# COPYING SEQUENCES

## Why copy sequences?

Mutable sequences can be modified.

Sometimes you want to make sure that whatever sequence you are working with cannot be modified, either inadvertently by yourself, or by 3rd party functions

We saw an example of this earlier with list concatenations and repetitions.

Also consider this example:

```
def reverse(s):
    s.reverse()
    return s
```

```
s = [10, 20, 30]

new_list = reverse(s)

new_list    → [30, 20, 10]
s           → [30, 20, 10]
```

We should have passed it a copy of our list if we did not intend for our original list to be modified

## Soapbox

```
def reverse(s):
    s.reverse()
    return s
```

Generally we write functions that do not modify the contents of their arguments.

But sometimes we really want to do so, and that's perfectly fine → in-place methods

However, to clearly indicate to the caller that something is happening in-place, we should not return the object we modified

If we don't return s in the above example, the caller will probably wonder why not?

So, in this case, the following would be a better approach:

```
def reverse(s):
    s.reverse()
```

and if we do not do in-place reversal, then we return the reversed sequence

```
def reverse(s):
    s2 = <copy of s>
    s2.reverse()
    return s2
```

## How to copy a sequence

We can copy a sequence using a variety of methods: `s = [10, 20, 30]`

Simple Loop

```
cp = []
for e in s:
    cp.append(e)
```

definitely non-Pythonic!

List Comprehension `cp = [e for e in s]`

The copy method `cp = s.copy()` (not implemented in immutable types, such as tuples or strings)

Slicing `cp = s[0:len(s)]` or, more simply `cp = s[:]`

The `copy` module

`list()` `list_2 = list(list_1)`

Note: `tuple_2 = tuple(tuple_1)` and `t[:]` does not create a new tuple!

## Watch out when copying entire immutable sequences

```
l1 = [1, 2, 3]
l2 = list(l1)            l2 → [1, 2, 3]     id(l1)    id(l2)

t1 = (1, 2, 3)
t2 = tuple(t1)           t2 → (1, 2, 3)     id(t1) = id(t2)     same object!

t1 = (1, 2, 3)
t2 = t1[:]               t2 → (1, 2, 3)     id(t1) = id(t2)     same object!
```

Same thing with strings, also an immutable sequence type

Since the sequence is immutable, it is actually OK to return the same sequence

## Shallow Copies

Using any of the techniques above, we have obtained a copy of the original sequence

```
s = [10, 20, 30]
cp = s.copy()
cp[0] = 100
```

cp → [100, 20, 30]   s → [10, 20, 30]

Great, so now our sequence s will always be safe from unintended modifications?   Not quite…

```
s = [ [10, 20], [30, 40] ]
cp = s.copy()
cp[0] = 'python'
```

cp → ['python', [30, 40] ]   s → [ [10, 20], [30, 40] ]

```
cp[1][0] = 100
```

cp → ['python', [100, 40] ]   s → [ [10, 20], [100, 40] ]

# Shallow Copies

What happened?

When we use any of the copy methods we saw a few slides ago, the copy essentially copies all the object references from one sequence to another

```
s = [a, b]            id(s) → 1000     id(s[0]) → 2000      id(s[1]) → 3000

cp = s.copy()         id(cp) → 5000    id(cp[0]) → 2000     id(cp[1]) → 3000
```

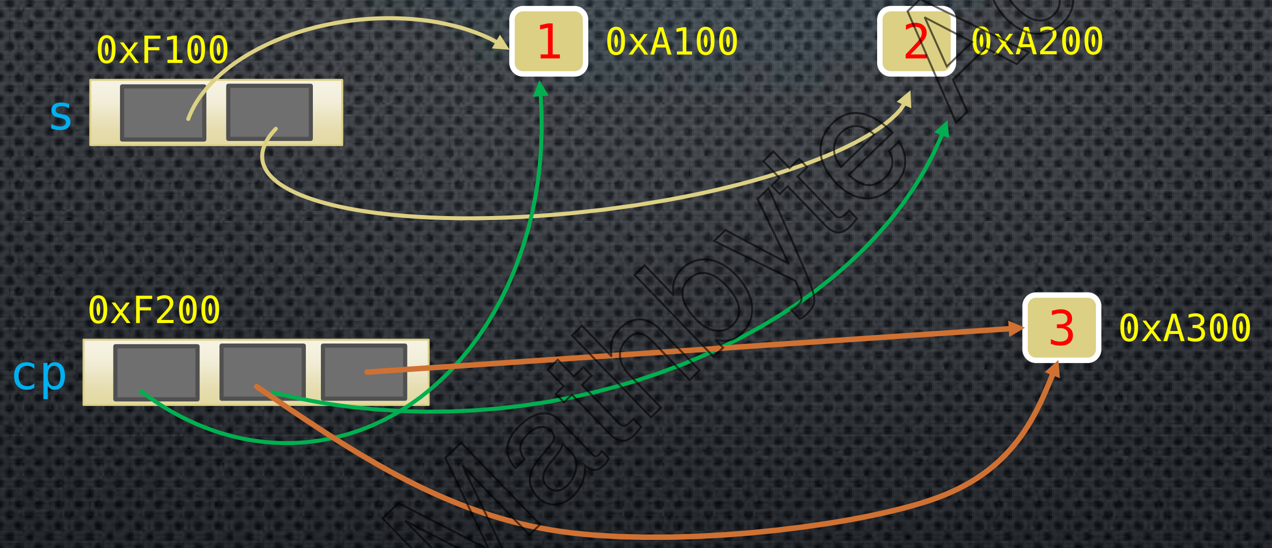
When we made a copy of `s`, the sequence was copied, but it's elements point to the same memory address as the original sequence elements

