Object Internals in Python

- **Mridu Bhatnagar**

Who am I?

- Python Enthusiast

- Backend Developer by profession.

- I love speaking at various meetup groups and conferences.

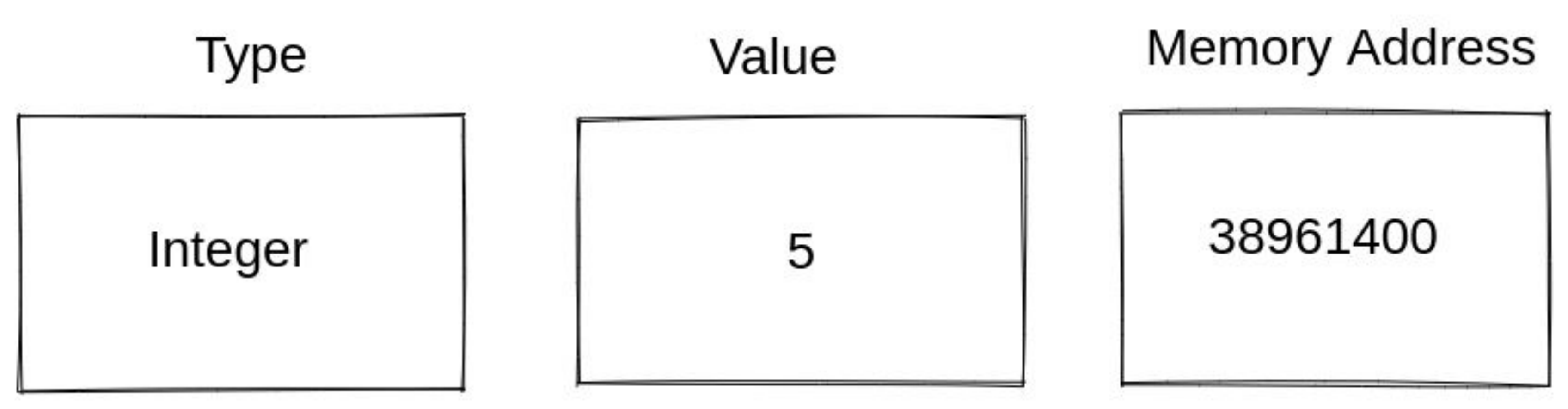
- Twitter handle - @Mridu__

Learning Objectives

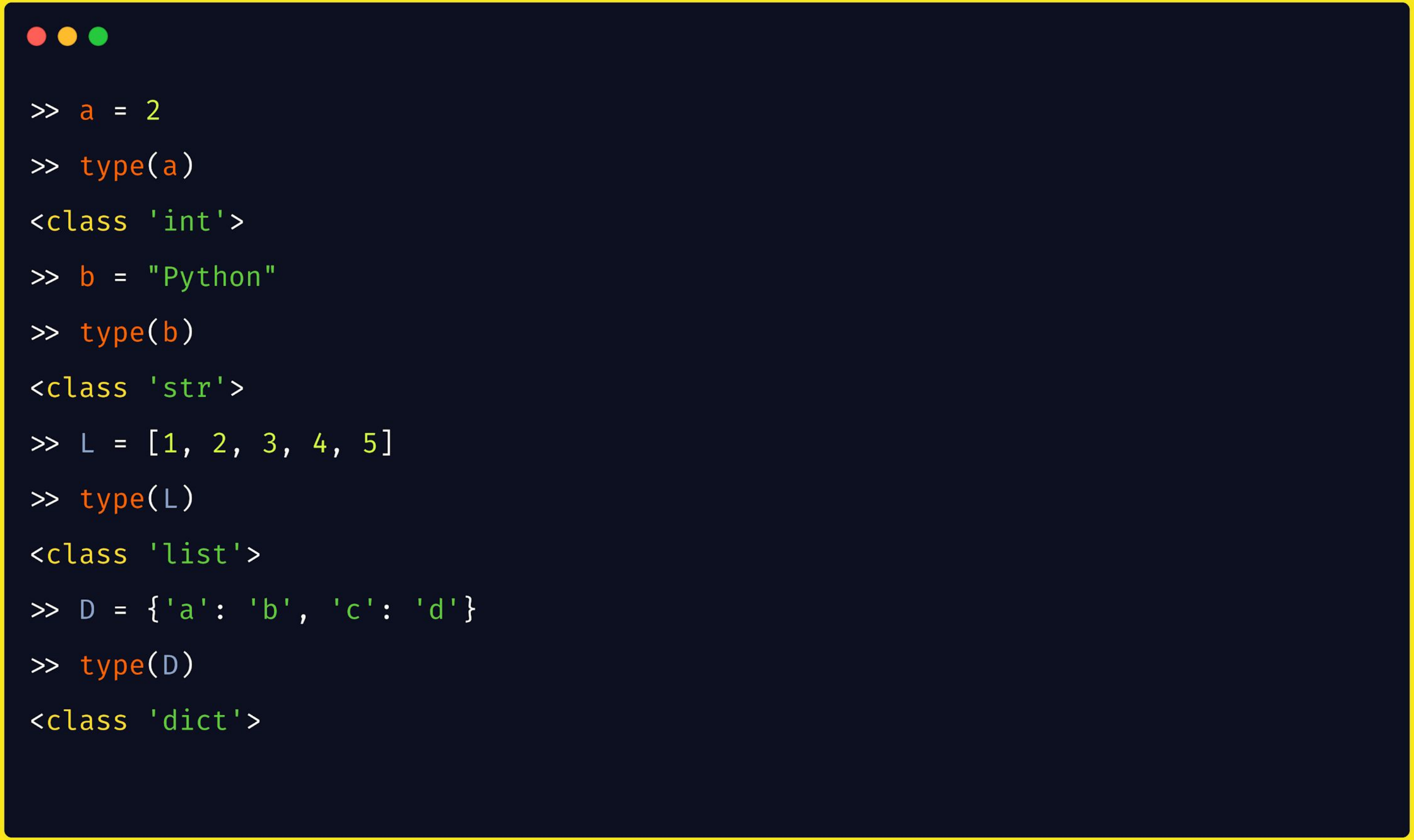
- objects
- memory address
- type of objects (mutable and immutable objects)
- difference between is vs == operator
- optimizations

An object is an entity that has **attribute** and **methods** associated with it.

Example:



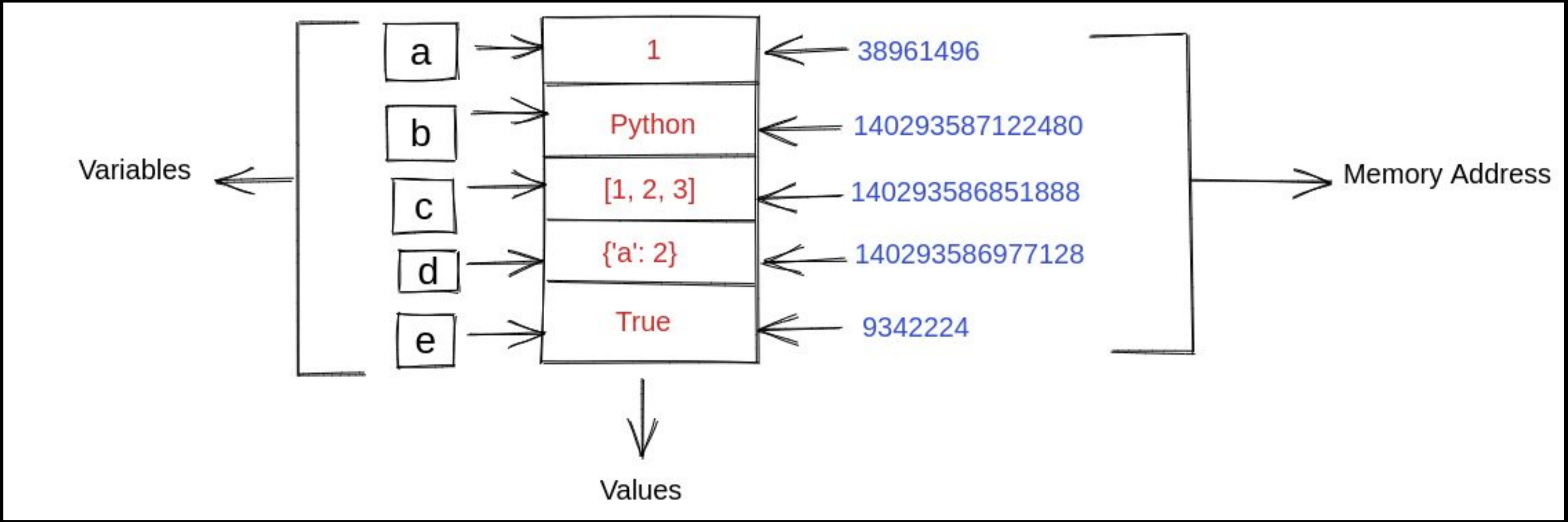
Type	Value	Memory Address
Integer	5	38961400



```
>> a = 2
>> type(a)
<class 'int'>
>> b = "Python"
>> type(b)
<class 'str'>
>> L = [1, 2, 3, 4, 5]
>> type(L)
<class 'list'>
>> D = {'a': 'b', 'c': 'd'}
>> type(D)
<class 'dict'>
```

The location where the object gets stored in memory is referred to as memory address.

