An introduction to Python for Scientific Computation

By Edward Smith

10th March 2017

Aims for today

- Further details of the Python language

  a) More on Python data structures.

  b) Use of functions and design of interfaces.

  c) Introduction to classes and objects.

  d) Structuring a project, importing modules and writing tests.

  e) Examples of usage for scientific problems.

Overview

- Review of last week (~20mins)
- Using functions for reading data (~10 min)
- Hands on session + break (~30 min)
- Classes and objects for scientific computing and data analysis (~20 min)
- Hands on Session + break (~30 min)
- Solutions and Summary (~10 min)

What we covered last week

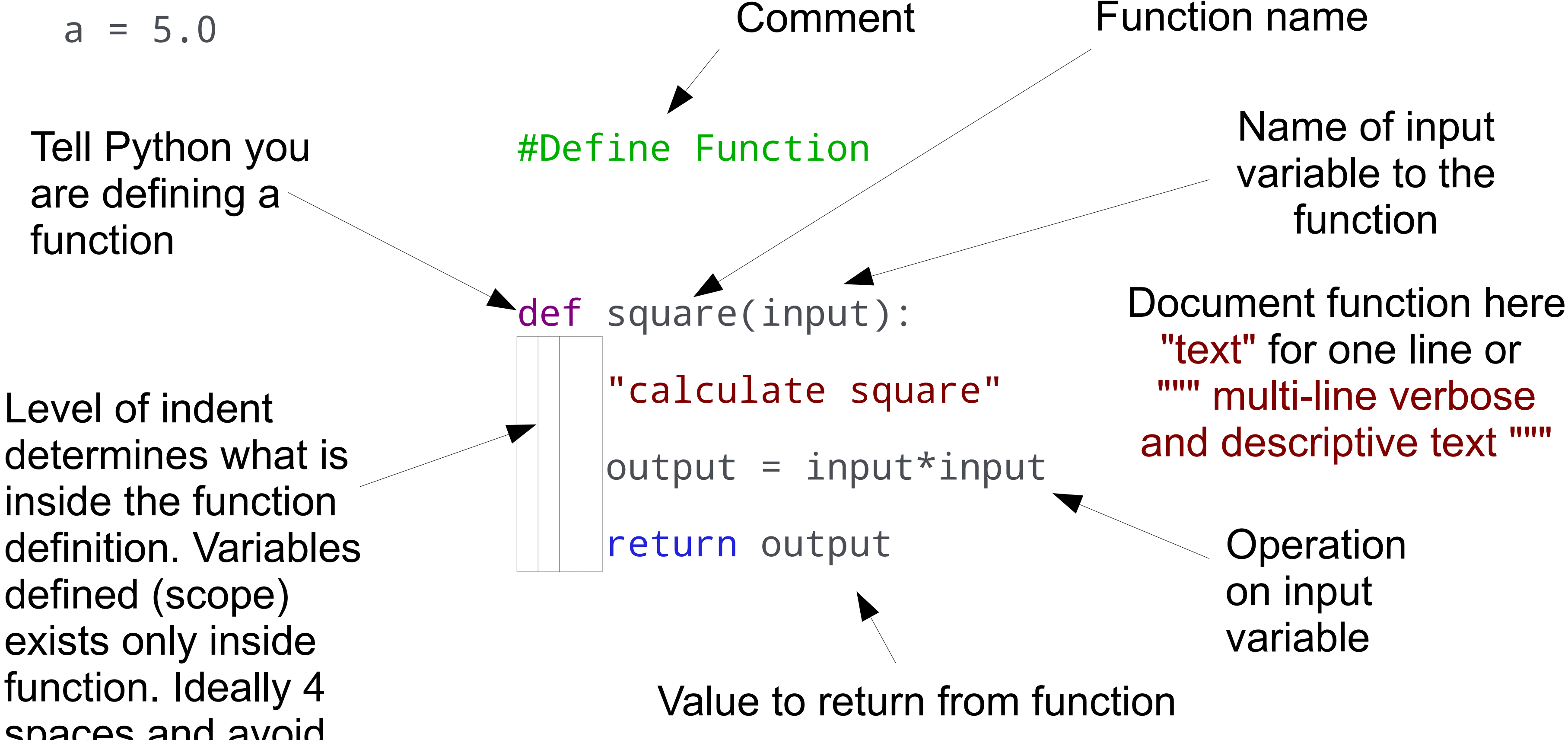
- Show how to use the command prompt to quickly learn Python
- Introduce a range of data types (Note everything is an object)

```
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

Functions

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



Comment

Function name

Tell Python you are defining a function

Name of input variable to the function

```
#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

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

Examples of Functions

- take some inputs
- perform some operation
- return outputs

Input → f(a, b, …, z) → Output

```
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
```

Optional variable. Given a value if not specified

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

```
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
```

Conditionals

- Allow logical tests

```
#Example of an if statement

if a > b:

    print(a)

else:

    print(a, b)
```

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

```
if type(a) is int:

    a = a + b

else:

    print("Error – a is type ", type(a))
```

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.find("x")                        Out: -1    #Not found

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

- We can make lists of any type

```
m = ["another string", 3, 3.141592653589793, [5,6]]
print(m[0], m[3][0])     #Note indexing starts from zero
```

- Iterators – loop through the contents of a list

```
m = ["another string", 3, 3.141592653589793, [5,6]]
for item in m:
    print(type(item), " with value ", item)
```

- To add one to every element we could use

```
l = [1,2,3,4]
for i in range(len(l)):
    l[i] = l[i] + 1
```

Note: will not work:

