Physics 514 – Basic Python Intro, Part II

Emanuel Gull

September 14, 2015

1 Introduction

This is part II of the python introduction. By now you should have installed Python on your computer and know how to use python as a pocket calculator. Today we will introduce some more syntax and conclude with some exercises. Next week we will have a look at some of the useful scientific libraries.

2 Flow Control (if, for, while)

2.1 Booleans and Comparisons

Boolean values (true, false) are another built-in data type:

```
>>> 1<4
True
>>> 5<4
False
>>>
```

We can use Booleans for comparison:

```
>>> a=5
>>> b=3
>>> a==b
False
>>> b=5
>>> a==b
True
```

Note here that the double `==` sign is a comparison operator, different from the single `=` sign which is an assignment operator. There is also an operator `is` for comparison. Note that `is` is subtly different from `==`: while `==` tests if the values are equal (value equivalence), `is` tests if the *objects* are identical. (Think of it like this: `==` checks if the value is the same, `is` if the memory address is the same). Example:

```
>>> L=[1,'x',4.]
>>> Q=[1,'x',4.]
>>> L==Q
True
>>> L is Q
False
>>> P=Q
>>> P is Q
True
>>> P is L
False
>>>
```

2.2 Conditional Statements: *if* and *else*

With this we have enough for a simple conditional statement. Python uses indentation (rather than brackets) to align code, so please pay attention that your code is always properly indented!

```
if x:
  if y:
    print z
  else:
    print q
else:
  print r
```

2.3 *while* loop

The statement for a `while` loop is:

```
x=10
while (x<20):
  print x
  x=x+1
```

Another example:

```
>>> x='spam'
>>> while x:
...   print x,
...   x=x[1:]
...
spam pam am m
>>>
```

There is no do ... while statement. Three additional keywords exist: `break`, if found inside a loop, exits the loop. `continue` jumps back to the top of the loop, and `pass` does nothing at all: it is an empty placeholder.

Advanced: write an example of a loop that uses `break`, `continue`, *and* `pass`.

2.4 *for* loops

The python `for` loop is:

```
for <target> in <object>:
  <statements>
```

for example:

```
>>> for x in ['spam', 'eggs', 'ham']:
...   print x,
...
spam eggs ham
```

This also works with strings, tuples, and so on:

```
>>> for x in 'lumberjack':
...   print x,
...
l u m b e r j a c k
```

The same, in one line:

```
>>> for x in [1,2,3,4]: print x**2,
...
1 4 9 16
```

In this context, the `range` function is very useful:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in range(5): print i,
...
0 1 2 3 4
```

Exercise: Program the sieve of Eratosthenes:

```
Input: an integer n > 1

Let A be an array of Boolean values, indexed by integers 2 to n,
initially all set to true.

for i = 2, 3, 4, ..., sqrt(n) :
  if A[i] is true:
    for j = i2, i2+i, i2+2i, ..., n:
      A[j] := false

Now all i such that A[i] is true are prime.
```

Exercise: Estimate `cos(x)` *by using the Taylor series to various orders. create a table of* `x` *and* `cos(x)` *values, compare them against the* `cos(x)` *function, make a plot in your favorite plotting program.*

3 Functions and Copying

functions in python are defined as follows:

```
def <name>(arg1, arg2, ..., argN):
  <statements>
```

In case of a return statement:

```
def <name>(arg1, arg2, ..., argN):
  <statements>
  return <value>
```

Here are some examples:

```
>>> def square(x):
...   return x**2
...
>>> square(3)
9
```

You can return tuples (more later):

```
>>> def square_and_cube(x):
...   return x**2,x**3
...
>>> square_and_cube(2.)
(4.0, 8.0)
```

The intersection of two strings:

```
>>> s1="SPAM"; s2="SPQM"
>>> def intersect(seq1, seq2):
...   res=[]
...   for x in seq1:
...     if x in seq2:
...       res.append(x)
...   return res
...
>>> intersect(s1,s2)
['S', 'P', 'M']
>>> set(s1) & set(s2)
set(['P', 'S', 'M'])
```

The last two lines compute an intersection using the built-in `set` type and the `&` operator, `help(set)` will give more information.

Function calls in Python are calls 'by reference' (by assignment, strictly, which creates a reference). A consequence of this is:

```
>>> def mysort(L):
...   L.sort()
...
>>> Q=['a','x','r','t']
>>> Q
['a', 'x', 'r', 't']
>>> mysort(Q)
>>> Q
['a', 'r', 't', 'x']
>>>
```

This is different from e.g. `C++`, where a function call by default is a 'call by value':

```
egull$ cat call.cc -o call
#include<iostream>
#include <algorithm>
#include<vector>

void mysort_by_value(std::vector<double> v){ std::sort(v.begin(), v.end());}
void mysort_by_reference(std::vector<double> &v){ std::sort(v.begin(), v.end());}
int main(){
  std::vector<double> v(3); v[0]=3.; v[1]=1; v[2]=2;
  mysort_by_value(v);
  std::cout<<v[0]<<" "<<v[1]<<" "<<v[2]<<std::endl;
  mysort_by_reference(v);
  std::cout<<v[0]<<" "<<v[1]<<" "<<v[2]<<std::endl;
}
egull$ g++ -o call call.cc
egull$ ./call
3 1 2
1 2 3
egull$
```

If you want to make sure that you pass a copy, you can use the `copy` command:

```
>>> import copy
>>> Q=['a','x','r','t']
>>> mysort(copy.copy(Q))
>>> Q
['a', 'x', 'r', 't']
>>>
```

Advanced: If you know the difference between shallow and deep copy: `copy` *does a shallow copy, use* `deepcopy` *for a deep copy* Note you can't just assign another variable first:

```
>>> Q=['a','x','r','t']
>>> P=Q
>>> mysort(P)
>>> Q
['a', 'r', 't', 'x']
```

The first `=` creates a reference of Q, which is then changed in the function. In other words: `P` and `Q` always point to the same memory!

4 Tuples

Tuples are very similar to lists. The only difference is that tuples are immutable, whereas lists are mutable. As a consequence: tuple assignments are not possible:

```
>>> 1,2,5,4,3
(1, 2, 5, 4, 3)
>>> z=1,2,5,4,3
>>> z
(1, 2, 5, 4, 3)
>>> z[2]
5
>>> z[2]=4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

This may be useful when you want to make sure that a list is not changed when it is being passed to a function. A `const` keyword does not exist in python. You can use the usual conversion routines to change lists to tuples and back:

```
>>> z=(1,2,3,4,5,6); print z
(1, 2, 3, 4, 5, 6)
>>> p=list(z); print p
[1, 2, 3, 4, 5, 6]
>>> q=tuple(p); print q
(1, 2, 3, 4, 5, 6)
>>>
```

You do not really need the brackets, here is a definition just with commas:

```
>>> z=3,2,5,4,6; print z
(3, 2, 5, 4, 6)
```

5 Files

Python contains the standard set of functions to access files on your hard drive: `open`, `read`, `write`, and `close`. `read(N)` reads the next N bytes, `readline` a line, including the end of line character; `write` and `writeline` are analogous to the `read` commands. `seek(N)` changes the file position offset to `N`, and `flush` writes all data to disk (without closing the file). Some examples:

5.1 writing

```
>>> myfile=open('myfile', 'w')
>>> myfile.write('hello text file\n')
>>> myfile.close()
```

Writing numbers to file:

```
>>> F=open('datafile.txt', 'w')
>>> F.write('%s,%s,%s\n' %(43,44,45))
>>> F.close()
```

the file now looks like this:

```
egull$ cat datafile.txt
43,44,45
```

5.2 reading

```
>>> myfile=open('myfile', 'r')
>>> myfile.readline()
'hello text file\n'
>>> myfile.readline()
''
>>>
```

One way of reading back the set of numbers: read a string, remove the trailing `\n`,then split it:

```
>>> G=open('datafile.txt')
>>> L=G.readline().strip().split(',')
>>> L
['43', '44', '45']
```

5.3 pickling objects

Python has a built-in storage routine called 'pickle'. Pickle can handle any native data type, e.g. dictionaries:

```
>>> import pickle
>>> D={'a':1, 'b':4, 'c':7}
>>> H=open('pickle.txt','w')
>>> pickle.dump(D, H)
>>> H.close()
```

