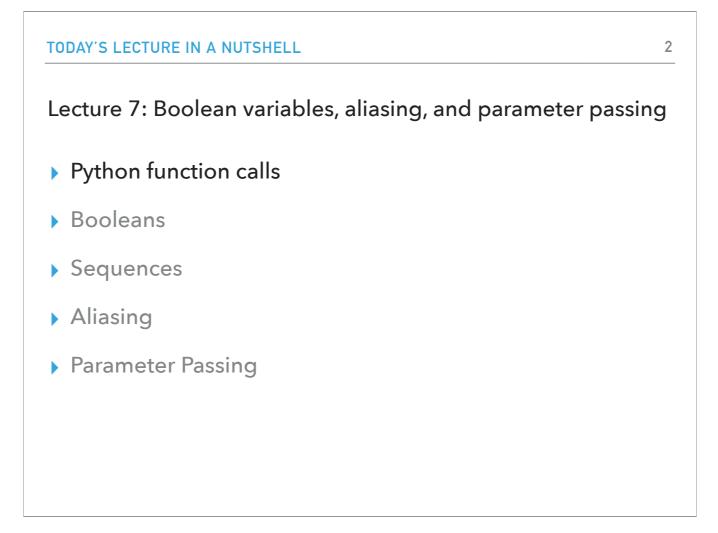




[record lecture on Zoom]

Welcome to lecture 7! We are officially together until the end of March :) Thank you for your willingness to help our department as we try to grow; your feedback about the past lecturers is extremely useful and important as we hope to grow the CS department.



Today, we will wrap up Lecture 6 by talking about tuples that you saw with Prof Ye in the lab. We will then get into a number of topics that will deepen our understanding of Python and programming languages in general. The first one will tackle the flow of our Python programs when we call functions.

PYTHON FUNCTION CALLS	3
What happens when a function gets called?	
Let's look at this simple file <u>function-calls.py</u>	
We can walkthrough what happens when we make a call to add_then_double, e.g., add_then_double(10, 20)	
1. The arguments to the function (10 and 20) are evaluated.	
 They could have been more complicated expressions, e.g., add_then_double(5+5, 40/2) 	
> 2. The function is then called and the parameters are associated with the arguments that passed in. Think of it like saying $a = 10$ and $b = 20$.	are
We now have two variables in the function that have the values of the arguments.	
The only tricky thing is that these variables are unique to this function, so even if they defined globally (outside any function), they will be different than the global variables	

Let's start by looking at the file function-calls.py. We can see that there is a function called add_then_double. Let's see what happens when we call add_then_double(10, 20). First, the arguments to the function (10 and 20) are evaluated. Notice that they could have been more complicated expressions that we pass as arguments, e.g., add_then_double(5+5, 40/2).

Second, the function is then called and the parameters are associated with the arguments that are passed in. Think of it like saying a = 10 and b = 20.

We now have two variables in the function that have the values of the arguments.

The only tricky thing is that these variables are unique to this function, so even if they're defined globally (outside any function), they will be different than the global variables.



