Graphics and Computer Vision

Passing Parameters

Let's review some concepts

What does the code below output?

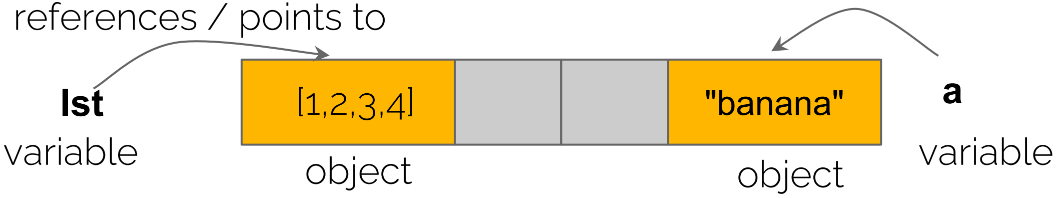
```
def nested_for(numbers):
  for x in numbers:
    for y in numbers:
      print(x, y)

nested_for([1,2,3])
```

y

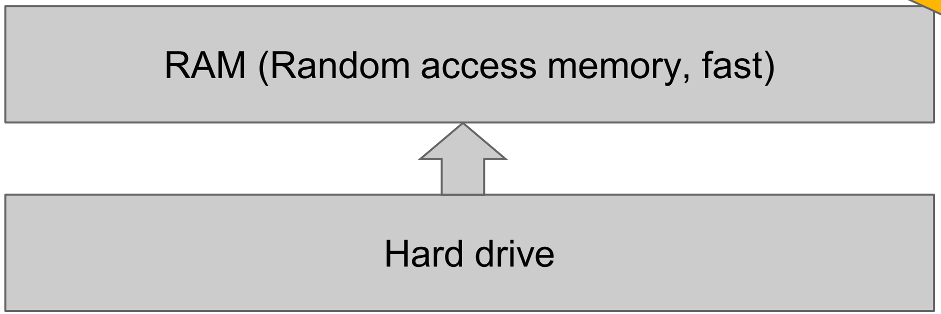
		1	2	3
	1			
x	2			