The file is not really human readable:

```
egull$ cat pickle.txt
(dp0
S'a'
p1
I1
sS'c'
p2
I7
sS'b'
p3
I4
s.
```

...but the data can easily be loaded again:

```
>>> H=open('pickle.txt')
>>> J=pickle.load(H)
>>> J
{'a': 1, 'c': 7, 'b': 4}
>>>
```

Binary storage is especially useful for large numerical data:

```
>>> import random
>>> J=[random.random() for x in xrange(1000)]
>>> K=open('pickle', 'wb')
>>> pickle.dump(J,K, protocol=2)
>>> K.close()
```

The 'protocol=2' flag switches to binary pickling. Loading works as with ASCII pickling.

6 Modules

6.1 creating and loading modules

We already know how to write functions, and how to write programs into file and run them from the command line. It is often convenient to split up a project into multiple files. These files then have to be `import`ed when functions in them are needed. Modules can have any name, but usually they end in `.py`. Here is a first example of a module, stored in file `module1.py`:

```
egull$ cat module1.py
def printer(x)
  print x
```

Within python, that module can be imported and functions called as follows:

```
>>> import module1
>>> module1.printer("hello, world!")
hello, world!
>>>
```

We can already check the functions available in this module:

```
>>> dir(module1)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'printer']
>>>
```

So far only the one function `printer` shows up, along with some auxiliary variables:

```
>>> print module1.__file__
module1.py
>>> print module1.__name__
module1
>>> print module1.__doc__
None
>>> help(module1)
```

Instead of importing the entire `module1` we can import just the one function `printer`, which then shows up as `printer`, instead of `module1.printer`:

```
>>> from module1 import printer
>>> printer("hello, world!")
hello, world!
```

A typical usage scenario is the math library:

```
>>> from math import pi,e,cos,sin,tanh
>>> sin(pi/2)
1.0
```

importing all functions into the current namespace saves some typing:

```
>>> from math import *
>>> log(2.)
0.6931471805599453
```

There are various pitfalls, most of which can be avoided in if the namespaces are not collapsed (see e.g. the module gotchas in the book).

7 Classes

Object oriented programming is a large and important topic in computer science. We will not introduce the concept in detail and we will skip most of the defining aspects (no polymorphism, inheritance, information hiding, ...). If you know about these topics, chapter V will be useful. Otherwise regard a class as a container for data and functions. This defines a class with two public member functions:

```
>>> class FirstClass:
...   def setdata(self, value):
...     self.data=value
...   def display(self):
...     print self.data
```

The line `self.data=value` sets the member variable `data` of the class to `value`. Note that these member variables need not be declared (unlike in `C++`!). The keyword `self` refers to the current instance of the class (`*this` in other languages). We then create two instances:

```
>>> x=FirstClass()
>>> y=FirstClass()
```

...and call their member functions to assign and print values:

```
>>> x.setdata("King Arthur")
>>> y.setdata(random.random())
>>> x.display()
King Arthur
>>> y.display()
0.426646841786
```

We can use classes like `structs` in `C`: as data fields without functions. The keyword `pass` is needed to specify that the class has no member functions:

```
>>> class rec: pass
>>> A=rec()
>>> A.data
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: rec instance has no attribute 'data'
>>> A.data=2
>>> A.data
2
```

8 Exercises

- Write a rectangular integration function. Integrate `cos` from zero to `pi`.

- Write a trapezoidal and a Simpson rule. Show that the convergence of the trapezoidal rule is $O(1/N^2)$, the one of the Simpson rule is $O(1/N^4)$, where N is the number of integration points. Produce a plot with log-log axes showing the polynomial scaling as a function of N.
- Write a class `func` implementing a function f. Change your Simpson integration to use `func.f` ; test your integration function with a class that implements f for complex and real functions.
- *Advanced:* Use inheritance to pass arbitrary functions by deriving from `func` and overloading f.
- *Advanced:* Overload the bracket () operator and use this one instead of f (check `__call__`).
- *Advanced:* Instead of calling a class member function, pass the function to be integrated as an argument to your integration routine and test it for `sin` and `cos`.