



# Passing by Reference and by Value



# Passing

- A program is built in small modular, functional pieces. These pieces are called functions (in procedural programming) or methods (in object oriented programming)\*.
- Each function has its own data. This data is known as “local” data and it exists *only* in it’s function.
- Each function does one-and-only one “job.”
- To get work done, functions must “hand off” (pass) data to each other for processing.
- Think of each function as a specialist that can only do *only* one job.
- Collections of functions make up an application.

*\* we will only talk about functions from here on*

# Background

- Anything can be passed: Variables, Arrays, Objects, Files, Data Structures... *Anything!* Even functions themselves can be passed to other functions.
- Different Languages handle passing differently but the concept is *always* the same.
- Some languages like C/C++ leave it up to you how to pass. You decide in every case what is passed and how it's passed and you have to write the code to determine how something is passed.
- Some languages like Python decide for you and do it automatically. You don't decide anything, the language does it, and you conform to it.
- Most languages fall in between these two points. Each language has some default behaviors, some you can override, some you can't.

# Two Ways to Pass/Return

- When you give “something” to a function it’s called *passing*.
- When a function gives “something” back, it’s called *returning*.
- We will just call both cases “passing” for now.
- Remember, “something” can literally be *anything*.

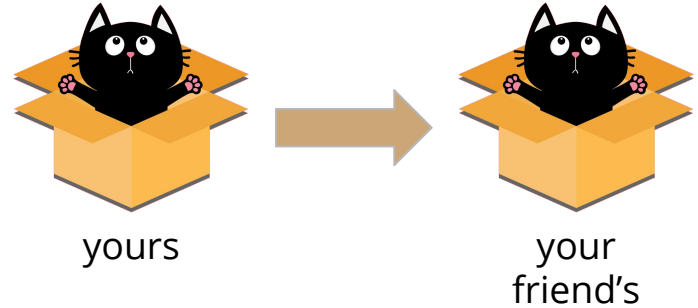
**Pass By Value** means that when a function hands off “something” it hands off a copy and it keeps the original. The original “something” and the copy “something” have *nothing* to do with each other.

**Pass By Reference** means that when one function hands off “something” it hands off the actual thing itself, not a copy. The original “something” and the passed “something” are the same thing!

# Pass By Value - Boxes and Kittens

Think of a variables as a box. In that box you have something, for now, let's say a kitten (data).

Now imagine you want to give your friend the kitten, but you want to keep your kitten too. Solution? Pass it by value! Tell your friend to get a box. Clone your kitten, give your friend the clone, you keep the original kitten and you keep your box.



# Pass By Value - Boxes and Kittens

The moment you give (i.e. pass by value) your kitten, there are two identical boxes with two identical kittens. However, they have nothing to do with each other, and they are owned by two different people (i.e. functions).



Your kitten (data),  
in your box  
(variable), owned  
by you (a function)



Your friend's kitten  
(new data), in their  
box (new variable),  
owned by them  
(another function)

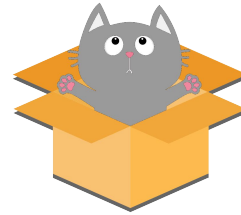
Now suppose your friend dyes their kitten grey. It's their kitten, they can do what they want with it. Then your friend decides to return their kitten to you. You need another box to keep it in and then you will have two kittens!

# Pass By Value - Boxes and Kittens

Your friend returns the grey kitten to you and they go away. You don't hear from your friend again unless you *call* them back. Now you have two kittens. One black one, which is your original, and one grey one which is the one your friend changed (processed).



Your original kitten (data), in your original box (variable), owned by you (function).



Your new kitten (new data), in your new box (another new variable), owned by you (function).

# Pass By Value In code

Imagine instead of a black kitten and grey kitten, we have the numbers 10 and 20.