Pictorial Representation



Variables

a

b

c

d

e

1

Python

[1, 2, 3]

{'a': 2}

True

38961496

140293587122480

140293586851888

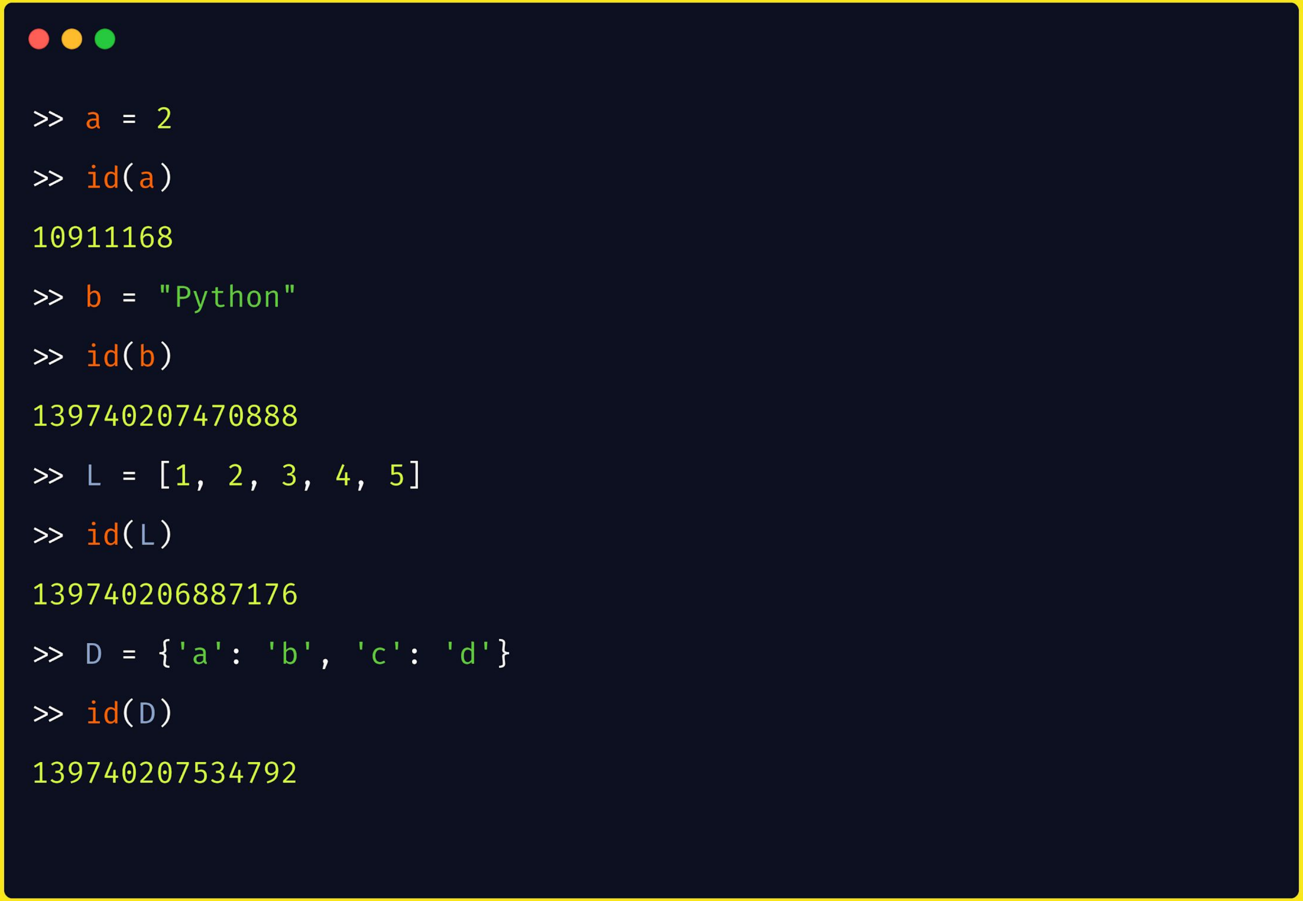
140293586977128

9342224

Memory Address

Values

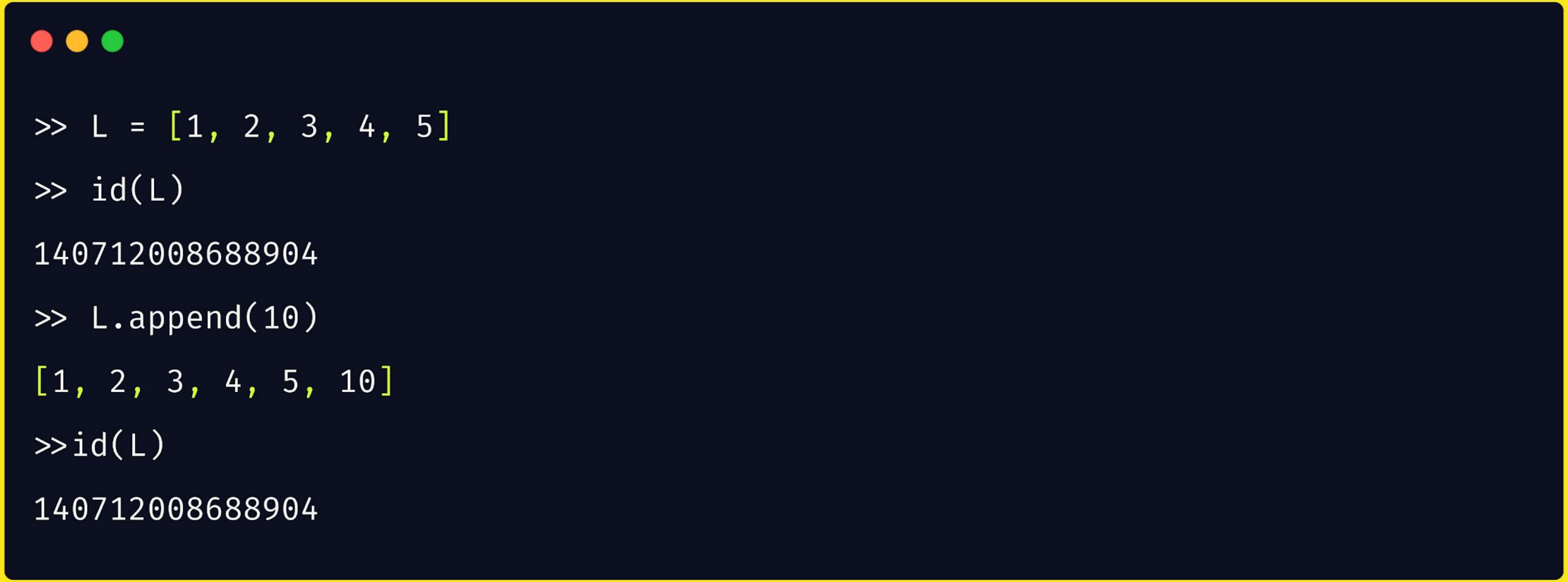
id(object) - id is a built-in function. It is used to determine the memory address of the object.



```
>> a = 2
>> id(a)
10911168
>> b = "Python"
>> id(b)
139740207470888
>> L = [1, 2, 3, 4, 5]
>> id(L)
139740206887176
>> D = {'a': 'b', 'c': 'd'}
>> id(D)
139740207534792
```