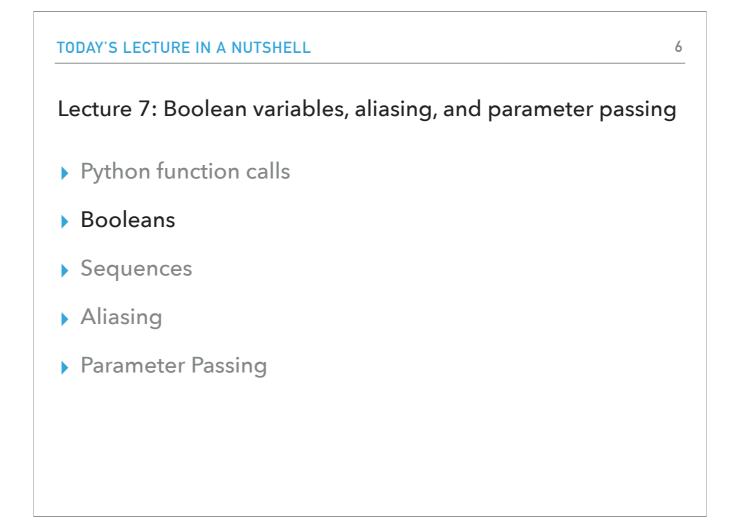


	What happens when a function gets called? (cont'd)
	3. The function body for add_then_double is executed one statement at a time.
	This will in turn call add(a, b). Like any other function call, we:
	evaluate the arguments, in this case, we evaluated a, which gives us 10, and b, which gives us 20,
	we then pass these values to add and associate them with parameters num1 and num2.
	add is only one line of code, it adds the number and then returns that value.
	num1 + num2 gives us 30 (10 + 20)
	the return statement means wherever this function was called, that is the value that call should represent.
	the call add(a, b) will then represent the value 30
	We then assign what add(a,b) returns to the variable added.
	 double is called with what was in the variable added (30) and following the same logic as above gives us back 60
	▶ 60 is then associated with the variable doubled.
	Finally, add_then_double returns doubled, which has the value 60.
The third step includes the execution of the	e add_then_double body, one statement at a time.

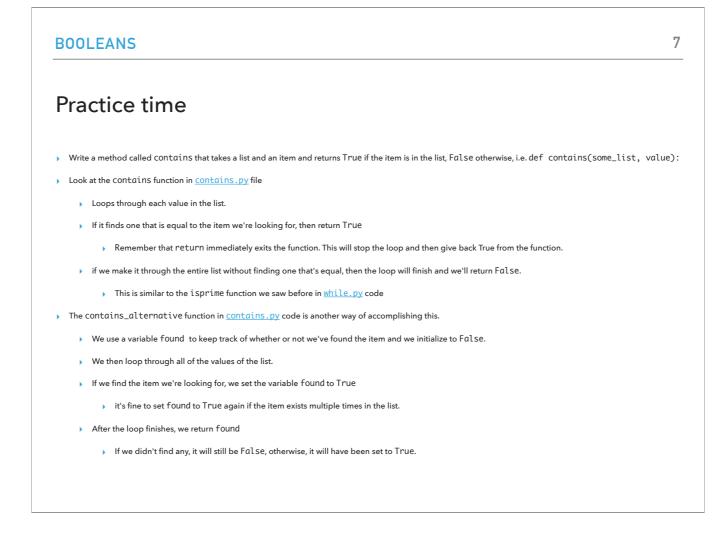
This will in turn call add(a, b). Like any other function call, we will start by evaluating the arguments (in this case, we evaluated a, which gives us 10, and b, which gives us 20). We then pass these values to add and associate them with parameters num1 and num2. add is only one line of code, it adds the number and then returns that value. num1 + num2 gives us 30 (10 + 20). The return statement means wherever this function was called, that is the value that call should represent. The call add(a, b) will then represent the value 30. We then assign what add(a,b) returns to the variable added. Then function double is called with what was in the variable added (30) and following the same logic as above gives us back 60. 60 is then associated with the variable doubled. Finally, add_then_double returns doubled, which has the value 60.

```
5
PYTHON FUNCTION CALLS
What happens when a function gets called? (cont'd)
> If we called add_then_double from the Python console/shell, we will see the value 60 echoed,
  not because it was printed out but because that call (add_then_double(10, 20)) now
   represents the value 60.
• We could just have easily done other things with it, e.g.,
  >>> add_then_double(10,20)
   60
  >>> add_then_double(10,20) * 2
   120
  >>> result = add_then_double(10,20)
   >>> result
   60
  >>> add_then_double(add_then_double(10,20),4)
  128
```

If we called add_then_double from the Python console/shell, we will see the value 60 echoed, not because it was printed out but because that call (add_then_double(10, 20)) now represents the value 60. But we could have easily done more complex things with our calls.



