CS102: Introduction to Python

The goal of this topic is to provide a brief introduction to **Python** to give you a feel for a language other than C. In many ways, Python is very different from C. It is generally considered to be a **scripting language**, although the distinction between scripting languages and other programming languages is not really clear-cut. Scripting languages tend to be interpreted rather than compiled; they tend not to require declarations of variables (the interpreter just figures it out); they tend to hide memory management from the programmer; they tend to support regular expressions; etc. In terms of usage, scripting languages tend to be useful for writing short programs quickly when you don't care too much about efficiency. Other languages that are typically considered to be scripting languages include Perl, Awk, and JavaScript.

Python supports several styles of programming, including (but not limited to) procedural programming (like C and C++), object-oriented programming (like C++ and Java), and functional programming (like Lisp). Note that it is not a mistake to include C++ in two categories, just as it is not a mistake to include Python in all three of these categories. The first version of Python was released in the late 1980s. Python 2.0 was released in 2000, and various improvements have been made in the Python 2.x chain of releases since that time. Python 3.0 was released in 2008, and again, various improvements have been made in the Python 3.0 chain of releases. Unfortunately, Python 3 is not backwards compatible with Python 2, and there seems to be debate as to which is the better version of Python to learn. On the one hand, in the future, Python 2 might become deprecated, and you might have to learn Python 3 to program in Python eventually. On the other hand, most of the available Python code on-line seems to use Python 2, there are some important libraries that have not yet been ported to Python 3, and some environments including Linux and Cygwin come with Python 2. All things considered, we will be exploring Python 2 (at the time this is written, the latest version is Python 2.7, and that is the version that seems to come installed with Cygwin and Ubuntu). We are also only considering procedural programming in Python, the goal being to get a general feel for how the language works. Interested students are encouraged to explore the language further on their own.

Let's look at our first Python program (sometimes called a **script**). As is customary, we will start with a simple program to display "Hello World!" to standard output on its own line. Here it is:

```
print "Hello World!"
```

That's it. The string itself can be surrounded by either double quotes or single quotes. (I would say that another common property of scripting languages is that there tends to be more flexibility, giving you multiple ways to do exactly the same thing. This is a good thing if you want to choose your own style, but it can also be bad because there is less consistency between code samples.) To avoid an automatic newline character after the printed string, you could place a comma after the ending quote, although this would still add a space at the end of the string.

Let's say you have used a text editor to create a file containing this program and named it "hello.py" (where ".py" is the common extension for a Python script). To run the program from a command line, you would just type "python hello.py", and assuming that Python is available, the program will run.

If you are coming from a background in which you have only used compiled languages such as C, this might seem strange. After all, we have learned that you can only run an executable that has been compiled to (or written in) the machine code recognized by the current environment. That is still true. The executable that you are running here is called "python". This type of executable is an **interpreter** (in this case, it is an interpreter for the Python language). If you want to know where the executable is stored on your system, in Linux/Unix/Cygwin you can type "which python" and it will tell you. In order for the system to find the interpreter in must be on your **path**. The path is a special environment variable (named "PATH") that tells the system where to look for executable files in order to run them. In Cygwin or Linux, for example, you can view the path and other environment variables by typing "printenv". If you only want to view the PATH variable, you can type "printenv | grep PATH". Here, "grep" is a separate Linux/Cygwin command to print all lines from the input that contain the specified pattern. (You can look for much more complex patterns using "grep", but we will not cover that.) The vertical bar, '|', designates a **pipe** that redirects the standard output of the command that is left of the pipe to the standard input of the command that is right of the pipe. There are also ways to set the path for different environments, but we will not cover that.

In the example above, you are running the Python interpreter, which in turn inspects and executes the commands specified in "hello.py", which is just a text file containing your Python code. Note that there is no equivalent to the "main function in a C program. Python, by default, just starts interpreting commands from top to bottom.

One more thing; if you are using Python 3, this simple program would not work! The "print" statement was removed from the language and replaced by a "print" function, so you need parentheses around the string. (I will not be pointing out differences in Python 3 in general, but I just wanted to give an example of something very basic that changed, and to remind students that Python 3 is not backwards compatible with Python 2.)