	3			

Data is stored on your PC like this:

```
lst = [1,2,3,4]
a = "banana"
```

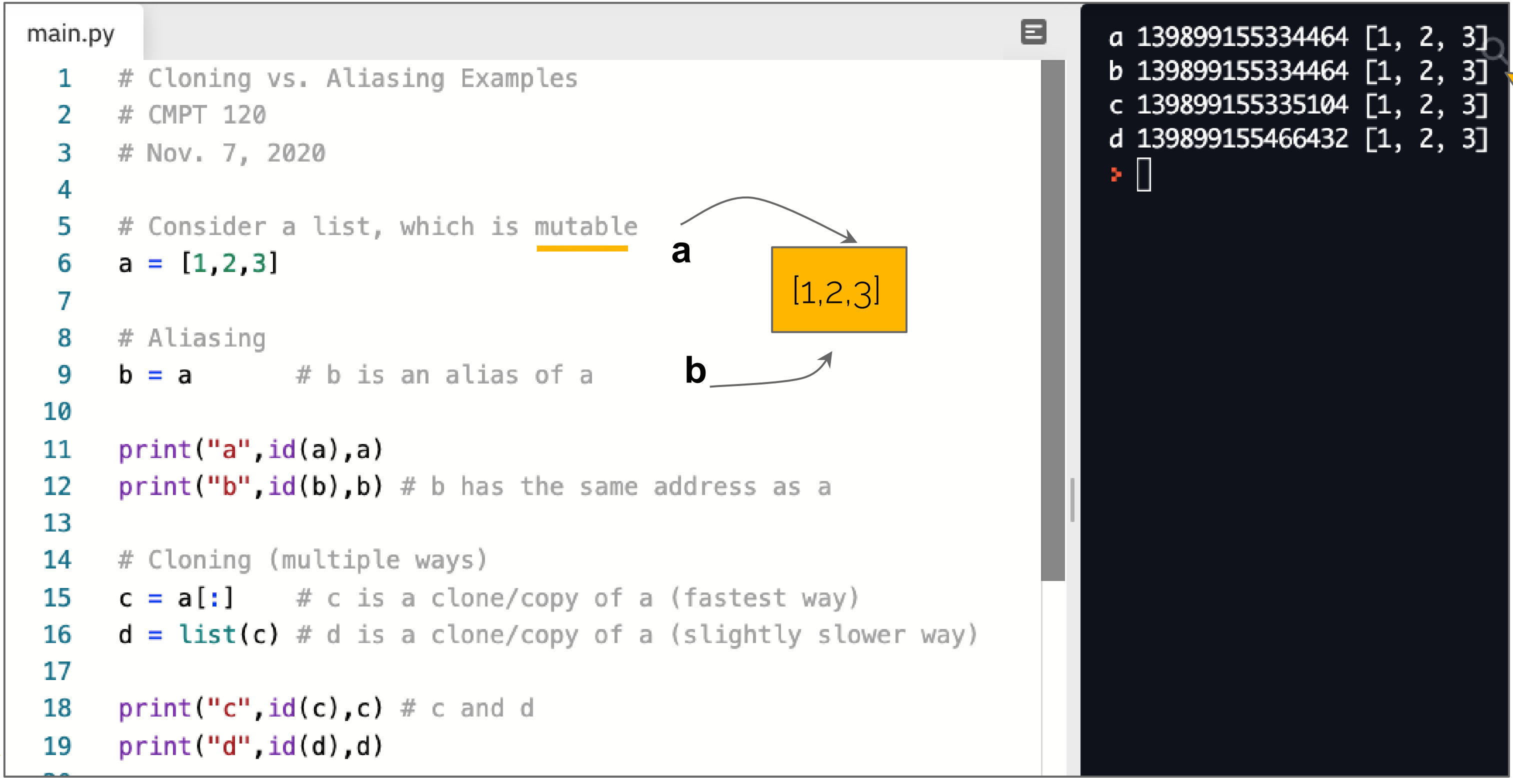


Your program's data is stored as *objects* in a semi-contiguous fashion in RAM while your program is running. Each object has an *address* in memory, and variables are a *reference* (pointer) to that address.



