Name: _____

Week 5:
Dictionaries, For-Loops, Nested Value Types, Intro To Classes

---

Dictionaries

With arrays, we're able to store multiple items, and then look them up by **position**. This works really well when we have things that we want to put in order, and when it makes sense to get them based on that order. But we have a lot of times when that doesn't really make as much sense - we frequently want to store things and look them up based on some value that is not the position. For example, we might want to keep track of the ages of our pets, and look up the ages based on the name of the pet, or keep track of the cost of store items based on product name.

We can do this with **dictionaries**. A dictionary stores multiple values, just like an array, but rather than storing them in order, we specify a **key** for every **value** we put into the dictionary. This would look something like this, in the example of keeping track of our pet's ages:

Key:	Value:
"gato"	12
"spot"	21
"blub"	2
"tweety"	9

In Python, we use two different types of syntax with dictionaries. To create them we use squiggly brackets, like so:

```
my_dict = {}
```

Just like with any value, we can put dictionaries into variables! And just like [] creates an empty array, {} creates an empty dictionary. Keys and values can be of any value type, just like array values can.

We can also create dictionaries with values in them, just like arrays. We simply enter each key/value pair as $KEY: $VALUE, with commas between pairs. The dictionary above would be created like this:

```
animal_ages = {"gato": 12, "spot": 21, "blub": 2, "tweety": 9}
```

When we want to get a value out of a dictionary, we use the other kind of syntax, square brackets, just like we use square brackets to read values out of an array, like so:

```
int_gato_age = animal_ages["gato"] # Sets int_gato_age to 12.
```

We can also overwrite values in the dictionary, just as with arrays:

```
animal_ages["gato"] = 13
print(str(animal_ages)) # Prints "{"gato": 13, "spot": 21, "blub": 2, "tweety": 9}"
```

You will also notice we can convert dictionaries to strings and print them out as well.

Unlike arrays, we can add new elements to our dictionary using the square bracket notation. If you remember, attempting to write:

```
some_nums = [1, 2, 3]
some_nums[3] = 4
```

results in `IndexError: list index out of range`. With a dictionary though, this is totally valid:

```
animal_ages["thumper"] = 5
```

Every key must be unique. You can have the same value in the dictionary multiple times, but you can't make two entries with the same key. Trying to do so will result in either an error, or editing the existing key/value pair.

Just like for strings and arrays, we have a few useful functions for our dictionaries:

Function:	Explanation:
len(dict)	`len()` works on dictionaries as well, and will tell you the number of entries in the dictionary.
del dict[key]	Removes the key/value pair for key from the dictionary. Will throw an error if the key doesn't exist.
key in dict	Just like we can use `value in array` to check if a value is in an array, or `substring in string`, we can check if a value is in a dictionary's keys.
dict.keys()	Returns the dictionaries keys. However, it does **not** return them as an array! You will have to call the conversion function list() on the returned result if you want to use them as an array.

dict.values()	Similar to .keys(), returns the values in the dictionary. Also does not return them as an array.

Example Problems:

Code:	What does it print?
`my_dict = {3: 2, 2: 3, 1:4}` `print(str(my_dict[1]))`	# Remember, it's not an array!

`pet_counts = {"cat": 3, "dog": 5}`
`pet_counts["cat"] += 1`
`print(str(pet_counts["cat"]))`
	# Not a trick question! += works here.

`foo = {}`
`foo["bar"] = "baz"`
`foo["baz"] = "bat"`
`print(foo[foo["bar"]])`
	# Python does things in brackets first!

`entries = {"a": 1.0, "b": 2.0, "c": 3.0}`
`del entries["b"]`
`print(str(entries))`	

`print(str(list({1.1: 1, 2.2: 2, 3.3: 3}.keys())))`	

For Loops

Now that we're working with arrays and dictionaries so much, it would be nice to have an easier way to loop over them. For example, currently, to print all the key/value pairs in a dictionary on their own lines, we would have to do something like this:

```
int_i = 0
key_values = list(dict.keys())
while int_i < len(key_values):
    print(str(key_values[int_i]) + ": " + str(dict[key_values[int_i]]))
    i += 1
```

We have to get our keys, make them into an array with `list()` so we can index them, get our counter, increment it, print out the key at that position, and then use the key at that position to get the corresponding value out of the dictionary and print that too. That's a lot of overhead! Fortunately, there's a much simpler way to do this, with **for loops**.