One other thing to point out is that, while there are integrated development environments (IDEs) that can be used to program with Python (e.g., Eclipse can be used with an appropriate plugin), it seems to me that it is more common to use a text editor and to run the interpreter from the command line. I am not exactly sure why, but this seems to be another common property of scripting languages. Alternatively, you can use Python in **interactive mode** by just typing "python" from the command line. This will bring you to the Python prompt (it will probably be ">>>"), and you can start typing commands. To exit interactive mode you can type "quit()".

Comments in Python start with "#", which is used similar to "//" in C. You can also have multiple-line comments that start and end with three quotes in a row (either single or double quotes), but this seems to be less common. (It is more common to start each line with "#".)

Python has more built-in **types** than a language such as C. These types include "int" (for integers), "float" (for floating point values), "complex" (for complex numbers), "str" (for strings), "list" (similar to a linked list, but it is implemented as a resizable array, and can store items of different types), "dict" and "set" (both implemented as hash tables, with "dict" having the ability to associate a value with each key), etc. There are many additional types, and every type supports

a lot of functionality beyond what we will be covering. The "dict" type is similar to associative arrays in some other languages, and the existence of such a type is also something that is common to scripting languages. In C, you would need to implement your own data structures to support some of these constructs (don't worry about it if you are not familiar with some of the concepts, such as hash tables). In languages such as C++ and Java, some of these constructs are supported with provided classes.

As in other languages, one very important concept is that of **variables**. Variables can store values of any of the provided types. Unlike most compiled languages, but similar to many other scripting languages, you do not have to declare variables in Python! The interpreter will determine the type of a variable to the left of an equal sign based on the value that is assigned to it. If you attempt to use an undefined variable to the right on an equal sign (either alone or as part of an expression), the interpreter will display an error message. In Python documentation, you will often see it stated that the variable name refers to an **object**; this is taken from object-oriented terminology. If you do not have experience with any object-oriented language, just remember that the object is the thing that the variable name refers to.

Below is an example of a Pythons script that sets and displays a few variables. This example also shows multiple ways to produce messages that combine strings and other types. The "str" function can convert integers or floats to strings; the '+' operator concatenates strings. The "print" statement can use codes that are similar to the "printf" function in C. Here is the script:

```
i = 10
f = 20.5
str1 = "Hello World!"

print "i = " + str(i)
print "i =", i # comma adds space
print "f = %.1f" % f
print "f =", f
print "str = " + str1
print "str = %s" % str1
```

The output looks like this:

```
i = 10
i = 10
f = 20.5
f = 20.5
str = Hello World!
str = Hello World!
```

Each text file containing Python code and ending in ".py" is called a **module**. We have already seen that within a module, variables can be assigned values and statements can be executed. The interpreter parses the lines of the module from top to bottom. Functions can also be defined within modules, and the lines within functions only get processed when the function is called. Variables that are assigned values outside of any function are **global variables** in the module.

A function definition in Python starts with the keyword "def" followed by the name of the function, then a parameter list in parentheses (if there are any parameters), followed by a colon. This is followed by a **block** of code (a.k.a. a **compound statement**) representing the statements within the function. Unlike C and most other languages, there is no special symbol to represent the start or end of a code block such as a function body. Rather, the interpreter pays attention to whitespace and considers indentation! Any inward indentation compared to the previous line starts a new block. When the indentation goes back to the previous level, this ends the block.