You can check the unique identifier of any object by using **id(obj)**

Aliases



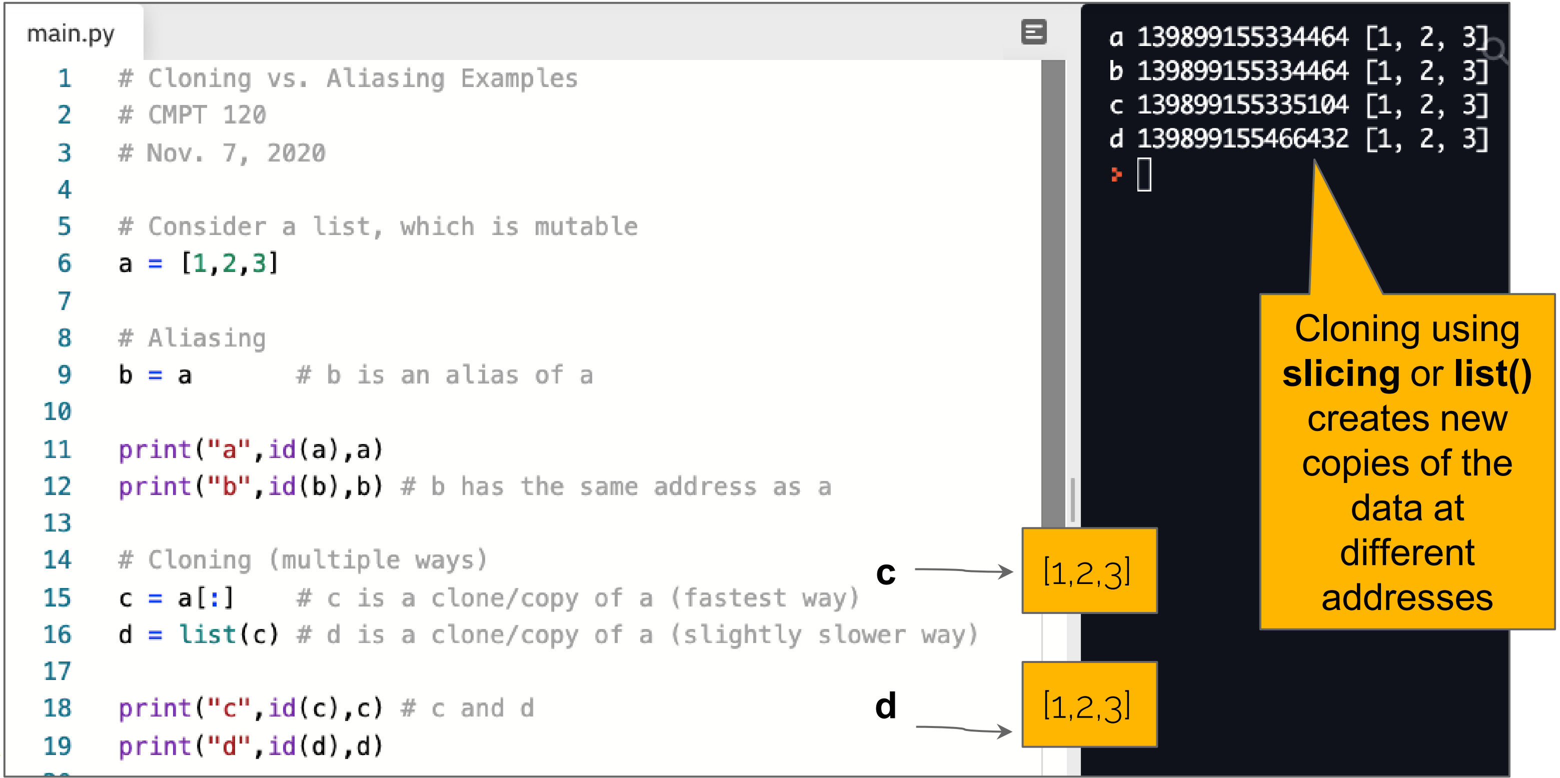
main.py

```python
# Cloning vs. Aliasing Examples
# CMPT 120
# Nov. 7, 2020

# Consider a list, which is mutable
a = [1,2,3]

# Aliasing
b = a       # b is an alias of a

print("a",id(a),a)
print("b",id(b),b) # b has the same address as a

# Cloning (multiple ways)
c = a[:]    # c is a clone/copy of a (fastest way)
d = list(c) # d is a clone/copy of a (slightly slower way)

print("c",id(c),c) # c and d
print("d",id(d),d)
```

```
a 139899155334464 [1, 2, 3]
b 139899155334464 [1, 2, 3]
c 139899155335104 [1, 2, 3]
d 139899155466432 [1, 2, 3]
›
```

Aliases point to the **same** object in memory, with the same address

Copies (aka clones)



```python
# Cloning vs. Aliasing Examples
# CMPT 120
# Nov. 7, 2020

# Consider a list, which is mutable
a = [1,2,3]

# Aliasing
b = a       # b is an alias of a

print("a",id(a),a)
print("b",id(b),b) # b has the same address as a

# Cloning (multiple ways)
c = a[:]    # c is a clone/copy of a (fastest way)
d = list(c) # d is a clone/copy of a (slightly slower way)

print("c",id(c),c) # c and d
print("d",id(d),d)
```