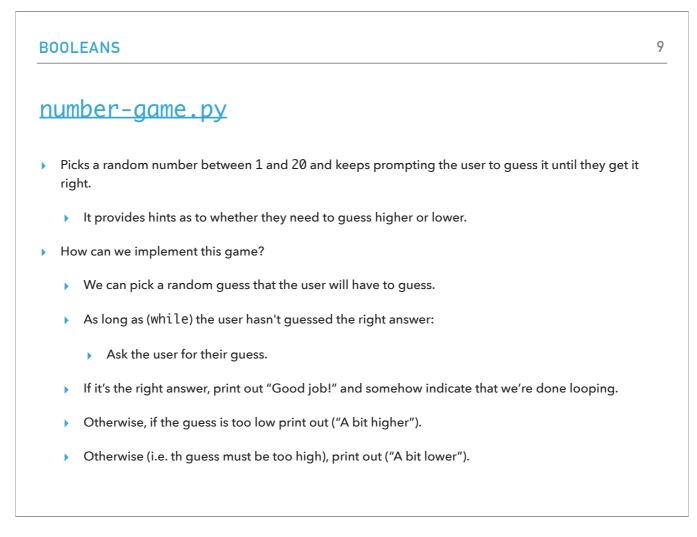
We will switch gears now and talk about booleans which we have seen that are represented with the type bool.



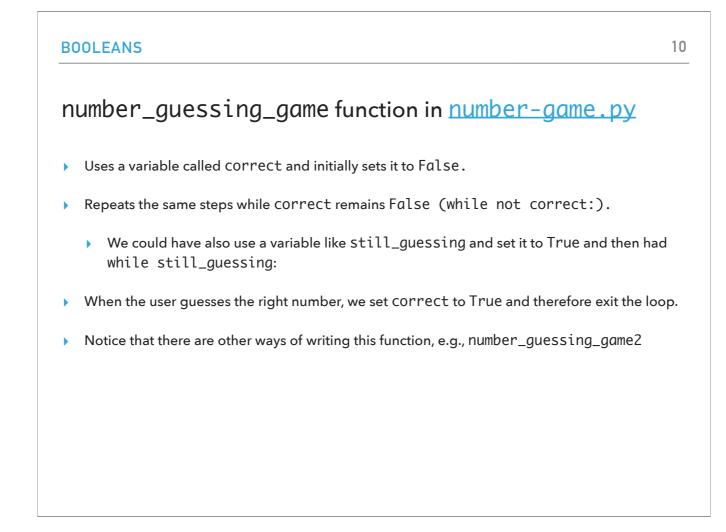
To practice, let's start by writing a method contains that will take as a parameter a list and an item and will return True if the item is in the list and False otherwise. You can look at the file contains.py to see two versions of contains that use a similar idea (a while loop) to go over the list and compare each of the elements of the list with the provided item. The difference is that in the second version we will use an intermediate bool variable to keep track of whether we have located the item while in the first version, we will return True/False directly.

BOOLEANS	
Built-in contains functionality	
Python has a built-in contains functionality in the form	>>> 2 in [1, 2, 3]
of the <code>in</code> keyword that works for lists, strings, and	True
tuples (i.e. sequences).	>>> 5 in [1, 2, 3]
Notice that for lists, the item has to be in the list as one	False
of the individual items, we cannot check for a slice.	>>> "banana" in [1, 2, 3]
	False
But this is not a problem in strings. We can check what has a string a section on article substring a string of the section	>>> [1, 2] in [1, 2, 3]
whether a string contains an entire substring not just a single character.	False
single character.	>>> [1, 2] in [[1, 2], 2,
	True
	>>> "a" in "banana"
	True
	>>> "ana" in "banana"
	True

