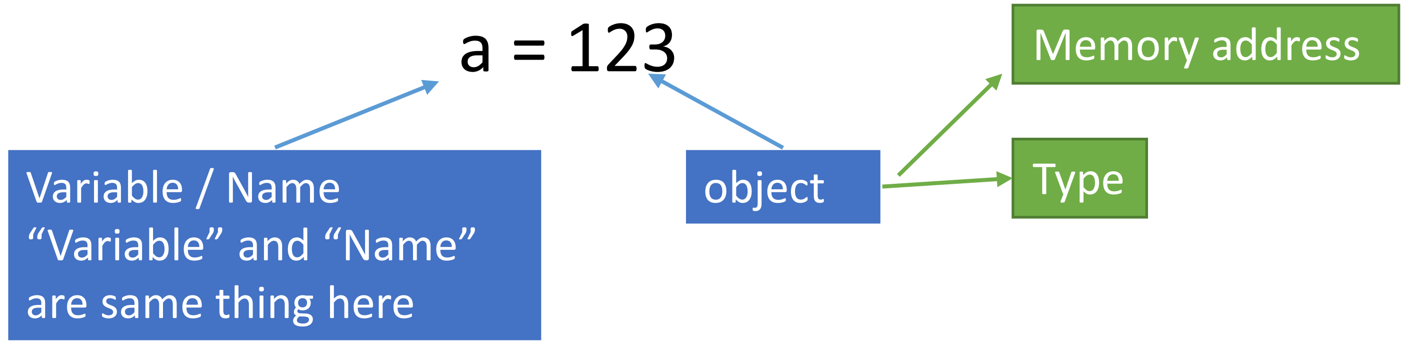
What is “assignment”

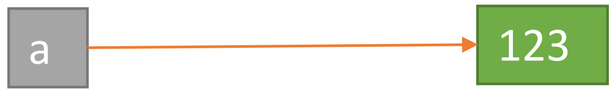


“assignment” is to bind an object to a name.
“assignment” is to bind an object to a variable.

a = 123 in technical writing:
Assigning object 123 to variable ‘a’.
Variable ‘a’ is assigned to object 123.
Assigning object 123 to name ‘a’.
Name ‘a’ is assigned to object 123.

Object has type, memory address, etc.
Strictly speaking, variable ***a*** (or name ***a***) does **not** have type or memory address.

What is “variable” or “name” ?



Variable ‘a’ is a pointer, which points to the memory address of object 123. Variable ‘a’ is used as a way to ***refer*** to object 123, so called “***reference***”. “memory address” is the memory block id where the object is stored in computer.
The following 2 statements are equivalent:
Variable ‘a’ is a reference of object 123.
Variable ‘a’ is a pointer pointing to object 123 (‘s memory address).

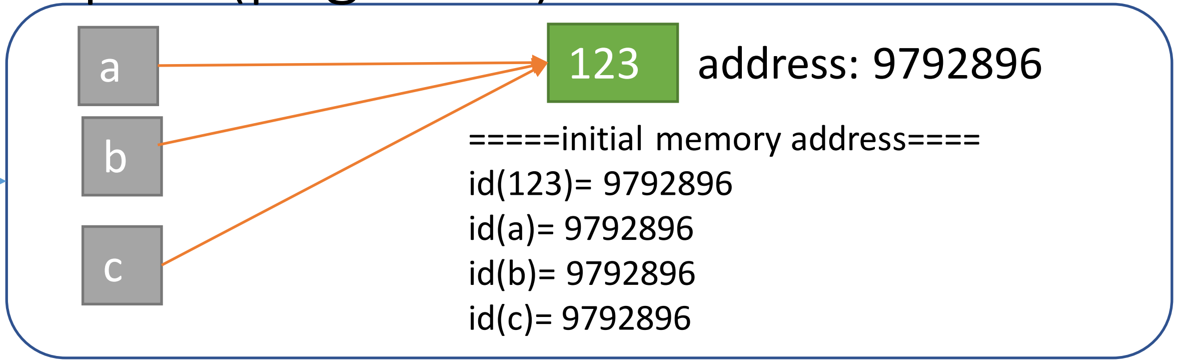
What is “dereference” ?