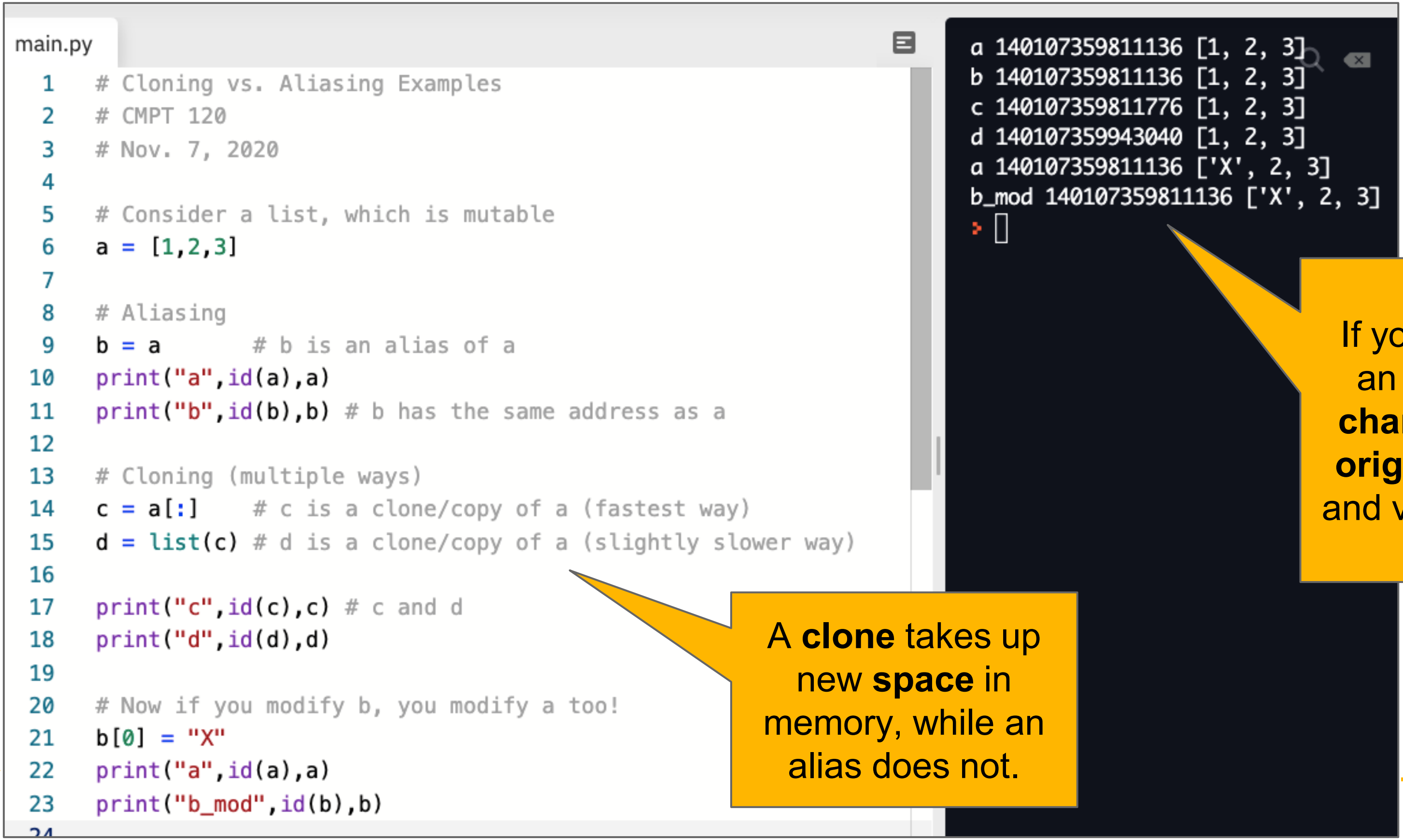
```
a 139899155334464 [1, 2, 3]
b 139899155334464 [1, 2, 3]
c 139899155335104 [1, 2, 3]
d 139899155466432 [1, 2, 3]
> 
```

c → [1,2,3]

d → [1,2,3]

Cloning using **slicing** or **list()** creates new copies of the data at different addresses

Why is this important?



```python
# Cloning vs. Aliasing Examples
# CMPT 120
# Nov. 7, 2020

# Consider a list, which is mutable
a = [1,2,3]

# Aliasing
b = a        # b is an alias of a
print("a",id(a),a)
print("b",id(b),b) # b has the same address as a

# Cloning (multiple ways)
c = a[:]     # c is a clone/copy of a (fastest way)
d = list(c) # d is a clone/copy of a (slightly slower way)

print("c",id(c),c) # c and d
print("d",id(d),d)

# Now if you modify b, you modify a too!
b[0] = "X"
print("a",id(a),a)
print("b_mod",id(b),b)
```

```
a 140107359811136 [1, 2, 3]
b 140107359811136 [1, 2, 3]
c 140107359811776 [1, 2, 3]
d 140107359943040 [1, 2, 3]
a 140107359811136 ['X', 2, 3]
b_mod 140107359811136 ['X', 2, 3]
› ▯
```

If you modify an **alias**, **it changes the original**, **too**, and vice versa!