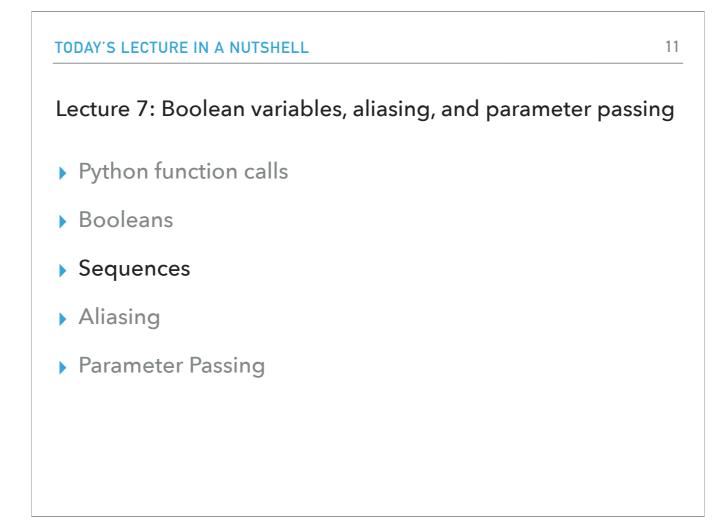
Actually, Python has a built-in contains functionality in the form of the in keyword that works for all sorts of sequences. Notice that in behaves slightly differently in lists and strings when it comes to searching for multiple elements.



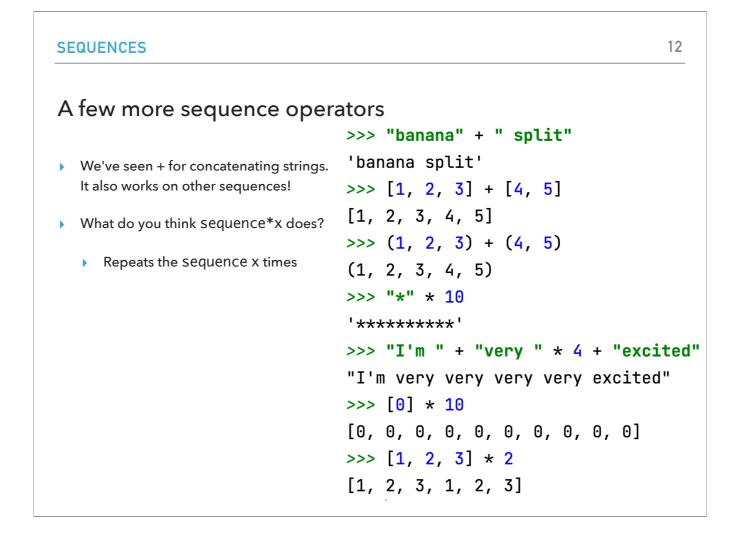
Time to play a game! If we run at the number-game.py file, we will see that it picks a random number between 1 and 20 and asks us to guess it by giving us hints until we get it :) (what a nice game!).



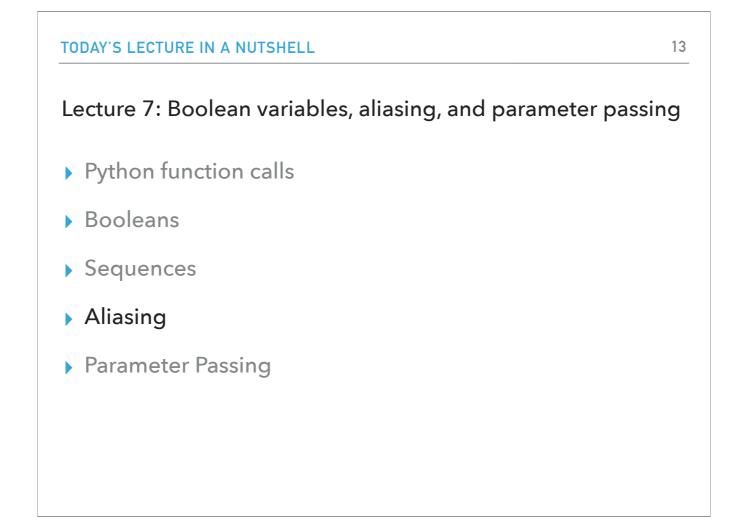
If we look at how the game is actually implemented, we will notice a number_guessing_game function that uses a variable set to False and a while loop that keeps checking whether that variable has been changed to True. If not, it will keep prompting us to keep making guesses and it will give us hints as we provide our guess. Notice that there are other ways of writing this function, e.g., number_guessing_game2



Let's go back momentarily into sequences, data structures that store data sequentially (e.g., lists, strings, tuples).



