Introduction to High Performance Scientific Computing

Autumn, 2017

Python lecture 1

Imperial College
London

Getting started with Python

Command-line basics; numbers; modules; lists, tuples, and strings

Python overview

- Python ‘core’: General-purpose programming language
- Python is an *interpreted* language: code does not need to be compiled as in c or fortran

Python overview

- Python ‘core’: General-purpose programming language
- Python is an *interpreted* language: code does not need to be compiled as in c or fortran
- Matlab-style scientific computing via add-on modules:
  - numpy: basic linear algebra
  - matplotlib: 2d plotting
  - scipy: large range of capabilities
    - Differential equations
    - Signal processing
    - Optimization
    - Statistics
- Will often see code like:

```
In [3]: import numpy as np

In [4]: import scipy.optimize as sco
```

Python overview

Can use Python is several ways:

- At the terminal (like the Matlab command window)
- Writing scripts in a text editor
- Creating notebooks (like Mathematica)
- Canopy and Anaconda Python distributions provide nice interfaces for each of these approaches

Getting started at the terminal

- The basic python terminal is… basic

- So, we use *interactive* python or *ipython* along with *qtconsole:*

```
$ jupyter qtconsole
```

- Nice help features, user-friendly

- *qtconsole* adds user-friendly features to *ipython* which can be launched with:

```
$ ipython
```

Getting started at the terminal

- The basic python terminal is… basic

- So, we use *interactive* python or *ipython* along with *qtconsole:*

```
$ jupyter qtconsole
```

- Nice help features, user-friendly

- *qtconsole* adds user-friendly features to *ipython* which can be launched with:

```
$ ipython
```

- Will almost always want *numpy* and *matplotlib*

- Can be easily imported with:

```
In [1]: %pylab
```

Getting started at the terminal

Can use terminal as a calculator:

```
In [13]: x=2

In [14]: y=3

In [15]: x+y
Out[15]: 5

In [16]: sin(x)
Out[16]: 0.90929742682568171

In [17]: sqrt(2.)
Out[17]: 1.4142135623730951
```

Getting help

A few different approaches:

- Using “?”
- <tab> completion
- google

Getting help

At the terminal, try: `sin?`

or: `numpy?`

<tab> completion:

Try sin **<tab>:**

```
In [93]: sin
sin          sinc          single        singlecomplex  sinh
```

or numpy. **<tab>**

Getting help

Can use these approaches for user-defined variables as well:

```
In [112]: x=4+2j

In [113]: x?
Type:        complex
String form: (4+2j)
Docstring:
complex(real[, imag]) -> complex number

Create a complex number from a real part
and an optional imaginary part.
This is equivalent to (real + imag*1j) where
imag defaults to 0.

In [114]: x. <tab>
x.conjugate  x.imag      x.real
```

Number types

We have integers, floating-point (real) numbers, Booleans, (and we have seen complex numbers)

```
In [130]: type(3)
Out[130]: int

In [131]: type(3.)
Out[131]: float

In [132]: 1<2
Out[132]: True

In [133]: type(1<2)
Out[133]: bool
```

Number types

Booleans are only useful if we know Python’s relational operators:

```
x == y                 # Produce True if ... x is equal to y
x != y                # ... x is not equal to y
x > y                 # ... x is greater than y
x < y                 # ... x is less than y
x >= y                 # ... x is greater than or equal to y
x <= y                 # ... x is less than or equal to y
```

Number types

Booleans are only useful if we know Python’s relational operators:

```
x == y               # Produce True if ... x is equal to y
x != y              # ... x is not equal to y
x > y               # ... x is greater than y
x < y               # ... x is less than y
x >= y               # ... x is greater than or equal to y
x <= y               # ... x is less than or equal to y
```

…which can also be combined:

```
In [141]: x=pi

In [142]: y=3

In [143]: (x>y) and (y>0)
Out[143]: True

In [144]: (x>y) or (y<0)
Out[144]: True
```

Containers

Containers: ‘collections’ of data – strings, tuples, lists, dictionaries

```
In [185]: type("this is a string")
Out[185]: str

In [186]: type(("this","is","a","tuple"))
Out[186]: tuple

In [187]: type(["this","is","a","list"])
Out[187]: list

In [188]: type({"num1":12.3,"num2":24.0})
Out[188]: dict
```

Strings

Useful for processing input, producing nicely-formatted output

Use <tab> completion to see built-in Python capabilities:

```
In [212]: name="first last"

In [213]: name. <tab>
```

For example:

```
In [2]: name="first last"

In [3]: name
Out[3]: 'first last'

In [4]: name.center(20)
Out[4]: '     first last     '
```

Tuples

Immutable collection of items

```
In [33]: a=(1,2,"three","four")

In [34]: a[2]
Out[34]: 'three'

In [35]: a[2]=3
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-1e8e286566a0> in <module>()
----> 1 a[2]=3

TypeError: 'tuple' object does not support item assignmentme="first last”
```

Can’t change 3rd element

Tuples

Often used as input/output for functions

Useful for “switching” values of variables:

```
In [47]: x,y=2,3

In [48]: x,y
Out[48]: (2, 3)

In [49]: x,y=y,x

In [50]: x,y
Out[50]: (3, 2)
```

Lists

Mutable collection of items

```
In [52]: a=[1,2,"three","four"]

In [53]: a[2]
Out[53]: 'three'

In [54]: a[2]=3

In [55]: a
Out[55]: [1, 2, 3, 'four']
```

Lists

Python has a lot of nice features for working with lists…

```
In [61]: a
Out[61]: [1, 2, 3, 'four']

In [62]: b
Out[62]: [5.0, 6, 7]

In [63]: a+b
Out[63]: [1, 2, 3, 'four', 5.0, 6, 7]

In [64]: a*2
Out[64]: [1, 2, 3, 'four', 1, 2, 3, 'four']

In [65]: a. <tab>
a.append   a.count    a.extend   a.index    a.insert   a.pop      a.remove   a.reverse  a.sort
```

List indexing

- **List indices run from zero to N-1 (*where N=len(list)*)**
- **Can also use negative indices as shown in example:**

```
In [7]: a=[1,2,'three','four']

In [8]: [a[0],a[1],a[2],a[len(a)-1]]
Out[8]: [1, 2, 'three', 'four']

In [9]: [a[-1],a[-2],a[-3],a[-4]]
Out[9]: ['four', 'three', 2, 1]
```

List slices

- List slices take the form list[*start*:*end*:*step*]
- This gives the values from *start* to *end-1* with stepsize given by *step*
- The default value of *step*=1 is used if step is omitted
- The default values *start*=0 and *end*=len(list) are used if *start* and *end* are omitted

```
In [1]: a=[1,2,'three','four']

In [2]: a[1:3]
Out[2]: [2, 'three']

In [3]: a[1:4:2]
Out[3]: [2, 'four']

In [4]: a[1::2]
Out[4]: [2, 'four']
```