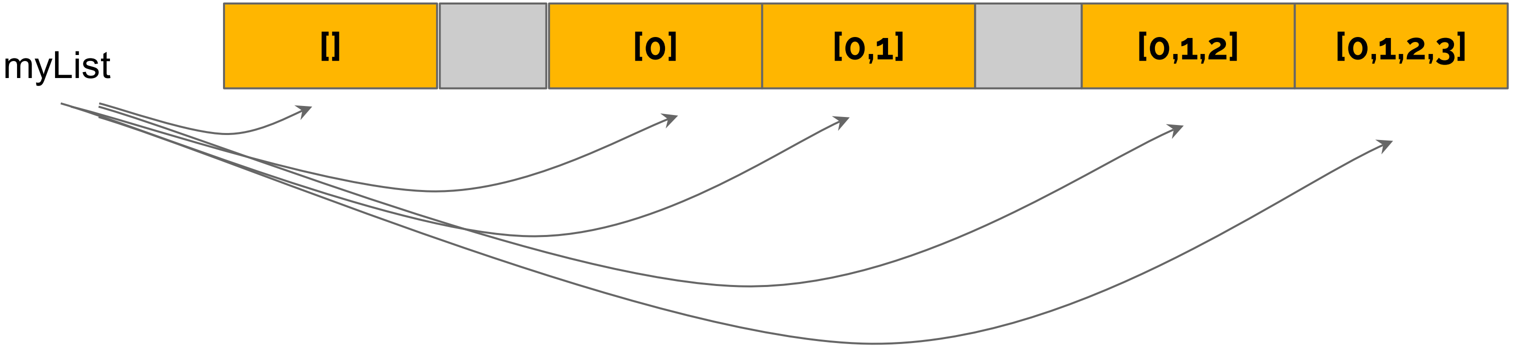
A **clone** takes up new **space** in memory, while an alias does not.

Concatenation vs. Append

```
myList = []
for i in range(4):
  myList = myList + [i]
```

If you replace this line with **myList.append(i)**, you'd overwrite the same area in memory, reducing the storage space needed for your program! :)

Under the hood, **5 copies** of myList are created and use up space, and myList is given a new address each time. Not very space efficient!!