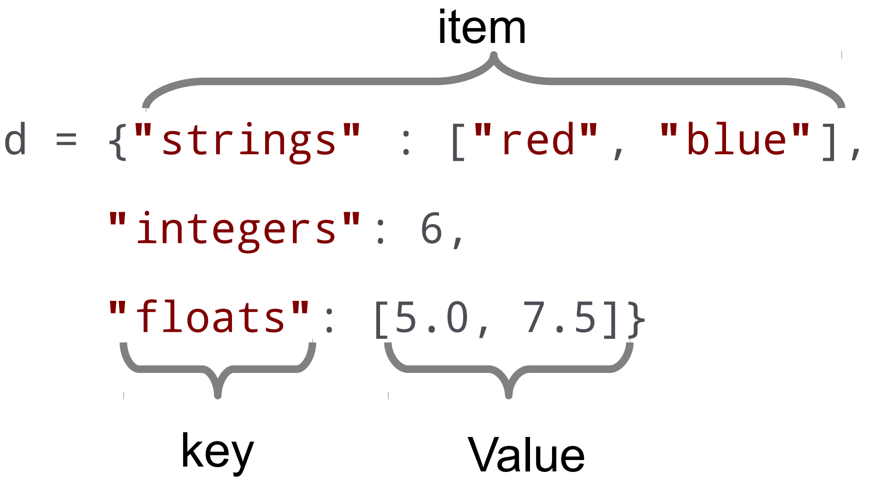
```
for i in l:
    i = i + 1
```

List comprehension

l = [i+1 for i in l]

Dictionaries

- Dictonaries for more complex data storage



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

```
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])
```

Numerical and Plotting Libraries

- Numpy – The basis for all other numerical packages to allow arrays instead of lists (implemented in c so more efficient)
  - `x = np.array([[1,2,3],[4,5,6],[7,8,9]])`
  - `mean, std, linspace, sin, cos, pi, etc`
- Matplotlib – similar plotting functionality to MATLAB
  - `plot, scatter, hist, bar, contourf, imagesc (imshow), etc`
- Scipy
  - Replaces lots of the MATLAB toolboxes with optimisation, curve fitting, regression, etc
- Pandas
  - Dataframes to organise, perform statistics and plot data

NOTE: Downloading and installing packages is trivial with “pip” or conda

Importing Numerical and Plotting Libraries

- 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]])
```

$$x = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

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

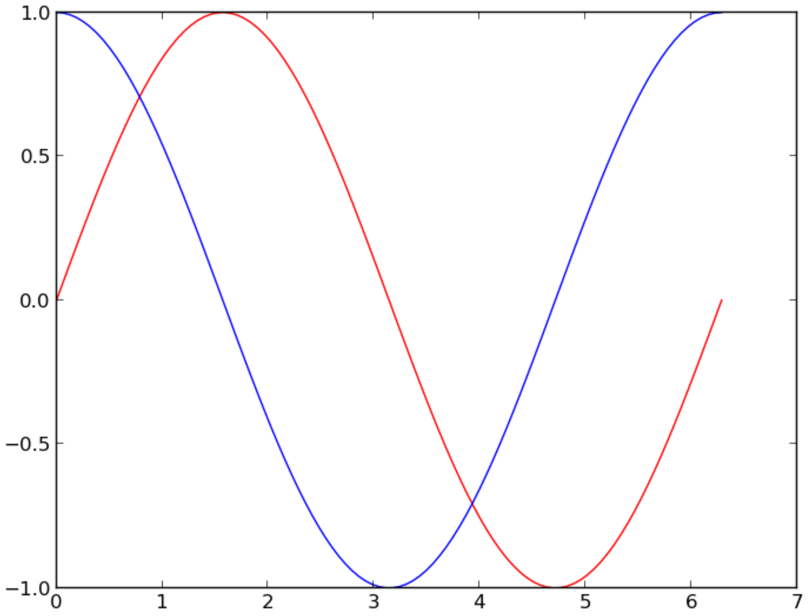
- matplotlib – similar plotting functionality to MATLAB

```
import matplotlib.pyplot as plt
plt.plot(x)
plt.show()
```

Use tab in ipython to see what code is available (or look online)

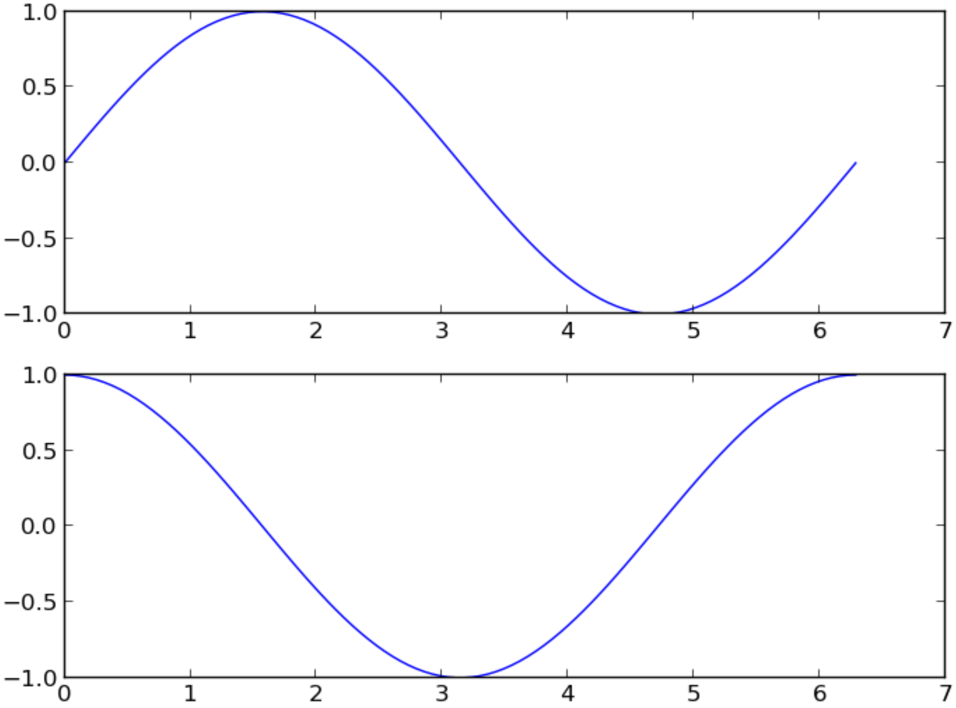
An Example plot

