**Dictionaries, Classes, and Objects**

The dictionary is another useful python type, similar to a list. However, a dictionary has O(1) lookup, insertion, and deletion time and is built using a hash table (which we will discuss later). In contrast, a python list has O(1) insertion time if we add to the front, but deletion and lookup can take O(n) time. However, a dictionary is not ordered, so it can't be sliced up like a list can.

A python dictionary is composed of key-value pairs. Given a unique key we can quickly look up the corresponding value. This is similar to a C++ map. Here is an example using integers ad the key and a string as the value. We can use other types for the key if we like. This makes a dictionary of three students, accessible by student ID:

```
students = {30510231: 'Bob', 31293931: 'Ted', 30984142: 'Carol'}
print(students[30510231])   # Bob
print(students[30984142])   # Carol
print(students[11111111])   # Key error
```

To test for membership based on a key we can use "in". There is also a "get" method:

```
if (11111111 in students):
    print(students[11111111])
else:
    print("ID 11111111 not found")
```

We can add to a dictionary using the array notation:

```
students[11111111] = 'Alice'
```

If you want to iterate over a dictionary, use the keys, values, or items:

```
for k in students.keys():
    print(k)
```

```
30510231
31293931
30984142
11111111
```

```
for v in students.values():
    print(v)
```

```
Bob
Ted
Carol
Alice
```

```
for i in students.items():
    print(i)                # tuples, access with i[0] or i[1]
```

```
 (30510231, 'Bob')
(31293931, 'Ted')
(30984142, 'Carol')
(11111111, 'Alice')
```

We can also easily delete from a dictionary using del:

```
del students[31293931]                    # Deletes Ted
```

**Classes**

Python lets you define classes just like Java or C++, except there aren't quite as many access controls. This makes the code simpler but there is a little less flexibility.

To define a class you use the keyword "class" followed by your class name. For example:

```
class Robot:
    def __init__(self,name = 'Bobo',version = 1.0):
        self.name = name
        self.version = version

    def greeting(self, num):
        for i in range(0,num):
            print("Hello from", self.name, self.version)

robot1 = Robot()
robot2 = Robot("Tobor",2.2)
print("Robot 1's name is", robot1.name)
robot1.greeting(3)
robot2.greeting(2)
```

This creates a class named Robot. The class has two instance variables, a name and version. It has one function or method, greeting. We create an instance of it in robot1 and robot2. We can then access the variables using the dot notation, just like in Java. Similarly we can call the function or method with a dot followed by the name.

The __init__ method is a special method. It serves like a constructor, although technically the object is created before the method is called, so initializer is a more accurate term. The __init__ method is called after the Robot object is created. There can be only one __init__ method, you can't overload them. However, you can supply default arguments and python will apply them appropriately as in the example of robot1 and robot2.

The self parameter is somewhat odd. It is the same thing as "this" in Java. It refers to the calling object. In the example, it is myrobot whose name has been changed to "tobor". We have to explicitly add self as the first parameter for any functions we define as a method but we don't specify it when we call the method. Python adds it automatically.

In Java and C++ we talk a lot about data hiding and making variables or methods private vs. public vs. protected. We don't really have any of that in python. Everything is public. But there are common conventions that are used.

A private variable is prefixed by two underscores. To make name a private variable we would have:

```
class Robot:
    def __init__(self,name = 'Bobo',version = 1.0):
        self.__name = name
        self.version = version

    def greeting(self, num):
        for i in range(0,num):
            print("Hello from", self.__name, self.version)
```

There are some protections from changing __name from outside the class (e.g. name mangling) but it's not as strong as Java or C++.

A protected variable is supposed to be accessible only within the class and its subclass. To denote protected, as a convention, use a single _ in front of the variable name. For example, if we want name to be protected it would be:

```
class Robot:
    def __init__(self,name = 'Bobo',version = 1.0):
        self._name = name
        self.version = version

    def greeting(self, num):
        for i in range(0,num):
            print("Hello from", self._name, self.version)
```

Note that we can still change _name from outside the class, but it would be violating common python conventions.

