L01_Introduction

December 15, 2015

#

1 Scientific Python

Kevin Stratford kevin@epcc.ed.ac.uk Emmanouil Farsarakis farsarakis@epcc.ed.ac.uk
Other course authors: Neelofer Banglawala Andy Turner Arno Proeme

www.archer.ac.uk support@archer.ac.uk
[Intro] Course overview
Course website https://hpcarcher.github.io/2015-12-14-Portsmouth/
Course material as above
[Intro] Scientific computing
Typical workflow

- Generate data usually from simulation on HPC facilities (also from experiment!)
- Process data to generate appropriate results
- Visualise results to understand the significance of our work and gain scientific understanding
- Communicate results through publications, presentations, web, etc.

[Intro] Why Python?
Python is a very flexible tool

- Simple to learn
- Can program "procedurally" or use full-blown OOP
- High level: focus on what code does not how to write it
- Interactive python shell aids rapid prototyping
- Extensive standard library (and packages)

[Intro] For scientific computing?
Rich set of scientific computing functionality

- Standard numerical and scientific libraries
- Extensive graphical functionality
- Can interface with existing C/C++/Fortran code
- Good for control of complex "workflow"

[Intro] And...

- Free and Open Source
- Interactive Python is an alternative to Matlab, R

- Currently a “growth area”
- Useful links
- https://www.python.org/
- http://www.scipy.org/

[Intro] Core packages for scientific computing

- IPython - an advanced interactive shell
- NumPy - tools for manipulating numerical arrays
- Matplotlib - plotting in 2D and 3D
- SciPy - High-level scientific routines for common algorithms e.g. numerical integration, optimisation, Fourier transforms

[Intro] Other useful packages

- mpi4py : message passing parallel programming
- pandas : data analysis library
- scikit-learn : machine learning

. . . and many more . . .
[Intro] Some cautions

- Some backward compatability problems between versions 3.x and 2.x
- Many packages can risk complex dependency “sprawl”
- It’s interpreted
- HPC tools (debuggers, profilers) lag behind compiled languages

[Intro] How to run Python?
Python code is executed by an interpreter: python
Python has two modes of operating:

- non-interactive
- interactive

[Intro] Non-interactive Python
In non-interactive mode, run the Python interpreter with a file

- ~$ python myscript.py
- Python module files end in .py extension
- ideal for persistent, resueable, large (complex) code

[Intro] Non-interactive Python : Hello world!
Try it! Run “helloworld.py” from the command-line (same directory as notebook)
~$ python helloworld.py Hello world!
[Intro] Interactive Python : the Python shell
Interactive mode : run Python interpreter without a file

- Interpreter runs as a Python shell (interactive Python runtime environment)
- Type Python commands directly into the Python shell (after the prompt >>>)

- Ideal for trying out commands, experimenting
- Use Ctrl+d to exit
- By default, lose session history when you exit shell

[Intro] Interactive Python : Hello world!
Try it! Launch a Python shell and after the prompt type : print "Hello world!"
~$ python Python 2.7.10 (default, Jul 01 2015, 09:00:00) [GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin Type "help", "copyright", "credits" or "license" for more information. >>> print "Hello world!" Hello world!
[Intro] IPython shell

- IPython is an enhanced Python shell
- Ideal for interactive data manipulation and visualisation
- Has features and built-in commands that make it easier to use than the standard Python shell
- Launch an IPython shell like a Python shell (without a script): ipython
- Useful links : http://ipython.org/

[Intro] IPython shell : useful commands

- TAB completion to list available functions in a module e.g. module.a<TAB>
- Query what function does with ? e.g. function?
- 'Magic' commands with %
- %hist to see commands issued so far
- %save to save commands issued so far
- Paste code straight into IPython shell
- Access system commands using !, e.g. !ls -l
- Command quickref for a summary of capabilities

[Intro] IPython shell : Hello world!
Try it! Launch an IPython shell and print "Hello world!"
IPython 4.0.0 – An enhanced Interactive Python. ? -> Introduction and overview of IPython's features. %quickref -> Quick reference. help -> Python's own help system. object? -> Details about 'object', use 'object??' for extra details. In [1]: print "Hello world!" Hello world!
[Intro] Python basics recap : data types

- Data types
- integer e.g. -1, float e.g. 3.1412
- string e.g. 'a string' or e.g. "a string also"
- Dynamically typed, no explicit declaration
  - e.g. can have x=100, followed by x="100" without error
- Can cast one data type into another e.g. x=int("100")

[Intro] Python basics recap : data types II

- list : elements can be a mix of types e.g. [3, "a", False]

- dictionary : like lists with generalised keys e.g. {“key1”:“value1”, “key2”:“value2”}
- tuple : an immutable sequence e.g. (1,2,3) or 1,2,3
- empty tuple: ( )
- “one-tuple” must have comma: e.g. (1,)