```
#python
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y)
plt.plot(x, z)
plt.show()
```



A Plot of Two Axes

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


x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
z = np.cos(x)
fig, ax = plt.subplots(2,1)
ax[0].plot(x, y)
ax[1].plot(x, z)
ax[1].set_xlabel("x axis", fontsize=24)
plt.show()
```



Two subplots, ax is a list of so called axis handles and we use the plot method of these handles.

An Plot Example Using Data from a csv File

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


#Read data from comma seperated variable file
data = np.genfromtxt("./file.csv", delimiter=',')


#Store columns as new variables x and y
x = data[:,0]
y = data[:,1]
plt.plot(x,y,"-or")
plt.show()
```

MATLAB syntax for plot line (-), point (o) in red (r)

file.csv

```
x,    y
1.0, 1.0
2.0, 4.0
3.0, 9.0
4.0, 16.0
5.0, 25.0
6.0, 36.0
```

Any Questions?

- We will build on these concepts in this session
  - Running from a script

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

```
s = "some string"
t = s + " with more"
s.split(" ")
```

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

```
if a < b:
    out = a
else
    out = b
```

```
import numpy as np
import matplotlib.pyplot as plt
#Read data from comma seperated variable file
data = np.genfromtxt("./file.csv", delimiter=',')
```

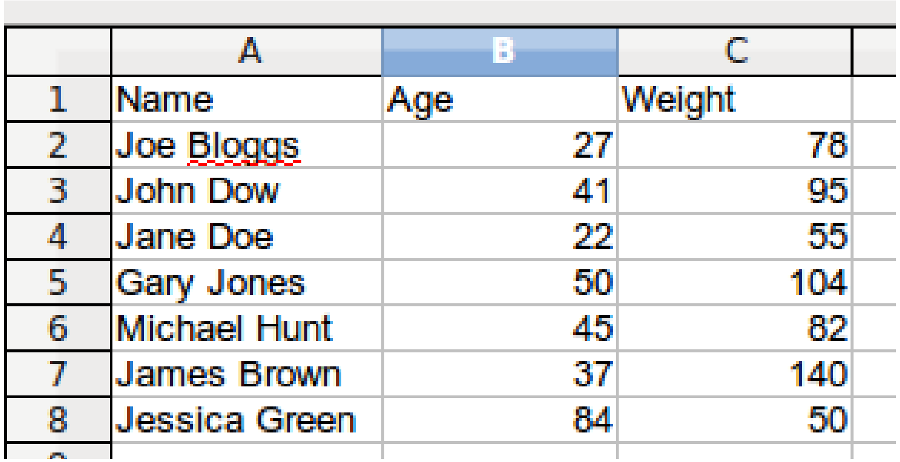
A Typical Postprocessing Workflow

- Get data from some source: experiments, numerical simulation, surveys/studies, an internet database, etc.
- Import it into python as a single numpy array, a list of numpy arrays, a dictonary of values, etc.
- Play around with various plots and data analysis techniques.
- Take the most promising output and save the script which generates this exactly, add labels and format to publication quality.
- We can develop an automated process from data to figure with minimal user input. This is useful because
  - Easy to make changes when required by reviewers
  - Clearer mapping from data to output (opendata movement)
  - Create functions to break the analysis down and reduce errors
  - You can use the same scripts to analyse similar data

An Example using data from a spreadsheet

```
import numpy as np


#Save data from spreadsheet into comma seperate file

data = np.genfromtxt("./sample_spreadsheet.csv", delimiter=',')
```

```
data = array([[  nan,   nan,   nan],
              [  nan,   27.,   78.],
              [  nan,   41.,   95.],
              [  nan,   22.,   55.],
              [  nan,   50.,  104.],
              [  nan,   45.,   82.],
              [  nan,   37.,  140.],
              [  nan,   84.,   50.]])
```

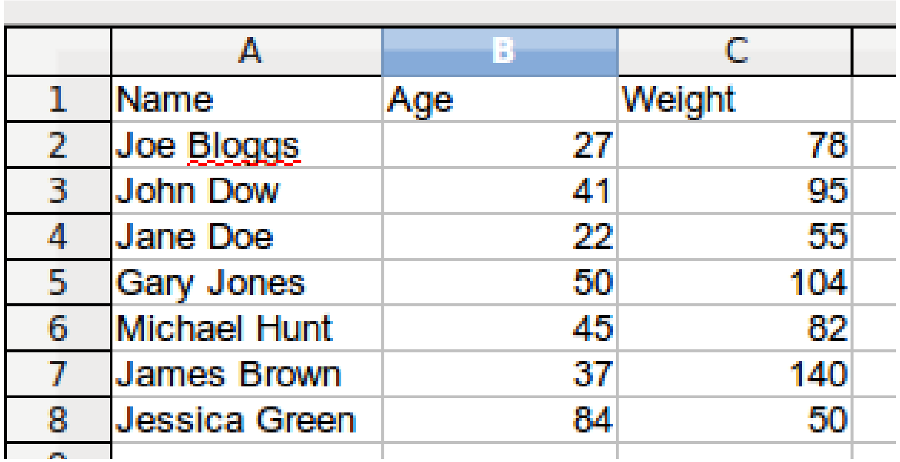


	A	B	C
1	Name	Age	Weight
2	Joe Bloggs	27	78
3	John Dow	41	95
4	Jane Doe	22	55
5	Gary Jones	50	104
6	Michael Hunt	45	82
7	James Brown	37	140
8	Jessica Green	84	50

An Example using data from a spreadsheet

