Introduction to Programming with Python

Written by Dave Pawlowski, October 3, 2011

Indroduction

There are several reasons that I chose Python for this course over other, more typically used languages like C or C++. For one, though it may not seem like it now, there is a very wide range of programming tasks that you may be asked to do in your careers. Today, python is used more broadly than most other languages out there. It is used for sophisticated scientific programming, it is used for unix scripting tasks, it is used for web application implementation. Python can even be extended in C and C++.

Another reason that I choose Python is that it is both an interactive language, and it is an interpreted language. You are familiar with interactive languages– Gnuplot. You can directly interact with Python via the Python prompt. This makes testing code snippits and debugging your code much easier than with a language that is not interactive. So what is meant by "interpreted language"? There are two types of programming languages, interpreted and compiled. A compiled language is one where you write a code, save it in a file, and then run a set of commands using a machine specific compiler that converts the code that you wrote into code that the computer can understand. The high-level code that you write is called the souce code, and the translated program is called object code or the executable. Fortran and C++ are examples of compiled languages.

Python is an interpreted language, which means that the code is executed by the programming language itself, not the processor. Usually, the language processes the program a little at a time, alternately readling lines and performing computations. The cost of this is that interpreted languages tend to be slower than compiled languages. But the advantage is that interpreted languages tend to be platform independent.

Many modern languages are both compiled and interpreted. They are first compiled into a lower level language called byte code, and then interpreted by a program called a virtual machine. Python actually does this, but because of how programers interact with it, it is usually considered an interpreted language.

Finally, the last reason we will be using python in this course is that it is free. You can download and run python on any system. Because of this, you will find it pre–installed on most non–windows computers. Macs come with a python installation, and the EMU systems yipe and emunix both have python installed.

Object oriented programming

Python is designed to be used as an object oriented language. We probably wont be using many of these features, but it is important to know what this means. There are several different types

of programming styles. One of them is object oriented programming (OOP). OOP is a type of programming in which programmers define not only the data type of a data, but also the types of operations that can be applied to the data. In this context, data are treated as objects, in that an object is something that includes both data and functions (in OOP, functions can be called methods). Think about it this way: say you were writing a program that uses information about a group of people. It is logical to treat a person as an object, because you would define characteristics and maybe functions that use these characteristics that apply to a person object. For example, you might declare a method that sets the height of a person. By defining a person as an object, your program can create new people that automatically "inherit" the methods and characteristics that belong to the person class.

The other common programming style is called modular programming. Modular programming has been around a lot longer than OOP. In modular programming, code is written such that code that may be used more than once is broken up into functions or procedures that can be called when needed. In many cases, these things are placed in to seperate files, so that the function or procedure can be used by multiple programs. You are expected to be able to use modular programming style, but if you want, and find it useful to use OOP you may. This may require some additional work on your part, because as I mentioned, I won't be spending much time, if any, on OOP techniques.

I mentioned OOP up front because one of the things that makes python so popular is that everything in Python is an object. In other OOP languages, this isn't necessarily the case, which means that some data structures may be able to have methods associated with them, while others didn't. In Python, every data structure is an object. Whether you actually use them as objects is up to you.

Python Basics

Indentation

Lets get this out of the way up front:

Python uses indentation to show block structure. Indent one level to show the beginning of a block. Out–indent to show the end. For example:

```
if (x)
   if (y)
      print 'yes and yes'
   else
      print 'yes and no'
else
   print 'no'
end
```

In Python, this would be:

```
if x:
   if y:
      print 'yes and yes'
   else:
      print 'yes and no'
else:
   print 'no'
```

The number of spaces for each level of indentation is typically 4, not a tab (though it actually doesn't matter). Emacs has a "python mode" that will correctly indent for you if you use tab!

Starting Python

Ok, now lets see how to actually use Python.

There are 2 ways to use python: *shell mode* and *script mode*. In shell mode you type statements into the Python shell and the interpreter immediately prints the result. In the command line, to start python type:

```
% python
```

You will see a little information about the version and be given access to the Python shell.

```
{Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1+1
```

The default python prompt is >>> and lets you know that Python is waiting for instructions. Use C-d to exit the Python shell.

Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file, i.e. write a Python script. Create a file called first.py (the .py ending is used by convention, but as usual is not mandatory). In the file add the line:

```
print 1+1
```

You can execute the program by typing:

```
% python first.py
```

and python should output "2" before returning control to the unix shell.

