PHYS4038/MLiS and AS1/MPAGS

Scientific Programming in python™

mpags-python.github.io

Steven Bamford

University of Nottingham
UK | CHINA | MALAYSIA

An introduction to scientific programming with

python™

Session 2:

Introduction to Python, continued

Python 2.x versus 3.x

- The 3.x branch was released in 2008
- Intentionally backwards incompatible
- Various improvements, removal of obsolete code, …
- But considered quite annoying by many!

- print a ➞ print(a)
- 1/3 == 1//3 == 0 ➞ 1/3 == 1.0/3.0 == 0.333…, 1//3 == 0
- … various others …

- Python 2 now considered completely deprecated – don’t use it!

- But there is still a lot of legacy Python 2 code around
- Tools exist to help with conversion, e.g. 2to3

Continue looking at basic Python,

but getting a bit more advanced…

List comprehensions

- A short, readable way of creating lists without writing a loop

```
fruit = ['  banana', '  loganberry ', 'passion fruit  ‘]
# conventional loop
newfruit = []
for f in fruit:
    newfruit.append(f.strip())
print(newfruit)
['banana', 'loganberry', 'passion fruit’]

# or in one line
newfruit = [f.strip() for f in fruit]
print(newfruit)
['banana', 'loganberry', 'passion fruit']
```

List comprehensions

- A short, readable way of creating lists without writing a loop

```
my_fav_num = [3, 17, 22, 46, 71, 8]

# extract all even values and square
even_squared = []
for n in my_fav_num:
    if n%2 == 0:
        even_squared.append(n**2)

# in one line:
even_squared = [n**2 for n in my_fav_num if n%2 == 0]

# both produce [484, 2116, 64]
```

Basic file I/O

```
>>> fname = 'myfile.dat'

>>> # read whole file in one go
>>> f = open(fname)
>>> lines = f.readlines()
>>> f.close()

>>> # read individual lines
>>> f = open(fname)
>>> firstline = f.readline()
>>> secondline = f.readline()

>>> f.seek(0)  # go back to start of file without re-opening

>>> # iterate over lines
>>> for l in f:
>>>     print(l.split()[1])
>>> f.close()
```

Basic file I/O

```
>>> # write out to a file
>>> outfname = 'myoutput'
>>> outf = open(outfname, 'w')  # second argument denotes writable
>>> outf.write('My very own file\n')
>>> outf.close()

>>> # append to a file
>>> outf = open(outfname, 'a')  # second argument denotes append
>>> outf.write('Another line\n')
>>> outf.close()

>>> # better to use a 'context manager'
>>> with open(outfname, 'w') as outf:
...     outf.write('My very own file\n’)
...
>>> # don't need to remember to close the file
```

Context managers often used for resource access

Better reading and writing of scientific data covered later…

Scope, functions, globals and classes

```
# This is a global variable
a = 0

if a == 0:
    b = 1  # another global variable

def my_function(c):
    # this is a local variable
    d = 3
    print(a, b)
    print(c, d)

# Call the function
my_function(7)

# a and b still exist
print(a, b)
# c and d don't exist anymore
print(c, d)
```

Scope, functions, globals and classes

- When trying to use functions correctly, common to end up passing lots of arguments around
    - e.g.,

```
def print_particle(index, x, y, z, vx, vy, vz, m0, q,
                   ke, name, decay_rate, lorentz_factor)
```

- One solution: global variables

Scope, functions, globals and classes

```
a = 0

def good_func(b):
    a = 2 * b  # this is a local 'a'
    print('In good_func a is', a)

good_func(7)
print('After good_func a is', a)

def hacky_func(b):
    global a
    a = 2 * b  # this is the global 'a'
    print('In hacky_func a is', a)

hacky_func(7)
print('After hacky_func a is', a)
```

```
Output:
In good_func a is 14
After good_func a is 0
In hacky_func a is 14
After hacky_func a is 14
```

Scope, functions, globals and classes

- When trying to use functions correctly, common to end up passing lots of arguments around
    - e.g.,
      ```
      def print_particle(index, x, y, z, vx, vy, vz, m0, q,
                         ke, name, decay_rate, lorentz_factor)
      ```
- One solution: global variables
    - **best avoided**
- Better solution: classes

Classes

```
>>> class MyLineClass(object):
...     def __init__(self, m, c=0.0):
...         self.m = m
...         self.c = c
...
...     def calc(self, x):
...         return self.m * x + self.c
...
```

```
>>> m = MyLineClass(2, 4)
>>> m.calc(0)
4
>>> m.calc(3)
10

>>> m.c
4

>>> m.c = 3
>>> m.calc(3)
9
```

```
>>> p = MyLineClass(5, 1)
>>> p.calc(0)
1
>>> p.calc(3)
16


>>> m.calc(3)
9

>>> print(m)
<__main__.MyLineClass
   object at 0x10c53d810>
```

Classes

```
>>> class MyLineClass(object):
...     def __init__(self, m, c=0.0):
...         self.m = m
...         self.c = c
...
...     def __call__(self, x):
...         return self.m * x + self.c
...
...     def __str__(self):
...         return f'{self.m} * x + {self.c}'
...

>>> m = MyLineClass(2, 5)

>>> m(1)
7

>>> print(m)
2 * x + 5
```

If you want to know more, search for object oriented programming in Python

Modules

- Modules can contain any code
- Classes, functions, definitions, immediately executed code
- Can be imported in own namespace, or into the global namespace

```
>>> import math
>>> math.cos(math.pi)
-1.0

>>> math.cos(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined

>>> from math import cos, pi
>>> cos(pi)
-1.0

>>> from math import *
```