The sequence was copied, but it's elements were not

This is called a shallow copy

## Shallow Copies

```
s = [ 1, 2 ]
cp = s.copy()
```




```
cp.append(3)

cp[1] = 3
```

If the elements of s are immutable, such as integers in this example, then not really important

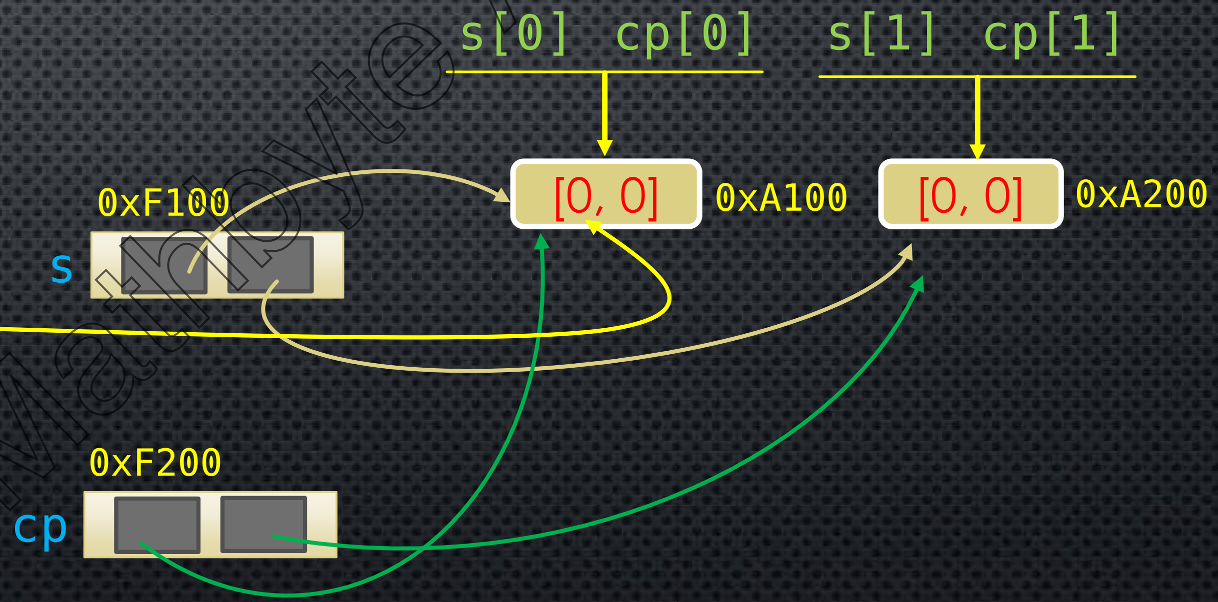
## Shallow Copies

But, if the elements of s are mutable, then it can be important

```
s = [ [0, 0], [0, 0] ]

cp = s.copy()

cp[0][0] = 100
```

cp → [ [100, 0], [0, 0] ]

s → [ [100, 0], [0, 0] ]