There are many other special methods we can override. The special methods have __ around the method name. Here are some of them:

| | |
|---|---|
| __str__ | like toString |
| __add__ | for overriding + operator |

```
    def __str__(self):
        return str(self.name) + " version " + str(self.version)

    def __add__(self, otherRobot):
        newName = self.name + otherRobot.name
        newVersion = self.version + otherRobot.version
        return Robot(newName, newVersion)

robot1 = Robot()
robot2 = Robot("Tobor",2.2)
print(robot2)
robot3 = robot1 + robot2;
print(robot3)
```

```
Tobor version 2.2
BoboTobor version 3.2
```

Other special methods:

| | |
|---|---|
| __eq__ | For == |
| __lt__ | For < |
| __le__ | For <= |

For a larger list see http://www.diveintopython3.net/special-method-names.html

Without eq defined:

```
robot1 = Robot("Tobor",2)
robot2 = Robot("Tobor",2)
print (robot1 == robot2)           #False, compares object references
```

With eq:

```
    def __eq__(self, otherRobot):
        return (self.name == otherRobot.name and self.version == otherRobot.version)
        # == on floats can cause problems just like C++ and Java

robot1 = Robot("Tobor",2)
robot2 = Robot("Tobor",2)
print (robot1 == robot2)        #True, uses return value from method
```

We can also define class attributes, which are shared among all instances, like static variables in a class:

```
class Robot:
    power = "Battery"

    def __init__(self,name = 'Bobo',version = 1.0):
        self.name = name
        self.version = version
        self.movement = None

robot1 = Robot("Tobor", 1)
robot2 = Robot()
print(robot1.power)                # Battery
print(robot2.power)                # Battery
Robot.power = "Solar"
print(robot1.power)                # Solar
print(robot2.power)                # Solar
```

However, if we set robot1.power then we create power as an instance variable.

```
robot1.power = "Solar"
print(robot1.power)                # Solar
print(robot2.power)                # Battery
```

You can also define subclasses and inherit properties of the parent class. To do so, give the parent name in parenthesis. Here is a somewhat simplified Robot and subclasses named TreadedRobot and WheeledRobot:

```
class Robot:
    def __init__(self,name = 'Bobo',version = 1.0):
        self.__name = name
        self.__version = version

    def __str__(self):
        return str(self.__name) + " version " + str(self.__version)

    def greeting(self, num):
        for i in range(0,num):
            print("Hello from", self.__name, self.__version)


class TreadedRobot(Robot):
    def __init__(self, name = 'Bobo', version = 1, treads = 2):
        Robot.__init__(self, name, version)
        self.__treads = treads

    def movement(self):
        return str(self.__treads) + " treads."

    def __str__(self):
        return Robot.__str__(self) + " treads: " + str(self.__treads)

class WheeledRobot(Robot):
    def __init__(self, name = 'Bobo', version = 1, wheels = 4):
        Robot.__init__(self, name, version)
        self.__wheels = wheels

    def movement(self):
        return str(self.__wheels) + " wheels."

    def __str__(self):
        return Robot.__str__(self) + " wheels: " + str(self.__wheels)

def foo(r):
    try:
        print(r.movement())     # Invokes movement() associated with object r
    except:
        print("Error")          # catch any exception if movement doesn't exist

walle = TreadedRobot("Wall-E", 2, 2)
print(walle)            # Calls __str__ in TreadedRobot which calls __str__ from parent
walle.greeting(2)       # Inherited method

spirit = WheeledRobot("Spirit", 1, 6)
print(spirit)

foo(spirit)             # Prints 6 wheels
foo(walle)              # Prints 2 treads
```

There’s a lot more python can do, especially when we start adding in other libraries, but that should be enough to get started!  We will use more python in class examples as we go on to build up your python skills and provide a contrast to C++.