# Mutability and Lists

March 30, 2020

## Python variable strangeness

Let’s look at some code snippets.  This will work like you expect it to:

```
a = 1
b = a
b += 1
print(a)
```

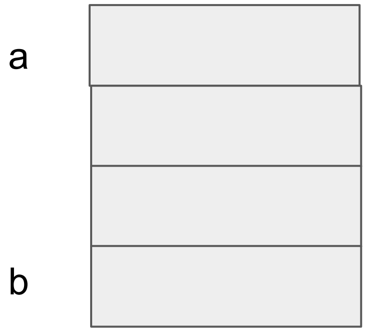
But this might produce some surprising output:

```
c = [1, 2, 3, 4]
d = c
d.append(5)
print(c)
```

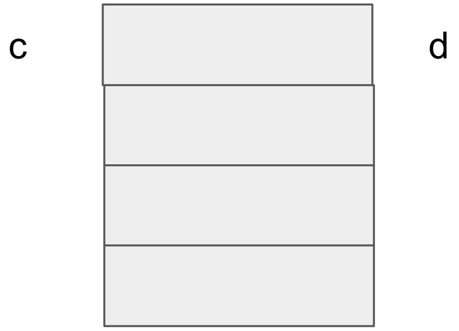
## So why did that happen?

Python handles assignment different for different types of variables. There are two basic categories: “immutable” variables, and “mutable” variables.

For simple types like int, float, str,  Python just copies the value:  b=a copies the value of a into a new location, labeled b. a and b point to different locations in memory. These types are immutable.



For types that are built out of other types, Python handles assignment by passing a reference to the original variable. c and d now point to the same locations in memory. These types are “mutable”



## What do “mutable” and “immutable” mean?

“Immutable” - cannot be changed after being created. Ever. For any reason.

- Wait? What about

  a = a + 1?

  Doesn’t that change the variable “a?”

  Not really. It actually creates whole new object - a new location in memory - where it stores the new value. The previous object is destroyed.

“Mutable” - can be changed after being created. Subject to rules.

- Appending a value to a list changes, or mutates, the list.
- So does inserting a value, popping a value, or removing a value

But making another assignment creates a whole new list object!!

## Illustrations using the Python “id” statement

```
a = 1
id(a)
a += 1
id(a)

b = a
id(b)
b += 1
id(b)
```

```
x = []
id(x)
x.append(1)
id(x)
x.insert(1,2)
id(x)

x = [1,2,3,4]
id(x)
```

```
c = [1,2,3,4]
id(c)
d = c
id(d)
d.append(5)
id(d)
print (c )
```

## Strings are immutable. But aren’t they like lists, and lists are mutable?

Try this:

```
  state = ‘Maryland’
  state[0] = ‘G’
```

It fails. Because strings are really immutable. Once you create a string, you can’t change it. You can only create a new string (with the same name) in a separate location in memory. You have to say:

```
  state = ‘Maryland’
  state = ‘Garyland’
```

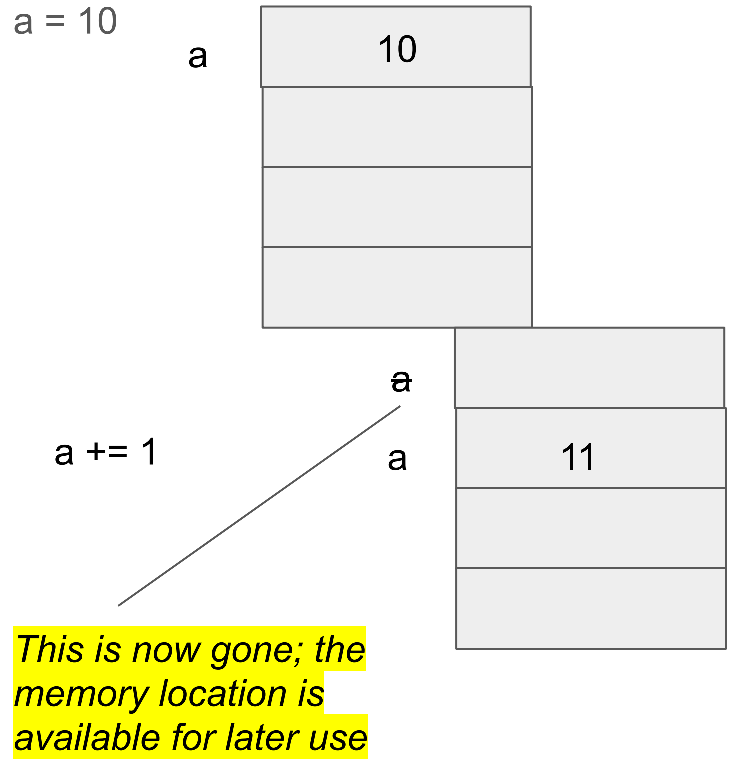
But wait - we can ***read*** individual letters or slices of a string, right?

