Data Structures in Python Chapter 1

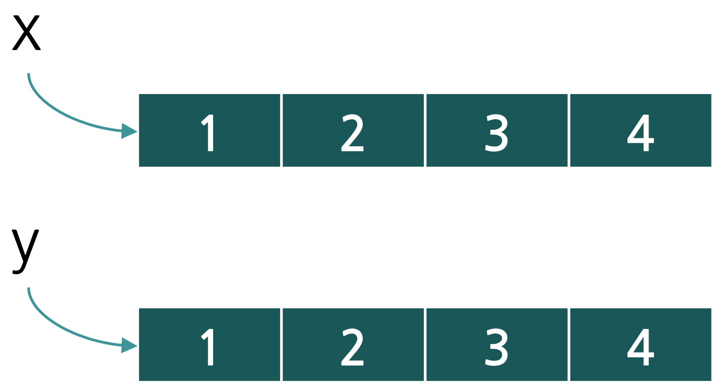
너는 청년의 때에 너의 창조주를 기억하라 곧 곤고한 날이 이르기 전에, 나는 아무 낙이 없다고 할 해들이 가깝기 전에 (전12:1)

Agenda

- Topics:
  - Python Review
    - Objects in memory
    - References
    - Equality
    - Mutability vs. Immutability
    - List operations (methods)
    - Shallow copy vs. Deep copy
- References:
  - DSpy: Chapter 1: Python Review(1) ~ (7)
  - Problem Solving with Algorithms and Data Structures using Python
    - Chapter 1

Objects in memory

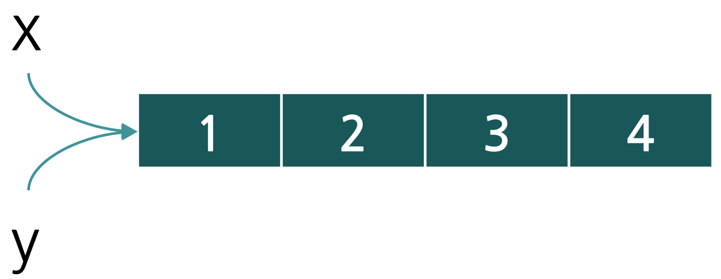
- Value equality



Two different objects that store the same information.

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

- Reference equality



Two different references (or names) for the same object.

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

Different ways to compare equality

- ==
  - Calls a method of the object
  - Typically involves checking the contents of the objects.
  - We should always use this for literals.

- is
  - Checks the references of the objects.
  - Evaluates to True if they are the same object.

```
x = [1, 2, 3, 4]
y = [1, 2, 3, 4]
print(x == y)
print(x is y)
```

```
x = [1, 2, 3, 4]
y = x
print(x == y)
print(x is y)
```

String

- Every **UNIQUE string** you create will have it's own address space in memory

```
a = 'foo'
b = 'foo'
print(id(a))
print(id(b))
print(a == b)
print(a is b)
```

```
2026110321456
2026110321456
```

immutable object

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

```
2026159683136
2026159685184
```

mutable object

Mutable and Immutable objects

- An immutable object is an object whose state cannot be modified after it is created.

- Examples of **immutable** objects:
  - **integer, boolean, float, string, tuple**

- Examples of **mutable** objects
  - **lists, dictionaries, sets,** most data structures studied in this course

```
a = 'hello'
b = 'hello'
print(id(a))
print(id(b))
```

```
2026159684288
```

```
a = 'hello'
print(id(a))
a = 'jello'
print(id(b))
```

```
2026159684288
```

Lists are mutable

- Lists are **mutable**
  - i.e. We can change lists in place, such as reassignment of a sequence slice, which will work for lists, but raise an error for tuples and strings.

- Example:
  - rgb = ['red', 'green', 'blue']
  - rgb[0] = 'RED'
  - rgb still points to the same memory when you are done.

```
rgb = ['red', 'green', 'blue']
print(id(rgb))
rgb[0] = 'RED'
print(id(rgb))
print(rgb)
```

```
2026159684288
```

Tuples are immutable

- Strings and tuples are immutable sequence types: such objects cannot be modified **once created.**
  - i.e. you can't change a tuple.

- Example:

```
rgb = ('red', 'green', 'blue')
rgb[0] = 'RED'
```

```
TypeError: 'tuple' object does not
support item assignment
```

- The immutability of tuples means they are **faster** than lists.

Operations on Strings

- Whenever you call a method of an object, make sure you know if **changes** the contents of the object or **returns** a new object.

