Python - Lesson 3

Loops, Modules, Object Orientation, Lists, Tuples, Sets, Command Line Arguments

Previously… in Lesson 2

- Comments
  - # comments - single line
  - Triple-quote comments - multi line
  - Most important to explain WHY you are doing something.
- Variables
  - We assign values to variables with the assignment operator ‘=’.
  - We use variables to stop repeating ourselves and to reuse values.

Previously… in Lesson 2

- Conditions
  - We use conditions to test if something is true. We can use many operators in conditions, such as the ‘>’, ‘<’, ‘!=’ and ‘==’ operators.
  - We can put together several conditions with the use of the ‘and’ and ‘or’ operators.
- ‘if’ statements
  - We use conditions in an if statement to conditionally run some code if the condition is true.
  - The ‘if’ statement has two friends - ‘elif’ and ‘else’.

Previously… in Lesson 2

- Style
  - Python standard says we must indent blocks of code with four spaces.
  - The Python standard is wrong! Use tabs instead.
- Standard library
  - Python comes with lots of functions we can use out of the box, without any effort needed to write them.
- Calling functions
  - We can call functions by typing the name of the function and passing it any arguments needed.

Previously… in Lesson 2

- Functions can return a value, which is the result of the function's processing. This value can be stored in a variable for later use.

- Input/output
  - We can use the raw_input() function to get input from the user on the command line.
  - We can give raw_input() a prompt as an argument, which will be presented to the user.

Fruit Loops

- Last week we spoke about control flow and ‘if’ statements. These are really useful but there is another element of control flow we need to learn.
- A loop is a block of code which is executed a given number of times, or until a condition becomes true.
- There are two types - ‘while’ and ‘for’.

‘while’ Loops

- A ‘while’ loop executes while a condition remains true.
- We must indent the block of code inside the loop, in the same way as we did with an ‘if’.

```
counter = 1

while counter <= 10:
    print 'counter is now %s' % counter

    counter = counter + 1

print 'loop finished'
```

‘for’ Loops

- ‘for’ loops are similar to ‘while’ loops, except it is used to loop over something called a ‘sequence’ or ‘iterable’.
- We will venture more into sequences later on, but for now:

```
numbers = [1, 2, 5, 3, 9, 0, -2, 4]

for number in numbers:
    print 'next number is %s' % number

print 'loop ended'
```

Have a Break

- Sometimes when we are working with loops, we need to exit the loop immediately without waiting for the ‘while’ condition to be false, or for the ‘for’ sequence to end.
- When we need to do this, we can use the ‘break’ statement.
- When the ‘break’ statement is used, the loop is stopped and the code after the loop is run.

Have a Break

```
counter = 1

while counter < 100:
    print 'counter is now %s' % counter

    if counter == 20:
        print 'calling break statement'

        break

    counter = counter + 1

print 'loop ends with counter at %s' % counter
```

Break's Over - Continue Your Work!

- As well as the 'break' statement, there is one more fun loopy thing we can do, it is called the 'continue' statement.
- The 'continue' statement tells your program to go back to the top of your loop to test the 'while' condition, or to go to the next item in your 'for' sequence.
- It can be used when you want to skip code.

Break's Over - Continue Your Work!

```
counter = 1

while counter < 100:
    if counter > 20 and counter < 90:
        print 'calling continue statement'

        continue

    print 'counter is now %s' % counter

    counter = counter + 1

print 'loop ends with counter at %s' % counter
```

- Can you see the mistake with this code? How would you fix it?

Modules

- Last week we talked about the Python standard library and how it has lots of code that we can use for common things instead of us having to write everything from scratch.
- The standard library is actually organised into a hierarchy of folders filled with Python code files. These code files are called ‘modules’.

Modules

- Actually, all Python programs you write yourself are called modules too! A module is just a regular Python script file.
- Python is organised into these modules to make it ‘modular’.
  - Programs that want to use the standard library don’t need to load in everything all at once, they just need to load the modules that the program needs to use.
- To use a module in your program, import it.

Importing Modules

- In the following code, we have imported the ‘sys’ and the ‘os’ modules from the standard library into our program.
  - We then use functions in the two modules.
- Use ‘import’ statements to import modules. Several ‘import’ statements are fine.

```
import sys
import os

print 'this is a %s computer' % sys.platform
print 'current working directory is %s' % os.getcwd()
```

Module Hierarchy

- As we learnt before, Python modules are organised in a hierarchy.
- For example, the ‘os’ module in the standard library (which deals with the operating system), has a module called ‘path’ inside it.
  - The ‘path’ module deals with file paths.
- We can create our own module hierarchies too! But we’ll tackle this some other time.

Module Hierarchy

- When working with a hierarchy, to help us import the modules we want, we can use the ‘from … import ...’ statement.
  - What follows the ‘from’ is the path down the hierarchy to the module we want to import.
  - What follows the ‘import’ is the same as previous (the name of the module we want to import).
- We can also use this to import only specific parts, instead of the full module.

Module Hierarchy

```
# import the full os module.
import os

# import the 'path' module, which is inside the 'os' module.
from os import path

# import the 'join' function from within the 'path' module, which is inside the 'os'
module.
from os.path import join

# import the 'platform' variable from the 'sys' module.
from sys import platform

print 'This is a %s computer' % platform
```

Object Orientated Programming

- OO programming is a concept which refers to the use of ‘objects’ in your code, where objects ‘own’ things and can ‘do’ things.
  - You are an object - you ‘own’ a car and can ‘do’ the shopping.
- OO programming is very complex, we’ll only touch the very tip of the iceberg here.