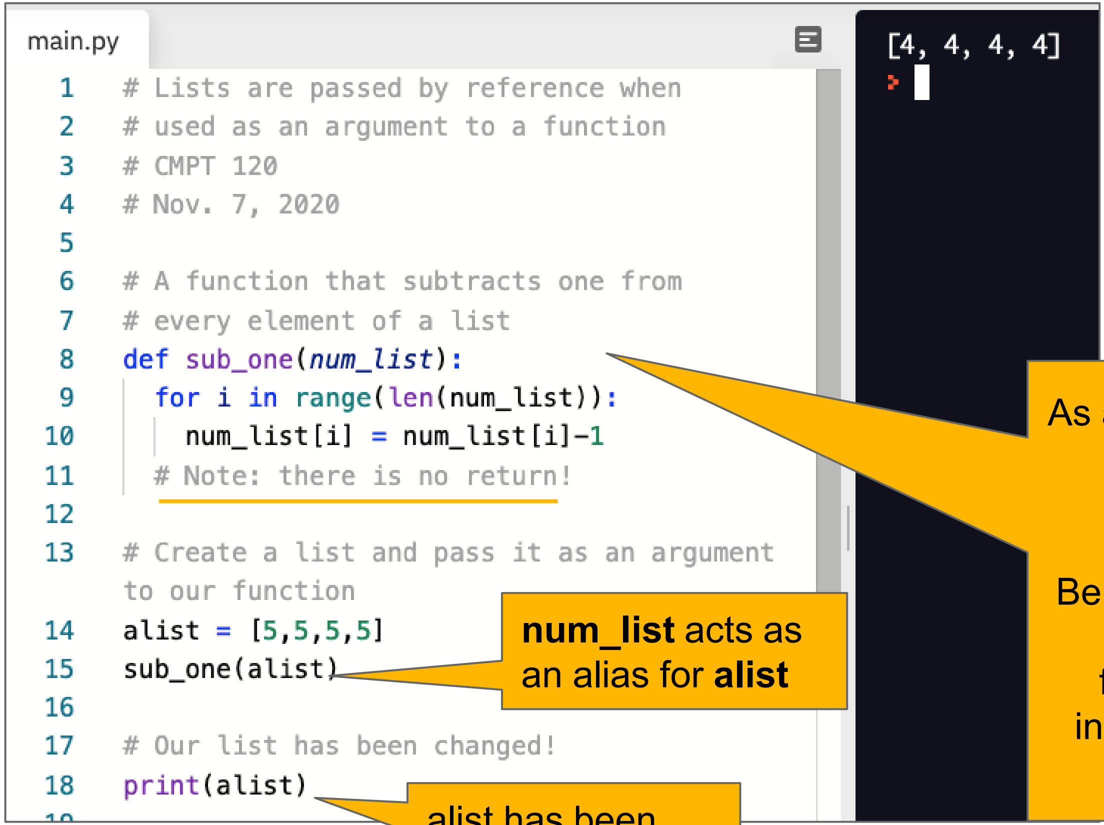


myList

[]		[0]	[0,1]		[0,1,2]	[0,1,2,3]

Note: Data is not necessarily stored in contiguous memory.

Lists are passed by reference



main.py

```python
# Lists are passed by reference when
# used as an argument to a function
# CMPT 120
# Nov. 7, 2020

# A function that subtracts one from
# every element of a list
def sub_one(num_list):
  for i in range(len(num_list)):
    num_list[i] = num_list[i]-1
  # Note: there is no return!

# Create a list and pass it as an argument to our function
alist = [5,5,5,5]
sub_one(alist)

# Our list has been changed!
print(alist)
```

```
[4, 4, 4, 4]
> 
```

As an optimization, lists are **passed by reference** when used as arguments to a function.

Be aware of this: there are no local copies of the list made in the function, so any modifications inside the function will happen to the original list passed in!

num_list acts as an alias for **alist**

alist has been modified :O

Note on cloning 2D lists

The **list slicing** and **list()** ways of cloning lists only works for 1-dimensional lists. For lists that contain sublists, only a **shallow copy** will take place of the outer list. The elements in the **sublists** will still point to the original objects.

In the final project, you will need to be aware of this.

If you do not wish to change the original image, create a new image using getBlackImage(), for example.

Efficiency and optimizing your program

What we learned:

- Consider whether you can modify your data in place or if you are creating extra copies where unnecessary (e.g. using append() vs. concatenation)
- Lists are *passed by reference* as function arguments to save space, but it's important to know this to avoid coding mistakes

Review: Clone vs Alias

- Which of these is true about alias and clone?
  a) Taking the **alias** of an object gives you a **new object.**
  b) Taking the **clone** of an object gives you a **new object.**
  c) Both are true.
  d) Neither is true.

- Which of these is true about alias and clone?
  a) Modifying an **alias** of an object **modifies the original.**
  b) Modifying a **clone** of an object **modifies the original.**
  c) Both are true.
  d) Neither is true.

Review: Passing to functions

- If a function is passed a list, which of the following is true?

  a) Trying to change a list passed to a function will crash the program.

  b) Changes to the list inside the function do not affect the original because lists are cloned when passed.

  c) Changes to the list inside the function affect the original because lists are passed by reference.

Review: Adding to a list

- Given the variable:

  ```
  names = [“Brian”, “Bhavana”, “Sue”]
  ```

  Which of the following adds “Max” to the list without creating extra copies of the list?

  a) `names += “Max”`
  b) `names += [“Max”]`
  c) `names.append(“Max”)`
  d) `names.append([“Max”])`