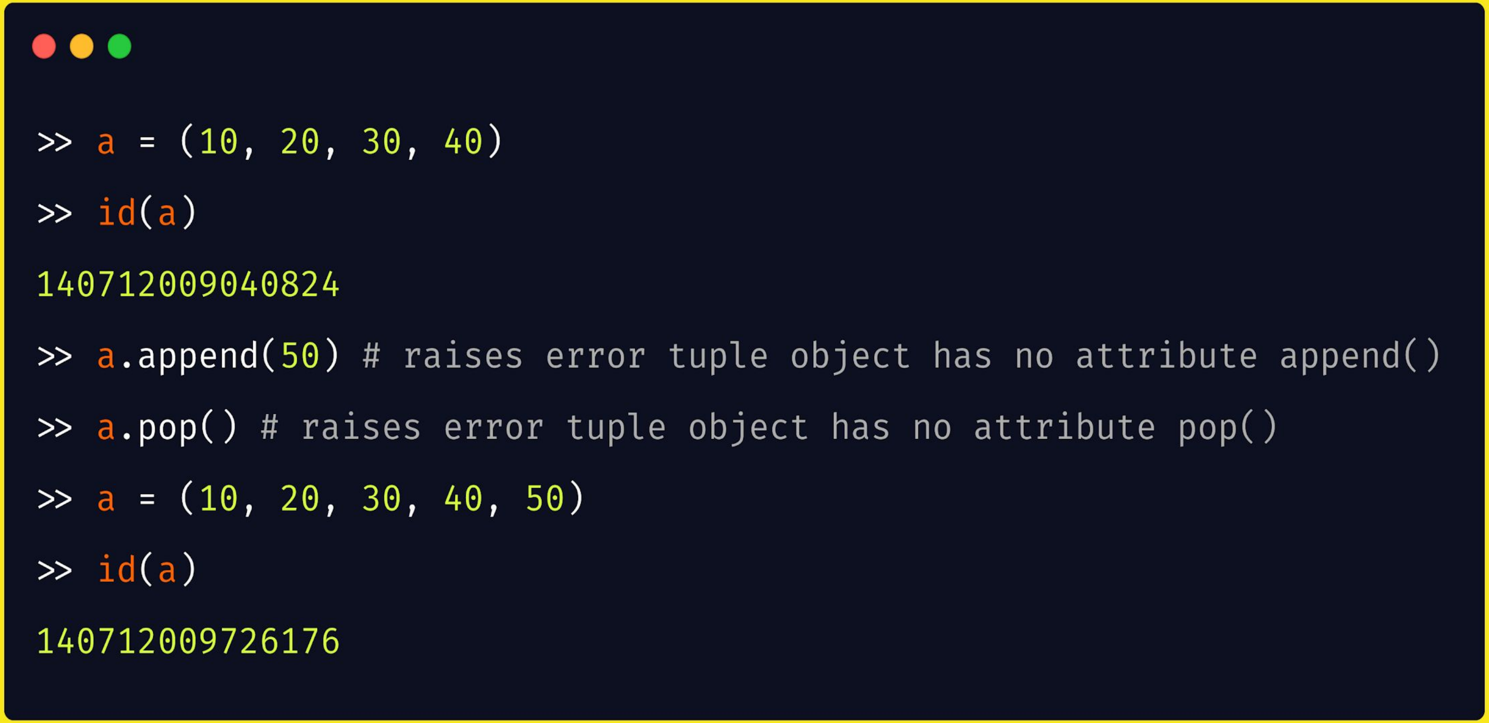
Mutable objects and Immutable objects.

Objects of built-in type (list, dictionary, sets) are mutable.



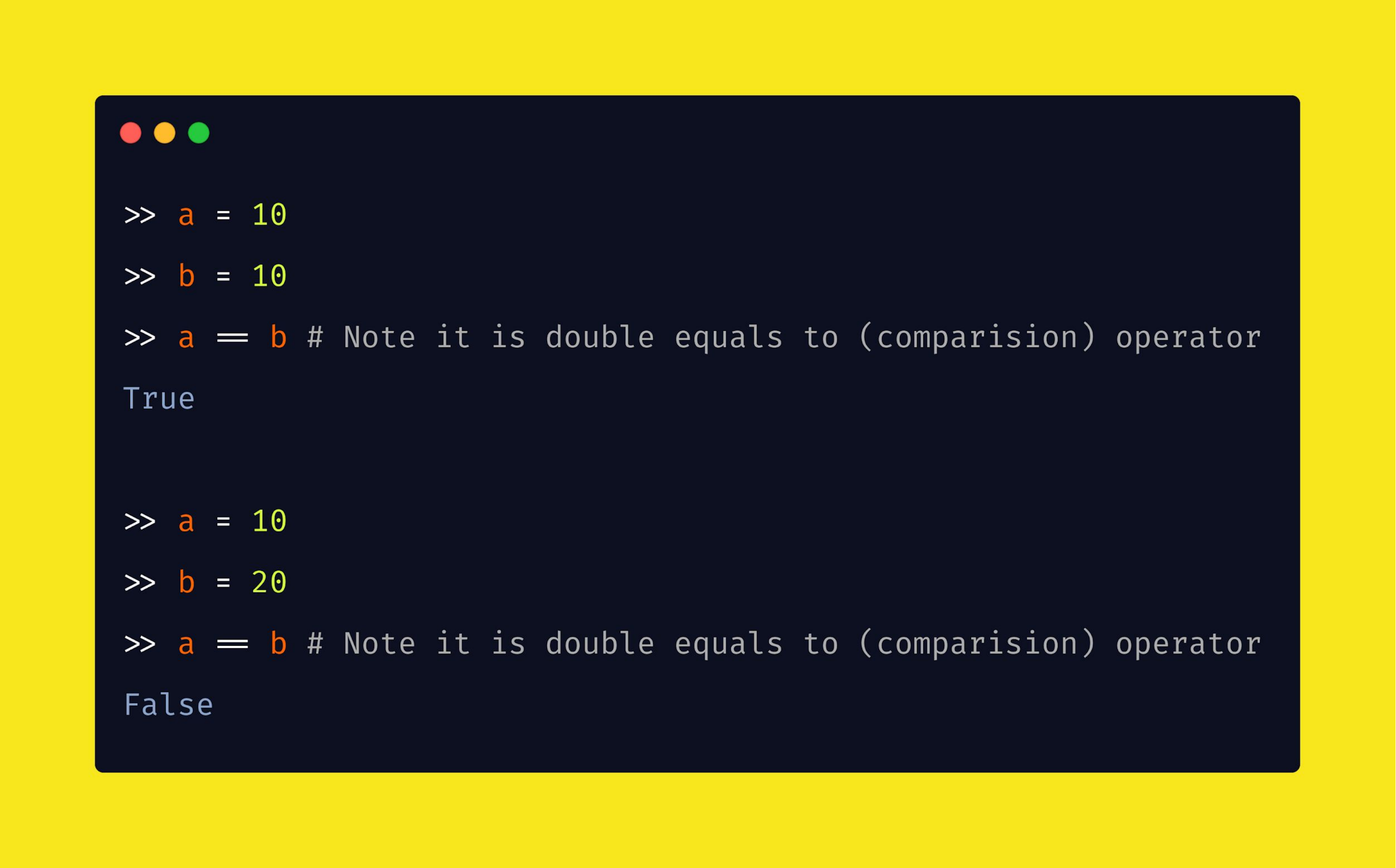
```
>> L = [1, 2, 3, 4, 5]

>> id(L)

140712008688904

>> L.append(10)

[1, 2, 3, 4, 5, 10]

>>id(L)

140712008688904
```

Objects of built-in type (int, float, bool, str, tuple) are **immutable**.