```
import numpy as np


#Save data from spreadsheet into comma seperate file

data = np.genfromtxt("./sample_spreadsheet.csv", delimiter=',',

                     names=True)
```

```
data =
array([(nan, 27.0, 78.0),
       (nan, 41.0, 95.0),
       (nan, 22.0, 55.0),
       (nan, 50.0, 104.0),
       (nan, 45.0, 82.0),
       (nan, 37.0, 140.0),
       (nan, 84.0, 50.0)],
dtype=[('Name', '<f8'),
        ('Age', '<f8'),
        ('Weight', '<f8')])
```



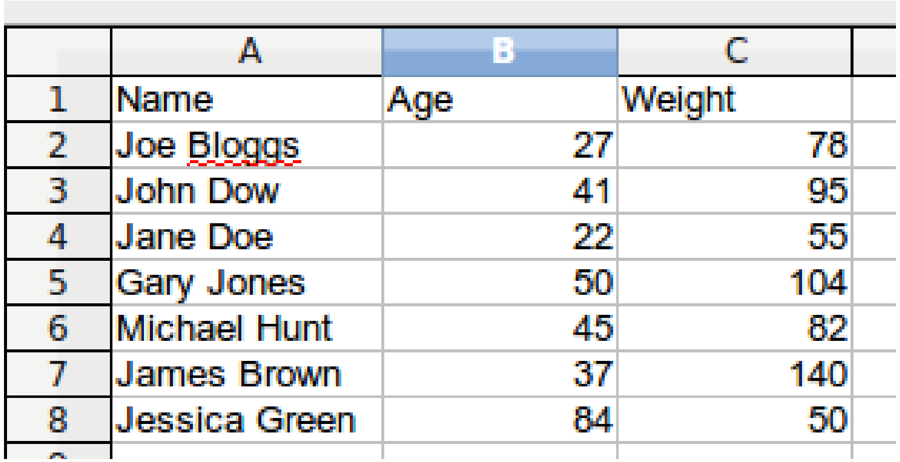
	A	B	C
1	Name	Age	Weight
2	Joe Bloggs	27	78
3	John Dow	41	95
4	Jane Doe	22	55
5	Gary Jones	50	104
6	Michael Hunt	45	82
7	James Brown	37	140
8	Jessica Green	84	50

We can access with data['Age'] like a dictionary

An Example using data from a spreadsheet

```
import numpy as np


#Save data from spreadsheet into comma seperate file

data = np.genfromtxt("./sample_spreadsheet.csv", delimiter=',',

                     names=True, dtype=object)
```

```
data =
array([('Joe Bloggs', '27',
'78'), ('John Dow', '41', '95'),
('Jane Doe', '22', '55'), ('Gary
Jones', '50', '104'),('Michael
Hunt', '45', '82'), ('James
Brown', '37', '140'),('Jessica
Green', '84', '50')],
dtype=[('Name', 'O'),
       ('Age', 'O'),
       ('Weight', 'O')])
```

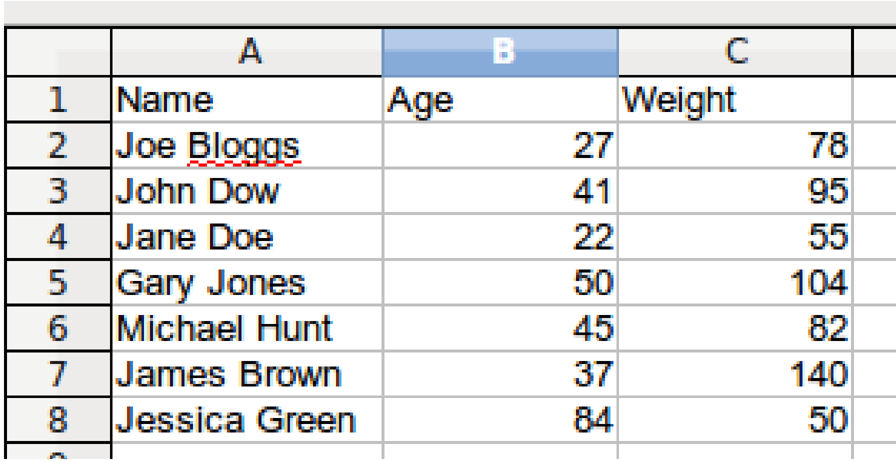


	A	B	C
1	Name	Age	Weight
2	Joe Bloggs	27	78
3	John Dow	41	95
4	Jane Doe	22	55
5	Gary Jones	50	104
6	Michael Hunt	45	82
7	James Brown	37	140
8	Jessica Green	84	50

An Example using data from a spreadsheet

```
import matplotlib.pyplot as plt

import pandas

data = pandas.read_csv("./sample_spreadsheet.csv")
```

```
data =          Name   Age   Weight
0       Joe Bloggs    27       78
1         John Dow    41       95
2         Jane Doe    22       55
3       Gary Jones    50      104
4     Michael Hunt    45       82
5      James Brown    37      140
6    Jessica Green    84       50
```



	A	B	C
1	Name	Age	Weight
2	Joe Bloggs	27	78
3	John Dow	41	95
4	Jane Doe	22	55
5	Gary Jones	50	104
6	Michael Hunt	45	82
7	James Brown	37	140
8	Jessica Green	84	50