- Example:

```
truth = 'Sola Gratia'
print(id(truth))
truth = 'Sola Fide'
print(id(truth))
```

```
2026160287536

2026160287088
```

a new String object is instantiated and given the data “Sola Gratia" during its construction.

- lower(), upper(), lstrip(), rstrip(), …
  - Return a new copy of the string

```
truth = 'Sola Gratia'
print(id(truth))
facts = truth.upper()
print(id(facts))
```

returns a new object.

Operations on Lists - append vs. extend

- extend() - extends the list by appending all the items in the given list (i.e. the argument is a list

```
x = [1, 2, 3]
x.extend([4, 5, 6])
print(x)
```

- append() - adds an item to the end of the lis.

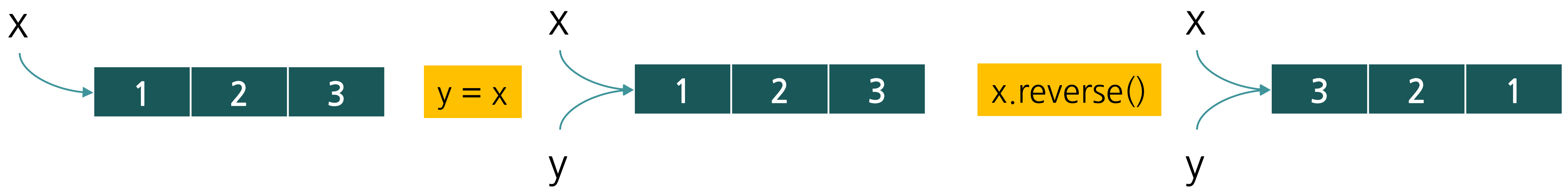
```
x = [1, 2, 3]
x.append([4, 5, 6])
print(x)
```

Operations on Lists - Reversing a list

- reverse() - reverses the list **in place** or alters the content of the list.

```
x = [1, 2, 3]
y = x
x.reverse()
print(x)
```

```
[3, 2, 1]
```



- sort() - sorts the list **in place** or alters the content of the list.

```
x = [1, 2, 3, -1]
x.sort()
print(x)
```

```
[-1, 1, 2, 3]
```

Exercise 1

- What is the output of the following code fragment? Why?

```
p = [1, 2, 3]
print (p[::-1])
print (p)
```

Aliases

- Two references to the same object are known as aliases.

```
x = [1, 2, 3, 4]
y = x
x.append(5)
print(x)
print(y)
```

- When an assignment is performed, **the reference** to the object on the right of the assignment is assigned to the variable on the left.

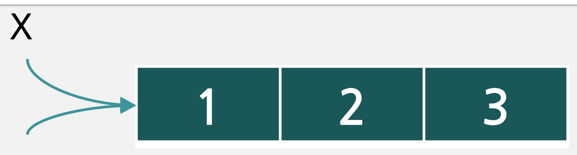
- When a method of an object is called,
  it sometimes **returns a value** and sometimes it **alters the object**.

Example

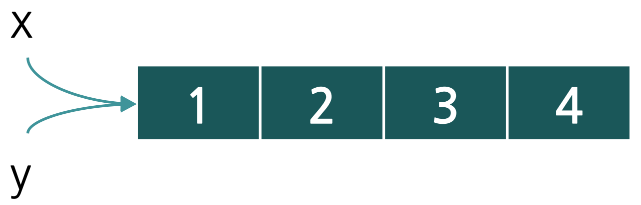
- What happens in the following cases? What is the output?

```
x = [1, 2, 3]
y = x

x += [4]

print(x)
print(y)
```

alters the object

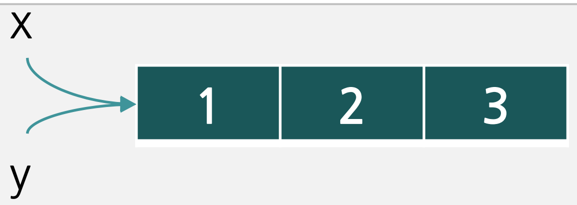


```
[1, 2, 3, 4]
[1, 2, 3, 4]
```



```
x = [1, 2, 3]
y = x

x = x + [4]

print(x)
print(y)
```

returns a new object;
x is replaced



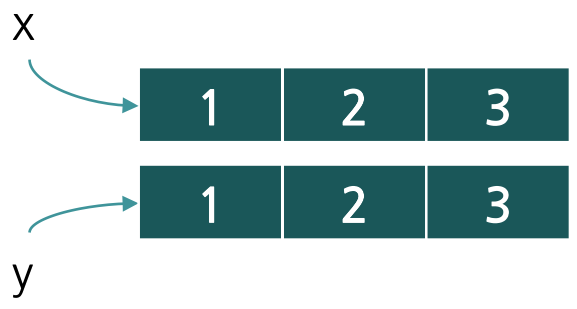
```
[1, 2, 3, 4]
[1, 2, 3]
```

Shallow copy

- Lists and dictionaries have a **copy()** method

```
x = [1, 2, 3]
y = x.copy()

print( x == y )
print( x is y )
```

```
True
False
```



```
a = [ [11], [22], [33] ]
b = a.copy()

print( a == b )
print( a is b )
print( a[0] is b[0])
```