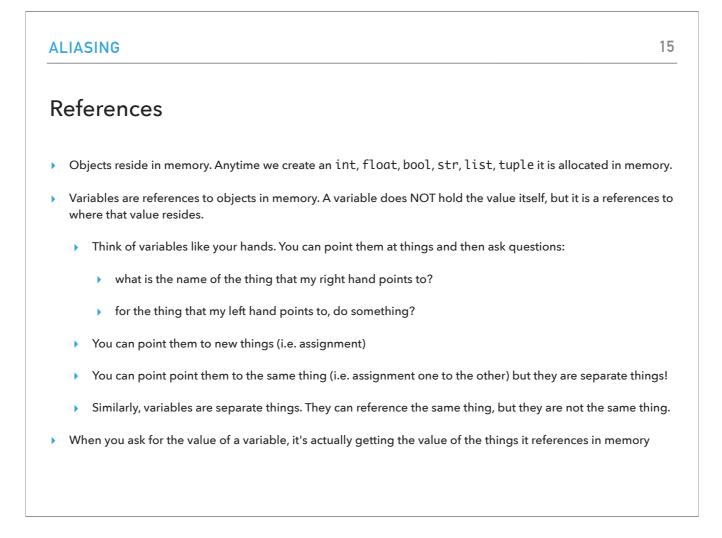
We have seen that strings can be concatenated with the + operator. We can apply the same operator for all types of sequences. Another operator that might come handy is the * which repeats the sequence as many times as we tell it. Cool, huh?



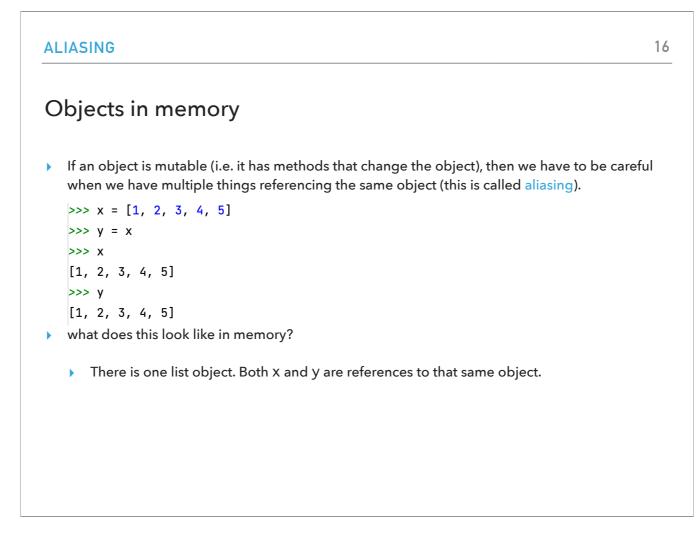
We're pivoting now to a very different topic called aliasing.

ALIASING		1
Assignment (i.e. =)	>>> x = 1	
	>>> y = 2	
when we say X = Y, what we're	>>> z = 3	
actually saying is let X reference the	>>> x = y	
same thing that y references.	>>> x	
	2	
• X and Y are still separate variables.	>>> y	
If we say $x = z$ that does NOT change	2	
the value of y.	>>> z	
	3	
	>>> x = z	
	>>> x	
	3	
	>>> y	
	2	
	>>> z	
	3	

Let's start with the basics of assignment which we have seen that is accomplished with the =. When we say x = y, what we're actually saying is let x reference the same thing that y references. But x and y are still separate variables. If we say now x = z, y will remain unaffected.