Global variables can be accessed within functions. If you do not assign the value of a global variable within some function, you can use or display the value of that variable without having to specify that the variable is a global. However, if the interpreter sees that a variable is set within a function, it is assumed to be a **local variable**. You cannot use or display the value of a local variable until it is assigned a value (that would lead to an error message from the interpreter). If you want to assign the value of a global variable within a function, you must specify that the variable is a global (with the keyword "global") at the start of the function. To explain this, consider the following Python module:

```
s = "Hello World!"

print "Location globalA: " + s

def func1():
    s = "Goodbye World!"
    print "Location func1A: " + s

def func2():
    global s
    print "Location func2A: " + s
    s = "Goodbye World!"
    print "Location func2B: " + s

print "Location globalB: " + s
func1()
print "Location globalC: " + s
func2()
print "Location globalD: " + s
```

When you run this script, the output is:

```
Location globalA: Hello World!
Location globalB: Hello World!
Location func1A: Goodbye World!
Location globalC: Hello World!
Location func2A: Hello World!
Location func2B: Goodbye World!
Location globalD: Goodbye World!
```

Let's try to understand what is happening here. First, the "globalA" print statement executes, displaying the global string "s"; that is straight-forward. Then come the two function definitions, but the code within these functions is not executed until the functions are called. So the next executed print statement displays the "globalB" message. Then "func1" is called. Since the value of "s" is assigned within this function, it is assumed to be a local variable (i.e., it is a separate variable from the global). We assign the variable a different string which gets displayed by the "func1A" print statement. Note that if there were a separate "print" statement before the assignment, this would lead to an interpreter error, because it would be interpreted as an attempt to display the value of a local variable before such a variable has a value assigned.

After "func1" returns, the "globalC" print statement is executed, and we see that the value of the global variable "s" has not been changed. This is followed by a call to "func2". Here, we specify that "s" within this function refers to the global "s". We display its value once at the "func2A" location, then change the value, then display the updated value. After "func2" returns, the final "globalD" print statement executes, and we see that the value of the global variable has indeed been changed.

Python does not allow you to change individual characters within a string; strings in Python are said to be immutable. However, it is very easy to access parts of strings, and with those, new strings can be created. Parts of strings can be obtained by placing ranges of indices within brackets; unlike C, the indices represent positions between characters (not the positions of the characters themselves). A 0 represents the position to the left of the first character, and for a string with N characters, N represents the position to the right of the last character. Consider the following example, which also shows that "\n" represents a newline character (as in C):

```
s1 = "Hello World!"
s2 = s1[:5]
s3 = s1[6:11]
s4 = s2 + " Planet" + s1[11:]

print s1+"\n"+s2+"\n"+s3+"\n"+s4
```

The output is:

```
Hello World!
Hello
World
Hello Planet!
```

Let's think about the last statement (involving the assignment to "s4") in more detail. In C, we would need to first compute the lengths of the three strings being combined ("s2", " Planet", and the right portion of "s1"), then allocate enough memory to hold the combined string, and then copy in the three strings to the new memory one at a time. The Python interpreter must be doing all of these steps also! Ultimately, the string that is formed must exist somewhere in the computer's memory, and surely all characters are occupying (at least) one byte. Python makes doing this simpler, for sure, but it is not any more efficient than C (and probably less so).

We'll eventually look in more detail about how to use some of Python's more interesting data types, but first let's get some other more basic constructs out of the way. Below is an example of a script that includes a Python "if" statement with "elif" and "else" clauses. It is also our first example that prompts the user for input. The "raw_input" function accepts a string from standard input. The "int" function converts a string to an integer. We have seen a similar program in C as part of an earlier topic:

```
print "Enter a positive integer: ",
x = int(raw_input())

if x < 10:
    print "x has only one digit"
elif x < 100:
    print "x has two digits"
elif x < 1000:
    print "x has three digits"
else:
    print "x has four or more digits"
    print "This program does not get more specific than that"
```

An interesting point to make here is that, assuming you have some background in C (or really any other programming language), you should be able to take a good guess at what this program does even if you have never seen these programming constructs in Python before. You might not understand that the "raw_input" function specifically returns a string (which is why the "int" conversion function is necessary), but given the surrounding lines, you might even be able to guess that. You can then start to make assumptions about Python syntax. For example, we see that the we don't need parenthesis around the condition of an "if" statement, but we do need a colon after the condition; and we see that "elif" in Python is similar to "else if" in C. We have already seen from a previous example that the Python interpreter pays attention to whitespace, and indentation is used to indicate blocks of code. This is not to say that you will be able to read all code in any language once you understand a single language. If, for example, you are looking at object-oriented code, and you do not have background in any object-oriented language, you are not likely to understand what is happening. However, there are certain constructs that are common to virtually all programming languages. These include things such as variables, "if" statements, and loops (which we will cover soon). None of these concepts should present any difficulty when you move to a new language.