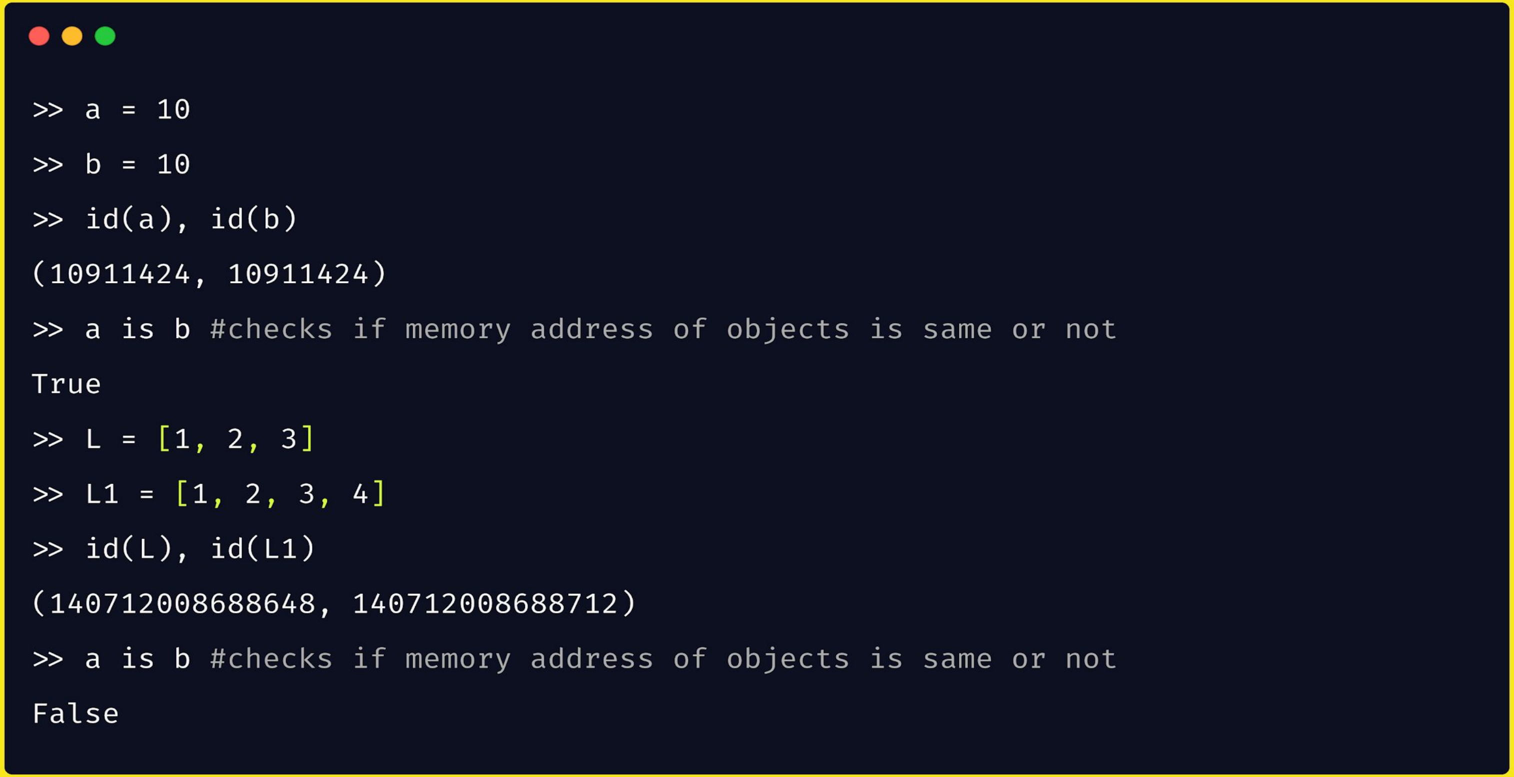


```
>> a = (10, 20, 30, 40)
>> id(a)
140712009040824
>> a.append(50) # raises error tuple object has no attribute append()
>> a.pop() # raises error tuple object has no attribute pop()
>> a = (10, 20, 30, 40, 50)
>> id(a)
140712009726176
```

Difference between is vs == operator



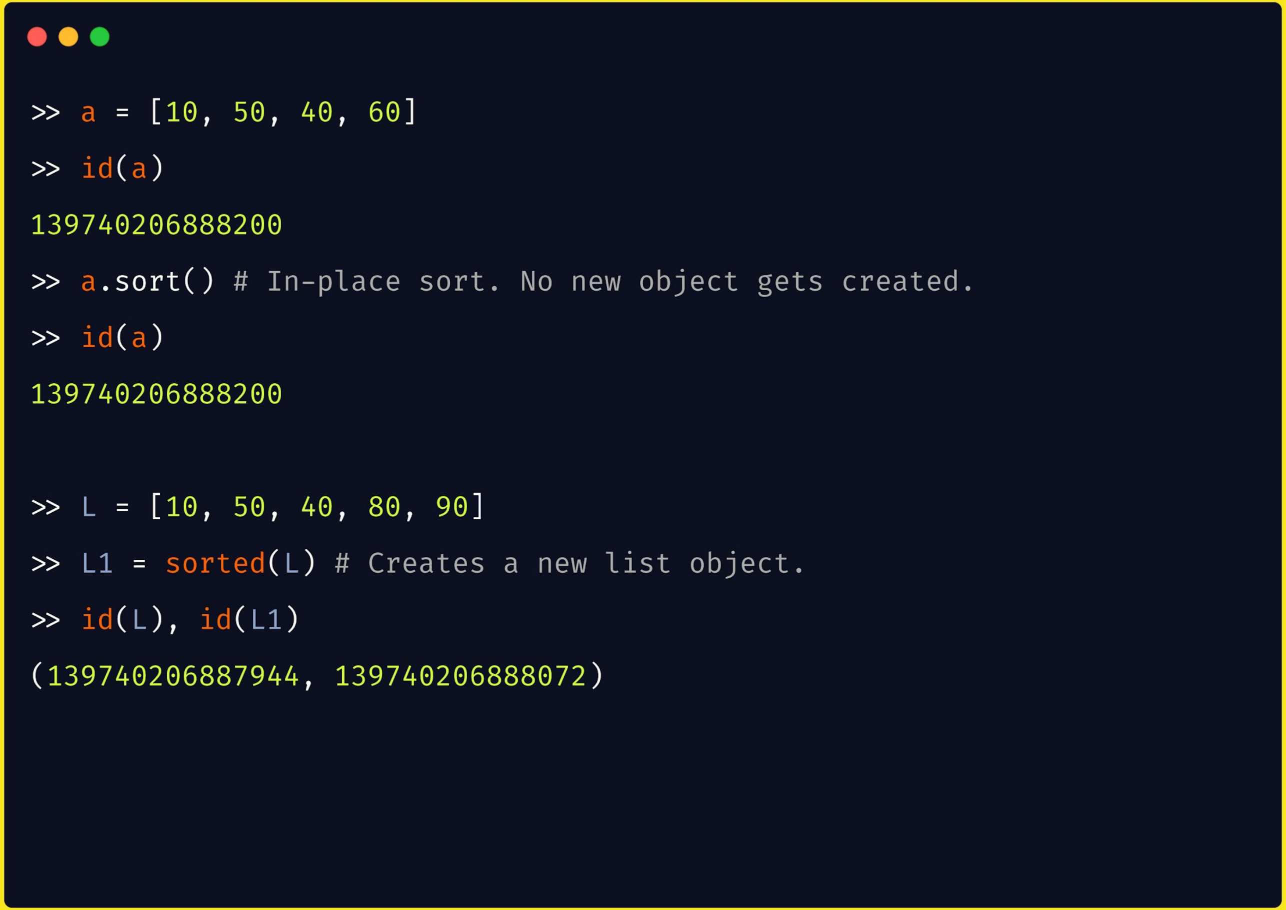
```
>> a = 10

>> b = 10

>> a == b # Note it is double equals to (comparision) operator

True


>> a = 10

>> b = 20

>> a == b # Note it is double equals to (comparision) operator

False
```



```
>> a = 10

>> b = 10

>> id(a), id(b)

(10911424, 10911424)

>> a is b #checks if memory address of objects is same or not

True

>> L = [1, 2, 3]

>> L1 = [1, 2, 3, 4]

>> id(L), id(L1)

(140712008688648, 140712008688712)

>> a is b #checks if memory address of objects is same or not

False
```