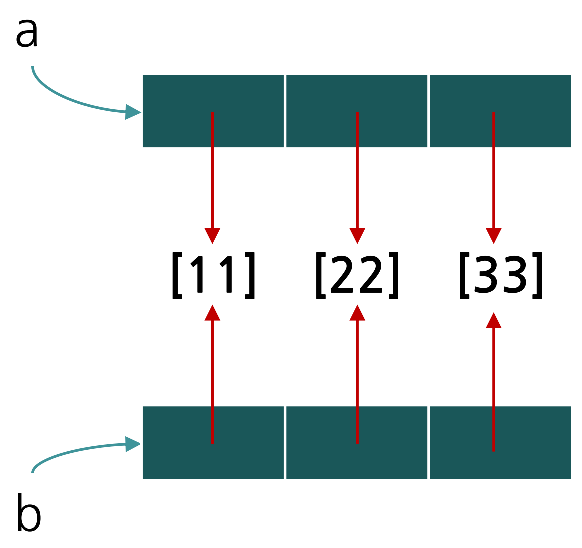
```
True
False
True
```

What does it mean?

a and b are different objects, but
a[0] and b[0] are referencing the same object.

Shallow copy

- New object created
  - Contents of the original object are copied
  - If the contents are references, then the *references* are copied



```
a = [ [11], [22], [33] ]
b = a.copy()

print( a == b )
print( a is b )
print( a[0] is b[0])
```

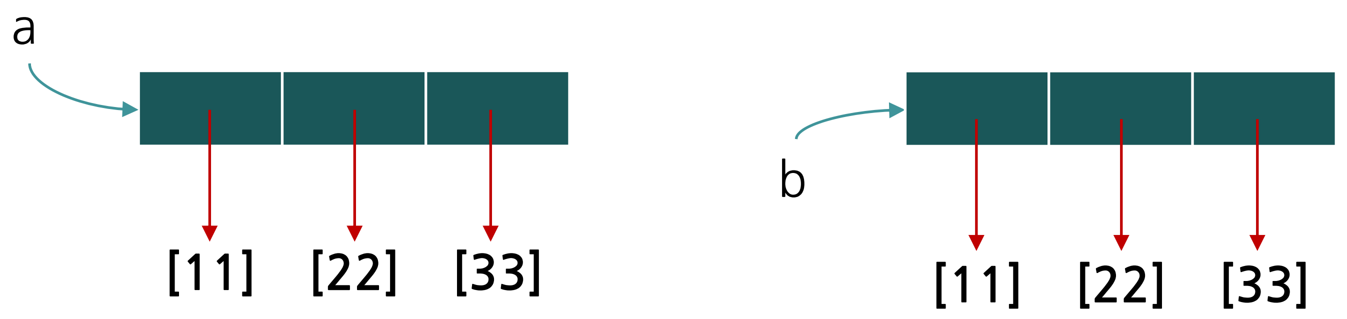
```
True
False
True
```

What does it mean?

a and b are different objects, but
a[0] and b[0] are referencing the same object.

Deep copy

- New object created
  - Contents of the original object are copied
  - If the contents are references, then **the copy the objects referred to** are copied



```
import copy

a = [ [11], [22], [33] ]
b = copy.deepcopy(a)

print( a == b )
print( a is b )
print( a[0] is b[0] )
```

```
True
False
False
```

b[0] has its own copy of the object.

Summary

- Variables store references to the objects, not the actual objects.
  - When you assign a variable, **a reference is copied**, not the object. Even it creates a new object and assigns its new reference to it in case of an immutable object.
- There are two kinds of equality.
  - Equality of content (value equality) can be tested with **==**
  - Equality of identity (reference equality) can be tested with is
- When a copy is created, it can be a shallow or deep copy.
  - A shallow copy copies the references.
  - A deep copy recursively copies the objects referred to.
- Lists slower but more powerful then tuples.
  - Lists can be modified and have lots of handy operations and methods.
  - Tuples are immutable and have fewer features.
- To convert between tuples and lists use the list() and tuple() function.