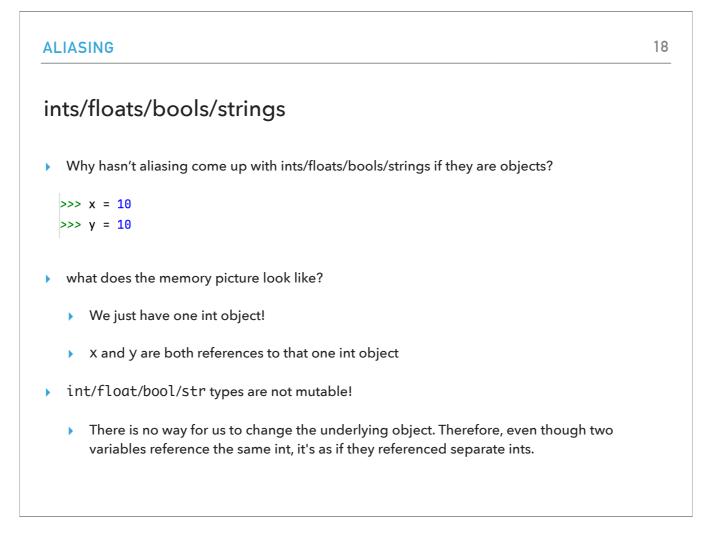
In general, all objects we create reside in memory. Variables are actually just references to objects in memory. You can think of variables as your arms (imagine having as many arms as there are variables in your program). When we assign a value to a variable, we point the variable to a memory location that stores that information.



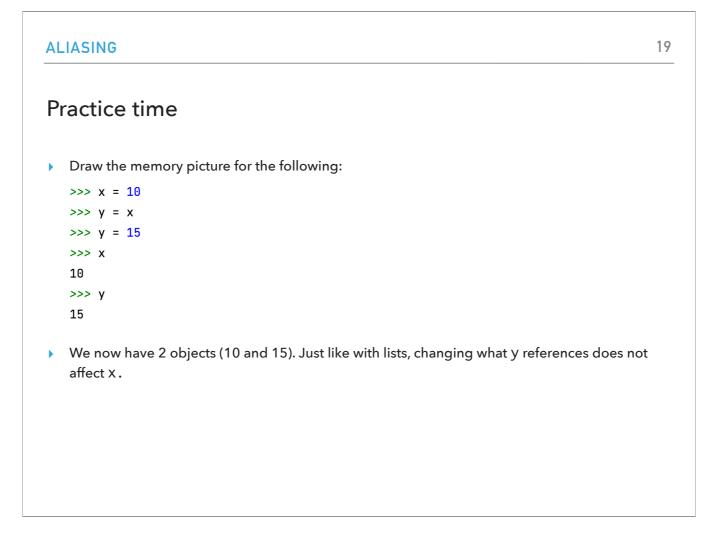
Remember that some objects can be mutable? That is they have methods (special functions called with the dot operator) that change them. In this case, we have to be careful when we have multiple things referencing the same object. This is called aliasing.

ALIASING	1	
Aliasing		
What happens when I call a method that	>>> y.reverse()	
changes/mutates the object?	>>> x	
	[5, 4, 3, 2, 1]	
	>>> y	
	[5, 4, 3, 2, 1]	
	>>> x[0] = 0	
	>>> x	
X and y are references to the same object!	[0, 4, 3, 2, 1]	
Statements that mutate the object that are done	>>> y	
on either variable will affect this object.	[0, 4, 3, 2, 1]	
	>>> y[0] = 15	
	>>> x	
	[15, 4, 3, 2, 1]	
	>>> y	
	[15, 4, 3, 2, 1]	

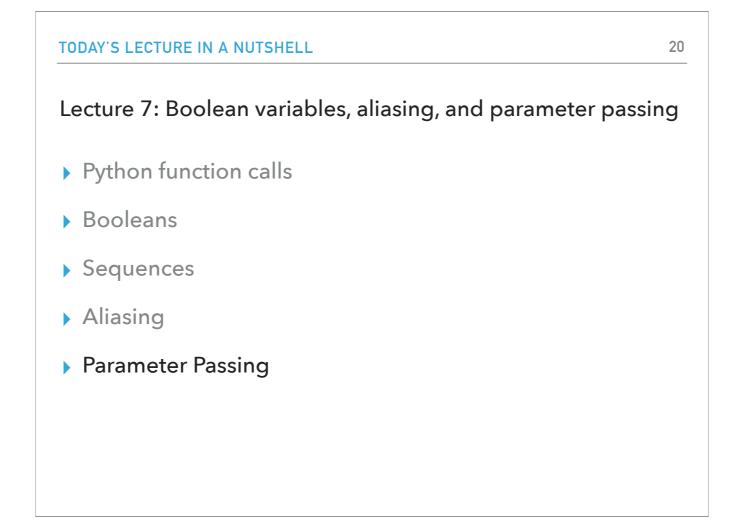
Look at the examples on the right. What x and y will point to might or might not surprise you.