print(a) # example python code

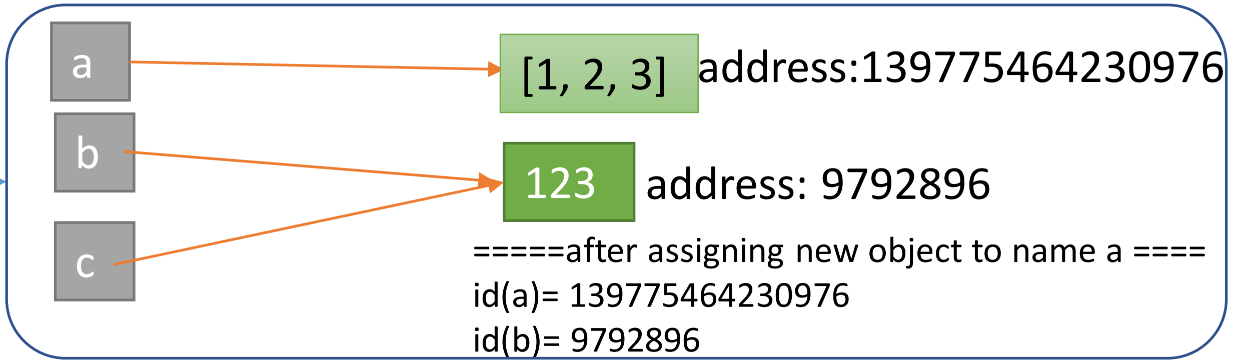
To retrieve object 123, through this pointer ‘a’, is called “dereference”. During “dereference”, computer uses pointer ‘a’ to locate memory address, gets to know memory blocks size, and translates information in those memory blocks back to proper computational values, etc.

Assignment Creates References, not Copies (page 308)

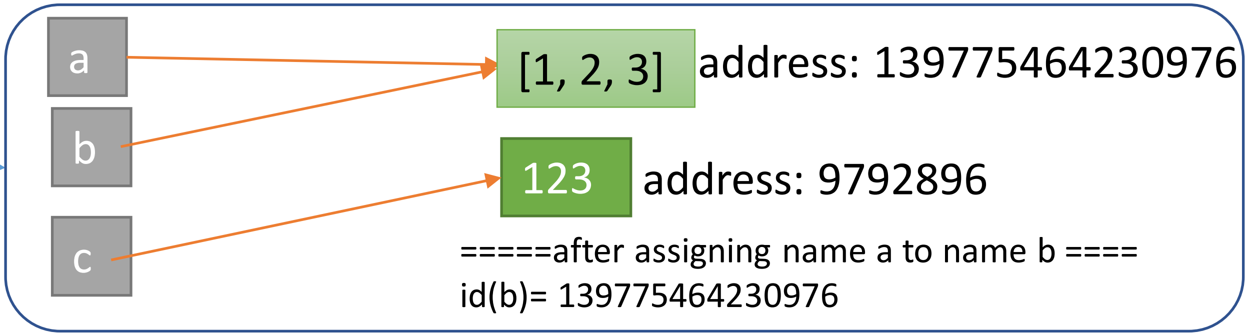
```
a=123
b=a
c=b
print("=====initial memory address====")
print("id(123)=", id(123))
print("id(a)=", id(a))
print("id(b)=", id(b))
print("id(c)=", id(c))

a=[1,2,3]
print("=====after assigning new object to name a ====")
print("id(a)=", id(a))
print("id(b)=", id(b))
print("=====after assigning name a to name b ====")
b=a
print("id(b)=", id(b))
```



```
=====initial memory address====
id(123)= 9792896
id(a)= 9792896
id(b)= 9792896
id(c)= 9792896
```



```
=====after assigning new object to name a ====
id(a)= 139775464230976
id(b)= 9792896
```



```
=====after assigning name a to name b ====
id(b)= 139775464230976
```

Python built-in function id():
Get memory address the variable is pointing to

b=a; c=b; these assignment do not create any clone copy
of existing objects. Assignment only creates references b, c. (pointers)