Your own modules

- Simply put code in a file with extension .py
- Import it in interpreter or another file
- Entire file immediately executed; objects defined in the file available

```
>>> import mymod
>>> mymod.dostuff()
```

- Reloading

```
>>> from importlib import reload
>>> reload(mymod)          # if you change the module code
```

- IPython autoreloading

```
>>> %load_ext autoreload
>>> %autoreload
```

Your own modules

- Can also collect files together in folders

```
myphdmodule/
   __init__.py                      contains code run upon import
   backgroundreader.py
   datareducer.py
   resultsinterpreter.py
   thesiswriter/                    submodule
        introwriter.py
        bulkadder.py
        concluder.py
```

```
>>> from myphdmodule import thesiswriter
>>> thesiswriter.concluder()
```

Executable scripts

```
#! /usr/bin/env python
import sys

def countsheep(n):
    sheep = 0
    for i in range(n):
        sheep += 1
    return sheep

if __name__ == "__main__":
    if len(sys.argv) > 1:
        n = int(sys.argv[1])
    else:
        n = 100
    s = countsheep(n)
    print(s)
```

`__name__ == "__main__"` is True only when module is run from the command line

Executable scripts (better)

```python
#! /usr/bin/env python
import sys

def countsheep(n):
    sheep = 0
    for i in range(n):
        sheep += 1
    return sheep

def main():
    if len(sys.argv) > 1:
        n = int(sys.argv[1])
    else:
        n = 100
    s = countsheep(n)
    print(s)

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

You will also need to make the file executable in Linux or OSX:

```
$ chmod u+x countsheep.py
$ ./countsheep.py
```

For more sophisticated parsing of command line arguments, see the documentation for `argparse`.

Documentation and examples

```
# My totally wicked function
def my_func(x, y=0.0, z=1.0):
    """This does some stuff.

    For example:
    >>> my_func(1.0, 3.0, 2.0)
    8.0

    Yes, it really is that good!
    """
    return (x + y) * z
```

- Comments before function, class, etc. are used to generate help
- "Docstrings"
  - preferred way of documenting Python code
  - automatically generate documentation (e.g. with pydoc or sphinx)
  - can contain examples, which can be automatically turned into tests!
    - see doctest module (more about testing later…)

Writing nice Python...

Coding guidelines

- **PEP8**
  - https://www.python.org/dev/peps/pep-0008/
  - several tools to test/enforce compliance – search 'pep8'
    - (I like Black: https://github.com/psf/black)
  - 4 spaces for indentation
  - 79 characters maximum line length (72 for docstrings)
    - format continuation lines consistently
  - use blank lines sparingly, but always around functions, etc.
  - whitespace
    - one space around operators (except = in function args)
    - after commas
  - ClassNames and function_names

Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Zen of Python

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Good programming practice

- Compromise between:
  - producing results quickly

    versus
  - easy reusability and adaptation of code
  - code that can be quickly understood

- Use meaningful names for variables, functions, etc.

```
r = 4.594 * l / A
```

versus

```
resistivity = 4.594
resistance = resistivity * length / area
```

Good programming practice

- Compromise between:
  - producing results quickly

    versus
  - easy reusability and adaptation of code
  - code that can be quickly understood

- Comment where code alone is unclear or to add details
- e.g.,

```
# Si doped with 10^15 atoms/cm^3 As at STP
resistivity = 4.594  # Ohm cm
```

Good programming practice

- Compromise between:
  - producing results quickly

    versus
  - easy reusability and adaptation of code
  - code that can be quickly understood

- Comment where code alone is unclear or to add details

- but better to make the code itself readable

```
# resistance is resistivity * length / area
r = 4.594 * l / A
```

versus

```
resistivity = 4.594
resistance = resistivity * length / area
```

Good programming practice

- Compromise between:
  - producing results quickly

    versus
  - easy reusability and adaptation of code
  - code that can be quickly understood

- Use functions

```
res_Ge_init = rho_Ge * length_init / area_init
res_Ge_final = rho_Ge * length_final / area_final
res_factor_Ge = res_Ge_final / res_Ge_init
#...
res_Si_init = rho_Si * length_init / area_init
res_Si_final = rho_Si * length_final / area_final
res_factor_Si = res_Si_final / res_Si_init
```

Good programming practice

- Better…

```
def calc_resistance(rho, length, area):
    return rho * length / area

def calc_res_factor(rho, length_init, length_final,
                    area_init, area_final):
    """ Calculate the multiplicative change in resistance."""
    res_init = calc_resistance(rho, length_init, area_init)
    res_final = calc_resistance(rho, length_final, area_final)
    return res_final / res_init

res_factor_Ge = calc_res_factor(rho_Ge,
                                length_init, length_final,
                                area_init, area_final)
#...
res_factor_Si = calc_res_factor(rho_Si,
                                length_init, length_final,
                                area_init, area_final)
```

Good programming practice

- Compromise between:
  - producing results quickly

    versus
  - easy reusability and adaptation of code
  - code that can be quickly understood

- Use modules
  - i.e. put code in different files and import where required
  - e.g. group utility functions into separate file
  - do not put every function in its own file
  - try to group code into standalone units
  - individual modules may be reused in different projects

Good programming practice

- Compromise between:
  - producing results quickly

    versus
  - easy reusability and adaptation of code
  - code that can be quickly understood

- **Science, not software development**
- You are your main co-developer and end user
- Write the code to do what you want, but…
- Keep your code DRY (don't repeat yourself)
- If your code is getting messy, take some time to 'refactor' it
- Future you will be happy you did!
- Keep your installed modules tidy and use version control