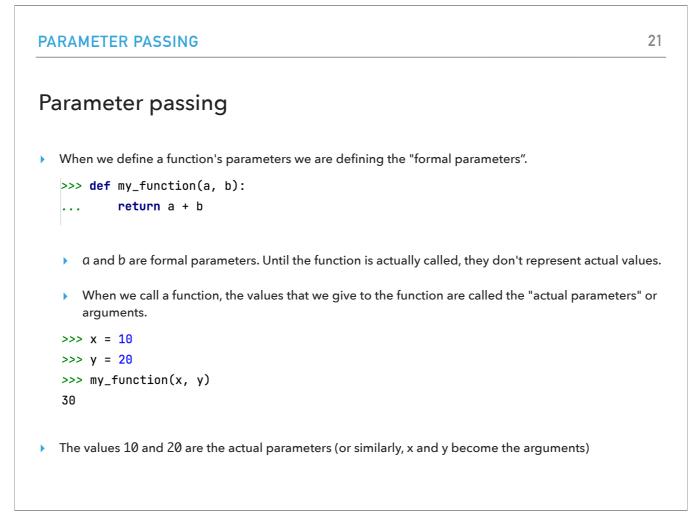
Aliasing comes up only with mutable objects (i.e. lists in our case). Any other type we have seen is immutable. If you take an int object, let's say the number 10, there's only one 10 in memory. Both x and y will point to that same 10 and they can't change it but they can point to a different number if we reassign them.



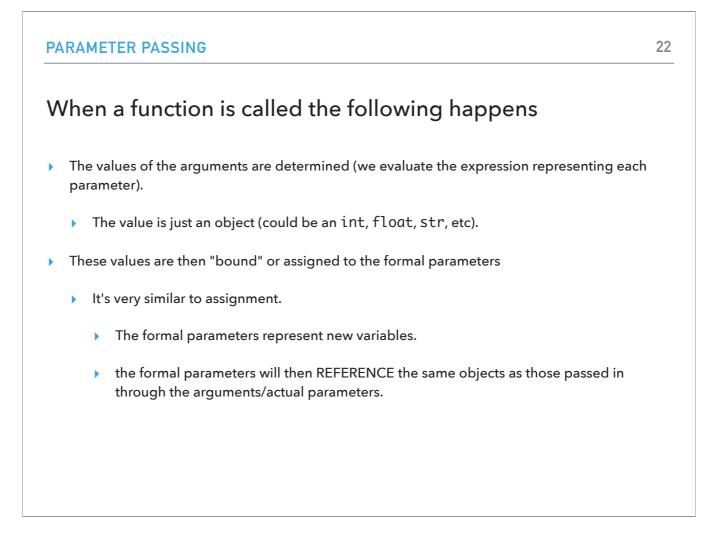
This affects what our memory allocation looks like when we run the example above.



