Week 2: Computing in python I, Introduction

MSc/MRes CMEE 2014-15

Samraat Pawar

Imperial College London

October 13, 2014

INTRODUCTION TO THE PYTHON WEEK

- Day 1: **Introduction and basic** `python`: Why Computer programs, `python` basics, Practical 0
- Day 2: Writing and running `python` code, control flow, Practical 1
- Day 3: **Writing** `python` **programs**: IPython, writing, debugging, using, and testing functions, Practical 1
- Day 4: **Writing** `python` **programs**: IPython, writing, debugging, using, and testing functions, Practical 2
- Day 5: **Writing** `python` **programs**: IPython, writing, debugging, using, and testing functions, Practical 2

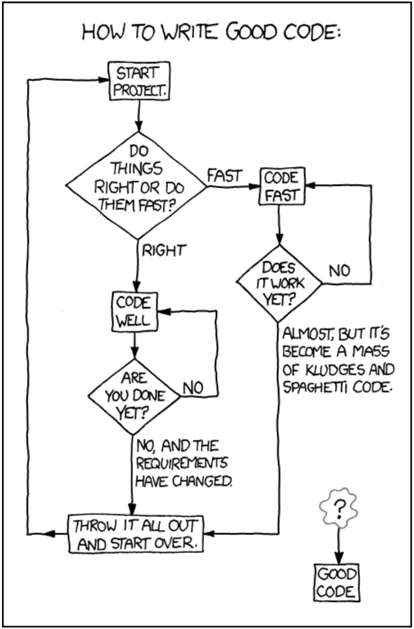
WHY WRITE COMPUTER PROGRAMS?

Donald Knuth, 1995: Science is what we understand well enough to explain to a computer. Art is everything else we do.

- Programs can do anything (that can be specified)
- As such, no software is typically available to perform exactly the analysis we are planning
- Permits success despite complexity, through modularization and precise specification
- Reproducibility – just re-run the code!
- Modularity – break up your complex analysis in smaller pieces
- Organised thinking – writing code requires this!
- Career prospects – good, scientific coders are in short supply in all fields!

WHICH LANGUAGES?

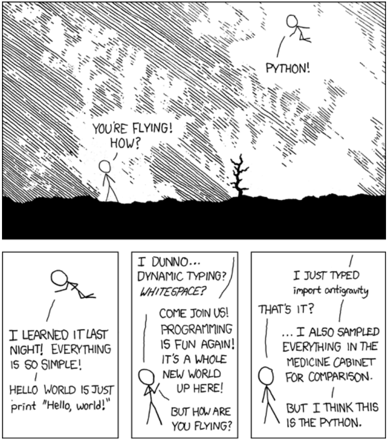
- Several hundred programming languages are currently available – which ones should a biologist choose?
  1. A fast, compiled (or semi-compiled) "procedural" language (e.g., `C`) – not so important for most of you
  2. A modern, easy-to-write, interpreted (or semi-compiled) language that is still quite fast, like `python`
  3. A mathematical/statistical software with programming and graphing capabilities like `R` (also, `mathematica`, `MATLAB`)
- One size doesn't fit all!


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle"

— Joe Armstrong, creator of Erlang

- `python` was designed with readability and re-usability in mind



http://xkcd.com/

Time taken by programming + debugging + running relatively lower than less intuitive or cluttered languages (e.g., `FORTRAN`, `perl`)

INSTALLING python

We will use 2.7.x, not 3.x (you can later, if you want)

- On Ubuntu/Linux, open a terminal (ctrl+alt+t) and type:

```
sudo apt-get install ipython python-scipy python-matplotlib
```

- Enable “Send Selection to Terminal” using `<Primary>Return` in `geany`

Reminder: All code in a colored box, and “\” means multi-line code (can be entered verbatim in bash/terminal)

python **WARMUP I**

- Open a terminal (`ctrl+alt+t`) and type `python`; Then,

```
>>> 2 + 2 # Summation; note that comments start with #
4

>>> 2 * 2
4

>>> 2 / 2
1

>>> 2 / 2.0
1.0

>>> 2 / 2.
1.0

>>> 2 > 3
False

>>> 2 >= 2
True
```