What is a copy
part 1 memory basics

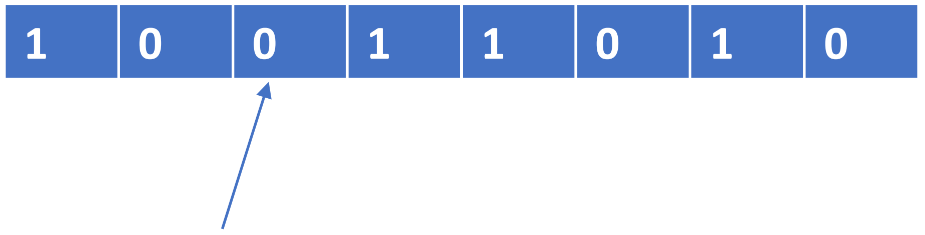
Real size Intel cpu

Real size Dell 16GB memory
able to hold 16 HD movies
Or 16*1024*1024*1024 characters
Or 4*1024*1024*1024 integers numbers

1 bit is either 0 or 1, i.e. 2 states.
So 8 bit can represent 2^8 different states. If we label all these states from 0 to n, which can map to 2^8 = 256 integer numbers, namely, from 0 to 255.
Computer can translate 1 byte in the range of 0 to 255, or -128 to 127

A character in programming language C/C++ is 1 byte, so “ASCII table” (google ASCII table) ranges up to 127.
An integer in programing language C/C++ is 4 bytes, which represent integer range:
-2,147,483,648 to 2,147,483,647

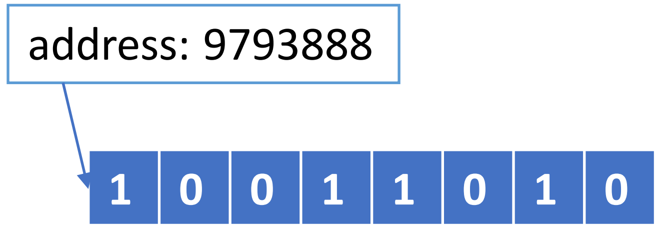
Memory size basic:
1 GB = 1024MB
1 MB = 1024KB
1 KB = 1024B
GB: gigabyte
MB: megabyte
B: byte
1 Byte = 8 bit



1 bit is either 0 or 1

Under the hood of assignment

x = 154



Object 154 is stored in the above byte (one memory block unit,
and in binary format: 10011010)

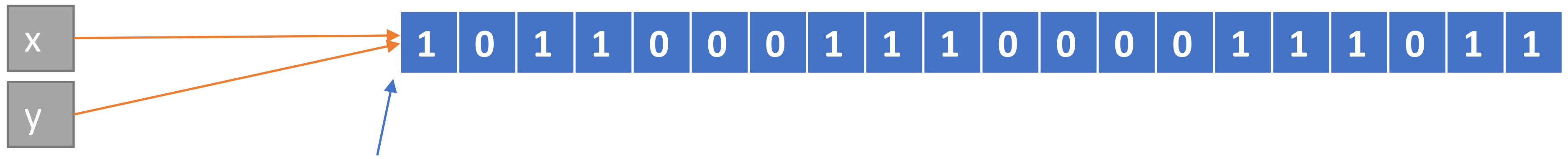
Variable ‘x’ is the memory address of object 154.
Memory address is the memory block id, where binary format of 154 are stored in memory.

y = x
z = y

These assignment only creates more pointers,
y, and z, which point to the same address 9793888, where Object 154 lives in memory.

What is a copy
part 2 a copy occupies new memory blocks

A simplified fake memory representation of list [‘a’, ‘1’, ‘zy’]