One other thing to point out is that if the user types text here, the interpreter will produce an error message when it tries to do the conversion. You might wonder how to prevent errors such as the one that happens if a user types text in the example above. Python supports exception handling, something that standard C does not. (**Exception handling** is part of many modern languages, including C++ and Java.) We are not going to discuss any details, except to say it provides a way of "trying" out code, and "catching" any exceptions that happen. Those of you who might have experience with exception handling in other languages will understand this better, but the rest of you should not worry about it.

Another thing to point out is that the "raw_input" function also allows you to specify the desired prompt as an argument. For example, we can write:

```
x = int(raw_input("Enter a positive integer: "))
```

However, there is one difference here from the behavior of the previous program, which is that "raw_input" automatically produces a newline after the message. In the previous example, this was avoided by including the comma after the print statement.

Now let's also add a "while" loop to make sure that the integer entered by the user is not negative. As with "if" statements, even if you are seeing this for the first time in Python, you should be able to guess what this does due to your background in C. Here is the code:

```
x = -1
while x <= 0:
    x = int(raw_input("Enter a positive integer:"))
```

One very important type in Python is called "list". In Python, lists are implemented as resizable arrays, and there are similar provided classes in C++ and Java. One interesting fact about Python lists is that you can combine different types of elements in a single list. Here is an example, which also introduces a Python "for" loop (which is quite different from a C "for" loop):

```
l1 = [1, 2, 3, 4]
l1.append(5)
l3 = [] # start empty list
for i in l1:
    print i,
    l3.append(i)
print # Just a newline

l2 = ["hello", "world"]
l2.append("goodbye")
for i in l2:
    print i,
    l3.append(i)
print

for i in l3:
    print i,
print
```

The output is:

```
1 2 3 4 5
hello world goodbye
1 2 3 4 5 hello world goodbye
```

The "range" function can be used to generate lists of integers with certain starting and ending values. The statement "range(N)" generates a list of integers from 0 to N-1; "range(a, b)" generates all integers from a to b-1; and "range(a, b, c)" generates all numbers starting at a, with increments of c, as long as they are still less than b. For example, the "range(10, 100, 20)" would generate the list [10, 30, 50, 70, 90]. This list can be used with a Python "for" loop, of course.

Next we will look at an example of a function that accepts a parameter and returns a value. Note that neither the type of the return value nor the type of the parameter needs to be specified. In this case, we are writing a function that takes a single parameter, assumed to be an integer, and it returns a list. The list will contain all of the Fibonacci numbers less than or equal to the input. (The Fibonacci sequence starts with [1, 1], and each additional number is computed by adding together the previous two.) Here is the code (mostly taken from an on-line Python tutorial):

```
# Create a list containing all Fibonacci numbers
# less than or equal to n
def Fib(n):
    res = [] # start with empty list
    a, b = 1, 1
    while a < n:
        res.append(a)
        a, b = b, a+b # generate next number in sequence
    return res

print Fib(100)
Fib2 = Fib(1000)
print Fib2
```

This program will display:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

There are a few things to notice here. First, only the name of the parameter needed to be indicated. Because we pass an integer argument, the interpreter treats the parameter "n" as an integer. The return value is a list, and again, the type did not need to be specified. We see the script calls the function twice, and the return value can be displayed directly or assigned to a variable. We also see that "print" knows how to display a list. Another thing introduced in this example is the ability to combine multiple assignments on a single line. For example, consider the line "a, b = b, a+b". The way this works is that the right-hand side expressions are evaluated first, and then the results are assigned to the left-hand side variables from left to right.