Start reading in **you()**

output

```
My box has 10 in it.  
My box still has 10 in it.  
My new box has 20 in it.
```

```
def friend(box):  
    # I prefer grey kittens (20) so I will change the contents of my box  
    box = box + 10  
  
    # Now I will give my friend back my grey kitten (20) and  
    # then I'm going to take my box and go away  
    return box  
    # now go back to where you left off below and keep reading  
  
def you():  
    # >>> START READING HERE <<<  
    # put your black kitten (10) in your box  
    myBox1 = 10  
  
    # tell everyone about it  
    print("My box has", myBox1, "in it.")  
  
    # now call your friend and pass them your box BY VALUE. this will let  
    # you keep your box, and keep your kitten. but you also know your friend  
    # will change their copy of the kitten (10) and give it back to you,  
    # so you need to prepare another box (myBox2) to hold it.  
    myBox2 = friend(myBox1)  
    # stop reading here, and go up and see what your friend is  
    # doing with your kitten (10)  
  
    # tell everyone about both your boxes  
    print("My box still has", myBox1, "in it.")  
    print("My new box has", myBox2, "in it.")
```

you()




# Pass By Value - In code

Now let's change some box (variable) names and see what happens. What if your friend's box is *ALSO* called myBox1? What changes?

```
def friend(myBox1):  
    myBox1 = myBox1 + 10  
    return myBox1  
  
def you():  
    myBox1 = 10  
    print("My box has", myBox1, "in it.")  
  
    myBox2 = friend(myBox1)  
  
    print("My box still has", myBox1, "in it.")  
    print("My new box has", myBox2, "in it.")
```

you()

Nothing changes! The myBox1 in you() has NOTHING to do with myBox1 in friend().



```
My box has 10 in it.  
My box still has 10 in it.  
My new box has 20 in it.
```

**NAMES OF LOCAL VARIABLES DO  
NOT MATTER!**

# Pass By Value - In code

Now let's see what happens if we don't have two boxes and we return to the same box. What changes?

```
def friend(box):  
    box = box + 10  
    return box  
  
def you():  
    myBox = 10  
    print("My box has", myBox, "in it.")  
  
    # this time we pass the same  
    #variable (myBox) as we return to  
    myBox = friend(myBox)  
  
    print("My box has", myBox, "in it.")
```

you()

Now something changes. You destroyed the old contents of myBox and replaced them with your friend's data. In kitten terms, the grey kitten killed and replaced your black kitten. Kittens will do that sometimes.



```
My box has 10 in it.  
My box has 20 in it.
```

# Pass By Value - Summary

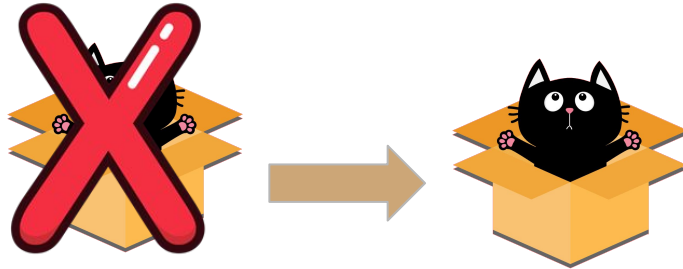
- Passing by value means you hand off a *copy* of data.
- The calling function passed the copy to the called function.
- Altering the copy has no effect on the original.
- Names of local variables do not matter.

Some additional notes that will become important later...

- Passing by value is “expensive” in computational terms.
- Passing by value is inefficient.
- In most languages, scalars (i.e. single value primitives) will pass by value automatically. Single value primitives (i.e. scalars) are things like ints and floats.

# Pass By Reference - Boxes and Kittens

Back to our boxes (variables) and kittens (data). Passing by reference means you do *NOT* make a copy. If you pass by reference your box and kitten to your friend, then you will give your actual box (variable) with your actual kitten (data) in it to your friend. Your friend now has complete control of your box and your kitten! Does this sound dangerous? Good, because it is.



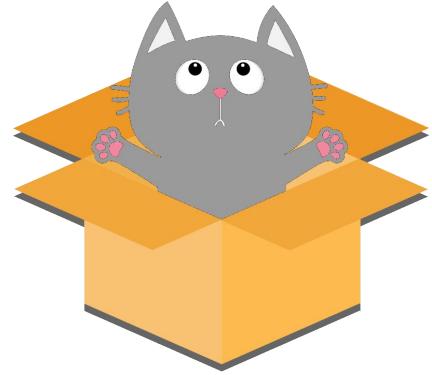
it still technically your box and kitten but your friend has complete control of it and you can't have it back until they let you.