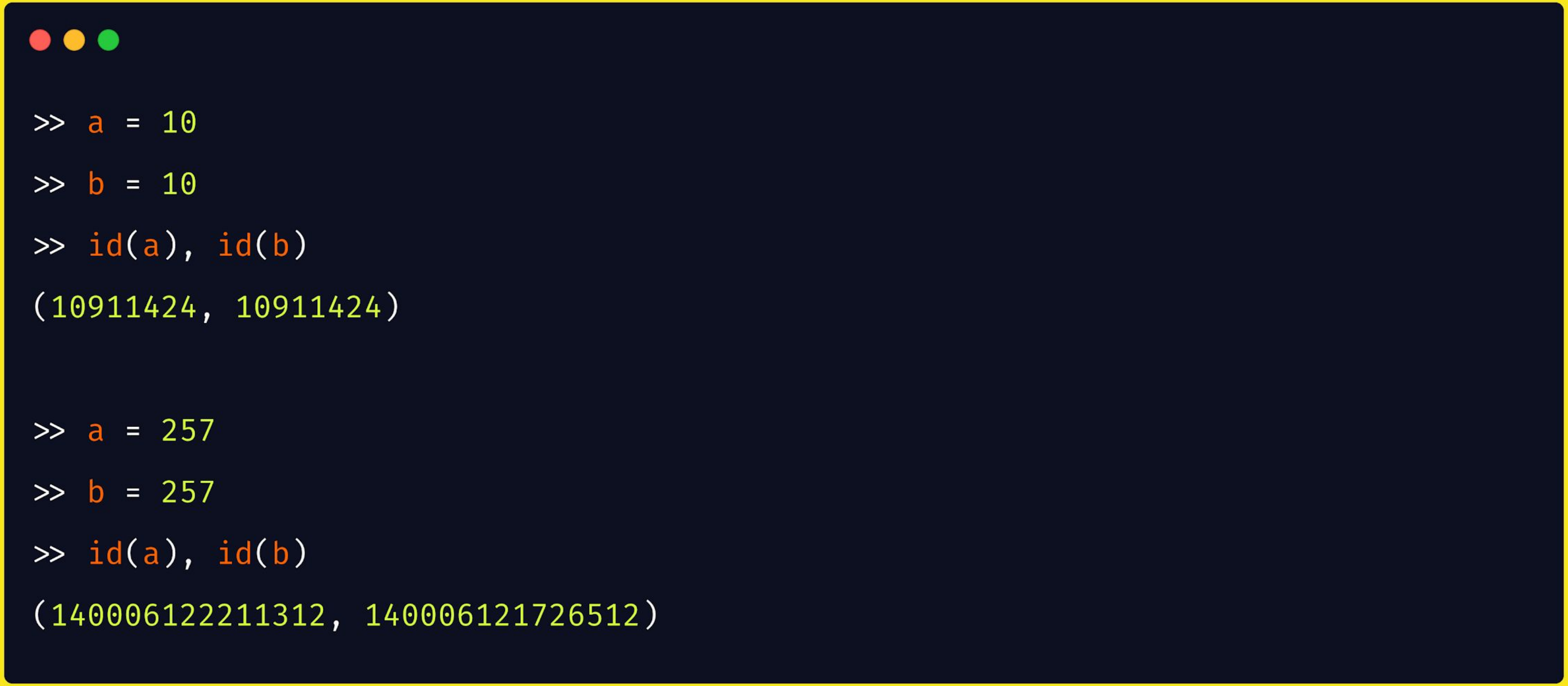
Different use cases to discuss memory optimization in Python

1. Sort and Sorted built-in methods



```
>> a = [10, 50, 40, 60]
>> id(a)
139740206888200
>> a.sort() # In-place sort. No new object gets created.
>> id(a)
139740206888200

>> L = [10, 50, 40, 80, 90]
>> L1 = sorted(L) # Creates a new list object.
>> id(L), id(L1)
(139740206887944, 139740206888072)
```

2. Concept of Integer Caching



```
>> a = 10
>> b = 10
>> id(a), id(b)
(10911424, 10911424)

>> a = 257
>> b = 257
>> id(a), id(b)
(140006122211312, 140006121726512)
```

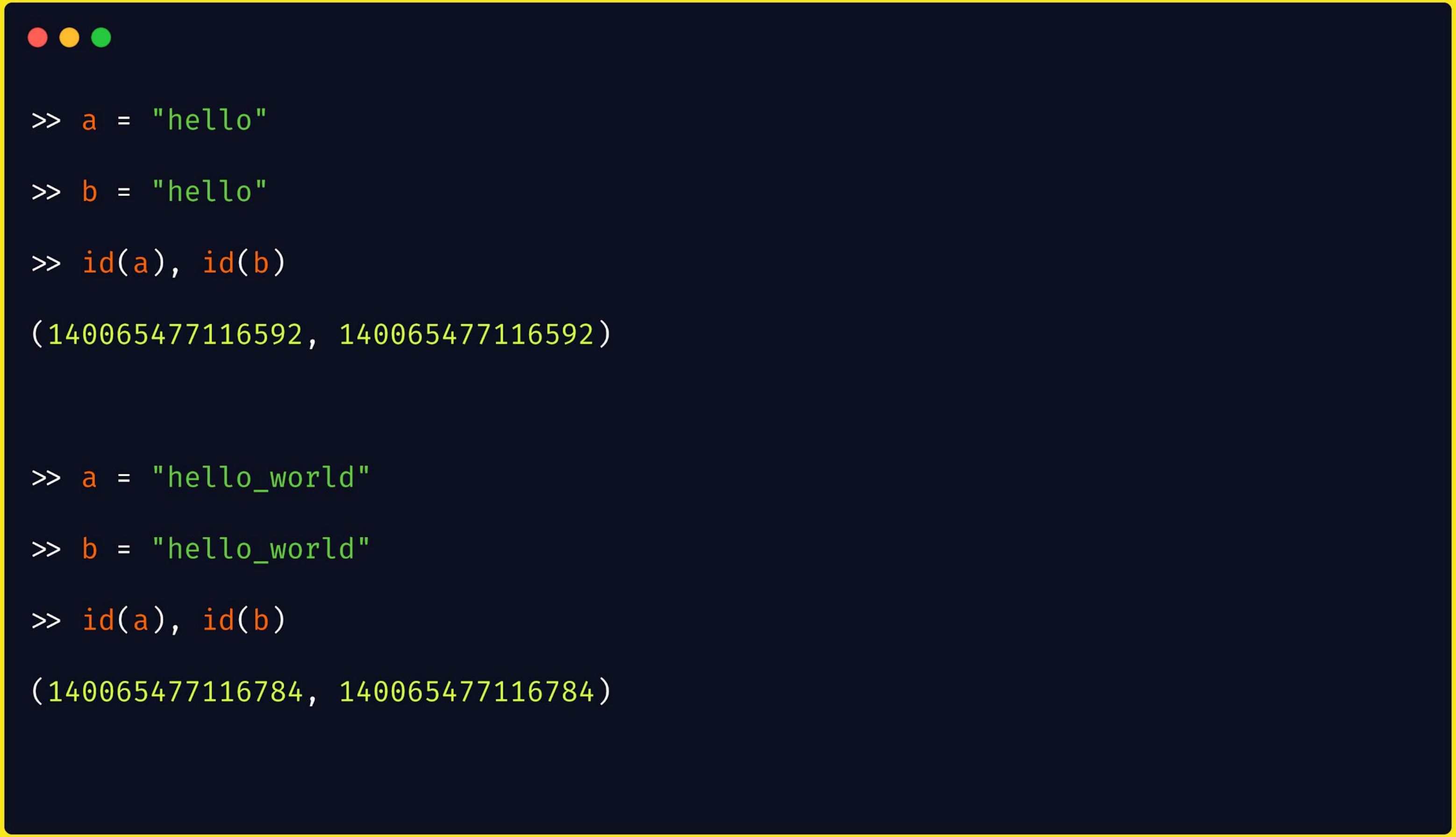
3. Concept of String Interning

As the Python code compiles identifiers are interned.

