Python for C programmers

The basics of Python are fairly simple to learn, if you already know how another structured language (like C) works. So we will walk through these basics here. This is only intended to be a quick overview, not a deep dive into how Python works. We will spend more time talking about certain topics (such as higher order functions) in later lectures, but for more details about the "basics," please talk to the instructors or the TA, or take a look at the Python reference pages (https://docs.python.org/2/reference/index.html)

One big difference between C and Python is that C is compiled while Python is interpreted. This means that to run a C program, you first have to compile it (e.g., with gcc) and then run it; but once you compile the program, you have a standalone executable (e.g., a.out). With a Python program, you do not have to compile the program but to run the program you need to run it in the Python interpreter (e.g., > python hw0.py)

In practice, at least for this class, this distinction will not really matter (it can matter more once you get to long-running programs that operate over large amounts of data)

Variables and Types

Perhaps the biggest difference between C and Python is that C variables are statically typed -- you need to say whether a variable x is an int or a float right up front. In Python, you don't:

```python
In [1]: x = 1
   ...: type(x)
```

Note that this means that we don't need to "declare" variables -- we can just use them whenever we need to.

What's interesting about Python, though, is that while we say the type of x is an int, what's really happening is that x is a reference to an integer object, which happens to have the value 1. x itself doesn't have a fixed type. We can re-assign it:

```python
In [2]: x = 1.2
   ...: type(x)
```

You can even make x a string:

```python
In [3]: x = "hello"
   ...: type(x)
```

Python does its type checking dynamically. It will not tell you until you try to do something with a variable whether the operation is legal or not:

```python
In [4]: len(x) #this will work because x is a string
   ...: x = 1.2
   ...: len(x) #what will happen here?
```
Python will also perform type coercion: when it makes sense, it will convert an object from one type to another to let an operation work:

```python
In []: p = 1
    print (type(p))  #the parentheses around the argument to print are optional

q = .2
    print (type(q))

r = p + q
    print (type(r))
    print ("value of r: {}".format(r))  #compare this to printf!
```

**Control statements**

Control statements in Python look a lot like their counterparts in C: if statements, while loops, for loops. The biggest difference is that in Python whitespace matters. We do not use { and } to separate blocks. Instead, we use colons (:) to mark the beginning of a block and indentation to mark what is in the block.

**If Statements**

Here is the equivalent of the C statement:

```c
if (r < 3) printf("x\n"); else printf("y\n");
```

```python
In []: if r < 3:
    print ("x")
else:
    print ("y")
```

And an example of multiline blocks:

```python
In []: if r < 1:
    print ("x")
    print ("less than 1")
elif r < 2:
    print ("y")
    print ("less than 2")
elif r < 3:
    print ("z")
    print ("less than 3")
else:
    print ("w")
    print ("otherwise!")
```

**While Loops**

while loops are similar:
In [ ]:
```python
x = 1
y = 1
while (x <= 10) :
    y *= x
    x += 1
print (y)
```

In [ ]:
```python
x = 1
y = 1
while (x <= 10) :
    if x % 2 == 0 :
        y *= x
    x += 1
print (y)
```

For Loops

For loops are a little trickier. They do not take the same form as C for loops. Instead, for loops iterate over collections in Python (e.g., lists). These are more like foreach loops that you might see in other languages (or the for (x : list) construct you see in Java). So let's start by talking about lists:

In [ ]:
```python
data = [1, 4, 9, 0, 4, 2, 6, 1, 2, 8, 4, 5, 0, 7]
print (data)
```

In [ ]:
```python
hist = 5 * [0]
print (hist)
```

Lists work like a combination of arrays in C (you can access them using []) and lists (you can append elements, remove elements, etc.)

In [ ]:
```python
length = len(data)
print ("data length: {} data[{}] = {}".format(length, length - 1, data[length - 1]))
```

In [ ]:
```python
data.append(8)
length = len(data)
print ("data length: {} data[{}] = {}".format(length, length - 1, data[length - 1]))
```

You can then iterate over the elements of the list:

In [ ]:
```python
for d in data :
    print (d)
```

In [ ]:
```python
for d in data :
    hist[d // 2] += 1
print (hist)
```
How do you write a `for` loop with an index variable that counts from 0 to 4, like you might in C? `for (int i = 0; i < 5; i++)`

Use the standard function `range`, which lets you count from a lower bound to an upper bound (with an optional step):

```python
In [ ]: r = range(0, 5)
    print (r)
In [ ]: for i in range(0, 5):
    print (i)
```

### Functions

Basic functions in Python work a lot like functions in C. The key differences are:

1. You don't have to specify a return type. In fact, you can return more than one thing!
2. You don't have to specify the types of the arguments
3. When calling functions, you can name the arguments (and thus change the order of the call)

```python
In [ ]: def foo(x) :
    return x * 2

print (foo(10))
```

```python
In [ ]: def foo2(x) :
    return x * 2, x * 4

(a, b) = foo2(10)
print (a, b)
```

```python
In [ ]: def foo3(x, y) :
    return 2 * x + y

print (foo3(7, 10))
print (foo3(y = 10, x = 7))
```

There are more complicated things you can do with functions -- nested functions, functions as arguments, functions as return values, etc. We will look at these in the lecture when we talk about Map and Reduce.