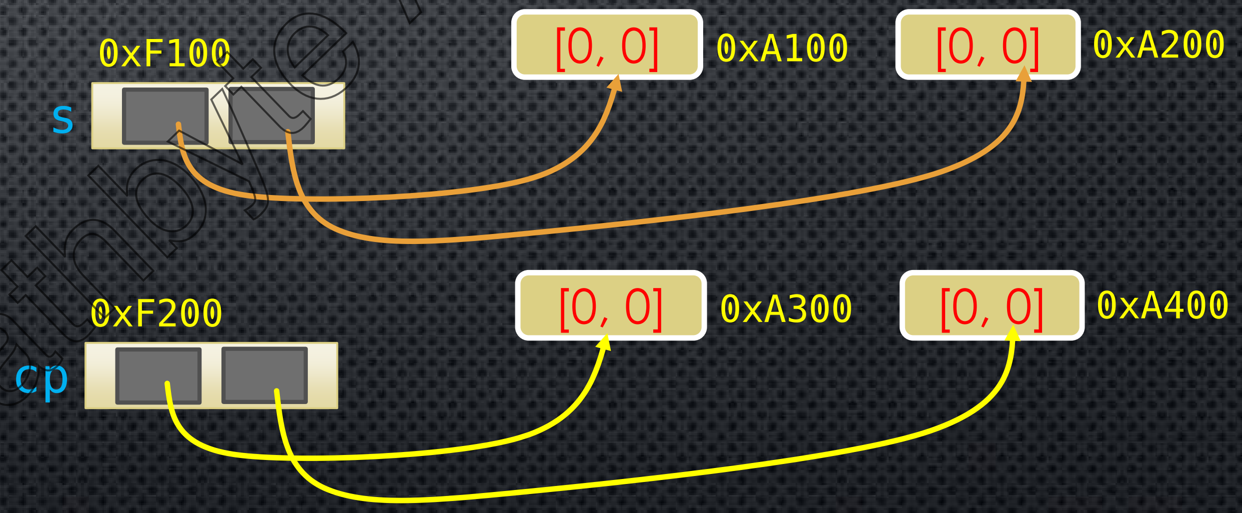
# Deep Copies

So, if collections contain mutable elements, shallow copies are not sufficient to ensure the copy can never be used to modify the original!

Instead, we have to do something called a deep copy.

For the previous example we might try this:

```
s = [ [0, 0], [0, 0] ]

cp = [e.copy() for e in s]
```




In this case:

cp is a copy of s

but also, every element of cp is a copy of the corresponding element in s

shallow copy

## Deep Copies

But what happens if the mutable elements of `s` themselves contain mutable elements?

```
s = [ [ [0, 1], [2, 3] ], [ [4, 5], [6, 7] ] ]
```

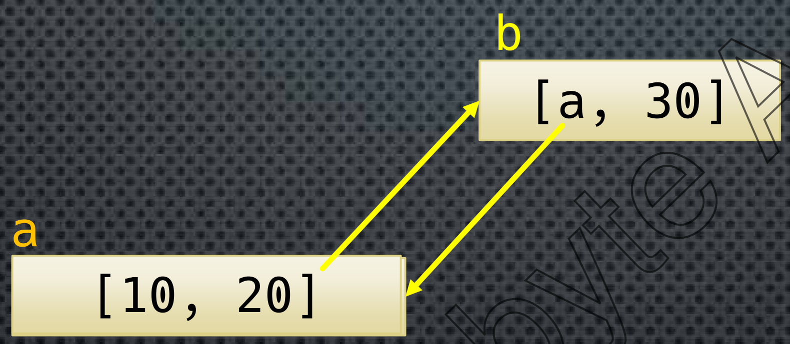
We would need to make copies at least 3 levels deep to ensure a true deep copy

Deep copies, in general, tend to need a recursive approach

## Deep Copies

Deep copies are not easy to do. You might even have to deal with circular references

```
a = [10, 20]
b = [a, 30]
a.append(b)
```




If you wrote your own deep copy algorithm, you would need to handle this circular reference!

# Deep Copies

In general, objects know how to make shallow copies of themselves

built-in objects like lists, sets, and dictionaries do - they have a `copy( )` method

The standard library `copy` module has generic `copy` and `deepcopy` operations

The `copy` function will create a shallow copy

The `deepcopy` function will create a deep copy, handling nested objects, and circular references properly

Custom classes can implement the `__copy__` and `__deepcopy__` methods to allow you to override how shallow and deep copies are made for you custom objects

We'll revisit this advanced topic of overriding deep copies of custom classes in the OOP series of this course.