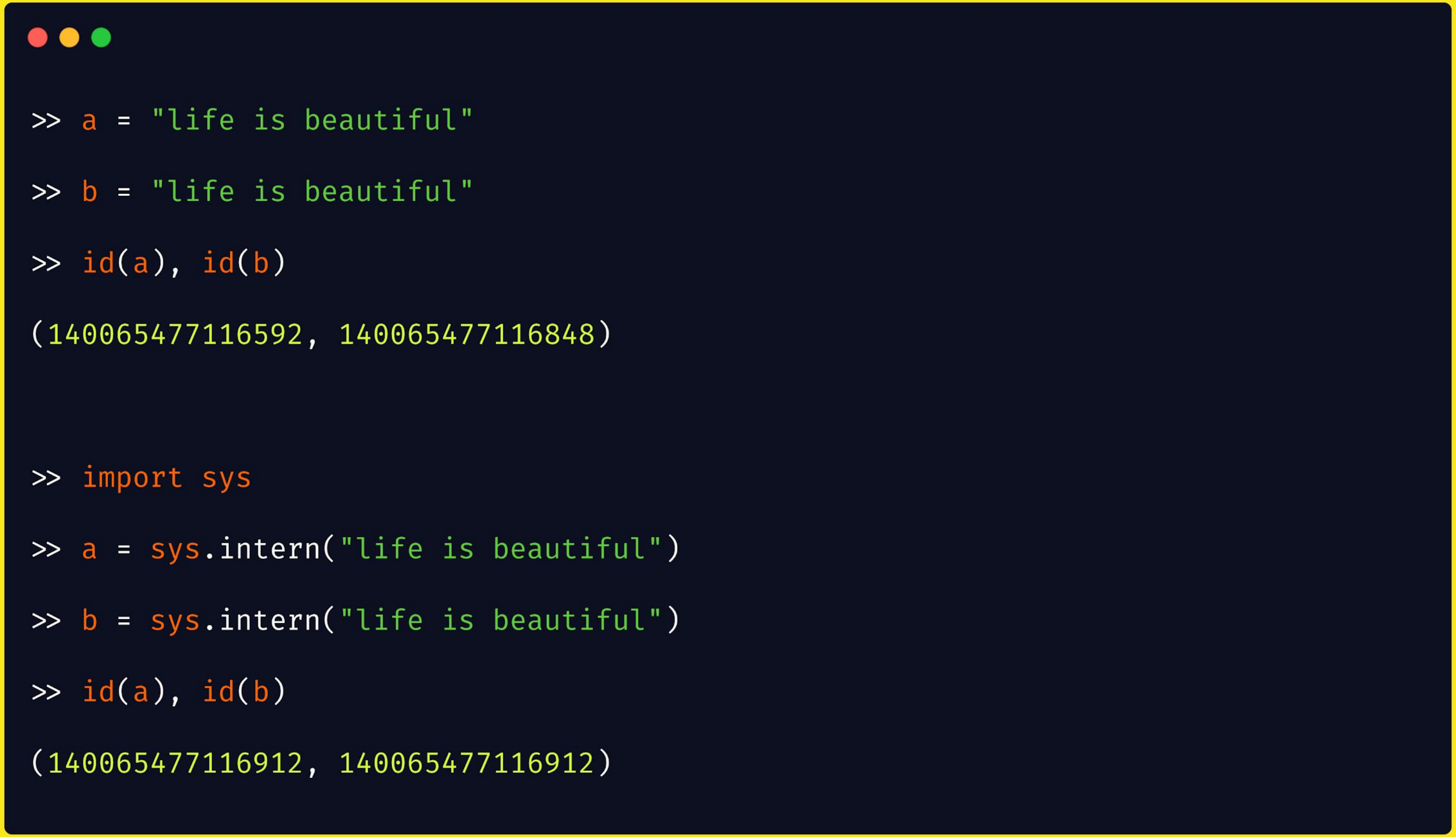
- variable names
- function names
- class names

Rule:

* start with _ or a letter.
* may contain _, letter, numbers.

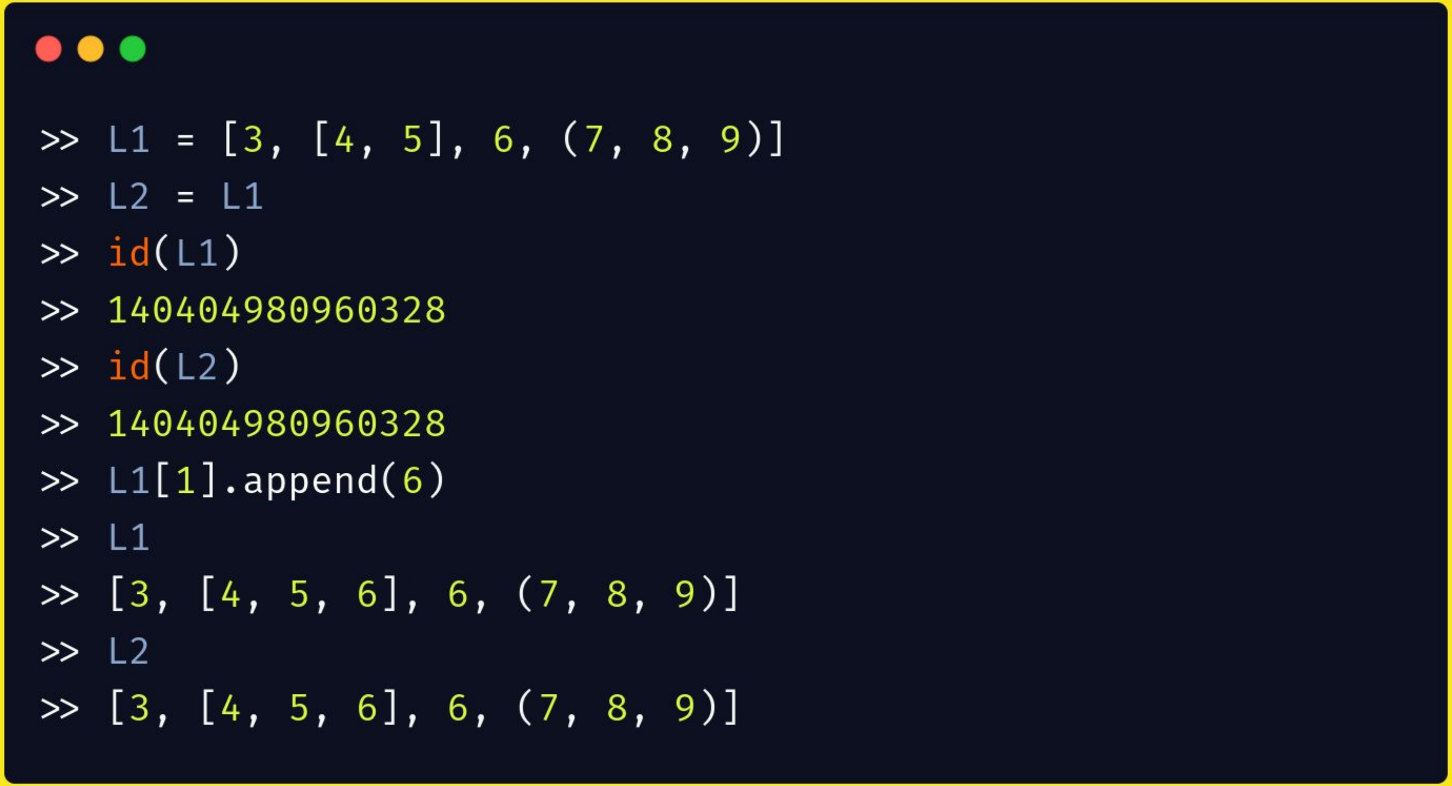


```
>> a = "hello"

>> b = "hello"

>> id(a), id(b)

(140065477116592, 140065477116592)


>> a = "hello_world"

>> b = "hello_world"

>> id(a), id(b)

(140065477116784, 140065477116784)
```



```
>> a = "life is beautiful"

>> b = "life is beautiful"

>> id(a), id(b)

(140065477116592, 140065477116848)


>> import sys

>> a = sys.intern("life is beautiful")

>> b = sys.intern("life is beautiful")

>> id(a), id(b)

(140065477116912, 140065477116912)
```