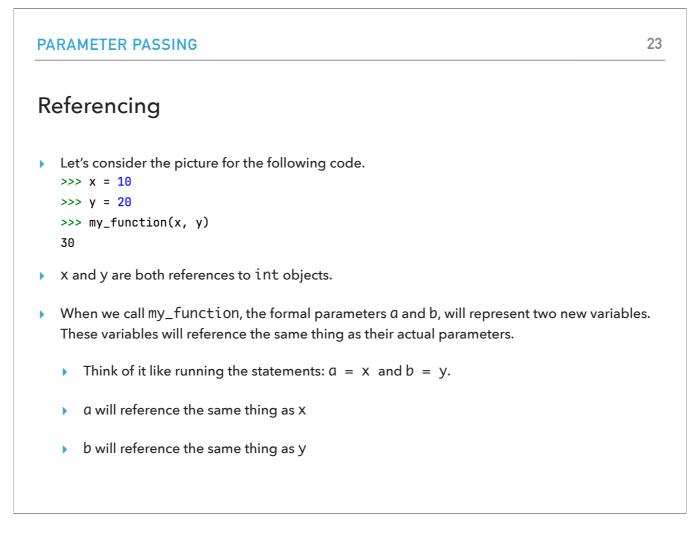
Finally, I want to talk about parameter passing which is central to working with functions.



We have already talked about parameters and arguments. Parameters are also called formal parameters and arguments actual parameters.



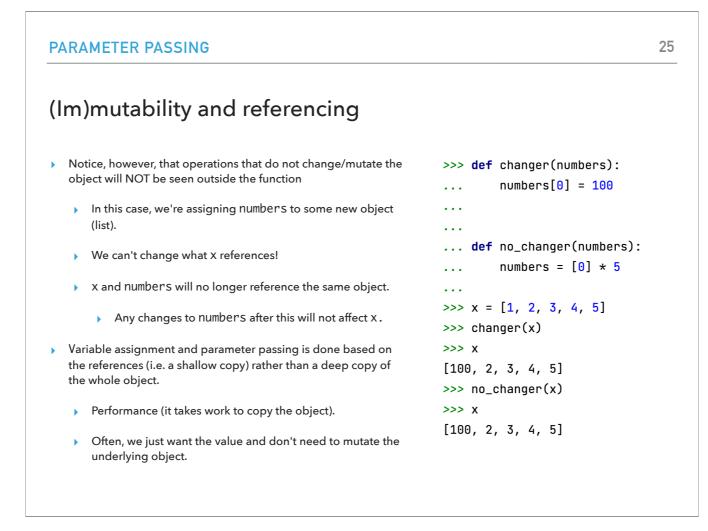
When a function is called, the values of the arguments are determined and then they are bound to the formal parameters, similar to assignment.



In the example above, both x and y are references to int objects. When we call my_function, the formal parameters a and b, will represent two new variables. These variables will reference the same thing as their actual parameters.

	\ . . . C .	
m)mutability and referencing	
	immutable objects, this whole story doesn't really matter. They	>>> def changer(numbers):
	uld be references to the same thing or copies but if we can't mutate object, the behavior is the same.	numbers[0] = 100
Но	w does this change for mutable objects, such as lists?	
•	X was a variable that references a list object	<pre> def no_changer(numbers): numbers = [0] * 5</pre>
•	When the function was called, the formal parameter numbers will represent a new variable.	
•	' numbers will reference the same thing as x.	>>> x = [1, 2, 3, 4, 5] >>> changer(x)
		>>> x
•	Think of it like running the statement	[100, 2, 3, 4, 5]
	<pre>numbers = x</pre>	
•	Because lists are mutable and since the formal parameter references the same object as what was passed in, changes made to the object referenced by numbers will also be seen in X .	

For immutable objects, this whole story doesn't really matter. They could be references to the same thing or copies but if we can't mutate the object, the behavior is the same. The story is different for mutable objects such as lists. Look at the changer function which alters the first element in the list numbers (therefore it alters what x points to).



In contrast, the function no_changer, changes the reference of numbers to a new list altogether. Now x and numbers no longer point to the same part of memory! In general, parameter passing is done based on the references (i.e. a shallow copy) rather than a deep copy of the whole object. There are performance and other reasons that we might not want to mutate the underlying object.

ASSIGNED READINGS AND PRACTICE PROBLEMS	26
Resources	
Textbook: Chapters 6	
<pre>function-calls.py</pre>	
contains.py	
number-game.py	
Homework	
 Assignment 3 (in progress) 	