`python` **WARMUP II**

- Now let's switch to `ipython`! (will explain later why)
- Type (ctrl+D): this will exit you from the python shell and you will see the bash prompt again)
- Now type `ipython`. You should now see (after some text):

```
In [1]:
```

- This is the interactive `python` shell (ergo, `ipython`)
  (If you don't like the blue prompt, type `%colors linux`)

python **WARMUP III**

- Now, let's continue our warmup (ignore the prompt numbering)

```
 In []: 2 == 2
Out []: True

 In []: 2 != 2
Out []: False

 In []: 3 / 2
Out []: 1

 In []: 3 / 2.
Out []: 1.5

 In []: 'hola, ' + 'me llamo Samraat' #why not two languages at the same
time?!
Out []: 'hola, me llamo Samraat'
```

- Thus, `python` has integer, float (real numbers, with different precision levels) and string variables

python **OPERATORS**

+	Addition
-	Subtraction
*****	Multiplication
/	Division
******	Power
%	Modulo
//	Integer division
==	Equals
!=	Differs
>	Greater
>=	Greater or equal
&, and	Logical and
|, or	Logical or
!, not	Logical not

Assigning and manipulating variables I

```
 In []: x = 5

 In []: x + 3
Out []: 8

 In []: y = 8

 In []: x + y
Out []: 13

 In []: x = 'My string'

 In []: x + ' now has more shit'
Out []: 'My string now has more shit'

 In []: x + y
Out []: TypeError: cannot concatenate 'str' and 'int' objects
```

Assigning and manipulating variables II

- No problem, we can convert from one type to another:

```
 In []: x + str(y)
Out []: 'My string8'

 In []: z = '88'

 In []: x + z
Out []: 'My string88'

 In []: y + int(z)
Out []: 96
```

In python, the type of a variable is determined when the program or command is running (dynamic typing) (like R, unlike C or FORTRAN)

`python` **DATA TYPES AND DATA STRUCTURES**

- `python` numbers or strings (or both) can be stored and manipulated in:
  - **List**: most versatile, can contain compound data, “mutable”, enclosed in brackets, [ ]
  - **Tuple**: like a list, but “immutable” — like a read only list, enclosed in parentheses, ( )
  - **Dictionary**: a kind of “hash table” of key-value pairs enclosed by curly braces, { } — key can be number or string, values can be any object! (well OK, a python object)
  - **numpy arrays**: Fast, compact, convenient for numerical computing — more on this later!

LISTS

```
 In []: MyList = [3,2.44,'green',True]

 In []: MyList[1]
Out []: 2.44

 In []: MyList[0] # NOTE: FIRST ELEMENT -> 0
Out []: 3

 In []: MyList[4]
Out []: IndexError: list index out of range

 In []: MyList[2] = 'blue'

 In []: MyList
Out []: [3, 2.44, 'blue', True]

 In []: MyList[0] = 'blue'

 In []: MyList
Out []: ['blue', 2.44, 'blue', True]

 In []: MyList.append('a new item') # NOTE: ".append"!

 In []: MyList
Out []: ['blue', 2.44, 'blue', True, 'a new item']

 In []: MyList.sort() # NOTE: suffix a ".", hit tab, and wonder!

 In []: MyList
Out []: [True, 2.44, 'a new item', 'blue', 'blue']
```

TUPLES I

```
 In []: FoodWeb=[('a','b'),('a','c'),('b','c'),('c','c')]

 In []: FoodWeb[0]
Out []: ('a', 'b')

 In []: FoodWeb[0][0]
Out []: 'a'

 In []: FoodWeb[0][0] = "bbb"  # NOTE: tuples are "immutable"
      TypeError: 'tuple' object does not support item assignment

 In []: FoodWeb[0] = ("bbb","ccc")

 In []: FoodWeb[0]
Out []: ('bbb', 'ccc')
```

- *Why assign these food web data to a list of tuples and not a list of lists?*

Tuples II

- Tuples contain immutable sequences, but their elements can be mutable:

```
 In []: a = (1, 2, [])

 In []: a[2].append(1000)

 In []: a
Out []: (1, 2, [1000])
```