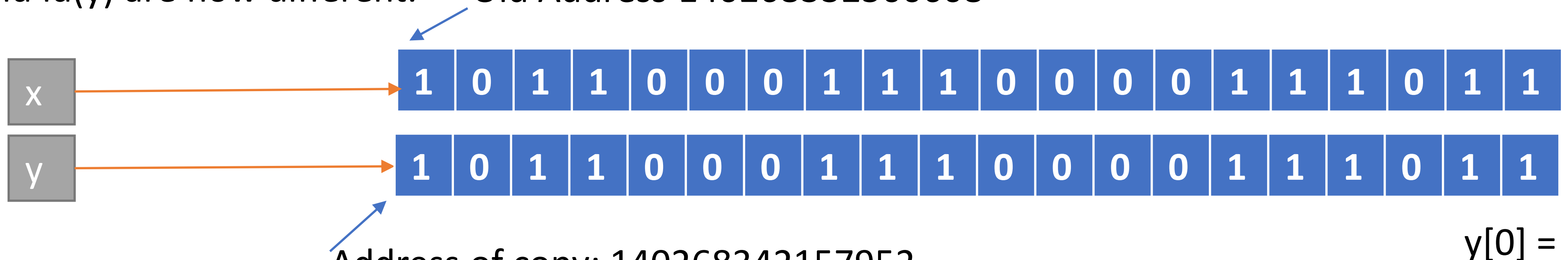
Address 140268332500608

x = ['a', '1', 'zy']
y=x # Assignment Creates References, not Copies (page 308)
id (x) and id(y) are the same:
140268332500608
Object [‘a’, ’1’, ‘zy’] is also called **“shared object”,** for x, y to sharing

Creating a copy:
y=x[:] # requesting new memory blocks
id (x) and id(y) are now different.

A copy is to clone the original memory content to new memory blocks

Old Address 140268332500608



Address of copy: 140268342157952

y[0] = ‘b’ has no impacts on old memory blocks

A copy occupies new memory blocks, with new address, with same memory content as original object. Each memory block has an unique address. **Modification** of a copy only applied to new memory blocks, and old memory block is not affected.

What is a copy
part 3 hold on!

Given: Assignment Creates References, not Copies (page 308)

y=x # Assignment Creates References, not Copies (page 308)

Why does the following assignment request new memory blocks and create a copy ???

y=x[:] # This is indeed an assignment! But assignment cannot create a copy!

(think about it for 20 seconds and see answers next page)

What is a copy
part 3 hold on! (continued)

Given: Assignment Creates References, not Copies (page 308)
y=x # Assignment Creates References, not Copies (page 308)
Why does the following assignment request new memory blocks and create a copy ???
y=x[:] # This is indeed an assignment! But assignment cannot create a copy!

Answer:
It is NOT the assignment that creats the copy. It is the operator [:] that creates a copy.
y is still a reference, because assignment creates reference not copy.
But y is now a reference to a new object, a newly copied/cloned object, with address at new memory blocks.

Page 183 line 4:
L2 = L1[:] # Make a copy of L1 (or list(L1), copy.copy(L1), etc.)
Page 183 second paragraph:
Here, the change made through L1 is not reflected in L2 because L2 references a copy of the object L1 references, not the original; that is, the two variables point to different pieces of memory.

So operator [:], or function list(), or function copy(), deepcopy(),etc are the function calls to request new memory spaces, and create copies. All name/variable assignment only create a reference to point to newly created object.
Thus, Assignment Creates References, not Copies (page 308) is true!

Go back to PPT slide 1: “What is assignment”
“assignment” is to **bind** an object to a variable. “bind” is not “create”.

Strictly speaking, technically, a “copy”, can only apply to an object. Only an object can have a “copy”. Talking about “variable copies” are meaningless, because in python all variables are names/references/pointers. Strictly speaking, technically, copy a variable like: y=x, is to “copy a reference”. But, these technical terms are too much to communicate, so “variable copies”, “object copies”, “creating a copy of the variable” are sometime causally used interchangeably.

Again, and finally, What is Assignment ?

assignment is to :

1. create a reference (or a pointer) to an object.
2. Bind this reference to a variable name.

Few people look into such details on these terminologies.
But I think it is important to know exact what do they mean.
For some python developers, assignment creating copy or not, has no impact on their day to day work. They don't care.

```
class Object:
    pass


Var = Object()
```

On the right-hand-side (rhs), a new copy of a class type: "Object" is created, new memory block is allocated.
On the left-hand-side (lhs), a reference is created and named "Var", this reference points to the left hand side object.

The assignment operator '=', never creates an object or a copy. It is always the lhs expression, such as a constructor call like Object(), or an operator [:], or a function like copy(), that creates the object.
The assignment operator '=' only creates a reference, and bind this reference to a variable name.
So repeat after me:
<u>Assignment Creates References, not Copies (page 308)</u>
"Assignment" here, I think to be precise, is the assignment operator '='.
It is the rhs expression that creates the object and allocate new memory blocks (the rhs expression can also be another reference, not necessarily involving new object creation).