```
#Some example operations

data.boxplot(); plt.show()  #Can call inbuilt plots

data.corr()    # Look at correlations
```

```
             Age    Weight
Age     1.000000 -0.246889
Weight -0.246889  1.000000
```

More Useful Application Of Python

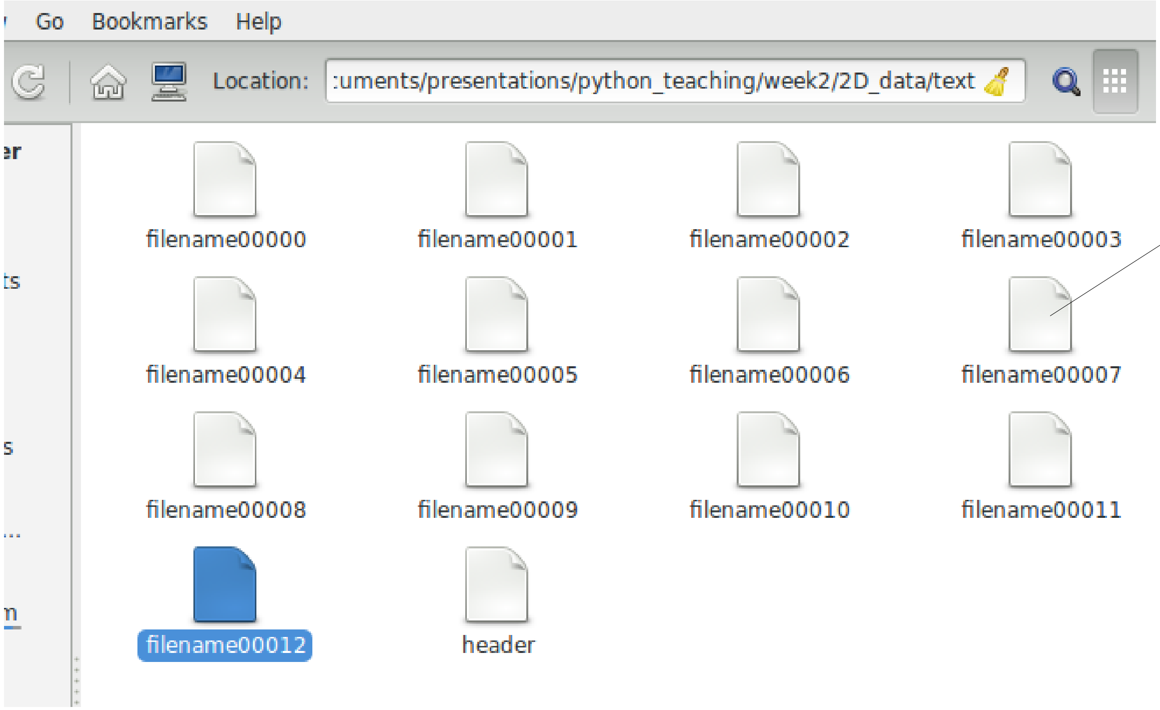
We have 2D data in multiple files with header (meta-data). We want

1) A function to read the header file and store parameters

2) A function to get the list of data files in the folder

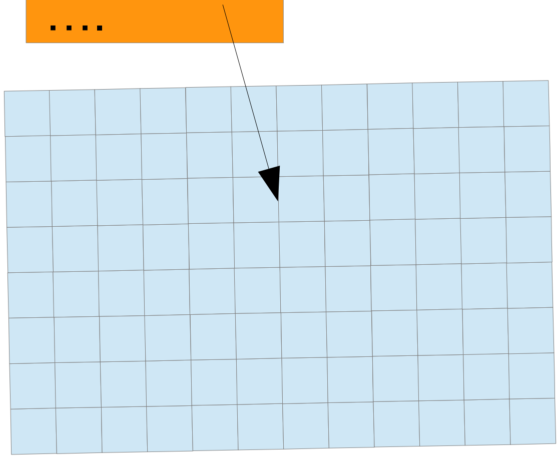
3) A function to read data using parameters from header

The data in each file is 4200 floats describing a 84 by 50 2D field



Location: :uments/presentations/python_teaching/week2/2D_data/text

1.025468
2.0198
-3.2471
….



Develop Three Functions to Postprocess

```
import matplotlib.pyplot as plt
# .. FUNCTIONS DEFINED HERE ..
# 1) read_header, 2) get_files and 3) read_file
foldername = './column/'
header = read_header(foldername+'header')
files = get_files(foldername, filename='filename')
for f in files:
    data = read_file(f)
    field = data.reshape(header['Nx'],header['Nz'])
    plt.imshow(field)
    plt.colorbar()
    plt.show()
```

Reading Header Files into Dictionaries

- Dictonaries are ideal for reading meta-data

```
header = {}

f = open('./header')

for l in f.readlines():

    key, value = l.split()

    header[key] = float(value)
```

string to list using spaces between words

header file

```
Nx 84
Nz 50
Lx 1560.41523408474
Lz 1069.90830902657
Nrecs 12
```

Data is a mix of integers and floats, we can use error handling to get type

```
    try:

        header[key] = int(value)

    except ValueError:

        header[key] = float(value)
```

Get All Files in Folder

- Write a loop to print 10 strings with names: "filename0", "filename1", … "filename9" (note str(i) converts an int to a string)