Classes

- In OO programming, we normally refer to ‘classes’ and ‘objects’.
- A class is a type of thing, and an object is an instance of that thing.
  - For example, humans are a class of animal, whereas you specifically are an object which belongs to the class of humans, as am I.

Classes

- Classes are like a specification document, describing what the thing is like, what it owns and what it's behaviours are.
  - Humans have body parts, a height, a weight, a deep unsettling sense of self loathing…
  - Humans can walk, can run, can swim.
- Objects describe as example of the class.
  - I am x metres tall, I am y kilos.
  - I can swim z metres per minute.

Classes

- In Python, we create an instance of a class (an object) by calling its ‘constructor’. This is a special function which builds an object. We can assign this object to a variable.
- When we have our object in our variable, we can tell it to do things by calling the class’s functions. To do this we use the ‘.’ (dot) operator.

Classes

```
# Create object of class Human by calling Human's constructor function.
simon = Human("Simon Allen")

# Tell the object to do things by calling the functions available to the class.
simon.goToWork()
simon.goToPub()

while 1 == 1:
    simon.drinkBeer()
    simon.sleep()

# Objects of class Human have a name, print out this one.
print simon.name
```

- What will happen when this code runs?

Classes

- Later on, we'll have a go at making our own classes and doing OO programming!
- For now though, we'll take a look at some of the built in classes in Python.
  - The classes that we will be looking at are called 'collection classes' because they allow us to store collections of things.

Lists, Sets and Dictionaries Oh My!

- Lists, sets and dictionaries are built-in collection classes in Python.
- They are all insanely useful and you will see them everywhere.
- We use collection classes to store items that belong together.
- Lists, sets and dictionaries are all similar in nature but achieve different ends.

Lists

- A Python list is just like a list in everyday life. It’s just a bunch of items with no particular order, one after the other.
- We can create a list by wrapping the list items we want together inside square brackets.
- We can append to our list, remove items from the list and select items from the list.

Lists

- To access items in a list, we use an ‘index’.
- An index is a number representing the place in the list.
  - eg: first item in the list, second item etc.
- In Python, the index starts from 0, not 1.
  - So the first item in the list is at index 0, the second is at index 1.
- To use an index, we put the number inside square brackets. See the example.

Lists

```
# Create our shopping list with some items.
shoppingList = ['applies', 'milk', 'cheese', 'horse', 'bread']

# We need to add a few more items to the list.
shoppingList.append('cake')
shoppingList.append('eggs')

# We need to remove an item from the list.
shoppingList.remove('milk')

# Print out some items in the list.
# The number is the index of the item we want to get.
print shoppingList[1]
print shoppingList[3]
```

Tuples

- Tuples are like lists but are read-only.
  - Once you create them, they cannot be changed.
  - This makes them more efficient for the computer to work with in some cases because the computer knows that they won't change.
  - This read-only feature is called 'immutability'.
- Tuples can usually be used the same way as lists, except that to construct a tuple we place the items in parenthesis, not brackets.

Tuples

```
# Create our tuple using parenthesis, not square brackets.
shoppingList = ('cake', 'bread', 'milk')

# This is an error, because tuples are read-only
shoppingList.append('fish')
```

Sets

- Sets are similar to lists except internally the data is stored very differently.
  - How the data is stored depends on several factors.
  - Internally, Python stores it’s sets as a hashtable.
- Sets store unique items only - they do not allow duplicates.
- The order you add items to the set is not necessarily the order they will come out as.
- You can not index a set.

Sets

```
# Create our new set.
shoppingList = set()

# Add some items to the set.
shoppingList.add('milk')
shoppingList.add('cake')

# This will not do anything, as milk is already in our set.
shoppingList.add('milk')

# This is an error, because we can't index a set.
print shoppingList[0]
```

Difference Between Lists and Sets

- The underlying data structures are very different, which means:
  - It takes less time to add an item to a list than a set.
  - It takes longer to search for an item in a list than a set.
  - Sets cannot be ordered, lists can have any order you like.
  - Set items must be unique.
  - Sets cannot be indexed (for example, you can not ask for the first item in the set).

Command Line Arguments

- You may have used command line programs and scripts before that take arguments on the command line.
- You can make programs in Python that do this too!
- Python has a module called ‘argparse’ that is great for this, but it’s fairly complex and well beyond the scope of this lesson.

Command Line Arguments

- Saying that, there is a simple way to make your program accept arguments.
- The ‘sys’ module has a variable in it called ‘argv’.
  - The ‘argv’ variable is a list which stores any arguments passed on the command line.
- For example, we might make a program that calculates Body Mass Index from height and weight arguments passed as arguments.

Command Line Arguments.

```
from sys import argv

# argv[0] is not an argument, it's the path to the script.
# Also, remember, we can use 'float' function to convert things to floats.
# We need to do this because the command line arguments are strings,
# and we can’t do math on strings!
weight = float(argv[1])
height = float(argv[2])

bmi = weight / (height ** 2)

print 'Your Body Mass Index (BMI) is %s' % bmi
```

- We can now run this program:
  - ./bmi.py 85 1.8

Homework

- Last week you (hopefully) wrote a really simple corrupt cop simulator.
- This time we want to use what we've learnt today to expand on this work.
  - Make your program accept your bribe as an argument on the command line.
  - If the cop attempts to haggle, ask the user for an offer in a loop until the value is 20% greater than the initial offer.
    - Store the haggle offers in a list and print them.