Name: ______________________

More On Functions and Classes

One of the most useful things about functions is that they allow us to solve a problem in general, and then apply that solution to specific cases. In this way, they're much like mathematical functions. For instance, take the function for squaring a number,

```
f(x) = x * x
```

We don't know what `x` is, but we have used a function to describe how to calculate `x` squared for any `x`. Later on, we might use this function, like so:

```
y = f(5)
```

Just like we want our math functions like the square function to be generic when we write them, so that we can apply them to any number, we also want our Python functions to be generic. In order to do this, Python lets you use placeholders in your functions, like how we use `x` in `f(x)` above, even though we don't know what `x` will be yet. These are our **parameters**. And just like we call `f(5)` with the value 5 when we want to actually use it with a real value, Python lets us pass in actual values as our **arguments**. In Python, this same function would look like this:

```
def square(x):
    return x * x
```

And the line using it would look like this:

```
int_y = square(5)
```

The **parameter** x is just like the variable `x` in `f(x)`. And the **argument** `5` is just like the `5` when we call `f(5)`! And just like when we do `f(5)` we compute the result by replacing `x` with `5` in our function, getting `5 * 5`, when we call `square(5)` we replace the parameter with the argument! The only difference is that in the math function we go through the whole function and replace the variable with the value all at once, while in a Python function we **store** the value of the argument **into** the parameter, since variables can be reassigned and changed later on.

Let's look at some more examples:

```
def cube(x): # x is our parameter.
    return x * x * x
```

```
cube(5) # 5 is our argument.
```

When we run the above code, the value of the argument, 5, will be stored into the variable x, the parameter, in function memory. So the result will be `5 * 5 * 5`, because the value of x is 5. (Python will, of course, calculate `5 * 5 * 5` before returning the result, so the return value will be `125`.)

```
def add(x, y): # x and y are our parameters.
    return x + y

add(3, 4) # 3 and 4 are our arguments.
num1 = 2
num2 = 5
add(num1, num2) # 2 and 5 are our arguments.
```

This time we have two places we're calling our function. The first time, the arguments are 3 and 4, so 3 will be stored into the variable x and 4 will be stored into the variable y. Then x + y will be computed and returned. Since x is 3 and y is 4, the returned value will be 7.
The second time, we see variables being passed in. When this happens, our arguments are the **values** stored inside those variables. In this case, those values are 2 and 5, so our arguments are 2 and 5 also. 2 will be stored into x, and 5 will be stored into y, and then x + y will be computed and returned. Since x is 2 and y is 5 this time, the returned value will be 7 again.

```
def double_if_even(x): # x is our parameter.
    if x % 2 == 0:
        x = 2 * x
    return x

num_to_double = 10
double_if_even(num_to_double) # 10 is our argument.
```

Our argument is the **value** stored in the variable num_to_double, which is 10. Since 10 is our argument, it will be stored inside our parameter, x. Once that's done though, x can be used just like a normal variable! So when this runs, since the value 10 stored in x is even, the if statement will be true, and x will get the number 20 stored into it. We then return the value stored in x, which will be 20, since it was updated by the `x = 2 * x` line, just like any variable would be.

Notice that in these examples, although we've said what is being returned, we have not printed any of these results or stored them into any variables - when we call these functions, we’re simply calculating the values, returning them, and immediately forgetting them.

Your turn to try some!

```
def double(x):
    return x + x

double(5)
```

What is the value of the argument when we call `double()`?

What is the name of the parameter when we call `double()`?

What is the value returned when we call `double()`?

```
def multiply(x, y):
    return x * y

multiply(5, 4)
multiply(3, 7)
```

What are the values of the arguments the first time we call `multiply()`?

What are the values of the arguments the second time we call `multiply()`?

What are the names of the parameters the first time we call `multiply()`?

What are the names of the parameters the second time we call `multiply()`?

Can the names of the parameters for `multiply()` ever change? Why not?

What is the value returned the first time we run multiply?

What is the value returned the second time we run multiply?

```
def invert(x):
    return -1 * x

num_to_invert = 100
invert(num_to_invert)
```

What is the name of the variable holding the value of the argument when we call `invert()`?

What is the value of the argument when we call `invert()`?

What is the name of the parameter when we call `invert()`?

What is the value returned by `invert()`?

```
def add_five(x):
    x = x + 5
    return x

to_add_to = 100
add_five(to_add_to)
```

What is the name of the variable holding the value of the argument when we call `add_five()`?