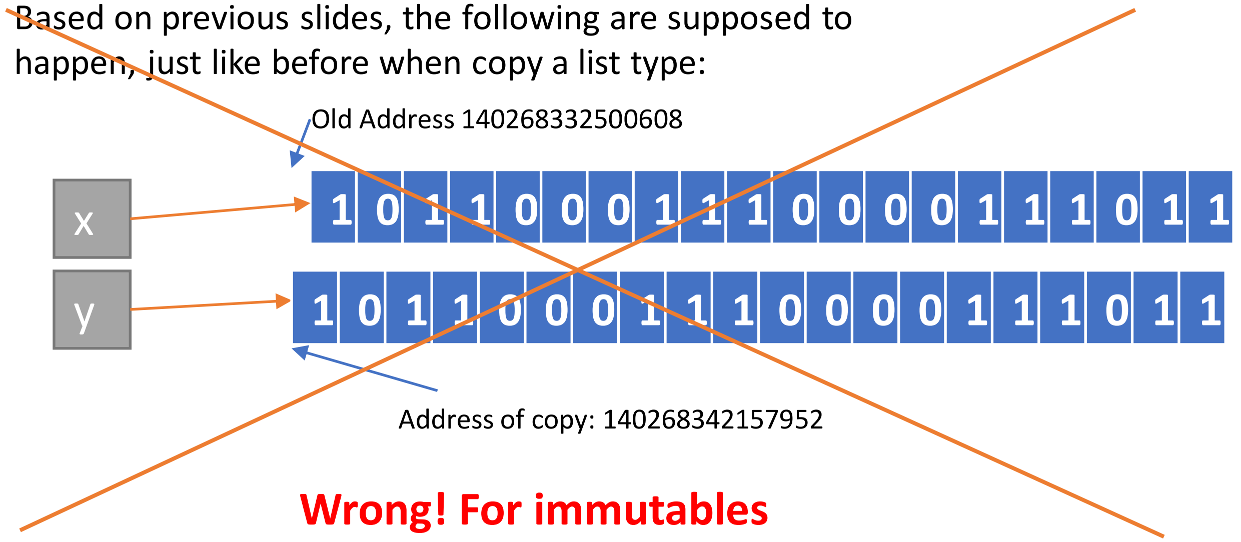
What is a copy part 4 tricky! Immutable vs mutable

How to create a copy of an immutable ?

```
from copy import deepcopy
x = (1, 2, 3, 4, 5)
y = deepcopy(x)
```

The above code perfectly creates a copy of a tuple (1,2,3,4,5), but, but, but, to your surprise, id(x) and id(y) are the same!!! id(x), id(y) are all the same: 140268331175424

Why?
Based on previous slides, the following are supposed to happen, just like before when copy a list type:



Wrong! For immutables

For immutables: numbers, string, tuples, once an object is created, it cannot be modified. It is a constant. There is no way to modify the content of an immutable object. So in python, it make sense to create only one single copy of such constant object, occupying only one memory blocks chunk, to save memory space. All "copies" of an immutable object, is the original object itself, all variables points to this single copy.

For mutables, it is different. A secondary clone copy can be created in memory.
Now review slide "What is a copy, part 2". **Modification** is highlighted for a reason. Because a mutable type (list, set, dictionary) can be modified. If we want to use a modified object, a real clone needs to be created at new memory blocks.

Finally, immutables are created directly and only once. There is no modification afterward.
x = "abc"
x = x + "d"
Variable x initially is assigned to immutable object "abc".
Later, x=x+"d", did we modify the immutable object "abc" to be "abcd"? No, we created a new immutable object "abcd", and assign the reference x to this new immutable object. We did not create a copy of immutable object "abc", then modified this object. We directly created a new immutable "abcd". Repeat after me:
immutables are created directly and only once. There is no modification afterward

For a mutable, we can do two steps: 1. create a real copy in memory, 2. modify the content of the memory blocks.
Thus a real copy is needed.