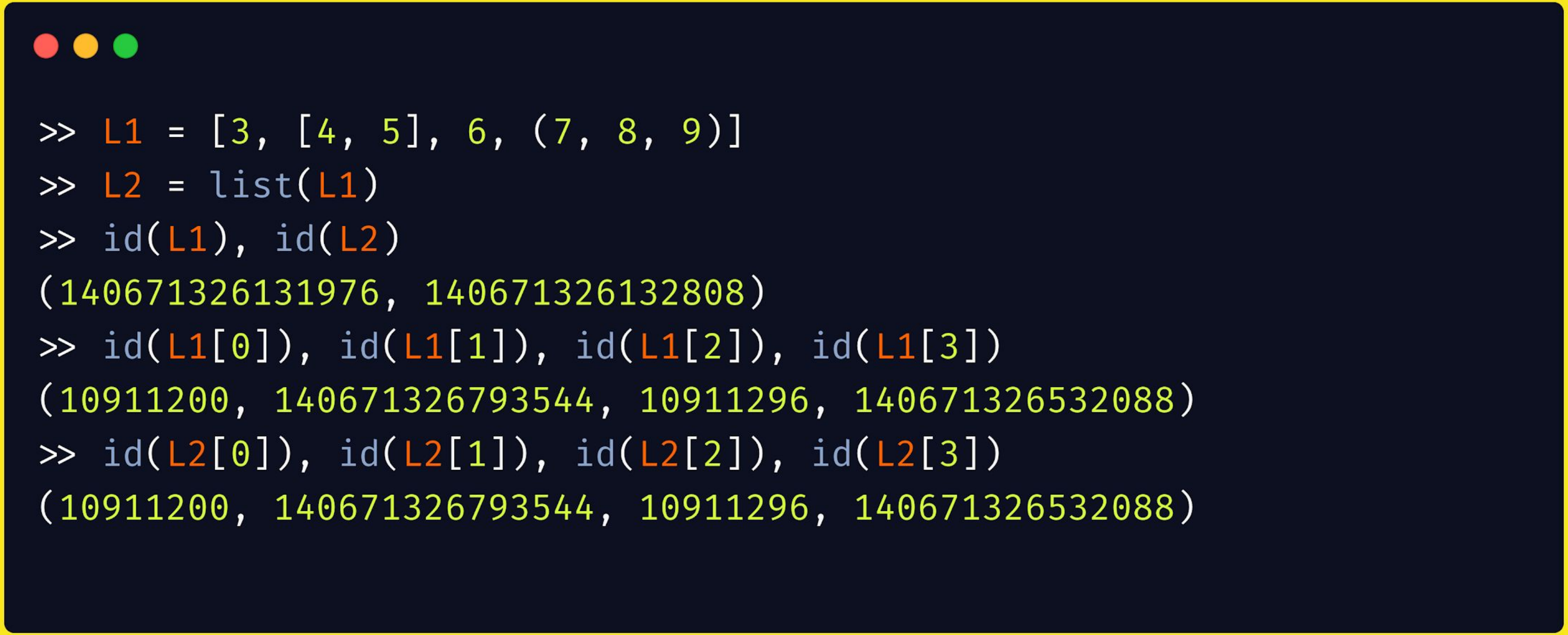
4. Copying List using Assignment operator



```
>> L1 = [3, [4, 5], 6, (7, 8, 9)]
>> L2 = L1
>> id(L1)
>> 140404980960328
>> id(L2)
>> 140404980960328
>> L1[1].append(6)
>> L1
>> [3, [4, 5, 6], 6, (7, 8, 9)]
>> L2
>> [3, [4, 5, 6], 6, (7, 8, 9)]
```

5. Shallow Copy in Lists

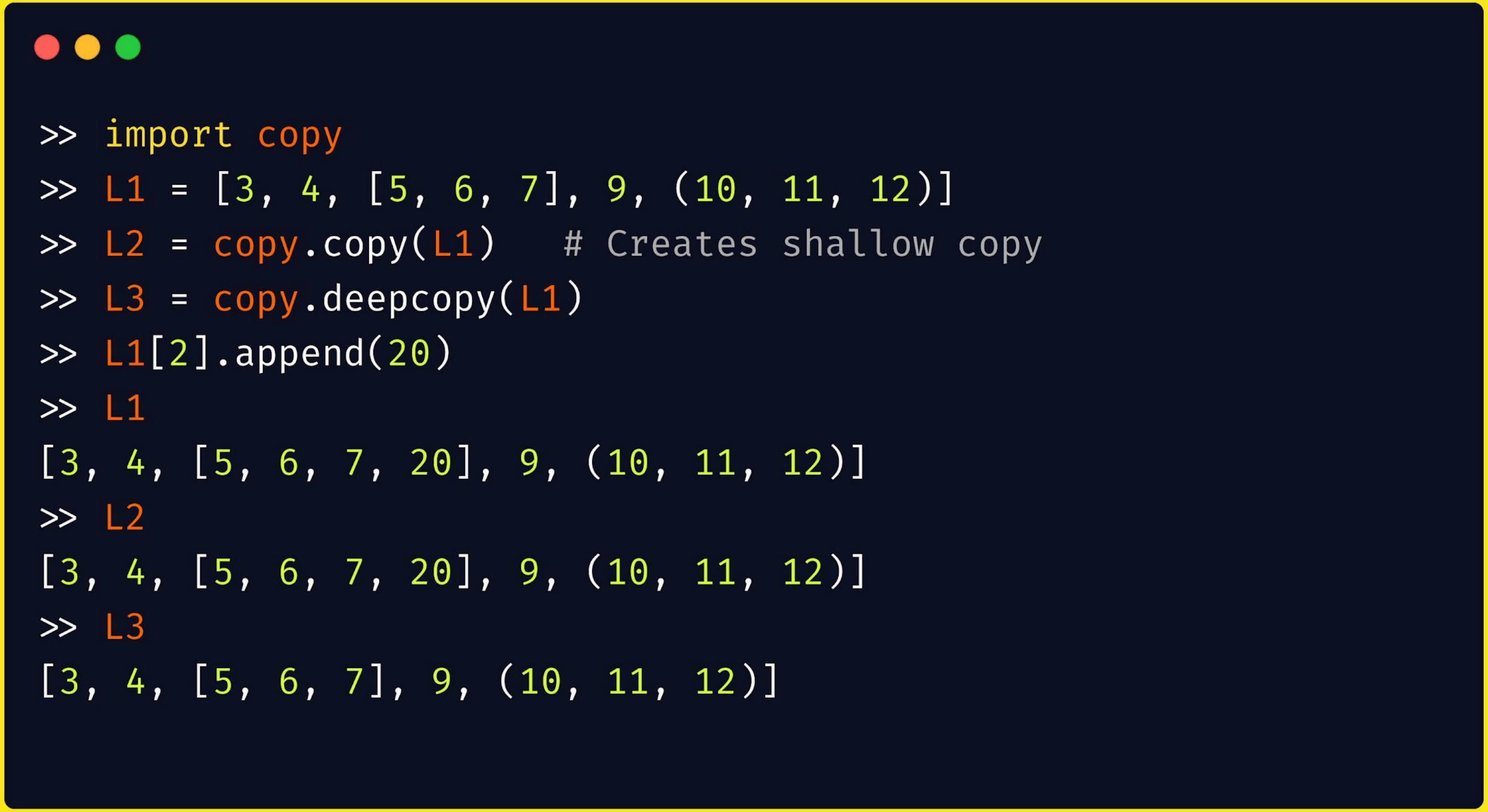
The outermost container is duplicated, but the copy is filled with references to the same items held by the original container.



```
>> L1 = [3, [4, 5], 6, (7, 8, 9)]
>> L2 = list(L1)
>> id(L1), id(L2)
(140671326131976, 140671326132808)
>> id(L1[0]), id(L1[1]), id(L1[2]), id(L1[3])
(10911200, 140671326793544, 10911296, 140671326532088)
>> id(L2[0]), id(L2[1]), id(L2[2]), id(L2[3])
(10911200, 140671326793544, 10911296, 140671326532088)
```