```
for i in range(10):
    print("filename0000" + str(i))
```

- More useful is the format method, with prepended zeros, so files are in displayed in order in folder (and read in order):

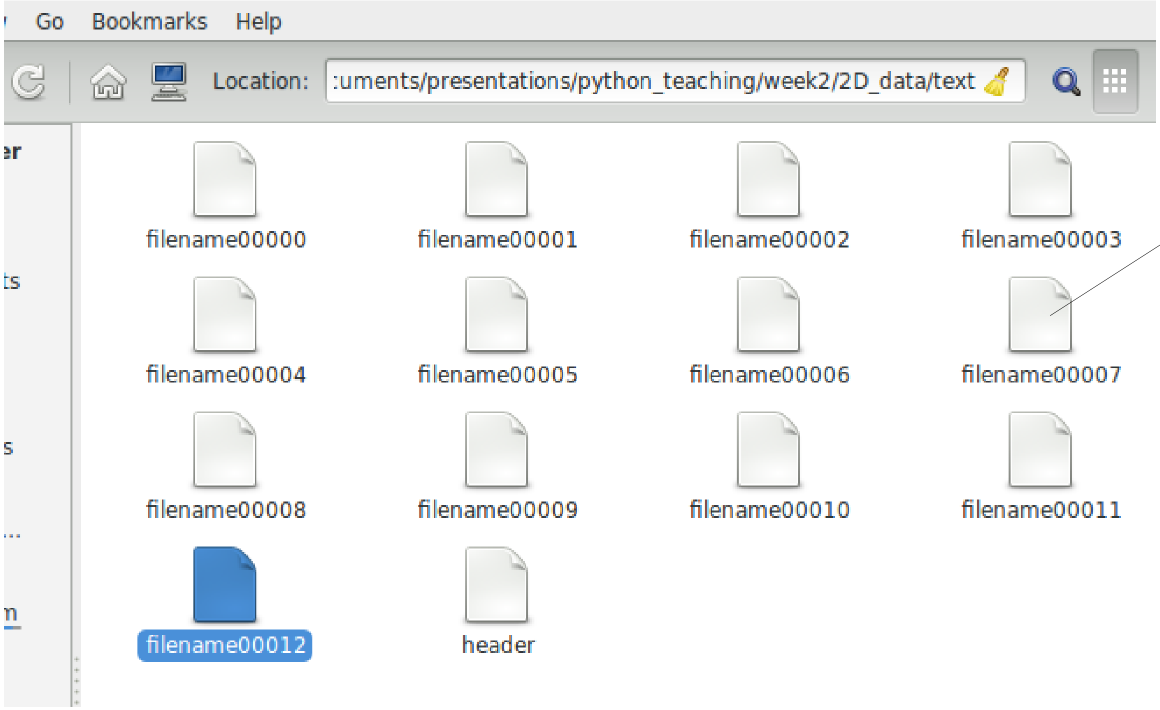
```
for i in range(13):
    print("filename{:05}".format(i))
```

- Get contents of all folder with same name

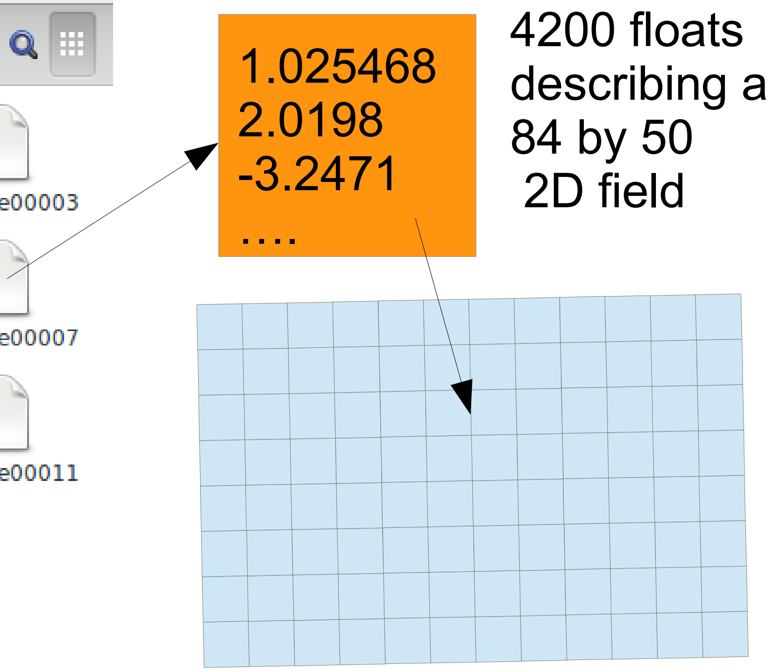
```
import glob
for i in glob.glob("filename*"):
    print(i)
```

Three Sample Datatypes

- Three formats of data for hands on exercise, choose one or use your own
  - Column - with a seperate header file
  - Binary - with seperate header (May not work for you, see Endianness)
  - Text - with included header file and other fields



Location: :uments/presentations/python_teaching/week2/2D_data/text



1.025468
2.0198
-3.2471
….

The data in each file is 4200 floats describing a 84 by 50 2D field

Reading 2D Data from a Column

- Reading data stored as a single column

```
# Numpy function to read column data

f = "./column/filename00001"

data = np.genfromtxt(f)

field = data.reshape(84, 50)
```

header file

```
Nx 84
Nz 50
Lx 1560.41523408474
Lz 1069.90830902657
Nrecs 12
```

Reorder to 2D based on Nx and Nz value we check for in the header file

Reading 2D Data from a Binary File

- Reading Binary Format data

```
# Numpy helper function to read binary

f = "./binary/filename00001"

data = np.fromfile(open(f ,'rb'), dtype='d')

field = data.reshape(84, 50)
```

Read binary flag

header file

Nx 84
Nz 50
Lx 1560.41523408474
Lz 1069.90830902657
Nrecs 12

Reading 2D Data from a Formatted Output File

