



COMPSCI 105 S1 2017

Principles of Computer Science

Equality, references and mutability



Exercises

- ▶ What is the output of the following code fragments?

```
names = ['Angela', 'Ann', 'Adriana']  
my_list = [len(x) for x in names]  
print (my_list)
```

```
def double(x):  
    return x*2  
  
my_list1 = [double(x) for x in range(5)]  
print(my_list1)
```

```
my_list2 = [double(x) for x in range(10) if x%2==0]  
print(my_list2)
```



Modeling objects in memory

▶ Value equality

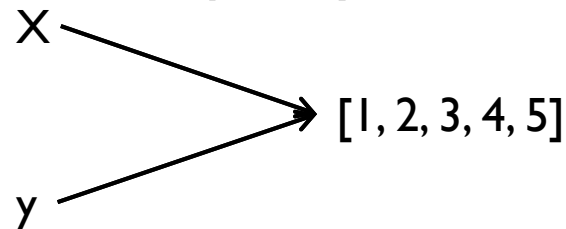
x → [1, 2, 3, 4, 5]

y → [1, 2, 3, 4, 5]

Two different objects that store the same information.

```
>>> x = [1, 2, 3, 4, 5]
>>> y = [1, 2, 3, 4, 5]
```

▶ Reference equality



Two different references / names for the same object.

```
>>> x = [1, 2, 3, 4, 5]
>>> y = x
```



Different ways to compare equality

▶ ==

- ▶ Calls a method of the object
- ▶ Programmer who defined the object decides how to determine equality
- ▶ 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 = [1,2,3,4,5]
>>> y = [1,2,3,4,5]
>>> x == y
>>> x is y
```

True
False

```
>>> x = [1,2,3,4,5]
>>> y = x
>>> x == y
>>> x is y
```

True
True



String

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

```
>>> a = 'foo'
>>> b = 'foo'
>>> id(a)
46065568
>>> id(b)
46065568
```

Same
memory
location

id(object) function
Return the “identity” of an
object.

```
>>> a is b
True
>>> a == b
True
>>>
```

```
>>> x = [1,2,3]
>>> y = [1,2,3]
>>> id(x)
47912776
>>> id(y)
47812296
```

```
>>> x is y
False
>>> x == y
True
```



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



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:
 - ▶ `li[0] = 10`
 - ▶ `li` still points to the same memory when you are done.

```
li = [1,2,3]
print(li)
print(id(li))
li[0] = 10
print(id(li))
print(li)
```

```
[1, 2, 3]
2681992
2681992
[10, 2, 3]
```



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:

```
tu = (1,2,3)  
tu[0] = 10
```

```
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.

```
name = "Angela"  
print(id(name))  
name = "Bob"  
print(id(name))
```

4217536
5699680

a new String object is instantiated and given the data "Bob" during its construction

- ▶ lower(), upper(), lstrip, rstrip...
 - ▶ Return a copy of s

```
name = "Angela"  
y = name.lower()
```

Return a new object



Operations on Lists

▶ append

- ▶ Add an item to the end of the list;

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

4997248
4997248
[1, 2, 3, 4]

▶ insert

- ▶ Insert an item at a given position.

```
x = [1, 2, 3]
print(id(x))
x.insert(0,4)
print(id(x))
print(x)
```

35668096
35668096
[4, 1, 2, 3]

▶ remove

- ▶ Remove the first item from the list whose value is x.

```
x = [1, 2, 3]
print(id(x))
x.remove(2)
print(id(x))
print(x)
```

34422912
34422912
[1, 3]



append Vs extend

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

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

5456000
5456000
[1, 2, 3, 4, 5, 6]

- ▶ append :- add **an item** to the end of the list

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

[1, 2, 3, [4, 5, 6]]

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

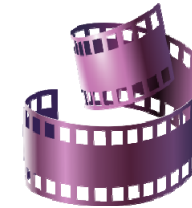
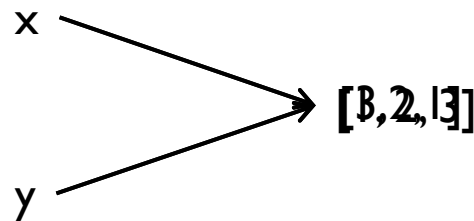
[1, 2, 3, 4, 5, 6]



Reversing a list

- ▶ The function `my_list.reverse()` **alters** the content of `my_list`

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



```
>>> x.reverse()
```

changes the contents
of the object

```
>>> x
```

[3, 2, 1]

```
>>> y
```

[3, 2, 1]

- ▶ Sort and Reverse

- ▶ Sort/reverse the items of the list, in place.