6. Deep Copy in Lists

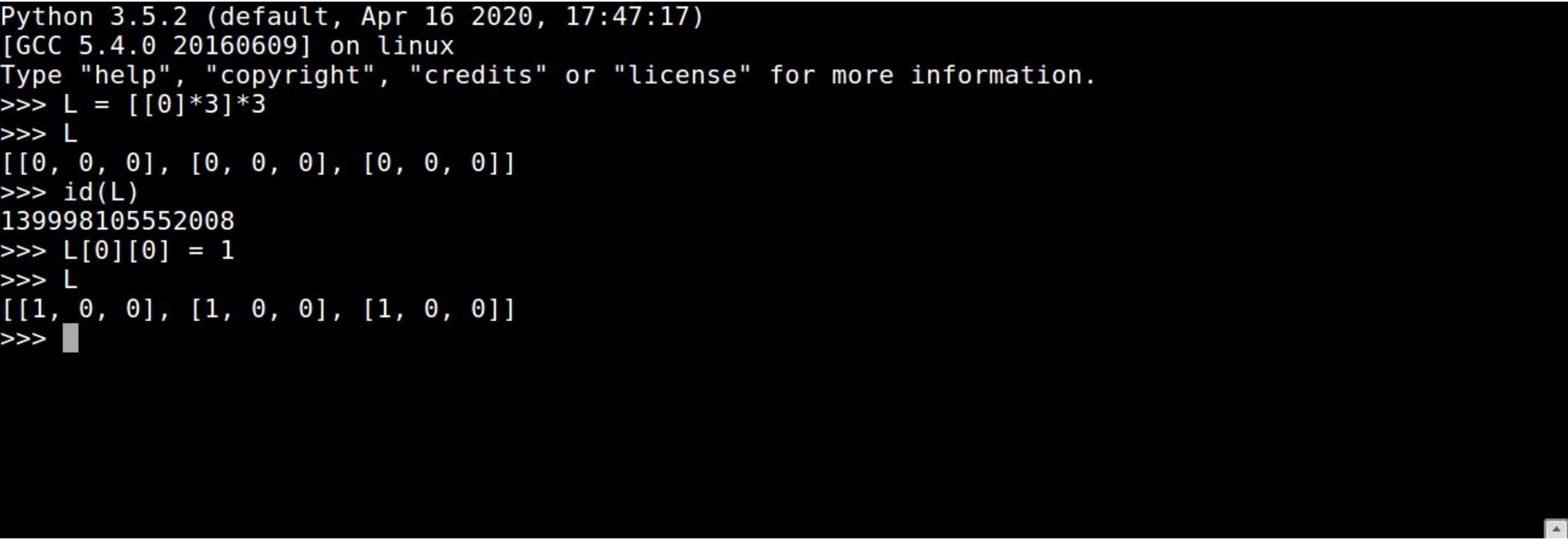
Duplicates do not share references of embedded objects.



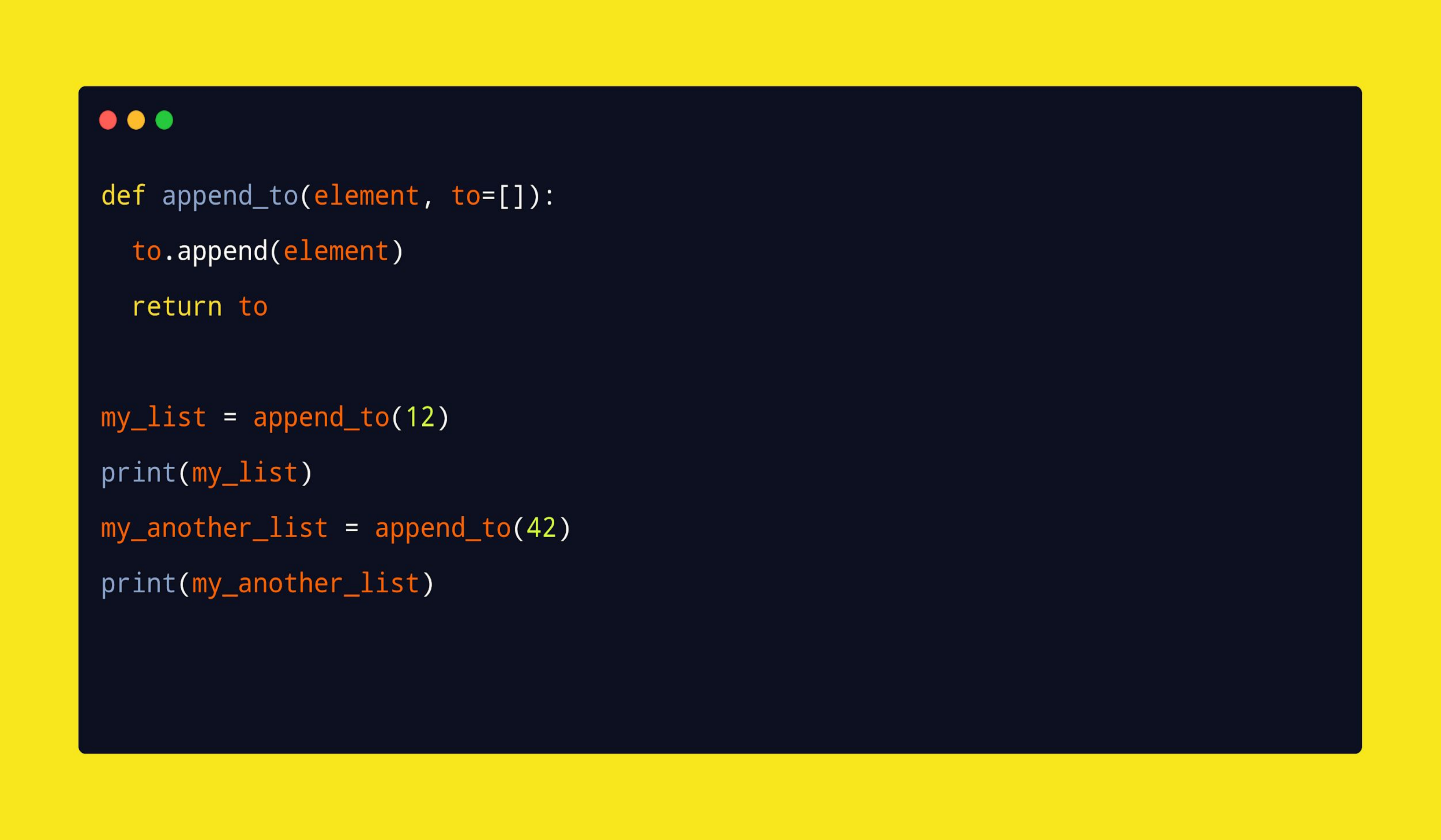
```
>> import copy
>> L1 = [3, 4, [5, 6, 7], 9, (10, 11, 12)]
>> L2 = copy.copy(L1)   # Creates shallow copy
>> L3 = copy.deepcopy(L1)
>> L1[2].append(20)
>> L1
[3, 4, [5, 6, 7, 20], 9, (10, 11, 12)]
>> L2
[3, 4, [5, 6, 7, 20], 9, (10, 11, 12)]
>> L3
[3, 4, [5, 6, 7], 9, (10, 11, 12)]
```

Why is it important to learn object internals?

* operator copies the memory references



```
Python 3.5.2 (default, Apr 16 2020, 17:47:17)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> L = [[0]*3]*3
>>> L
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> id(L)
139998105552008
>>> L[0][0] = 1
>>> L
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
>>> 
```



```python
def append_to(element, to=[]):
  to.append(element)
  return to

my_list = append_to(12)
print(my_list)
my_another_list = append_to(42)
print(my_another_list)
```

Summary

- object internals
- memory address use cases
- type of objects (mutable, immutable)
- difference between is vs == operator
- optimizations

FOR LETTING ME JOIN