SETS

- You can convert a list to an immutable “set” on which you can perform set operations
- A set is an unordered collection with no duplicate elements

```
 In []: a = [5,6,7,7,7,8,9,9]

 In []: b = set(a)

 In []: b
Out []: set([8, 9, 5, 6, 7])

 In []: c = set([3,4,5,6])

 In []: b & c
Out []: set([5, 6])

 In []: b | c
Out []: set([3, 4, 5, 6, 7, 8, 9])

 In []: list(b | c) # set to list
Out []: [3, 4, 5, 6, 7, 8, 9]
```

a - b	a.difference(b)
a <= b	a.issubset(b)
a >= b	b.issubset(a)
a & b	a.intersection(b)
a | b	a.union(b)

DICTIONARIES, I

- A set of values (any `python` object) indexed by keys (string or number), a bit like `R` lists

```
 In []: GenomeSize = {'Homo sapiens': 3200.0, 'Escherichia coli': 4.6,
'Arabidopsis thaliana': 157.0}

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
  'Escherichia coli': 4.6,
  'Homo sapiens': 3200.0}

 In []: GenomeSize['Arabidopsis thaliana']
Out []: 157.0

 In []: GenomeSize['Saccharomyces cerevisiae'] = 12.1

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
'Escherichia coli': 4.6,
'Homo sapiens': 3200.0,
'Saccharomyces cerevisiae': 12.1}
```

DICTIONARIES, II

```
 In []: GenomeSize['Escherichia coli'] = 4.6  # ALREADY IN DICTIONARY!

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3200.0,
 'Saccharomyces cerevisiae': 12.1}

 In []: GenomeSize['Homo sapiens'] = 3201.1

 In []: GenomeSize
Out []:
{'Arabidopsis thaliana': 157.0,
 'Escherichia coli': 4.6,
 'Homo sapiens': 3201.1,
 'Saccharomyces cerevisiae': 12.1}
```

So in summary,

- If your elements/data are unordered and indexed by numbers use **lists**
- If they are ordered sequences use **tuples**
- If you want to perform set operations on them, use **sets**
- If they are unordered and indexed by keys (e.g., names), use **dictionaries**
- *But why not use dictionaries for everything?*
  – Can slow down your code!

COPYING MUTABLE OBJECTS IS TRICKY

```
# First, try this:
a = [1, 2, 3]
b = a # you are merely creating a new ``tag'' (b)
a.append(4)
print b
# this will print [1, 2, 3, 4]!!

# Now, try:
a = [1, 2, 3]
b = a[:]  # This is a "shallow" copy
a.append(4)
print b
# this will print [1, 2, 3].

# What about more complex lists?
a = [[1, 2], [3, 4]]
b = a[:]
a[0][1] = 22 # Note how I accessed this 2D list
print b
# this will print [[1, 22], [3, 4]]

# the solution is to do a "deep" copy:
import copy

a = [[1, 2], [3, 4]]
b = copy.deepcopy(a)
a[0][1] = 22
print b
# this will print [[1, 2], [3, 4]]
```

python LOVES STRINGS

```python
s = " this is a string "
len(s)
# length of s -> 18

print s.replace(" ","-")
# Substitute spaces " " with dashes -> -this-is-a-string-

print s.find("s")
# First occurrence of s -> 4 (start at 0)

print s.count("s")
# Count the number of "s" -> 3

t = s.split()
print t
# Split the string using spaces and make
# a list -> ['this', 'is', 'a', 'string']

t = s.split(" is ")
print t
# Split the string using " is " and make
# a list -> [' this', 'a string ']

t = s.strip()
print t
# remove trailing spaces

print s.upper()
# ' THIS IS A STRING '

'WORD'.lower()
# 'word'
```

WRITING `python` CODE

Now let's learn to write and run python code from a .py file. But first, some some guidelines for good code-writing practices (see `python.org/dev/peps/pep-0008/`):

- Wrap lines to be $<$80 characters long. You can use parentheses () or signal that the line continues using a "backslash" $\backslash$
- Use 4 spaces for indentation, no tabs (I use tabs!)
- Separate functions using a blank line
- When possible, write comments on separate lines

Make sure you have chosen a particular indent type (space or tab) in geany (or whatever you are using) — indentation is all-important in `python`!