What is a copy
part 5 caching, garbage collection

x = “abc”
x = x + “d”
We talked about the above code, where two immutable objects are involved: “abc”, and “abcd”.
After x is assigned to the new immutable object “abcd”, what happens to the first object “abc”?

Python have two ways to the handle first object “abc”.
1. Cache it.
Save “abc” in that memory block, and later, if any one need an “abc” object, python does not create a new “abc” object, but instead, return the existing old object “abc” from the cache. The most common caching algorithm is the “least recently used” cache, or LRU cache, which is out of the scope of this slide.
2. Garbage collect it.
Python can also decide to wipe out the memory blocks of the object “abc”, and return this memory blocks to system for later use. “abc” will be destroyed, like “garbage”, no one use it anymore. Returning the memory blocks to system is called garbage collection.

1 or 2, which way should python do, is out of scope of this slide. Memory management and garbage collection involve advanced topics such like slab allocator, mark and sweep algorithm, etc.

Cache pronounces the exact same way as “cash”. A pool of memory blocks to save frequently used objects.
Searching for “python reference counting” to level up your understanding of this topic if time permit.

Revisit chapter 6 page starting at page 176, Variables, Objects, and References

In C/C++, Types are static, once declared, cannot change anymore

```
double x = 12.345;  //declare x initially as type "double"
x = "abc"; // compilation error, try to change x type to string
string s = "abc"; //declare s initially as type string
s = 123; // compilation error, try to change s type to int
```

In python, types are dynamic because every variable is a 'universal pointer', can points to any object at any time. All followings are OK.

```
x = 12.345
x = "abc"
s = "abc"
s = 123
```

This is called "dynamic typing" (vs. C/C++ static typing)

Python 's dynamic tying are all based on all variables are only 'universal pointer', and are only names that binded/assigned to objects, not the objects themselves. Only objects have types, and python objects still have "static typing" the same as C/C++. Python's dynamic typing applies to variables/names, not objects.

```
[1,2,3] + 1 #python code, objects are still static typing
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

To make different python type to work together, need type conversion operators, or functions.

number	string	tuple
list	set	dictionary

Python built-in types
Dark blue: immutable
Light grey: mutable

Understanding Argument-passing basics (page 523)

Part 1 preparation

What we should know by now:
What is variable
What is object
What is assignment
What is copy
What is reference
What is pointer
What is shared object
Difference of making a copy of:
immutable vs mutable

What is “assigning objects to a variable”
What is “variables are assigned to objects”

New terminologies:

1. function arguments:

[Python Function Arguments (w3schools.com)](#) <- click

```
def f(arg1, arg2):
   arg1 += 1
   return arg1 + arg2
```

function scope, or local scope

Values/objects of arg1, and arg2 are function arguments.

2. caller:

```
x = 1
y = 2
caller_need_result = f(x, y)
```

caller scope

The above 3 lines represent how a caller can use the function f.
caller usually does not care what happens in the function f.
caller usually does not care what python code in the function f.
caller simply invoke/call the function f to get result.

3. scope, local names

Starting from function header line: def f(arg1, arg2)
To line: return arg1 + arg2
This is the function local scope.
Starting from x=1, to caller_need_result = f(x, y)
This is a caller scope.

arg1, arg2 are names only visible in function scope, they are **local names**.
arg1 is never declared in caller scope so cannot be used, like
print(arg1) in caller scope:
NameError: name 'arg1' is not defined

4. Input and output for functions

```
def f(arg1, arg2):
   arg1.append(1)
   return len(arg1) + len(arg2)
```

arg1, and arg2 are input
len(arg1) + len(arg2) is output
But if arg1 is mutable, arg1 can also be an output

Example **:
x=[1,2,3]
y=[4,5]
z=f(x,y)
Now z is 6, and x is [1, 2, 3, 1].

Page 523 last line:
Mutable arguments can be input and output for functions.
Describing the x change above.

Understanding Argument-passing basics (page 523)
Part 2 reading between the lines item a)

Arguments are passed by automatically assigning objects to local variable names. Page 523

Removing "automatically, variable" -> **Arguments are passed by assigning objects to local names.**

In last page **Example **,**

