Week 2: Computing in python I, Writing and testing Python functions

MSc/MRes CMEE 2014-15

Samraat Pawar

Imperial College
London

October 14, 2014

THE ZEN OF PYTHON

Open a python shell and type `import this`

USING IPYTHON

- In IPython, you can `TAB` everything — a lot will be revealed!
- `TAB` leads to autocompletion
- `TAB` after "." reveals object's functions and attributes:

```
In [1]: alist = ['a', 'b', 'c']

In [2]: alist.
alist.append   alist.extend   alist.insert   alist.remove
alist.sort     alist.count    alist.index    alist.pop
alist.reverse

In [3]: adict = {'mickey': 100, 'mouse': 3.14}

In [4]: adict.
adict.clear       adict.items       adict.pop
adict.viewitems   adict.copy        adict.iteritems
adict.popitem     adict.viewkeys    adict.fromkeys
adict.iterkeys    adict.setdefault  adict.viewvalues
adict.get         adict.itervalues  adict.update
adict.has_key     adict.keys        adict.values
```

USING IPYTHON

- IPython also has “magic commands” (start with %; e.g., `%run`)
- Some useful magic commands:

`%who` Shows current namespace (all variables, modules and functions)

`%whos` Also display the type of each variable; typing `%whos function` only displays functions etc.

`%pwd` Print working directory

`%history` Print recent commands

`%cpaste` Paste indented code into IPython — very useful when you want to run just part of a function (indentation and all)

Let’s try the `%cpaste` function

USING IPYTHON

- First, type the following code in a file:

```
def PrintNumbers(x):
    for i in range(x):
        print x
    return 0
```

- Now launch IPython and type:

```
In []: x = 11

# Now copy the for loop from the file.
# Note that there is extra indentation!

In []: %cpaste
Pasting code; enter '--' alone on the line to stop
or use Ctrl-D.
:    for i in range(x):
:        print x
:--
11
11
11
11
11
11
11
11
11
11
```

USING IPYTHON

- Another useful feature is the question mark:

```
In []: ?adict
Type:        dict
Base Class: <type 'dict'>
String Form:{'mickey': 100, 'mouse': 3.14}
Namespace:  Interactive
Length:      2
Docstring:
dict() -> new empty dictionary
dict(mapping) -> new dictionary initialized from a mapping
object's
    (key, value) pairs
dict(iterable) -> new dictionary initialized as if via:
    d = {}
    for k, v in iterable:
d[k] = v
dict(**kwargs) -> new dictionary initialized with the
name=value pairs
    in the keyword argument list.
For example:  dict(one=1, two=2)
```

FUNCTIONS AND MODULES

- Ideally you should aim to compartmentalize your code into a bunch of functions, typically written in a single `.py` file: this is a Python "module"
- Why bother with modules?
  - Keeping code compartmentalized is good for debugging, unit testing, and profiling (coming up later)
  - Makes code more compact by minimizing redundancies (write repeatedly used code segments as a module)
  - Allows you to import and use useful functions that you yourself wrote, just like you would from standard python packages (see next slide)
- A Python package is simply a directory of Python modules (quite like an R package)

TIME TO EXPLAIN THIS WEIRD STUFF I

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

- `sys.argv` – try this in a file called `sysargv.py`:

```
import sys
print "Name of the script is: ", sys.argv[0]
print "Number of arguments are: ", len(sys.argv)
print "The arguments are: " , str(sys.argv)
```

- Now run `sysargv.py` with different numbers of arguments

```
run sysargv.py
run sysargv.py var1 var2
run sysargv.py 1 2 var3
```

TIME TO EXPLAIN THIS WEIRD STUFF II

- OK what about

```
if __name__ == "__main__"
```

- This directs the `python` interpreter to set the special `__name__` variable to have a value "`__main__`"
- This allows our `Python` script to act as either as a reusable module, or as a standalone program
- To see this, put an `import(control_flow)` and modify the `main` function of `boilerplate.py` so:

```
def main(argv):
print 'This is a boilerplate' # NOTE: indented using two tabs
print 'control_flow says: ' + control_flow.even_or_odd(10) + '!'
return 0
```

- Run it!