Yes, those are allowed because they don’t ***change*** the value of the string. They just access part of it.

```
  state = ‘Maryland’
  if state[:4] == ‘Mary’:
      print (‘yes’)
```

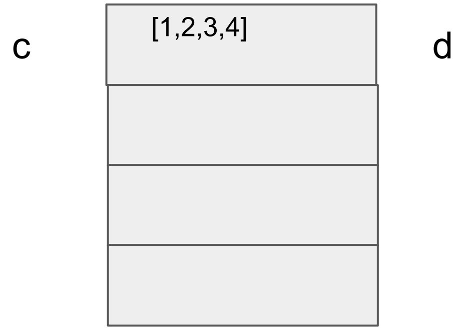
# What’s really happening in memory

a = 10



*This is now gone; the memory location is available for later use*

c = [1,2,3,4]
d = c



## Why on Earth would you create a language like this?

***No, this is not on the test!!!***

- Because immutable objects - variables - are much faster to access when the program is running.
  - It’s very hard to change them; you actually have to create a copy to add 1 to an integer
  - But recent experience shows us that lots and lots of coding involves objects that never actually change
- Mutable objects take longer to access, but they’re easier to change
- This design appears to reach a good trade-off between speed of access and cost to change

## How can you tell which is which?

Immutable objects: int, float, bool, str, tuple, unicode (don’t worry about those last two yet)

- Assignment works like the left side of the previous slide: b = a means copy the value stored in a into b.

Mutable objects: list, set, and dict.  (We’re going to get to “dict” on Thursday)

- Assignment works like the right side of the previous slide: d = c means that d gets the *address in memory* of c and thus they point to the same locations

“Mutable” means “can be changed”; “immutable” means “can’t be changed.”

## If that’s not strange enough - passing a list to a function

```
def strange_stuff (a_list):
  a_list.append(5)
  return

# This function doesn't return any value
# that's usable by the main program. So
# what will happen?
```

```
# a calling program
if __name__ == "__main__":
   main_list = [1, 2, 3, 4]
   strange_stuff(main_list)
   print(main_list)
```

## WHY??!!!!

Because constructed types like lists are mutable, and that mutability applies across function calls.

Mostly, because that’s the way Guido van Rossum (who originally wrote Python) thought it should work, and enough users agreed with him that this caught on.

# Deep Copy vs Shallow Copy

So what do you do if you want to just work with a copy of the original list?

Two options: use the list constructor function build a copy

```
list_copy = list(main_list)
```

Or use a slice:

```
list _copy = main_list[:]
```

These two operations copy the elements of the list into a new object, so you do not mess with the contents of the original.

This is fine if it’s a list of integers or floats or strings.  What if it’s a list of lists?

```
a = [
  [1,2,3],
  [4,5,6],
  [7,8.9]
]
b = list(a)
b[1].append(10)
```

## Shallow copy

b = list(a)

creates a *shallow copy* of a. b is a different location in memory than a. (id(b) != id(a))

But the *components* of b are the same locations in memory as the *components* of a.

id(b[0]) = id(b[1]), etc.

So you can still have unintended side effects

## Deep copy

To make a truly independent copy of a list, you have to make a copy of all of its components, not just of the top-level list.

If the list is composed of ints, strings, floats, booleans, then shallow and deep copies are effectively the same.

When a list is made of other lists, this is more complex

## What you need to take away from this lecture

“Mutable” means “can be changed after creation.” “Immutable” means “cannot be changed after creation.”

In Python, simple types - ints, floats, booleans, strings - are immutable. Complex types - lists - are mutable.

That means they behave differently when you make assignments. You can change the value of mutable variables without meaning to if you’re not careful.

To prevent unintentionally modifying a list, make a copy and work with that.