This iPython Notebook contains examples for the first lecture of the "Introduction to programming using Python" unit of CBMG688P at the University of Maryland in College Park. While all of the examples in this notebook share the same "scope" (variables defined in one example can be accessed in subsequent examples), I have made a point to make each example self-contained. This makes it possible to easily convert any example into a stand-alone script by:

I. Creating a new text document called script_name.py
II. Putting the line "#!/usr/bin/env python" at the top of the file (without the quotes)
III. Copying and pasting the complete code from any single example into the file

script_name.py can then be run in the normal ways by either typing "python path/to/script_name.py" in a terminal, "%run path/to/script_name.py" in IPython, or by making it executable with "chmod +x script_name.py" and then simply typing "path/to/script_name.py".

Printing

Simple printing

In [60]:
```
print "Hello, world!"
```

```
Hello, world!
```

Simple variable declaration

In [61]:
```
name = "Ted"
print "Hello,", name, "!"
```

```
Hello, Ted !
```

Concatenating strings

In [62]:
```
name = "Ted"
print "Hello,"+ name +"!"
```

```
Hello,Ted!
```

In [63]:
```
name = "Ted"
print "Hello, "+ name +"!"
```

```
Hello, Ted!
```

Formatting print statements

In [64]:
```
name = "Ted"
print "Hello, %s!" % (name)
```

```
Hello, Ted!
```

In [79]:
```
name = "Ted"
hometown = "Tulsa"
school = "UMD"
print "Hello, %s! How are things back in %s?" % (name, hometown)

print "Hello, "+name+"! How are things back in "+hometown+"?"

out_var = "Hello, %s! " % (name)
out_var = out_var + "How are things back in %s?" % (hometown)
print out_var
```

```
Hello, Ted! How are things back in Tulsa?
Things are good, thanks for asking.
Hello, Ted! How are things back in Tulsa?
Hello, Ted! How are things back in Tulsa?
```

Variable Types

In [80]:
```
name = "Ted"
hometown = "Tulsa"
print "Hello, "+ name +"! How are things back in "+ hometown +"?"
```

```
Hello, Ted! How are things back in Tulsa?
```

In [81]:
```
name = 7
hometown = 9
print "Hello, "+ name +"! How are things back in "+ hometown +"?"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-81-861692fe929b> in <module>()
      1 name = 7
      2 hometown = 9
----> 3 print "Hello, "+ name +"! How are things back in "+ hometown +"?"

TypeError: cannot concatenate 'str' and 'int' objects
```

In [84]:
```
print type("7")
```

```
<type 'str'>
```

In [83]:
```
print type("Ted")
```

```
<type 'str'>
```

Typecasting

In [87]:
```
print type(7)
```

```
<type 'int'>
```

In [88]:
```
print type('7')
```

```
<type 'str'>
```

In [89]:
```
print type(str(7))
```

```
<type 'str'>
```

In [91]:
```
name = 7
```

```
hometown = 9
print "Hello, "+ str(name) +"! How are things back in "+ str(hometown) +"?"
```

```
Hello, Ted! How are things back in 9?
```

Formatted printing cleanly casts variables into the desired type

In [93]:
```
name = 7
hometown = 9
print "Hello, %s! How are things back in %s?" % (name, hometown)
```

```
Hello, 7.00! How are things back in 9?
```

Python has four basic variable types:

In [94]:
```
print "Text is type: %s" % (type("some text"))
print "True/False are type: %s" % (type(True))
print "Whole numbers are type: %s" % (type(3))
print "Decimal numbers are type: %s" % (type(3.14))
```

. . .

Certain interactions between different types are handled automagically

In [95]:
```
print type(2.5)
print type(4)
print type(2.5*4)
2.5*4
```

```
<type 'float'>
<type 'int'>
<type 'float'>
```

Out[95]:
```
10.0
```

...but it's best to be careful

In [96]:
```
print type(7)
print type(9)
print type(7/9)
7/9
```

```
<type 'int'>
<type 'int'>
<type 'int'>
```

Out[96]: 0

Adding a decimal point to an integer will force it to be stored as a floating point number, which can be handy

```
Note: Python 3 will automatically convert integers to floating point values, as necessary.
```

In [97]:
```
type(7.)
```

Out[97]: float

In [98]:
```
7./9
```

Out[98]: 0.7777777777777778

Floating point numbers are base 2 approximations of base 10 numbers

```
For small numbers of significant digits, this is usually okay, but as the number grows, the approximation
breaks down.  This behavior is normal for most programming languages because the base 2 approximation uses
significantly less memory and allows calculations to be carried out significantly faster.
The official Python documentation provides a pretty good explanation:
http://docs.python.org/2/tutorial/floatingpoint.html
```

In [99]:
```
print "%.10f" % 0.1
print "%.20f" % 0.1
print "%.30f" % 0.1
print "%.40f" % 0.1
print "%.50f" % 0.1
```

```
0.1000000000
0.10000000000000000555
0.10000000000000000005551115123126
0.10000000000000000005551115123125782702118158340454l0
```