```
/*--------------------------------*- C++ -*----------------------------------*|
=========                 |                                                 |
| \      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \    /   O peration     | Version:  3.0.1                                 |
|   \  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \/     M anipulation  | NOTE - THIS IS A FAKE FILE FOR PYTHON TEACHING  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    Nx        84;
    Nz        50;
    Lx        1560.41523408474;
    Lz        1069.90830902657;
    Nrecs     12
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 3 -1 0 0 0 0];

internalField   nonuniform List<scalar>
4200
(
-0.0310257085323
-0.0625208281593
-0.0440674291947
```

Header information in file

84 times 50 = 4200 records between brackets

Develop Three Functions to Postprocess

```
import matplotlib.pyplot as plt
# .. FUNCTIONS DEFINED HERE ..
# 1) read_header, 2) get_files and 3) read_file
foldername = './column/'
header = read_header(foldername+'header')
files = get_files(foldername, filename='filename')
for f in files:
    data = read_file(f)
    field = data.reshape(header['Nx'],header['Nz'])
    plt.imshow(field)
    plt.colorbar()
    plt.show()
```

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) Open the sample spreadsheet in the examples folder and plot age against weight. Save as a csv file. Import into numpy using genfromtxt and plot age against weight.
2) Choose an input type from 2D_data folder (column easy, binary intermediate and text is complex). Write a function read_file with inputs foldername and filename which returns all data. Check data using plt.imshow(data.reshape(84,50))
3) Write a function get_header which has input of the header file name and reads the header file into a dictonary (Note header is part of filename for the example with text files, you will need to change this).
4) Write a function get_files to get a list of all 13 files containing filenames.000* in a given directory (Using header information value Nrecs or files in directory itself)
5) Write a script which calls header=get_header and file=get_files(), loops through all 13 files calling data=read_file(filename), reshapes using header['Nx'] and header['Nz'] and plots 2D data.

Solution

```
import matplotlib.pyplot as plt
# .. FUNCTIONS DEFINED HERE ..
# 1) read_header, 2) get_files and 3) read_file
foldername = './column/'
header = read_header(foldername+'header')
files = get_files(foldername, filename='filename')
for f in files:
    data = read_file(f)
    field = data.reshape(header['Nx'],header['Nz'])
    plt.imshow(field)
    plt.colorbar()
    plt.show()
```

The Function Interface

- The inputs to a function and returned output are like a contract with the user, 'give me this and I will give you that'
    - All three exercises returned the same data from different files
    - This means the same top level code could be used for any of the three data formats
    - This hides the form of the underlying data from the user, you only need to call read(filename) to get the data
- When releasing software, version number systems are based around this
    - From v1.0 to v1.1 the interface stays the same
    - If major number changes, e.g. v1.1 to v2.0, the interface has changed and is no longer backward compatible

Unit Testing in Python

```
import unittest

def square(a):

    return a**2


class test_square(unittest.TestCase):

    def test_float(self):

        assert square(2.) == 4.

    def test_int(self):

        assert square(2) == 4


unittest.main()
```

Required format for unittest (we'll explain classes next)

Assert raises an error if the logical statement is not true

Test Driven Development

```
import unittest

def some_fn(a):

    pass




class test_drive(unittest.TestCase):

    def test_float(self):

        assert some_fn(2.) == 8.

    def test_int(self):

        assert some_fn(2) == 4

unittest.main()
```

We start with some requirements

Test Driven Development

```
import unittest
def some_fn(a):
    pass

class test_drive(unittest.TestCase):
    def test_float(self):
        assert some_fn(2.) == 8.
    def test_int(self):
        assert some_fn(2) == 4
unittest.main()
```

and a function which fails to satisfy them

We start with some requirements

Test Driven Development

```
import unittest

def some_fn(a):

    if isinstance(a, float):

        return a**3

    elif isinstance(a, int):

        return a**2

class test_drive(unittest.TestCase):

    def test_float(self):

        assert some_fn(2.) == 8.

    def test_int(self):

        assert some_fn(2) == 4

unittest.main()
```

Not we can develop the function until it does satisfy the tests.

We start with some requirements

Classes in Python

- So what is a class?

```
class MyClass():
    a = 4.5
    i = 2
```

A way to organise and group data, e.g.

```
class Person():
    age = 24
    name = "John Doe"
```

Classes in Python

- So what is a class?

```
class MyClass():
    a = 4.5
    i = 2
```

A way to organise
and group data, e.g.

```
class Person():
    age = 24
    name = "John Doe"
```

```
#Use as a static container of data
print(MyClass.a, MyClass.i) #Out: 4.5, 2
#Or instantiate (create an instance) of MyClass – an object
c = MyClass()
print(c.a, c.i) #Out: 4.5, 2
#We can access, no private in python but convention is to prepend
with _ for internal variables/methods and __ to indicate private)
c.a = 5.5
```

Classes in Python

- A class can also include functions

```
class MyClass():
    a = 4.5
    i = 2
    def square(self, b):
        return b**2
c = MyClass()
c.square(4)          #Out: 16
c.square(c.a)       #Out: 20.25
```

The first argument must always be self in a function defined inside a class

Recall from the string manipulations, we said that s.capitalize() works like capitalize(s)

- It would be more useful if the functions automatically act on the data in the class

Classes in Python

- A class can also include functions

```
class MyClass():

    a = 4.5

    i = 2

    def square(self):

        return self.a**2

c = MyClass()

c.square()         #Out: 20.25

c.a = 5.

c.square()         #Out: 25.0
```

Square is now acting on the the variable “a” which is part of MyClass (note, self is how MyClass refers to itself inside)

```
class Person():

    age = 24

    name = "John Doe"

    def next_bday(self):

        return self.age += 1
```

Classes in Python

- A class can also include functions

```
class MyClass():
    def set_val(self, a):
        self.a = a
    def square(self):
        return self.a**2
c = MyClass()
c.square()  #AttributeError: MyClass
            #instance has no attribute 'a'
c.set_val(5.)
c.square()        #Out: 25.0
```

We write a function to set a value.

The square function returns it's square using c.square()

But, it is possible to create an instance of MyClass and square with no value of a. We would ideally prevent using MyClass without defining a!

Classes in Python

- A class can also include functions