WRITING `python` CODE

- Use "docstrings" to **document how to use the code**, and **comments to explain why and how the code works**
- Naming conventions (bit of a mess, you'll learn as you go!):
  - `_internal_global_variable` (for use inside module only)
  - `a_variable`
  - `SOME_CONSTANT`
  - `a_function`
  - Never call a variable `l` or `O` or `o`
    why not? – you are likely to confuse it with `1` or `0`!
- Use spaces around operators and after commas:
  `a = func(x, y) + other(3, 4)`

WRITING python CODE I

Let's start with a "boilerplate" code:

```
#!/usr/bin/python

"""Description of this program

   you can use several lines"""

__author__ = 'Samraat Pawar (s.pawar@imperial.ac.uk)'
__version__ = '0.0.1'

# imports
import sys # module to interface our program with the operating system

# constants can go here

# functions can go here
def main(argv):
    print 'This is a boilerplate' # NOTE: indented using 4 spaces
    return 0

if (__name__ == "__main__"): #makes sure the "main" function is called from commandline
    status = main(sys.argv)
    sys.exit(status)
```

- First line tells computer where to look for python
- Triple quotes start a "docstring" comment (more on this later)

WRITING python CODE II

- "__" signal "internal" variables (never name your variables so!)
- Last few lines tad esoteric, but `main` is important (look up `http://ibiblio.org/g2swap/byteofpython/read/module-name.html`)
- Type the function and save as `boilerplate.py` in `CMEECourseWork/Week2/Code`:

```
$ cd ~/Documents/../CMEECourseWork/Week2/Code
$ geany boilerplate.py
```

- After writing and saving `boilerplate.py`, in terminal, type:

```
$ ipython boilerplate.py
$ ipython -i boilerplate.py
```

- First command launches python and executes the file
- Second command launches python and "imports" the script (more on this later)
- You can use `python` instead of `ipython`

CONTROL STATEMENTS I

- OK, let's get deeper into `python` functions
- To begin, first copy and rename boilerplate.py:

```
$ cp boilerplate.py control_flow.py
```

- Then type the following script:

```python
#!/usr/bin/python

"""Some functions exemplifying the use of control statements"""
#docstrings are considered part of the running code (normal comments are
#stripped). Hence, you can access your docstrings at run time.
__author__ = 'Samraat Pawar (s.pawar@imperial.ac.uk)'
__version__ = '0.0.1'

import sys

def even_or_odd(x=0): # if not specified, x should take value 0.

    """Find whether a number x is even or odd."""
    if x % 2 == 0: #The conditional if
        return "%d is Even!" % x
    return "%d is Odd!" % x
```

```python
def largest_divisor_five(x=120):
    """Find which is the largest divisor of x among 2,3,4,5."""
    largest = 0
    if x % 5 == 0:
        largest = 5
    elif x % 4 == 0: #means "else, if"
        largest = 4
    elif x % 3 == 0:
        largest = 3
    elif x % 2 == 0:
        largest = 2
    else: # When all other (if, elif) conditions are not met
        return "No divisor found for %d!" % x # Each function can return a value or a variable.
    return "The largest divisor of %d is %d" % (x, largest)

def is_prime(x=70):
    """Find whether an integer is prime."""
    for i in range(2, x): #  "range" returns a sequence of integers
        if x % i == 0:
          print "%d is not a prime: %d is a divisor" % (x, i) #Print formatted text "%d %s %f %e" % (20,"30",0.0003,0.00003)

          return False
    print "%d is a prime!" % x
    return True

def find_all_primes(x=22):
    """Find all the primes up to x"""
    allprimes = []
```

CONTROL STATEMENTS III

```
    for i in range(2, x + 1):
      if is_prime(i):
        allprimes.append(i)
    print "There are %d primes between 2 and %d" % (len(allprimes), x)
    return allprimes

def main(argv):
    print even_or_odd(22)
    print even_or_odd(33)
    print largest_divisor_five(120)
    print largest_divisor_five(121)
    print is_prime(60)
    print is_prime(59)
    print find_all_primes(100)
    return 0

if (__name__ == "__main__"):
    status = main(sys.argv)
```

- Now run the code:

```
$ ipython -i control_flow.py
```

- You can also call any of the functions within `control_flow.py`:

```
In []: even_or_odd(11)
Out[]: '11 is Odd!'
```