Formatting numbers

In [102]:
```
import math
print "How many digits of pi do you know?"
print "  %.2f (Most people)" % (math.pi)
print "  %.5f (Enough for most scientists)" % (math.pi)
print "  %.22f... (You're a dick.)" % (math.pi)
```

```
How many digits of pi do you know?
  3.14 (Most people)
  3.14159 (Enough for most scientists)
  3.1415926535897931159980... (You're a dick.)
```

Boolean Operators and Comparisons

In [103]:
```
True
```

Out[103]: True

In [115]:
```
False
```

Out[115]: False

In [105]:
```
not False
```

Out[105]: True

In [116]:
```
True == False
```

Out[116]: True

In [124]:
```
False == False
```

Out[124]: False

In [122]:
```
True and False
```

Out[122]: False

In [109]:
```
True and not False
```

Out[109]: True

In [117]:
```
True or False
```

Out[117]: True

In [118]:
```
True & True
```

Out[118]: True

In [119]:
```
True | False
```

Out[119]: True

In [120]:
```
(True and False) == (True & False)
```

Out[120]: True

In [121]:
```
(True or False) == (True | False)
```

Out[121]: True

Control Structures (part 1)

Control structures allow users to control the flow of a program, repeating or skipping certain steps as needed.

Decisions are typically handled using "if" statements

In [129]:
```
if True:
    earth = "world"
    print "Hello, %s!" % earth
```

```
Hello, world!
```

In [130]:
```
if not True:
    print "Hello, world!"
```

Nothing is printed in this case because "not True" is false.

Useful tip: Most things other than False, None, 0, and empty containers are treated as True by Python

In [131]:
```
if "Steve":
    print "Hello, Steve!"
```

```
Hello, Steve!
```

In [132]:
```
if "":
    print "Hello, Steve!"
```

Nothing is printed because Python considers the empty string "" to be false.

In [133]:
```
name = "Ted"
hometown = "Tulsa"
if hometown:
    print "Hello, %s! How are things back in %s?" % (name, hometown)
else:
    print "Hello, %s!" % (name)
```

```
Hello, Ted! How are things back in Tulsa?
```

In [135]:
```
name = "Ted"
hometown = ""
if hometown:
    print "Hello, %s! How are things back in %s?" % (name, hometown)
else:
    print "Hello, %s!" % (name)
```

```
Hello, Ted!
```

In [ ]:
```
name = ""
hometown = ""
if hometown:
    print "Hello, %s! How are things back in %s?" % (name, hometown)
elif name:
    print "Hello, %s!" % (name)
else:
    print "welp, I'm at a loss."
```

Data Structures

Data structures allow users to store more than one value in a single variable. Different data structures store these values in different ways that are optimized for different situations. Python has four basic types of data structures. Learning how to effectively use each type can dramatically improve the performance of your code and spare you a lot of grief.

Note: Python (like most "scripting languages") allows different types of values to be stored in the same

```
data structure.  This is not true of lower-level languages like C!
```

Lists

Lists store data in a fixed order

In [ ]:
```
test_list = ["Ted", 3, 3.14, True]
print test_list
```

An individual value can be retreived from a list by providing the index for the position of the desired value

In [ ]:
```
test_list = ["Ted", 3, 3.14, True]
print test_list[1]
```

Note that lists are "zero-indexed", which means that the first position is accessed using index "0", the second with index "1", and so on...

In [ ]:
```
test_list = ["Ted", 3, 3.14, True]
print test_list[0]
```

Lists allow individual members to be modified (they are "mutable")

In [ ]:
```
test_list = ["Ted", 3, 3.14, True]
print test_list

test_list[3] = False
print test_list
```

List objects have a set of special functions associated with them, as do many types in Python

In [ ]:
```
test_list = [3, 3.14]
print test_list
```

```
test_list.append(True)
print test_list

test_list.insert(0, "Ted")
print test_list

test_list.sort()
print test_list

test_list.pop()
print test_list

test_list.pop(0)
print test_list
```

Tuples

Tuples are very similar to lists, except that the values can not be modified individually once the tuple is created (they are "immutible"). While this makes them less flexible than lists, tuples are faster and more memory efficient.

In [ ]:
```
test_tuple = ("Ted", 3, 3.14, True)
print test_tuple
```

Individual values can be read from Tuples by using their index, just as with lists.

In [ ]:
```
test_tuple = ("Ted", 3, 3.14, True)
print test_tuple[0]
```

Both lists and tuples allow ranges of values to be extracted using Python's "slice" notation (two numbers separated by a colon). Note that the slice stops before the second index is reached. Page 159 of Haddock and Dunn contains a helpful diagram for visualizing Python's slicing indexes.

In [ ]:
```
test_tuple = ("Ted", 3, 3.14, True)
```

```
print test_tuple[1:3]
```

A slice taken from a tuple in this way will return a tuple (note the parentheses in the example above). Likewise, a slice taken from a list will return a list.

