Name: ________________________

# Reference vs Value Variables

---

## Reference vs Value Variables

When calling a function, there are two ways to pass arguments: Pass-by-reference, and pass-by-value. Python only uses pass-by-value, you'll probably learn about pass-by-reference in CS I, but if you understand this section it shouldn't be too hard to learn pass-by-reference. When we call a function in Python, we pass the arguments by **value**, which means that the value of the argument is copied into the corresponding parameter. You can think of this as the computer looking at the value of the argument, writing that value on a piece of paper, and putting that paper into the box labelled with the parameter's name.

For classes (which includes built-in classes like arrays and dictionaries) this gets a little more complicated. Python doesn't ever actually store classes directly into our variables. Instead, it puts into our variables directions to where the object actually is in memory. A variable containing directions to an object is called a **pointer**, because it points the way to the object we care about. Think of the pointer as being like a treasure map. When we look at the treasure map, we can use it to find the object it's giving us directions to, and then we can use that object instead of using one stored directly in the box. It just takes us an extra step, following the map, instead of using the object immediately. (Python nicely hides from us. Whenever Python finds a pointer in a variable [i.e. a treasure map] inside one of our boxes, it will follow the directions on its own, and pretend the object it points to [i.e. the treasure]  is actually inside the box.)

What makes this different is the fact that since the class objects aren't actually stored inside our variables, they aren't actually in main memory! (Or function memory, or whatever memory our variables happen to be in.) And even more than that, we can have multiple pointers (treasure maps) to the same class object. Here's an example:

Let's say we make a car, from last week's packet, and it's a grey DeLorean. That car will be stored in the variable `my_car`. But wait! We just learned it's not actually in the variable. The variable only stores a pointer, or treasure map, telling Python how to find the class object (or **instance**) elsewhere in memory. So if we say `my_car_2 = my_car`, Python will look at the **value** in `my_car`, make a copy of it, and put that copy into `my_car_2`. But what was the value in `my_car`? It was just directions on how to get to the actual car object! So now we have two copies of the directions. However, just like with a real buried treasure, making a copy of our treasure map did not make a copy of the treasure - there is still only one car object, we can just have two maps to find it now.

What does this mean when we pass classes as arguments to functions? Well, when we pass a class object, like a Car, in to a function, Python will look at the value in the value in the variable (which will be a pointer) and copy **that** value, storing it into the parameter. So the parameter contains the pointer, and not the actual car object! And since we still only have one Car (we copied the **pointer**, not the Car!), any changes to that parameter will affect the original Car, since both pointers are pointing to it! This is kinda like giving a copy of a treasure map to your friend. If he then goes and changes the treasure, it will be changed when you go to find it as well. Here's a code example, for extra concreteness:

```
def add_speed(car):
    car.speed += 10

my_car = Car()
my_car.speed = 10
print(my_car.speed)
add_speed(my_car)
print(my_car.speed)
```

The first `print` statement will print `10`, but the second will print `20`! That's because the function was given a copy of the directions to the same car object that the main code was using, so when the function changed it, it changed for the main code as well. This won't happen if we use non-class variables like this though:

```
def add_ten(num):
    num += 10

my_num = 10
print(my_num)
add_ten(my_num)
print(my_num)
```

In this case, both `print` statements will print out `10`! This is because when Python passes an argument to a function, it always makes a copy of the argument, and store it into the corresponding parameter. In the bottom case, the argument is the number `10`, so Python copies that number and stores it into num. Changing num won't affect my_num, because they both have their own values! In the first example though, my_car contains a **pointer** to the actual car object, and that **pointer** is what is copied and stored into car. So both variables are pointing to the same car object! And because of that, when the function changes the car, it changes for both of them.

Just to reiterate: This will only happen for variables holding classes (including built-in classes), because they're the only ones where we have these pointers. For non-class variables, Python will just copy the value and the function will only be able to change the copied value!