# Pass By Reference - Boxes and Kittens

At the moment you give (i.e. pass by reference) your kitten, your friend owns your *BOTH* your box and your kitten.

Now suppose your friend dyes the kitten grey. *Your* cat is now grey! It's your kitten, but they can do what they want with it if you pass by reference.

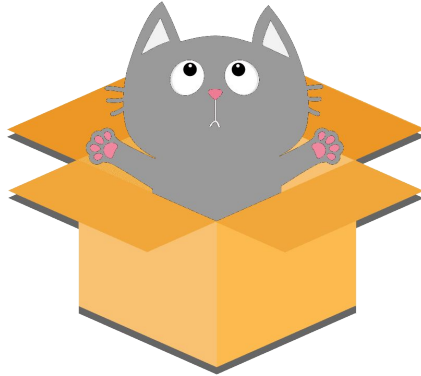
Still sound dangerous? Good, because it is.



Your kitten (data), in your box (variable), but now controlled by your friend!

# Pass By Reference - Boxes and Kittens

Your friend does not need to return your kitten. It's still yours. They were just borrowing it. When your friend goes away, your box and kitten goes back to you automatically because it is still yours. However, whatever your friend did to your kitten, you have to deal with it, because you let your friend control your box and your kitten. That's on you!



# Pass By Reference - In code

```
def friend(myList):  
    myList[0] = 100  
    myList[1] = 200  
    myList[2] = 300  
    # notice there is no return!  
  
def you():  
    myList = [10,20,30]  
  
    print("Check out my list", myList)  
    friend(myList) # no return!  
    print("Check out my list now", myList)  
  
you()
```



In the code shown here, we make a list in `you()`. It belongs to `you()`. We pass it to `friend()`. The pass is by reference (*Python does this automatically, more on this later*). Now `friend()` decides to alter the list but it's the same list from `you()`!

```
Check out my list [10, 20, 30]  
Check out my list now [100, 200, 300]
```

# Pass By Reference - In code

Now let's change the variable name in friend and see what happens. What if your friend's list is *NOT* called myList? What changes?

```
def friend(someList):  
    someList[0] = 100  
    someList[1] = 200  
    someList[2] = 300  
    # notice there is no return!
```

```
def you():  
    myList = [10,20,30]  
  
    print()  
    print("Check out my list", myList)  
    friend(myList) # no return!  
    print("Check out my list now", myList)  
    print()
```

```
you()
```



Nothing changes!

```
Check out my list [10, 20, 30]  
Check out my list now [100, 200, 300]
```

**Again... NAMES OF LOCAL  
VARIABLES DO NOT MATTER!**



# Pass By Reference - Summary

- Passing by reference means you hand off your actual variable and the data it contains. This is known as a *memory pointer* and will become *very* important later, especially in languages like C/C++.
- The calling function temporarily loses control of the variable and data passed to the called function which then gains control.
- Altering the passed data affects the original data because it is the same thing! As you can imagine, this is dangerous, but has benefits.
- You do not return the passed data from a pass by reference. You do not need to. It's the same data in both places.
- Names of local variables do not matter.

# Pass By Reference - Summary continued

Some additional notes that will become important later...

- Passing by reference is “cheap” in computational terms.
- Passing by reference is very efficient.
- In most languages, objects (i.e. multi-value data structures) will pass by reference automatically.
- In Python, lists, dictionaries, strings, tuples, arrays, and many other objects pass by reference automatically. Scalars pass by value automatically.
- In C/C++, you have to decide what you want to pass by value or by reference. This can be quite challenging, but gives complete control of... *everything*.
- In other languages, it will be a mix of value and reference passing and you can override those (mostly) as needed.

# Pass By Value and Reference Conclusion

Mastering passing by value and reference is fundamental to being a programmer. It is essential to not only understand it, but to truly master it.

**You cannot be a professional programmer in *any* language unless you master this concept!**

It is tempting to “enjoy” a language like Python that hides these details from you and makes these decisions for you, but it’s also a trap. If you stumble through programming without mastering these concepts, you will create security holes, bugs, inefficient code, and all sorts of problems.

It’s tempting to “suffer” a language like C/C++ which throws these details in your face and makes you manage them yourself. However, when you master C/C++ you will master this concept in all languages. This is one reason C/C++ is *the* language of computer science.