In [ ]:
```
test_tuple = ("Ted", 3, 3.14, True)
print test_tuple[1:3]
print type(test_tuple[1:3])

test_list = ["Ted", 3, 3.14, True]
print test_list[1:3]
print type(test_list[1:3])
```

Sets

Sets differ from lists and tuples in two major ways:

1) Values stored in sets are not stored in order and cannot be accessed with a numerical index.

In [ ]:
```
test_set = set(["Ted", "Keith", "Jonathan"])
print test_set
```

Instead, sets are typically used to efficiently test for membership.

In [ ]:
```
test_set = set(["Ted", "Keith", "Jonathan"])
"Ted" in test_set
```

In [ ]:
```
test_set = set(["Ted", "Keith", "Jonathan"])
"Steve" in test_set
```

2) Values stored in sets must be unique, but adding a non-unique value to a set will not produce an error. This can be useful for

removing duplicate entries from a data set.

In [ ]:
```
test_set = set(["Ted", "Keith", "Jonathan", "Ted"])
print test_set
```

It is important to note that the set() function only accepts a single argument. In the example above, I provided single Python list object, not just a comma-separated list of variables.

In [ ]:
```
test_list = ["Ted", "Keith", "Jonathan"]
test_set = set(test_list)
print test_set
```

Trying to pass a list directly without first storing it in a list object will produce an error.

In [ ]:
```
test_set = set("Ted", "Keith", "Jonathan")
print test_set
```

Python also has a suite of useful set-specific operators

In [ ]:
```
considered_vegetables = set(["broccoli", "carrots", "tomatoes", "squash"])
technically_fruits = set(["apples", "oranges", "tomatoes", "squash"])

union = considered_vegetables | technically_fruits
print union

intersection = considered_vegetables & technically_fruits
print intersection

technically_vegetables = considered_vegetables - intersection
print technically_vegetables
```

Dictionaries

Dictionaries (similar to "hashes" in PERL) store a set of unordered "keys", each of which is mapped to a "value".

In [ ]:
```
test_dict = dict([("key1","value1"), ("key2","value2"), ("key3","value3")])
print test_dict
```

Like sets, dictionaries are optimized for randomly accessing data.

In [ ]:
```
test_dict = dict([("key1","value1"), ("key2","value2"), ("key3","value3")])
print test_dict["key1"]
```

Also like sets, dictionaries can be used to effeciently test for membership.

In [ ]:
```
test_dict = dict([("key1","value1"), ("key2","value2"), ("key3","value3")])
"key2" in test_dict
```

In [ ]:
```
test_dict = dict([("key1","value1"), ("key2","value2"), ("key3","value3")])
"key4" in test_dict
```

The dict() function, like the set() function, accepts exactly one argument. Defining a dictionary with multiple key-value pairs requires that they first be arranged into a list of tuples.

In [ ]:
```
test_tuple1 = ("key1", "value1")
test_tuple2 = ("key2", "value2")
test_tuple3 = ("key3", "value3")
test_list = [test_tuple1, test_tuple2, test_tuple3]
test_dict = dict(test_list)
print test_dict
```

An arguably easier way to define dictionaries is using the alternate curly brace syntax.

In [ ]:
```
test_dict = {"key1":"value1", "key2":"value2", "key3":"value3"}
print test_dict
```

Nested Data Structures

In the above exmaple, I demonstrate that lists can contain tuples. With few exceptions, all data structures can contain other data structures.

In [ ]:
```
test_list_1 = ["thing1", "thing2"]
test_list_2 = ["thing3", "thing4"]
tuple_of_lists = (test_list_1, test_list_2)
tuple_of_lists
```

One notable exception is that sets can only contain immutable objects.

In [ ]:
```
test_list_1 = ["thing1", "thing2"]
test_list_2 = ["thing3", "thing4"]
set_of_lists = set([test_list_1, test_list_2])
print set_of_lists
```

In [ ]:
```
test_tuple_1 = ("thing1", "thing2")
test_tuple_2 = ("thing3", "thing4")
set_of_tuples = set([test_tuple_1, test_tuple_2])
print set_of_tuples
```

The same is true for dictionary keys.

In [ ]:
```
key_list_1 = ["key1a", "key1b"]
key_list_2 = ["key2a", "key2b"]
test_dictionary_keyed_using_lists = {key_list_1:"value1", key_list_2:"value2"}
print test_dictionary_keyed_using_lists
```

This is as good a time as any to mention that, while it's a good idea to choose descriptive variable names, it's also wise to keep them concise and reasonable.

In [ ]:
```
key_tuple_1 = ("key1a", "key1b")
key_tuple_2 = ("key2a", "key2b")
test_dictionary_keyed_using_tuples = {key_tuple_1:"value1", key_tuple_2:"value2"}
print test_dictionary_keyed_using_tuples
```

Unlike keys, dictionary values can be whatever you want. The same is true for lists and tuples.

In [ ]:
```
test_tuple = ("thing1", "thing2")
test_set = set([True, False])
test_list = [test_tuple, test_set, "stuff", "things"]
test_dict = {"key1":test_list, "spam":"eggs", "The Answer to the Ultimate Question of Life, the Univ
print test_tuple
print test_set
print test_list
print test_dict
```