Now, before we get into for loops, I want to give you a warning: For loops in Python are *very* different from for loops in almost every other language. So don't rely on them! If you're ever doing something with a for loop that you're not sure how to do with a while loop, stop yourself and figure out the while version. Otherwise, when you get into C++, you'll be very sad.

That said, here's how for loops work: For loops repeat code a number of times equal to the number of items in a collection, such as an array. They also create a "loop variable" that is automatically set equal to each element in the collection, so that we don't have to get each item by hand. So, for instance, the array [1, 2, 3], if we used a for loop on it, would make our for loop repeat 3 times, and the loop variable would become 1, then 2, then 3. To use a for loop, we simply write `for`, then a name for our loop variable, then the word `in`, and then the collection we want to loop over, and finally a colon. Then we put the code we want to repeat on following lines, indented by 4 spaces. (Additive, as always.) Here's an example:

```
for i in [1, 2, 3]:
    print(str(i))
# Prints "1\n2\n3\n", where \n is the newline character.
```

Like with all our commands, for loops can be nested inside if statements, while loops and functions, and if statements and while loops can be nested inside of them. (We cannot, however, nest a function definition inside a for loop. Function definitions [for now] should never be nested inside anything.)

We can also use for loops over some things that we can't index. For instance, we can loop over the .keys() method from a dictionary, even though it's not an array, like so:

```
dict = {1: 4, 2: 5, 3: 6}
for key in dict.keys():
```

```
    print(str(key))
# Prints "1\n2\n3\n".
```

This means we can simplify our initial code down to this:

```
key_values = dict.keys()
for key in key_values:
    print(str(key) + ": " + str(dict[key]))
```

Much simpler, right?

We also have a special function called **range**. The range function will generate a collection that we can loop over (not an array though) between two numbers. This, combined with a for loop, is an easy way to run code a specific number of times. All we have to do is call the function `range(start, stop)`, where `start` is the first value we want in the collection, and `stop` is one more than the last value we want in the collection. So, for instance, to get all the numbers between 0 and 9 inclusive, we would say `range(0, 10)`. And to get the numbers between 5 and 20 inclusive, we would say `range(5, 21)`.

Example Problems:

Code:	What does it print?
`for int_i in range(0, 10):` `    print(str(int_i))`	
`for str_i in ["a", "b", "d", "c"]:` `    print(str_i)`	

`my_dict = {"a": 1, "b": 2}`
`if len(my_dict) % 2 == 0:`
`    for str_letter in my_dict.keys():`
`        print(str_letter)`
`else:`
`    for int_num in my_dict.items():`
`        print(int_num)`	

Nested Value Types

When we have a value types that allows us to put things into them, such as arrays and dictionaries, we should make a brief note of a very useful fact: When we say we can put any value type into an array or a dictionary, that includes arrays and dictionaries! This might seem a little confusing at first, but it's not so bad if you just remember that, to Python, an array or a dictionary is a single object, and can be put into and taken out of a variable or container just like any other object.

For instance, to make an array of arrays, we would just write this:

```
nested_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And that's it! We just use our notation for making arrays, with the square brackets, and create arrays inside of another array creation! We can, of course, also use .append() to put arrays into an already existing array.

Dictionaries work similarly, except that the key has to be a simple type. We'll see some exceptions to this later, but for now assume you can't use an array or dictionary for the key - they can be values though, like so:

```
nested_dict =  {1: {"a": "b"}, 2: {"c": "d"}, 3: {"e": "f"}}
```

Just like with the nested array, we just take our dictionary creation syntax, and use it inside of a dictionary creation! Nothing special to it. We can continue on in this manner as well, putting arrays inside of dictionaries and vice-versa, or even nesting 3, 4, or more levels deep.

To access elements in nested structures, we just use our square-bracket indexing notation like normal. For instance, looking at our nested_array variable from above, `nested_array[0]` would be the first element, which in this case is the array `[1, 2, 3]`. What if we wanted to get a single element? Well, if `nested_array[0]` is the array [1, 2, 3], then we can use the index 2 to get the value 3 from it! So `nested_array[0][2]` will give us `3`. Dictionaries work similarly, with `nested_dict[1]["a"]` giving us "b".

Updating and adding values also work like normal, but we have to be careful to make sure we keep track of what item we're modifying. If we wanted to add an element to the array [7, 8, 9] in `nested_array`, for instance, `nested_array.append(10)` won't work. It would add an item to the **outer** array, and result in [[1, 2, 3], [4, 5, 6], [7, 8, 9], 10], which is an array with mixed types, something that we don't want to do, since it's an error in most languages. (Though Python will allow it.) We would instead first want to get the array [7, 8, 9] by saying `nested_array[2]`, and then call append on that array: `nested_array[2].append(10)`. Dictionaries work the same way, where we need to make sure we know if we're modifying our outer dictionary, or the value stored in a key/value pair.

Example Problems:

Code:	What is printed?
`foo = [[1], [2], [3]]` `print(str(foo[2]))`	
`foo = [[1], [2], [3]]` `print(str(foo[2][0]))`	
`bar = {"a": [1, 2], "b": [3, 4]}` `print(str(bar["a"]))`	
`baz = {"a": [1, 2], "b": [3, 4]}` `print(str(baz["a"][1]))`	
`bat = [{1: 10}, {2: 11}, {3: 12}]` `print(str(bat[1][2]))`	

`arr = []`
`for int_i in range(0, 5):`
`  arr.append([int_i])`
`print(str(arr))`
	# What type are we appending to arr?

`qux = {}`
`for i in range(0, 5):`
`  qux[i] = {str(i): str(i + 1)}`
`print(str(qux))`	

Classes

Do you notice that the things we're doing, and the objects and value types we have are getting more and more complicated? They are, and they're just going to keep getting more complex as you learn more and more. For example, writing a function to get the second item from a single array is pretty easy, we can just say

```
def getSecond(array):
    item = array[1]
    return item
```

However, if we want to get the 2nd inner item from an array of arrays (e.g., getting 2 out of [[1], [2], [3]], or getting 5 out of [[6, 5], [4]]), we'd have to write something like this:

```
def getSecond(nested):
    if len(nested[0]) >= 2):
        return nested[0][1]
    else:
        return nested[1][0]
```

And this is just going to get more complex if we were to want an item further in, like the 5th inner item, or if we were to nest arrays more than 2 layers deep. We might not be able to get around having to write a complicated function to do something to a complicated value type, after all, doing complicated things is why people want computers! What we can do though is to store our values and our functions together, so that we only have to write the function once, and it will automatically go along with our values. We do this using **classes**.

Classes are a way for us to bundle values, like integers, strings, arrays, nested arrays, and anything else (even other classes, as we'll see later!) together with functions that do something to those values. (Classes are still values! So they can be stored into variables, arrays, dictionaries, and all that.) A simple example would be a class called Animal, where we store the name of the animal, the sound it makes, and a method called speak() that prints the animal's noise. Let's look at how classes would handle that.

Before we get started, we need to know the difference between a **class**, and an **instance** of that class. What's an instance? Well, think about it like this. String, which we've been using a lot, is a class. But each individual string is different, since we can store different values in each one when we create it. Every string we make is an **instance** of the string **class**. The **class** tells us what a string looks like, what values it stores, what methods it has, etc., and the **instance** is when we then create a specific object of that class. So string is a class, and "cat", "dog", "", and

“aaa123” are instances of class string. For our example, an instance of our Animal class might be where the name is “sheep”, and the sound is “baa”.

The class, then, defines what the individual instances will look like. It does this by specifying **methods** and **instance variables**. Methods are functions that any instance of the class can call, like “abc”.toUpperCase(). Although there are exceptions to this, for now all methods have to be called on a specific instance of the class - we can’t say Animal.speak(), because Animal is the name of our class, not an instance of that class. Instance variables are what make one instance different from another, such as the array holding the characters for a string, or the name and sound variables in our Animal class. These are variables that are unique to each instance of the class, and can be set at create time or changed after the class already exists. If we have an instance of class Animal with the name “cat” and the sound “meow” and then decide we like the sound “mew” better, we can just change the instance variable!

We can create a new instance of a class by calling the class name as a function, such as saying