## Deep Copies

Suppose we have a custom class as follows:

```
def MyClass:
    def __init__(self, a):
        self.a = a
```

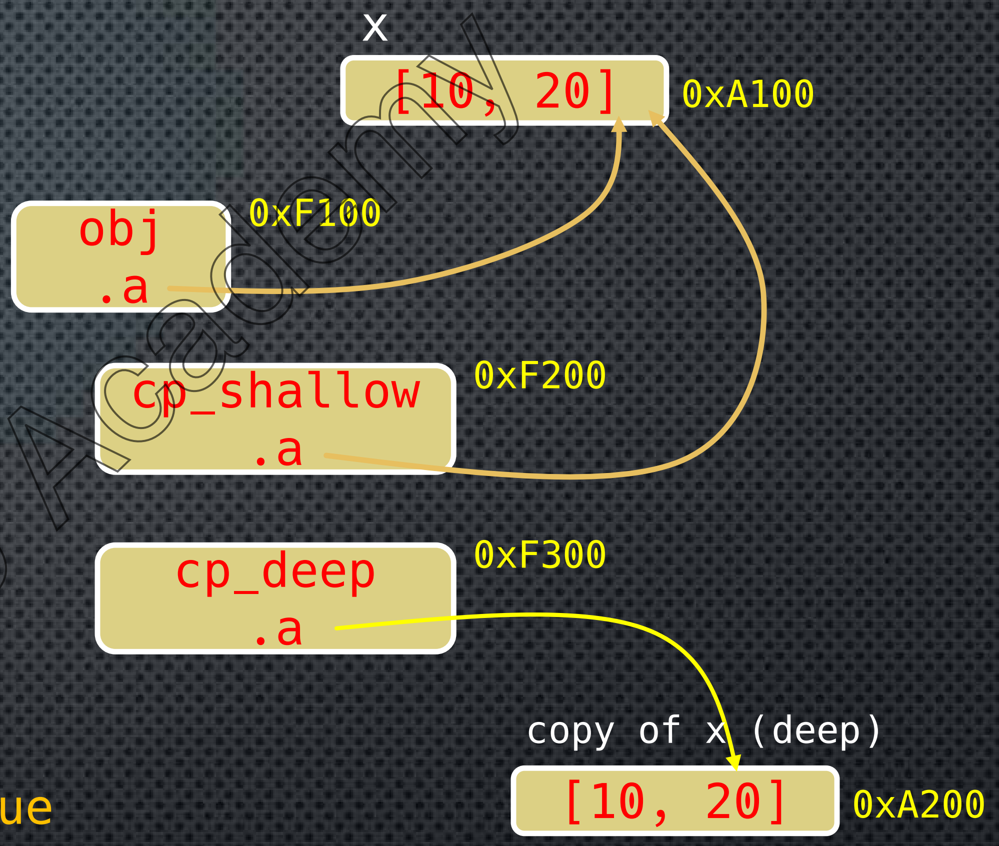
```
from copy import copy, deepcopy
```

```
x = [10, 20]
```

```
obj = MyClass(x)
```
x is obj.a → True

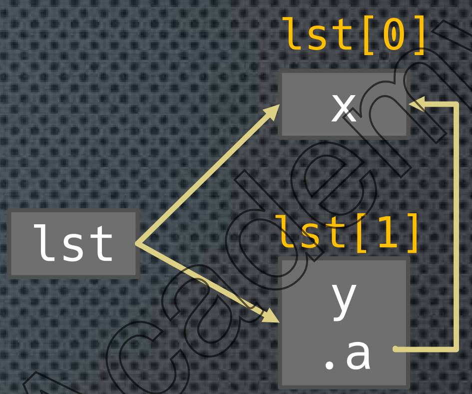
```
cp_shallow = copy(obj)
```
cp_shallow.a is obj.a → True

```
cp_deep = deepcopy(obj)
```
cp_deep.a is obj.a → False



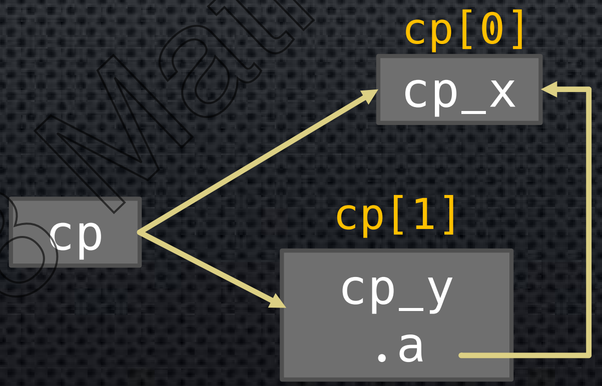
## Deep Copies

```
def MyClass:
    def __init__(self, a):
        self.a = a

x = MyClass(500)

y = MyClass(x)          y.a is x → True

lst = [x, y]

cp = deepcopy(lst)

cp[0] is x → False

cp[1] is y → False

cp[1].a is x → False


cp[1].a is cp[0] → True
```




this is not a circular reference

but there is a relationship between y.a and x




relationship between cp_y.a and cp_x is maintained!

# Code Exercises