Let's write another Python program, call it second.py:

second.py

```
#!/usr/bin/python
#This is a comment.  This program says something,
#sets a few variables, and does some math.

print "Hello World!"

x = 1.23e7
print x

x = 1
y = 2

print x/y

y = 2.

print x/y
```

This time, instead of running this like we did first.py, do this:

```
% chmod u+w second.py
% second.py
```

The first line of the program tells the unix shell to use python to interpret the file, just like when you used csh.

Now, the code. Setting variables is done very logically, just use the equals symbol to assign a number, string, result of an expression, etc to a variable name. In Python, you don't have to define the variable before you assign a variable to it (you do in other languages like C++ and Fortran).

Something strange happens when you do the first division, x/y; python says that the result of 1/2 is zero! This is something called integer division. Since you assigned the variables x and y the integers 1 and 2, python assumed that you wanted an integer when you divided the two numbers. Almost all programming languages would give you a zero here. This is because it is costumary to round down. Whenever you divide 2 numbers, you should always use floating–point division– make sure that the denominator is a float (unless you know you want to do integer division)..

Usually, you make a number a float by adding a decimal point. We'll talk later about how to convert a variable that was originially an integer into a float. In the Python shell, try out the type function:

```
>>> type(1.0)
```

This tells you the type of variable entered in the (), in this case, 1.0 is a float, which is sort for floating point. A floating point is a method of representing real numbers in a way that can support a wide range of variables. Floating point refers to the fact that the decimal point can float– be placed anywhere relative to the significant digits of the number. The alternative is fixed point notation, where the decimal point is always in the same place.

When y is reassigned to a float, 2., second.py gives you 0.5 as the result of the x/y expression, as expected. In the last line, the operator is used. That is the exponentiation operator.

Try the type function on a few other cases:

```
>>> type('hello!')
>>> type(5)
```

Variable names and keywords

Programmers usually choose names for their variables that are meaningful– the document what they are used for. The names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. By convention, only lower case letters are used for variable names, but that choice is up to you. But remember– case matters. Dave and dave are not the same variables.

The underscore character (_) is allowed in a name, and can be useful for for names with multiple words. Illegal names will give you a syntax error:

```
>>> 16denard = "awesome"
  File "<stdin>", line 1
    16denard = "awesome"
           ^
SyntaxError: invalid syntax
>>> denard$ = 'Good?'
  File "<stdin>", line 1
    denard$ = 'Good?'
          ^
SyntaxError: invalid syntax
>>> class = "fun?"
  File "<stdin>", line 1
    class = "fun?"
          ^
SyntaxError: invalid syntax
```

As mentioned, variable names can't start with a number, which is why the first example gives you an error. In the second example, an error is returned because the dollar sign is an illegal character. The third example uses the Python keyword "class" as a variable name. In each case, Python tries to help you figure out what the error is by telling you the type of error, and the line in which it occured. The ^ symbol even tells you where in the line the issue is. But be careful, these may not always tell you what the error is. Many times, this is where the error is caught,

but the error itself may actually be earlier in the code.

Input

There are two built–in functions in Python for getting keyboard input. Try this script:

```
n = raw_input("Please enter your name: ")
print 'Hi ' + n
n = input("Enter a number: ")
print n
```

Running this should give you something like this:

```
dpawlows@casey: ~/Phy379 >> python input.py
Please enter your name: Dave
Hi Dave
Enter a number: 16
16
```

In the script, you refer to the variable using only the variable name, not $ like in the unix shell. This means that you should never 'quote' variables. In this example, we use '+' to add the string 'Hi' to n. In this context, there is no addition taking place. Since the plus sign is operating on two strings, it is used to perform **concatenation**, which means join together the two operands by linking them end–to–end.

Operations on strings

In general, you can't perform mathematical operations on strings, even if the strings look like numbers:

```
>>> print 5 + '6'
```

will give you an error, since '6' is a string and 5 is an int. As mentioned though, you can use the plus sign for two strings:

```
>>> team = 'Michigan'
>>> status = ' is good'
>>> print team+status
```

Does in fact do what you would expect. In addition the operator works on strings:

```
>>> print team*5
```

You will see that the symbol causes repetition.

Composition

So how do you put all of these elements of a program together: variables, expressions, statements, etc.

One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>> print 17 + 3
```

In reality, the addition has to happen before the printing, so the actions arent actually happening at the same time. The point is that any expression involving numbers, strings, and variables can be used inside a print statement.

```
>>> hour = 10
>>> minute = 13
>>> print "Number of minutes since midnight: ",hour*60+minute
```