```
animal = Animal()
```

(Usually we’ll want to store this into a variable, or Python won’t bother remembering it!)
We can also pass information in at the time we’re creating it, such as the name and sound for our animal, like this:

```
animal = Animal("cat", "meow")
```

This will make a new instance of class Animal, with name of “cat” and sound of “meow”. The parameters that are accepted at create time, as well as the ordering of the parameters, is specified when we define the class, which we’ll see later.

Once we have an instance, `animal` in this case (with a lowercase first letter - this is a common naming convention between classes and instances), we can alter the instance variables either directly via dot access, such as

```
animal.sound = "mew" # animal's sound is now "mew"
```

or by calling a method which will then alter them for us, such as

```
animal.setSound("mew") # The function (we assume) also sets animal's sound to "mew"
```

(We would, of course, have to write the setSound() method in the class definition first, in order for that to work.)

You’ll notice we’re using the dot notation again, just like we did earlier with strings when we saw things like “abc”.toUpperCase(). Whenever we put a function or instance variable into a class, such as putting setSound() and sound into Animal, we can’t use the function or variable name by itself anymore. This is due to the fact that we could make multiple copies of our class, creating several different Animal instances, for instance. And when we have multiple instances of our class, that means we also have multiple copies of our instance variables! So in order to let the computer know which one we want, we need to get an instance of our class first, and

then reference the instance variable or function on that specific instance with the dot notation. For example:

```
animal1 = Animal("bird", "tweet")
animal2 = Animal("cat", "meow")
print(animal1.sound) # Prints "tweet"
print(animal2.sound) # Prints "meow"
animal1.speak() # Prints "tweet"
animal2.speak() # Prints "meow"
```

If we didn't have the thing before the dot, Python wouldn't know which animal's sound we wanted! Note that if we'd said Animal.sound, we would have gotten an error.

So, internally, each instance of the Animal class will look similar to this:

.name	"cat"
.sound	"meow"
.speak()	<Function>

The left column is the name of the instance variable or function, and the right column shows a possible value. If we had a different instance, the instance variables would have different values, though .speak() would still hold the same function.

The logical next question is, how do we make a class in Python?

To make a class in Python, we simply type the word `class`, then a class name, a pair of parentheses "()", and a colon ':'. Everything we want to put into the class should be indented four spaces, additive, as always. So the declaration looks like this:

```
class Animal():
    # Class contents go here.
```

Python has a special way of writing functions that use this dot notation. I mentioned earlier that you can sorta think of dot notation as just passing the thing before the dot as the first parameter, so `"abc".toUpperCase()` would be similar to `toUpperCase("abc")`, and it turns out that's actually similar to how Python handles it. (Do note, this is another Python-specific thing. Most languages do this in another way.) When we write a function inside a class, like how we want to put speak() inside of Animal, we give it a special first parameter that will get the value of the instance the function is called on. By convention, we use the word `self` for this special parameter. We can use that special first parameter to access the specific values in the instance the function was called on - like "abc" in our .toUpperCase() example earlier. Once we have that instance, we can access its member variables (and even call other functions on it) using dot notation, just like we saw above in our example.

The `self` parameter is a little bit magic, in that Python will automatically set it equal to the thing that comes before the '.', and we don't have to worry about that. If we have other parameters,

they will be pushed back. So, for instance, `animal.giveCommand(“roll over”)` would become `giveCommand(animal, “roll over”)`.

Putting a function inside a class is an exception to the rule that functions should not be nested inside of something - when they go inside a class they are indented by 4 spaces in order to show they are part of the class definition. Other than that, and remembering to add the special `self` parameter, it’s pretty much like normal! Let’s see that by adding the .speak() and .giveCommand() functions to our Animal class:

```
class Animal():

    def speak(self):
        print(self.sound)

    def giveCommand(self, command):
        if command == "speak":
            self.speak()
```

There are two important things to note here. First, we can access our instance variables (sound, name, etc) by using dot notation on the `self` parameter. Although we only read them here, we could set them as well by assigning into them. (We’ll see an example of this in a second.) Second, we can call other class functions (which we call **methods**) on the `self` parameter as well, as we do in giveCommand(). This just calls the method like any other function call, running the statements inside the other method. And since `self` is the thing before the dot, it is what gets passed in as the `self` parameter for the new function - in other words, the function will be called on the same instance.

Now, how do we create instance variables, and how do we set values at create time? It turns out, we do both of those in the same way. Every function has a special method called __init__(), which is called automatically when we create a new instance of the class. So when we say `Animal()`, Python will create a new instance of class Animal, and call Animal’s __init__() method. __init__() uses the `self` parameter just like other methods (remember, method is just a fancy name for “function inside a class”), but it does something special - the __init__() method also *creates* the instance variables by using the dot notation on `self`. Do create a new instance variable, all you do is pick a name, assign a value into `self.$NAME`, and you’re done! __init__() can also take arguments beyond `self`, which are passed inbetween the parentheses when we’re creating the class, like how we saw Animal(“cat”, “meow”) above. We’ll assume that our __init__() method for class Animal will take in two parameters (plus `self`), one for the name, and one for the sound. Adding that to our class would look something like this:

```
class Animal():

    def __init__(self, inName, inSound):
        self.name = inName
```

```
        self.sound = inSound
        self.age = -1

    def speak(self):
        print(self.sound)

    def giveCommand(self, command):
        if command == "speak":
            self.speak()
```

It's typical to put the __init__() function first inside our class, since that's where people will look to figure out what our instance variables are.

Also, you'll notice I set `self.age`, even though there wasn't a parameter for it - this is because we should use __init__() to create and set all of our instance variables, even if we aren't going to use them yet. Maybe later I will add a .setAge() method that lets the user set their pet's age, but I don't want to include it at create time like inName and inSound. I should still assign something (and something of the correct type!) to it in my __init__() method, so that the variable exists when I want to use it later. (In Python you can get around this in some ways, but it's a bad habit, and other languages won't be so nice. So practice declaring and assigning all your variables in __init__()!)

And that's that! We now have a class that will keep track of the entire concept of an animal, all in one place, and we can use this in our code, all nicely bundled together. For example, we can write the following to use our new class:

```
# This code is outside the class
def make_cat():
    return Animal("cat", "meow")

def make_dog():
    return Animal("dog", "woof")

my_cat = make_cat()
my_dog = make_dog()

my_cat.speak() # Prints "meow"
my_dog.speak() # Prints "woof"

my_cat.sound = "nyan"

my_cat.giveCommand("speak") # Prints "nyan"
my_dog.giveCommand("speak") # Prints "woof"
```

Example Problems:

Code:	What is printed?

`class Foo():`
`  def __init__(self):`
`  self.bar = "A"`
`  self.baz = "B"`
 	
`foo = Foo()`
`print(foo.baz)`	

`class Bar():`
`  def __init__(self, inNum):`
`  self.num = inNum`
 	
`bar = Bar(5)`
`print(str(bar.num))`	

`class Baz():`
 	
`  def __init__(self, inColor):`
`  self.color = inColor`
 	
`  def printColor():`
`  print(self.color)`
 	
`foo = Baz()`
`foo.printColor()`	

`class Foo():`
`  def __init__(self):`
`  self.bar = "A"`
`  self.baz = "B"`
 	
`foo = Foo()`
`foo.baz = "C"`
`print(foo.baz)`	

```python
class Qux():
    def __init__(self, inLetter):
        self.letter = inLetter

    def setLetter(self, inLetter):
        self.letter = inLetter

    def printLetter(self):
        print(self.letter)

my_var = Qux('R')
my_other_var = Qux('R')
my_var.setLetter('L')
my_other_var.printLetter()
```

```python
class Adder():
    def __init__(self, inNum):
        self.num = num

    def add(self, inNum):
        self.num += inNum

adder = Adder(5)
adder.add(10)
adder.add(5)
print(str(adder.num))
```

```python
class Multiplier():

    def __init__(self, inNum):
        self.num = num
    def mult(self, inNum):
        self.num *= inNum

foo = Multiplier(5)
bar = Multiplier(7)
foo.mult(10)
foo = bar
print(foo.num)
```