```
def f(arg1, arg2):
   arg1.append(1)
   return len(arg1) + len(arg2)

x=[1,2,3]
y=[4,5]

z=f(x,y)
```

Local names are arg1, and arg2,
f(x,y) is to do:
arg1 = x
arg2 = y

Assigning objects to local names.
Remember assignment creates reference not copies.
So books says: "Objects passed as arguments are never automatically copied."

Book quote "Function arguments—references to (possibly) shared **objects** sent by the caller"
Be carefull, arguments are objects, not names. So arguments are [1,2,3], and [4,5], not arg1, arg2.
arg1 and arg2 are literally "parameters", or local names, the objects they point to are arguments, e.g. [1,2,3], [4,5].

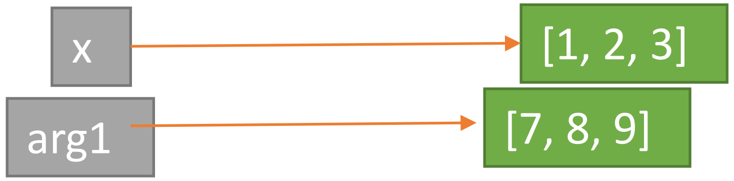
But, but, in real world, few people distinguish such subtle difference.
For most python developers, "arguments, parameters, arg1, arg2," are the same thing. Which is technically wrong, but few people pay attention.

Understanding Argument-passing basics (page 523)
Part 2 reading between the lines item b)

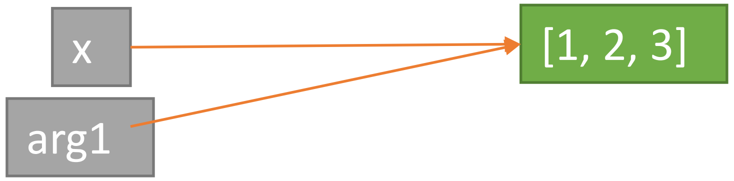
Assigning to argument names inside a function does not affect the caller. Page 523

The author is very precise here. What is “argument names” ?
“argument names” are just local names, local variables.
argument names are not arguments.
Remember, arguments are objects, real values.
argument names are names binded to thoese objects.

```
def f(arg1, arg2):
    arg1 = [7,8,9] #*line
    return len(arg1) + len(arg2)
```



```
x=[1,2,3]
y=[4,5]
z=f(x,y)
```



Does *line change the caller scope variable x? No.
x in caller scope is still pointing to [1,2,3]
arg1 is pointing to new object [7,8,9]

Book says: “There is no aliasing between function argument names and variable names in the scope of the caller.”

Aliasing:
Aliasing refers to : the same memory location can be accessed using different names.

Tricky here!
Yes, x and arg1 initially are pointing to the same memory location after line:
z=f(x,y)

But x and arg1 are not “aliasing”, they are not “forced” to always point to the same memory location.
So after line:
arg1=[7,8,9]
Assignment creates a new reference to variable arg1, not to x.

If they are “aliasing” (not the case in python function scope parameters), Then x will also points to [7,8,9], because they are “forced” to always point to the same memory location. This is not the case in python function argument passing.

Comparing to **Example ****, at part 1. to see the difference of “in place change of shared object” vs. no aliasing here.

Understanding Argument-passing basics (page 523)
Part 3 reading between the lines item c)

Changing a mutable object argument in a function may impact the caller. Page 523
After part1 and part2, this part3 should be self-explanatory.

Next Very tricky part
Immutable arguments are effectively passed “by value.” page 524
Mutable arguments are effectively passed “by pointer.” page 524

“pass by value”, is to create a new copy, at new memory blocks when an argument is assigned to a function local variable name, at function call. Python, conceptually, do “pass by value” only for immutables.
But, but, but, revisit “what is a copy, part 4”, there is no additional copy for immutable types. All immutables are created only once and never copied, never modified. So that’s why “pass by value” only existing in concept in python. In reality, all argument passing are by pointers/reference.

The author here is to explain to C/C++ programers, that “pass by value” does not really exist in python.
You just need to understand his statement:
Of course, if you’ve never used C, Python’s argument-passing mode will seem simpler still—it involves just the assignment of objects to names, and it works the same whether the objects are mutable or not.