TIME TO EXPLAIN THIS WEIRD STUFF III

- OK, finally, what about this bit:

```
sys.exit(status)
```

- It's just a way to exit in a dignified manner (from anywhere in the program)!
- Try putting it elsewhere in a function/module and see what happens

MODULES

There are different ways of to **import** a module:

- `import my_module`, then functions in the module can be called as `my_module.one_of_my_functions()`.
- `from my_module import my_function` imports only the function `my_function` in the module `my_module`. It can then be called as if it were part of the main file: `my_function()`.
- `import my_module as mm` imports the module `my_module` and calls it `mm`. Convenient when the name of the module is very long. The functions in the module can be called as `mm.one_of_my_functions()`.
- `from my_module import *`. Avoid doing this!
  Why?

MODULES

There are different ways of to **import** a module:

- `import my_module`, then functions in the module can be called as `my_module.one_of_my_functions()`.
- `from my_module import my_function` imports only the function `my_function` in the module `my_module`. It can then be called as if it were part of the main file: `my_function()`.
- `import my_module as mm` imports the module `my_module` and calls it `mm`. Convenient when the name of the module is very long. The functions in the module can be called as `mm.one_of_my_functions()`.
- `from my_module import *`. Avoid doing this!
  Why? – to avoid name conflicts!

MODULES

There are different ways of to **import** a module:

- `import my_module`, then functions in the module can be called as `my_module.one_of_my_functions()`.
- `from my_module import my_function` imports only the function `my_function` in the module `my_module`. It can then be called as if it were part of the main file: `my_function()`.
- `import my_module as mm` imports the module `my_module` and calls it `mm`. Convenient when the name of the module is very long. The functions in the module can be called as `mm.one_of_my_functions()`.
- `from my_module import *`. Avoid doing this!
  Why? – to avoid name conflicts!
- You can also access variables written into modules: `import my_module`, then `my_module.one_of_my_variables`

PYTHON INPUT/OUTPUT I

- Let's look at importing and exporting data
- Make a textfile called `test.txt` in `Week2/Sandbox/` with the following content (including the empty lines):

```
First Line
Second Line

Third Line

Fourth Line

```

PYTHON INPUT/OUTPUT II

- Then, type the following in `Week2/Code/basic_io.py`:

```
##############################
# FILE INPUT
##############################
# Open a file for reading
f = open('../Sandbox/test.txt', 'r')
# use "implicit" for loop:
# if the object is a file, python will cycle over lines
for line in f:
    print line, # the "," prevents adding a new line

# close the file
f.close()

# Same example, skip blank lines
f = open('../Sandbox/test.txt', 'r')
for line in f:
    if len(line.strip()) > 0:
        print line,

f.close()

##############################
# FILE OUTPUT
##############################
# Save the elements of a list to a file
list_to_save = range(100)
```

PYTHON INPUT/OUTPUT III

```
28 f = open('../Sandbox/testout.txt','w')
   for i in list_to_save:
30     f.write(str(i) + '\n') ## Add a new line at the end

32 f.close()

34 ############################
   # STORING OBJECTS
36 ############################
   # To save an object (even complex) for later use
38 my_dictionary = {"a key": 10, "another key": 11}

40 import pickle

42 f = open('../Sandbox/testp.p','wb') ## note the b: accept binary files
   pickle.dump(my_dictionary, f)
44 f.close()

46 ## Load the data again
   f = open('../Sandbox/testp.p','rb')
48 another_dictionary = pickle.load(f)
   f.close()
50
   print another_dictionary
```