What is the value of the argument when we call `add_five()`?

What is the name of the parameter when we call `add_five()`?

What value is in that parameter at the beginning of the function?

What value is in that parameter at the end of the function?

What is the value returned by `add_five()`?

```
add_five(x):
    x = x + 5
    return x

x = 100
add_five(x)
```

What is the name of the variable holding the value of the argument when we call `add_five()`?

What is the value of the argument when we call `add_five()`?

What is the name of the parameter when we call `add_five()`?

What value is in that parameter at the beginning of the function?

What value is in that parameter at the end of the function?

What is the value returned by `add_five()`?

Why didn't anything change, even though we used `x` in two places this time?

Classes (Now with 100% more cars!)

If we look at a car, for instance, there are a number of different attributes about a car that we might notice: The color, model, the speed at which it is driving, whether the headlights are on, and so on. Cars can also do a bunch of different actions: Brake, accelerate, honk, put on the turn signal, and so on. And while different cars may have different attributes, with one being red and one being blue, or one having the headlights on and another having them off, the actions will be the same for all of them: All cars can accelerate, all cars can brake, all cars can honk, etc.

Classes are just a way of taking these observations, and using them to represent objects in Python. We first make a **definition** of the class. that tells us two things. First, what aspects (color, model, speed, etc) we want to keep track of - the values of these aspects can change from one **instance** to another, just as the color, model and speed change from car to car, but the aspects we are keeping track of remain the same, just like every car still has a color, even if the colors are different. Second, it tells us what actions the object can do, like honking, braking, etc. These will be the same for every object, just like all cars have the same basic method for braking (apply brake pads and slow down), but they may have some differences based on aspects of the object. (For instance, all cars accelerate in the same way, but a Bugatti Veyron will accelerate faster than a Ford Model T - the action is the same, but the aspects affect how it is executed.)

In Python, the aspects correspond to **class variables**, and the actions correspond to **class methods**, or **class functions**. We use the class variables to keep track of the aspects, so we might have a string_color variable for a car's color, and a float_speed variable for its speed, and we use the class functions to describe how the object does its actions - in the case of a car, we might have an accelerate() function that describes how to accelerate, a brake() function that describes how to brake, and a honk() that describes how to honk.

For some of these actions though, we need to know what the aspects are, in order to properly describe how to do them. For instance, our accelerate() function might want to increase the speed by 24 mph/s when a Veyron is accelerating, but only by 1.9 mph/s if it's a Model T doing the accelerating. In order to do this, we need a way for our class functions to be able to look at the aspects of the object. Since our aspects are stored in our class variables, we can use them just like normal variables, and the only tricky bit is knowing how to get the right ones. After all, if we have both a Veyron and a Model T, we want to make sure that when we call accelerate() on the Veyron we don't look at the model for the Model T by accident! That would not result in an impressive take off.

This is where `self` comes in. The `self` variable is Python's way of saying, "Here, these are your aspects." Or, "these are your class variables". `self` represents the car that is currently doing the action, so if we call accelerate() on the Model T, then `self` would represent the Model T.

Let's see this in action. For now, we're just going to look at the honk() and accelerate() functions and how we call them - we'll worry about creating new cars later. Making class functions is mostly easy - we just nest them inside of our class. The only catch is that every class function has to take the `self` as the first parameter, so that we can tell what object (e.g. car) is doing the action (e.g. accelerating)! This `self` parameter will be handled automatically by Python, so you don't need to worry about having an argument for it to match with:

```python
class Car():
    def honk(self):
        print(self.horn_noise)

bugatti.honk() # Prints "HONK". When this honk runs, self is bugatti.
model_t.honk() # Prints "AAOOGHA". When this honk runs, self is model_t.
```

Every car has a different horn noise, so that is an aspect we can keep track of. Let's assume that we are storing it in a class variable called horn_noise. When the function honk() is called, we need to print out a different sound based on which car is doing the honking - the Veyron should make a normal honk sound, while the Model T should make one of those old-timey "Aaoogha!" noises. In order to tell what to print, we have to look at the horn noise aspect of the car which is doing the honk action. Since self represents the object (car, in this case) which is doing the action (e.g. honking), that means we have to look at `self`, and use it to find the horn_noise class variable. The good news is, that's easy! We access the class variable with `self.horn_noise`, and can use it just like any other variable. Let's see another example, with accelerate:

```python
class Car():
    def accelerate(self):
        self.speed = self.speed + self.acceleration
        if self.acceleration > 10:
            print("VROOM!")

bugatti.accelerate() # Prints "VROOM!"
model_t.accelerate() # Prints nothing.
```