[Intro] Python basics recap : importing I
A module is a file (e.g. mymodule.py) containing Python definitions and statements
To use functions defined within a module, need to import functionality
[Intro] Python basics recap : importing II
There are several ways to import functionality

1. import mymodule, call function as mymodule.afunc
2. from mymodule import afunc, call function as afunc
3. To save typing, can use alias for imported module

- import mymodule as mod, then use mod.afunc

4. Avoid importing everything from a module

- from module import *

[Intro] Python basics recap : code structure
Whitespace matters, code blocks are indented with a tab or 4 spaces (need colons!) * for item in list: #do some stuff * if condition: #do some stuff * def myfunc(arg1, arg2): #do some stuff
[Intro] Python basics recap : functions I

- A number of built-in functions
  - Available from the interpreter, e.g., len(), int()
- Methods related built-in datatypes (objects)
  - Via dot notation, e.g., list.append(), list.sort()

[Intro] Python basics recap : functions II

- Very large selection of standard library packages:
- Always available as part of python
- import required package
- invoke method as package.method()
- Growing seleection of additional packages
- May need to install appropriate version
- May need addition to PYTHONPATH
- import package and invoke as above

[Intro] Python basics recap : script files

- At the top of script files you may see:
- #!/usr/bin/env python
- Locate interpreter from your environment’s $PATH
- At the bottom of script files you may see:
- if __name__ == “__main__”: import sys function(sys.argv)
- Makes file usable as a script as well as an importable module

[Intro] Python basics recap : references

- Careful - variables are references
- variables are references to objects: let a = 3, b = a; if b = 5 then a = 5 (try it!)
- Getting help: make use of online documentation
- Documentation
- https://docs.python.org/
- https://www.codecademy.com/learn/python

[Intro] Warm-up : exercise I

Define a function age that takes a list of years between 1950 and 2015 and returns the median age, and the two ages closest to it, where age = 2015 - year. Assume the input list is randomly ordered and has an odd number of elements N, where $N >= 3$. So for:

- years = [1989, 1955, 2011, 1943, 1975], age returns [26, 40, 60]
- Note: for a sorted list of numbers, the median is the number in the middle of the list.

[Intro] Warm-up : exercise II

Steps to take:

1. You will need to create the input list. You can do this manually or use Python... [Hint: you may want to use random.randint(start, stop)]
2. You will need to sort the list
3. Use list indexing where possible e.g. list[3:5]
4. Make sure to test your function.

[Intro] Warm-up exercise : a solution

```
In [ ]: # function to calculate the median age and
        # its two neighbours from a list of years

        def medianage(years):
            ages=[];
            for y in years:
                ages.append(2015-y);
            ages.sort()
            n = len(ages);
            mid = n/2;
            return ages[mid-1:mid+2];

        years = [1989, 1955, 2011, 1943, 1975];
        medianage(years)
```

[Intro] Warm-up : optional I

Read the list of years from a text file, 'years.txt', which should have the total number of years N in the first line, followed by a numbered list of years:

total number of years 1 year ...

[Intro] Warm-up : optional II

To generate the input file 'years.txt', you may need:

import sys, yearsfn=open(filename, "w"), input.close(), output.write("{0:2d} {1:2d}".format(i, j))

To read the input file 'years.txt', you may need:

input=open(filename, "w"), line = infile.readline(), line.rstrip(), line.split(), int("9"),

Could you use list comprehension (if you haven't already)? E.g. squared = [x*x for x in list]

In [ ]:
```
# extended exercise : create an input file containing
# years

from random import randint;
import sys;
import age;

years = [1989, 1955, 2011, 1943, 1975];
medianage(years);

# using list comprehension
years = [randint(1950,2015) for x in range(9)];

# save years to file
outyears = open("years.txt",'w');
i=1
# output total number of years
outyears.write("{0:2d}\n".format(len(years2)))

for y in years2:
    outyears.write("{0:2d} {1:2d}\n".format(i, y));
    i+=1;

outyears.close();

# read file
inyears = open("years.txt","r");
line = inyears.readline()
# remove any newline characters
N=int(line);

ages=[];
yrs=[];
for i in range(N):
    line = inyears.readline()
    line = line.rstrip();
    tokens = line.split();
    yrs.append(int(tokens[1]));

inyears.close()

# check...
print yrs

# get median age
medianage(yrs)
```

[Intro] Summary
We have considered why Scientific Computing in Python is a worthwhile pursuit
We have reviewed how to use Python together with some Python basics
We have also been introduced to the IPython shell
We are now ready to explore the packages that constitute the backbone of scientific python
Next session : NumPy