PYTHON INPUT/OUTPUT IV

- The `csv` package makes it easy to manipulate CSV files (get `testcsv.csv` from master repo):

```
1  import csv
2
3  # Read a file containing:
4  # 'Species','Infraorder','Family','Distribution','Body mass male (Kg)'
5  f = open('../Sandbox/testcsv.csv','rb')
6
7  csvread = csv.reader(f)
8  temp = []
9  for row in csvread:
10     temp.append(tuple(row))
11     print row
12     print "The species is", row[0]
13
14 f.close()
15
16 # write a file containing only species name and Body mass
17 f = open('../Sandbox/testcsv.csv','rb')
18 g = open('../Sandbox/bodymass.csv','wb')
19
20 csvread = csv.reader(f)
21 csvwrite = csv.writer(g)
22 for row in csvread:
23     print row
24     csvwrite.writerow([row[0], row[4]])
25
26 f.close()
27 g.close()
```

PYTHON INPUT/OUTPUT V

- type this in `Week2/Code/basic_csv.py`!

UNIT TESTING, I

What do you want from your code? Rank the following by importance:

1. it gives me the right answer
2. it is very fast
3. it is possible to test it
4. it is easy to read
5. it uses lots of 'clever' programming techniques
6. it has lots of tests
7. it uses every language feature that you know about

- If you are very lucky, your program will crash when you run it
- If you are lucky, you will get an answer that is obviously wrong
- If you are unlucky, you won't notice until after publication
- If you are very unlucky, someone else will notice it after publication

UNIT TESTING, II

- Ultimately, most time is spent debugging (coming up!), not writing code
- Unit testing prevents the most common mistakes and yields reliable code
- Indeed, there are many reasons for testing:
  - Can you prove (to you) that you code do what you think it do?
  - Did you think about the things that might go wrong?
  - Can you prove to other people that your code works?
  - Does it still all still work if you fix a bug?
  - Does it still all still work if you add a feature?
  - Does it work with that new dataset?
  - Does it work on the latest version of R/Python?
  - Does it work on Mac, Linux, Windows?
  - 64 bit and 32 bit?
  - Does it work on an old version of a Mac
  - Does it work on Harvey, or Imperial's Linux cluster?

UNIT TESTING, III

- The idea is to write *independent* tests for the *smallest units* of code (*why smallest units?* – to be able to retain the tests upon code modification)
- Let's try `doctest`, the simplest testing tool in python: simple tests for each function are embedded in the docstring
- Copy the file `control_flow.py` into the file `test_control_flow.py` and edit the original function:

```
1  #!/usr/bin/python
3  """Some functions exemplifying the use of control statements"""
5  __author__ = 'Samraat Pawar (s.pawar@imperial.ac.uk)'
   __version__ = '0.0.1'
7
   import sys
9  import doctest # Import the doctest module
11 def even_or_odd(x=0):
       """Find whether a number x is even or odd.
13
       >>> even_or_odd(10)
```

UNIT TESTING, IV

```
15      '10 is Even!'
17      >>> even_or_odd(5)
        '5 is Odd!'
19
        whenever a float is provided, then the closest integer is used:
21      >>> even_or_odd(3.2)
        '3 is Odd!'
23
        in case of negative numbers, the positive is taken:
25      >>> even_or_odd(-2)
        '-2 is Even!'
27
        """
29      #Define function to be tested
        if x % 2 == 0:
31          return "%d is Even!" % x
        return "%d is Odd!" % x
33
## I SUPPRESSED THIS BLOCK: WHY?
35
# def main(argv):
37      # print even_or_odd(22)
        # print even_or_odd(33)
39      # return 0

41  # if (__name__ == "__main__"):
        # status = main(sys.argv)
43
doctest.testmod()    # To run with embedded tests
```