Notice how we assign a value to self.speed, just like we would to any other variable?
Now, the Bugatti printed "VROOM!" like we expected, but how can we see what the actual speeds are? In other words, how can we access the class variables for the Bugatti and the Model T? Just how `self` represents the object doing an action when we're in a function, when we're not in a function the variable where an object is stored holds that representation. So just like we can say `self.speed` and get the speed of the car doing the accelerating when we're in the accelerate function, we can call `bugatti.speed` or `model_t.speed` and access their function variables directly.

```python
class Car():
    def accelerate(self):
        self.speed = self.speed + self.acceleration
        if self.acceleration > 10:
            print("VROOM!")

bugatti.accelerate() # Prints "VROOM!"
model_t.accelerate() # Prints nothing.
print(bugatti.speed) # Prints 24
print(model_t.speed) # Prints 1.9
```

Okay, so we know how to access our class variables, but how do we actually make a new car? This is where Python gets tricky, but I'll try to make it as simple as possible. First, make a list of all the aspects you want to keep track of. Write them all down one after the other:

```
speed
color
model
acceleration
horn_noise
```

Next, assign the values you want them to have when a new object (car) is initially created, just like variables.

```python
speed = 0
color = "black"
model = "Porsche 911"
acceleration = 14.6
horn_noise = "HONK"
```

Now put `self.` at the front of each line.

```python
self.speed = 0
self.color = "black"
self.model = "Porsche 911"
self.acceleration = 14.6
self.horn_noise = "HONK"
```

That's the main part! There's one tricky step left - put all of this inside a function called "__init__" in your class. Just like all the other functions, it will take `self` in. You don't need to fully understand this function, just know that it is called automatically when we make new instances (cars).

```python
class Car():
    def __init__(self):
        self.speed = 0
        self.color = "black"
        self.model = "Porsche 911"
        self.acceleration = 14.6
        self.horn_noise = "HONK"
```

And that's it! We have specified all of the aspects we want when we make a new car, and they will now get set to the values we typed. Now we can make new cars like this:

```python
porsche = Car()
```

But wait, that only lets us make Porsche 911s! What if we want another car? Well, there are two things we can do. The first is to take advantage of the fact that all of those aspects we made are class variables, and we can access them if we want to. We can make a new car, then set all of its aspects by hand by setting the class variables like this:

```python
ford_gt = Car()
ford_gt.model = "Ford GT Turbo"
self.acceleration = 6.5
self.color = "blue"
# The other aspects stay the same - we don't change them!
```

This is a lot of work though! Fortunately, there is an easier way. Even though __init__ is magic, we can still do some normal function things with it. Most importantly, we can still give it parameters like a normal function. So we can change our code to create new cars to this:

```python
class Car():
    def __init__(self, inModel, inAcceleration, inColor):
        self.speed = 0
        self.color = inColor
        self.model = inModel
        self.acceleration = inAcceleration
        self.horn_noise = "HONK"
```

Now we can pass arguments in, and they will be stored in those parameters, letting us create different types of cars more easily. But wait - we never call __init__! It's called by magic when we say `Car()`! That's true, but fortunately Python has some extra magic to help us out - if we put our arguments between the parentheses when we say `Car()`, Python will pretend that we used those arguments when calling __init__. So we can do this:

```python
ford_gt = Car("Ford GT Turbo", 6.5, "blue")
```

The arguments will be "Ford GT Turbo", 6.5, and "blue", and will be stored into inModel, inAcceleration, and inColor, respectively. Those values will then be stored into the class variables just like the ones we had written there before were! (Notice again that the `self` parameter was handled automatically by Python, so we didn't have to pass an argument for it.)

Your turn to try this out!

What are three aspects we might keep track of if we were simulating an analog clock?

What are three aspects we might keep track of if we were simulating a dog?

What are two actions an analog clock might have?

What are three actions a dog might have?

Two different instances of the same class (e.g., different cars) will have the same functions, but they can have different what?

What variable lets us know which instance is doing an action?

What variable is in every class function?

How would we set up a Clock class to make new analog clocks, given those aspects?

How would we set up a Dog class to make new dogs, given those aspects?

How might one of those clock actions look as a function?

How might one of those dog actions look as a function?