Loops

```
# for loops in Python
for i in range(5):
    print i

my_list = [0, 2, "geronimo!", 3.0, True, False]
for k in my_list:
    print k

total = 0
summands = [0, 1, 11, 111, 1111]
for s in summands:
    print total + s

# while loops  in Python
z = 0
while z < 100:
    z = z + 1
    print (z)

b = True
while b:
    print "GERONIMO! infinite loop! ctrl+c to stop!"
# ctrl + c to stop!
```

CONTROL FLOW EXERCISES I

```
# How many times will 'hello' be printed?
# 1)
for i in range(3, 17):
    print 'hello'

# 2)
for j in range(12):
    if j % 3 == 0:
        print 'hello'

# 3)
for j in range(15):
     if j % 5 == 3:
        print 'hello'
     elif j % 4 == 3:
        print 'hello'

# 4)
z = 0
while z != 15:
    print 'hello'
    z = z + 3

# 5)
z = 12
while z < 100:
    if z == 31:
        for k in range(7):
            print 'hello'
```

CONTROL FLOW EXERCISES II

```python
    elif z == 18:
        print 'hello'
    z = z + 1

# What does fooXX do?
def foo1(x):
    return x ** 0.5

def foo2(x, y):
    if x > y:
        return x
    return y

def foo3(x, y, z):
    if x > y:
        tmp = y
        y = x
        x = tmp
    if y > z:
        tmp = z
        z = y
        y = tmp
    if x > y:
        tmp = y
        y = x
        x = tmp
    return [x, y, z]

def foo4(x):
    result = 1
```

CONTROL FLOW EXERCISES III

```
    for i in range(1, x + 1):
        result = result * i
    return result

# This is a recursive function
# meaning that the function calls itself
# read about it at
# en.wikipedia.org/wiki/Recursion_(computer_science)
def foo5(x):
    if x == 1:
        return 1
    return x * foo5(x - 1)
```

- We have been running scripts from our beloved terminal or bash shell
- To execute script file from within python shell: `execfile(\filetoload.py")`
- In IPython: `%run filetoload.py`

LIST COMPREHENSIONS I

- A way to combine loops, functions and logical tests in a single line of code:

```
## Let's find just those taxa that are oak trees from a list of species

taxa = [ 'Quercus robur',
         'Fraxinus excelsior',
         'Pinus sylvestris',
         'Quercus cerris',
         'Quercus petraea',
       ]

def is_an_oak(name):
    return name.lower().startswith('quercus ')

##Using for loops
oaks = set()
for taxon in taxa:
    if is_an_oak(taxon):
        oaks.add(taxon)
print oaks

##Using list comprehensions
oaks = set([t for t in taxa if is_an_oak(t)])
print oaks

##Get names in UPPER CASE using for loops
oaks = set()
for taxon in taxa:
```

LIST COMPREHENSIONS II

```
    if is_an_oak(taxon):
        oaks.add(taxon.upper())
print oaks

##Get names in UPPER CASE using list comprehensions
oaks = set([t.upper() for t in taxa if is_an_oak(t)])
print oaks
```

- Don't go mad with list comprehensions — code readability is more important than squeezing lots into a single line!
- Call me a fool, but I rarely use list comprehensions!

FOR YOUR READING PLEASURE

- The Zen of python: open a python shell and type `import this`
- Code like a Pythonista: Idiomatic `python` (Google it)
- Also good: the Google `python` Style Guide
- Browse the python tutorial `docs.python.org/tutorial/`

PRACTICAL 0: MAKE SURE THE BASICS WORK

1. Review and make sure you can run all the commands, code fragments, and functions we have covered today and get the expected outputs
2. Run `boilerplate.py` and `control_flow.py` from the terminal (try both python and ipython)
3. Run `boilerplate.py` and `control_flow.py` from within the python and ipython shells
4. Open and complete the tasks/exercises in `dictionary.py`, `tuple.py`, `lc1.py`, `lc2.py` (in this order)
5. Keep all code files organized in `CMEECourseWork/Week2/Code`
6. Version control all your work; the updated bitbucket repository should contain: `boilerplate.py`, `control_flow.py`, `lc1.py`, `lc2.py`, `dictionary.py`, `tuple.py`