```
class MyClass():
    def __init__(self, a):
        self.a = a
    def square(self):
        return self.a**2


c = MyClass(4.5)
c.square()        #Out: 20.25
```

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

- So, using this idea, we could design a number class which can perform a range of useful operations for numbers

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
```

Classes for Postprocessing

- We can use classes with the data readering 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 readering 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('./binary/', 'header', 'filename')
for f in pp.files:
    data = read_file(f)
    field = data.reshape(pp.header['Nx'],pp.header['Nz'])
    plt.imshow(field)
    plt.colorbar()
    plt.show()
```

Hands-On Session 2

1) Use test driven development (i.e. write the tests first) to design a function which gives the cube of a number.
2) 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). Don'f forget to add the self argument to each function's input.
3) Write a constructor for the postproc class to take foldername, header and filename. Saves these values in the class (e.g. self.foldername = foldername). Remove input arguments from get_files and get_header function and use self values in functions.
4) In the constructor, include self.header=get_headers() and self.files=get_files(). Write another function to get the maximum number of records from header['Nrec'] and check against number of records from len(files).
5) Instantiate postproc class in a script, read header, get data files, read them and plot
6) Add a new function to the class which takes in a record number (between 0 and 12) and returns the field data formatted as a 2D numpy array ready to plot
7) Move class to a new file postproclib.py and use import postproclib as ppl
8) Try steps 3) to 8) for a different input data format by creating a new class.
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

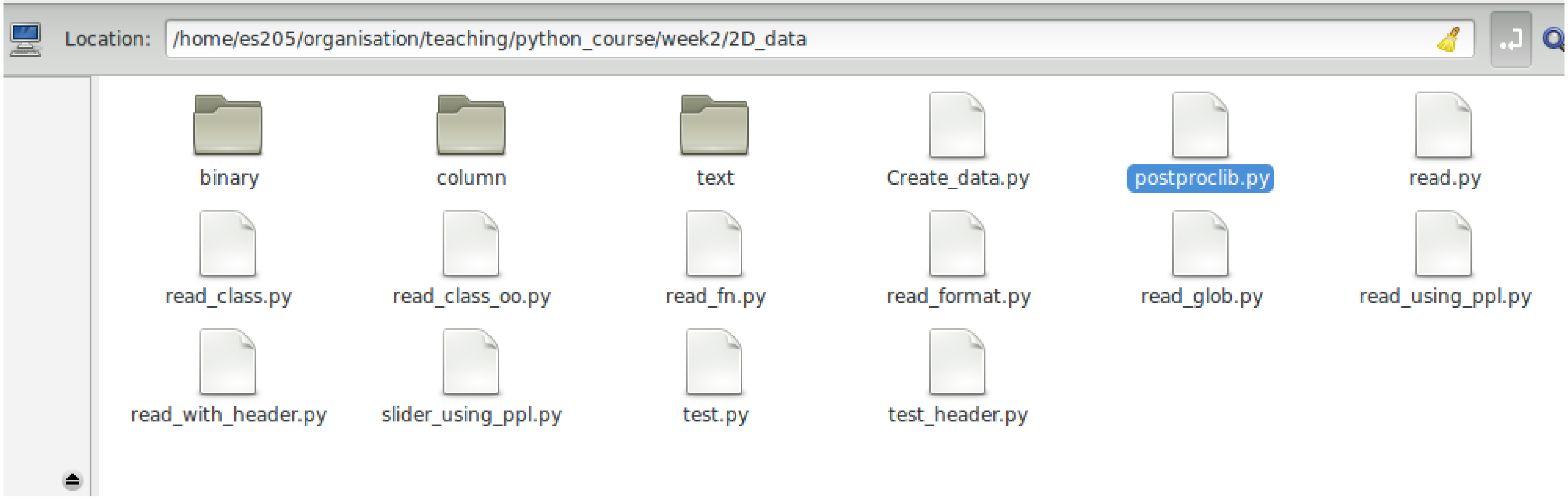
- In this hands on, we have gone beyond scripts and functions

  a) Classes group data processing information together with the methods to collect that data

Making a Module - Postproc Library

- Simply copy code to a new file, for example postproclib.py. Any script in the same folder can then import this,

```
import postproclib as ppl

pp = ppl.postproc('./binary/', 'header', 'filename')
```



Location: /home/es205/organisation/teaching/python_course/week2/2D_data

Using a Module - Postproc Library

```
import matplotlib.pyplot as plt
import postproclib as ppl


pp = ppl.postproc(foldername='./binary/',
                  headername='header',
                  filename='filename')
for i in range(pp.get_Nrecs()):
    field = pp.read_field(i)
    plt.imshow(field)
    plt.colorbar()
    plt.show()
```

Summary

- In this hands on, we have gone beyond scripts and functions

  a) Classes group data processing information together with the methods to collect that data

  b) Similar code is collected into a module which can be imported in any script and easily plotted

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/',
         '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()
```

A Hierarchy of Classes for Postprocessing

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


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)
```

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

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....

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

Summary

- In this hands on, we have gone beyond scripts and functions

  a) Classes group data processing information together with the methods to collect that data

  b) Similar code is collected into a module which can be imported in any script and easily plotted

  c) Object inheretance allows us to resuse most of the code and with only small changes, read all three data formats with the same interface (and more).

Summary

- Further details of the Python language

  a) More on Python data structures.

  b) Use of functions and design of interfaces.

  c) Introduction to classes and objects.

  d) Structuring a project, importing modules and writing tests.

  e) Examples of usage for scientific problems.