UNIT TESTING, V

Now type `run test_control_flow.py -v`:

```
In []: run  test_control_flow.py -v
Trying:
    even_or_odd(10)
Expecting:
    '10 is Even!'
ok
Trying:
    even_or_odd(5)
Expecting:
    '5 is Odd!'
ok
Trying:
    even_or_odd(3.2)
Expecting:
    '3 is Odd!'
ok
Trying:
    even_or_odd(-2)
Expecting:
    '-2 is Even!'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   4 tests in __main__.even_or_odd
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

UNIT TESTING, VI

- You can also run doctest "on the fly", without writing `doctest.testmod()` in the code by typing in a terminal: `python -m doctest -v your_function_to_test.py`
- For more complex testing, see documentation of `doctest` at `docs.python.org/2/library/doctest.html` , the package `nose` and the package `unittest`
- Start testing as early as possible!
- But don't try to test everything either!
- Easier to test if code is compartmentalized into functions (*why?*)
- R has formal testing framework, but the function `stopifnot()` can be useful
- OK, so you unit-tested, let's go look at life through beer-goggles... BUT NO! WHAT IS THIS I SEE?! A BLOODY BUG!

DEBUGGING I

- Bugs happen! You need to find and debug them
- Banish all thoughts of littering your code with `print` statements to find bugs — enter the debugger
- The command `pdb` turns on the python debugger
- Type the following in a file and save as `debugme.py` in your `Code` directory

```
def createabug(x):
    y = x**4
    z = 0.
    y = y/z
    return y

createabug(25)
```

- Now run it

DEBUGGING II

```
In []: %run debugme.py
[lots of text]
createabug(x)
      2     y = x**4
      3     z = 0.
----> 4     y = y/z
      5     return y
      6

ZeroDivisionError: float division by zero
```

- OK, so let's `%pdb` it

```
In []: %pdb
Automatic pdb calling has been turned ON

In []: run debugme.py
[lots of text]
ZeroDivisionError: float division by zero
> createabug()
      3     z = 0.
----> 4     y = y/z
      5     return y

ipdb>
```

DEBUGGING III

Now we're in the debugger shell:

- `n` move to the next line
- `ENTER` repeat the previous command
- `s` "step" into function or procedure (i.e., continue the debugging inside the function, as opposed to simply run it)
- `p x` print variable x
- `c` continue until next break-point
- `q` quit
- `l` print the code surrounding the current position (you can specify how many)
- `r` continue until the end of the function

DEBUGGING IV

- So let's continue our debugging:

```
ipdb> p x
25
ipdb> p y
390625
ipdb> p z
0.0
ipdb> p y/z
*** ZeroDivisionError: ZeroDivisionError
('float division by zero',)
ipdb> l
      1 def createabug(x):
      2     y = x**4
      3     z = 0.
----> 4     y = y/z
      5     return y
      6
      7 createabug(25)

ipdb> q

In []: %pdb
Automatic pdb calling has been turned OFF
```

DEBUGGING WITH BREAKPOINTS I

- You may want to pause the program run and inspect a given line or block of code (*why?* — impromptu unit-testing is one reason)
- To do so, simply put this snippet of code where you want to pause and start a debugging session and then run the program again:

```
import ipdb; ipdb.set_trace()
```

- Or, `import pdb; pdb.set_trace()`
- Alternatively, running the code with the flag `%run -d` starts a debugging session from the first line of your code (you can also specify the line to stop at)
- If you are serious about programming, please start using a debugger (R, Python, whatever...)!

PRACTICAL 1 I

- **As always, add and commit all your new (functional) code and data to the version control repository**: `basic_io.py`, `basic_csv.py`, `testout.csv`, `bodymass.csv`, `tesp.p`, `test_control_flow.py`, `debugme.py`, `profileme.py`

PRACTICAL 1 II

Now,

1. Open and run the code `test_oaks.py` — there's a bug, for no oaks are being found! (where's `TestOaksData.csv`?)
2. Fix the bug (hint: `import ipdb; ipdb.set_trace()`)
3. Now, write doctests to make sure that, bug or no bug, your `is_an_oak` function is working as expected (hint: `>>> is_an_oak('Fagus sylvatica')` should return `False`)
4. If you write a good doctest, you will note that you found another error that you might not have just by debugging (hint: what happens if you try the doctest with 'Quercuss' instead of 'Quercus'?)
5. How would you fix the new error you found using the doctest?

READINGS/RESOURCES

- Browse the python tutorial: `docs.python.org/tutorial/`
- For functions and modules: `learnpythonthehardway.org/book/ex40.html`
- For IPython: `ipython.org/ipython-doc/stable/interactive/tips.html` `wiki.ipython.org/Cookbook`