We'll look at one more provided type, called "dict". As mentioned earlier, this is similar to associative arrays in other languages. You can think of a "dict" as being a set of (key, value) pairs. The keys are often strings, but they can also be integers, floats, etc., and a single "dict" can have a mix. Let's say you use a "dict" to represent common spelling errors and their corrections. The following code sets up a very simple "dict" to represent some common corrections. It then allows the user to type words one at a time; the words are either corrected or left the same.

```
corrections = { "teh":"the", "adn":"and", "mispell":"misspell" }

while True:
    word = raw_input("Enter a word:")
    if word == "quit":
        break
    if word in corrections:
        print corrections[word]
    else:
        print word
```

We see here how a "dict" can be initialized, and how a key can be mapped to its corresponding value (see "corrections[word]" in the code above). We also see that "in" can be used to check if an item is an existing key in a "dict". We also see how string comparison works, and we see that we can "break" out of a loop. The values "True" and "False" are recognized keywords in Python. You should be able to figure out the rest, or try out the code for yourself.

There is so much more to Python that we haven't discussed. There are many other provided types, and a lot more functionality to the types we did discuss. There is also exception handling (briefly mentioned earlier), and the ability to create classes. The Python **standard library** provides many additional modules providing further functionality, dealing with topics such as advanced mathematical concepts, regular expressions, Internet communication, and much, much more. Then there are other common libraries available (not part of the standard library, but made available by their creators) to help out with graphics, graphical user interfaces, web development, natural language processing, database interaction, etc. In fact, I would say it is the usefulness of this vast array of available libraries that really makes Python so valuable.

We will look at one module from the standard library, just to show an example of how to **import** a module and use its routines. I have chosen the "random" module (not that it is particularly important, but it is simple and provides some interesting functions). We will look at only two of the routines. The function "random" generates a random float in the range [0.0, 1.0) (inclusive of 0.0, not inclusive of 1.0). The function "randint" takes two parameters, call them "a" and "b", and it generates a random integer in the range from "a" to "b" (inclusive of both). Now consider the following code:

```
import random

for i in range(10):
    x = random.random() # random float in range [0.0, 1.0)
    print x,
print

for i in range(10):
    x = random.randint(1, 10) # random integer from 1 to 10
    print x,
print
```

The actual name of the module we are importing from the standard library is "random.py", but you leave out the ".py" extension in the "import" statement. Important a module provides access to its defined functions. (If there were statements in the imported module outside of functions, these would also be executed, but this would not be standard.) You can also import a module that you write yourself in another module.

This code includes examples that combine the "range" function, which produces a list, with the Python "for" loop. Within the two loops, we see calls to two functions from the "random" module that produce pseudo-random results. To call these functions, we specify the name of the module, followed by a period, followed by the function names. I cannot specify the exact output of this program, because it will likely be different every time the program is executed. However, it will produce two rows of output, each containing ten comma-separated pseudo-random values as described. The reason I say "pseudo-random" is that a computer is a deterministic device that is not capable of true randomness. The results are based on formulae that we are not aware of, plus a seed which, by default, is a representation of the date and time of day. The important thing is that the results appear random to us, and we cannot predict the values (because we do not know the seed or the formulae).

Note that it is also possible to produce pseudo-random numbers in C using the "rand" function, whose prototype statement is in the "stdlib.h" header file; this function returns a pseudo-random integer in the range of 0 to RAND_MAX (a pre-defined constant). However, if the programmer wants the program to produce different results every time, they must explicitly seed the random-number generator (using a different function called "srand") with the date and time of day (using the "time" function, whose prototype statement is in the "time.h" header file). If a C programmer wants to produce pseudo-random floats or pseudo-random integers in some specific range, they must use their own formulae to convert the return values of "rand".

How does Python compare to C overall? It is certainly much easier to quickly write code that can perform some powerful tasks (especially if there is an available module that directly applies to your task). However, as is the case with other scripting languages, I think most programmers would agree that Python is not the best choice for writing large applications that need to be very efficient. More debatably – and I may be in the minority on this point – I personally believe that C is a better language to learn first. C forces you to understand to a much greater extent what is happening in memory, and I find that programmers who understand what is happening in memory are usually much better programmers. In any case, the purpose of this introduction was to give a brief introduction to Python, and I hope that some of you choose to explore it in much more depth on your own.