp.get_field(i)
    out1 = mlab.contour3d(field, contours=[0.3])
    mlab.savefig('./surface{:05d}'.format(i)+'.obj')
```

Object file, a format recognised by blender

Blender python interface

- Use python plugin to import isosurface, set material and save render

```
import bpy, bmesh
#Blender file saved with correctly setup camera/light source
bpy.ops.wm.open_mainfile('./scene.blend')
for i in range(minrec,maxrec):
    #Load object file from mayavi
    file ='/surface{:05d}'.format(i)
    bpy.ops.import_scene.obj(file+'.obj')
    obj = bpy.context.selected_objects[:][0]
    #Set material and render
    mat = bpy.data.materials['shiny_tranparent']
    obj.data.materials[0] = mat
    bpy.data.scenes['Scene'].render.filepath =file+'.jpg'
    bpy.ops.render.render( write_still=True )
    #Delete last object ready to load next object
    bpy.ops.object.select_all(action='DESELECT')
    bpy.context.scene.objects.active = obj
    obj.select = True
    bpy.ops.object.delete()
```

Blender python interface

Blender python interface

Hands-On Session 2

1) Create a Number class with a constructor to take in a variable a and store it as self.a, then add methods to square, cube and halve self.a.
2) Create four different instances of Number using an integer, float, list and numpy array. Try calling square, cube and half, what do you notice about duck typing?
3) In python, use pickle to save a list l=[1,2,3] then use a script to load and print
4) Inherit str to create a new `class` MyString(str) and add a new method first_letter which returns the first letter. Create an instance of MyString and get the first letter
5) Use subprocess to run the commad 'echo hello' ten times
6) Plot a 3D random walk using matplotlib with x=np.cumsum(np.random.randn(1000)*0.01) with similar for y and z.

<u>Data reader exercise</u>

7) Refactor the three functions from the first hands-on: get_files, read_header and read_data into a single class postproc (use examples if you didn't finish this).
8) Write a constructor for the postproc class to take foldername, header and filename. Remove input arguments from get_files and get_header function and use self values in functions, include self.header=get_headers() and self.files=get_files().
9) Make postproc a base class and using Python inherentance syntax, e.g. `class` postproc_binary(postproc):, create a range of different data readers.

Summary

- Using Python to read files and plot.
- Designing a basic Graphical User Interface.
- Unit testing frameworks and version control.
- Running parameter studies by calling executables repeatedly with subprocess.
- Other libraries and how to wrap your own code from fortran, c++, etc