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)
```

Alter the object

[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]

- ▶ 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, 4]
y = x

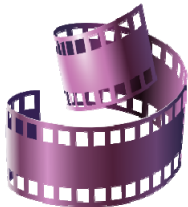
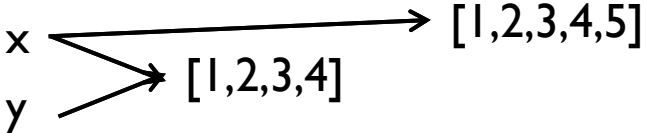
x = x + [5]

print(x)
print(y)

```

Return a new object

But replaced



```

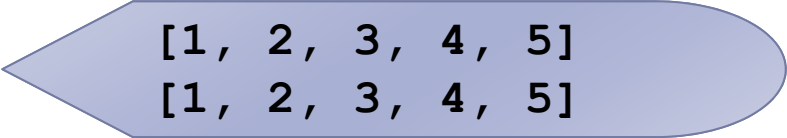
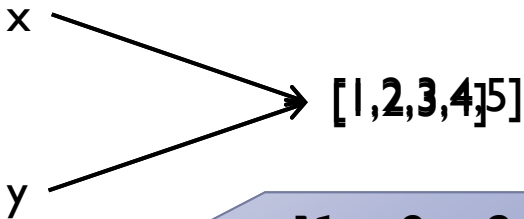
x = [1, 2, 3, 4]
y = x

x += [5]

print(x)
print(y)

```

Alter the object





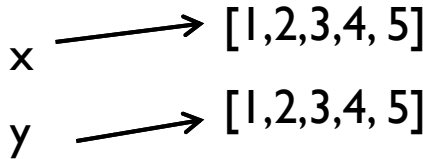
Shallow copies

- ▶ Lists and dictionaries have a copy method

- ▶ `data.copy()`

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

False
True



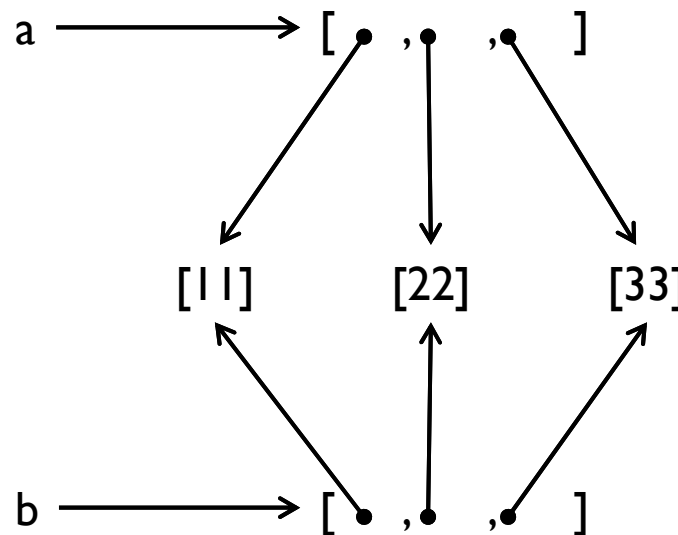
```
a = [ [11], [22], [33] ]
b = a.copy()
print( a is b )
print( a[0] is b[0] )
```

False
True



Shallow copy

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



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

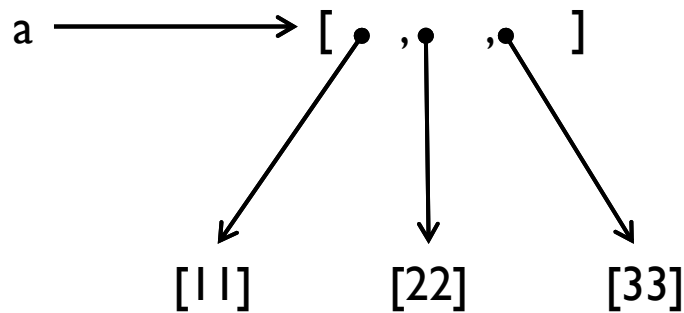
False
True



Deep copies

Example01.py

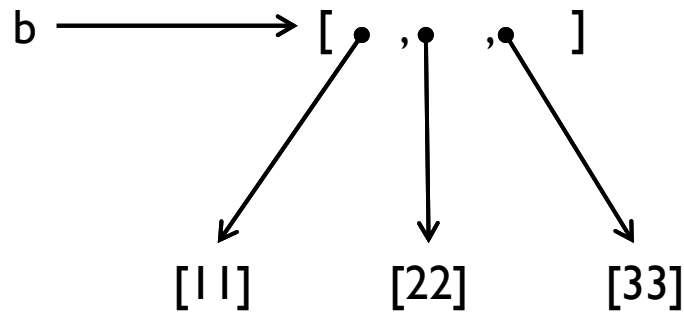
- ▶ New object created
 - ▶ Contents of the original object are copied
 - ▶ If the contents are references, then the copy the objects referred to



```
import copy

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

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



False
False



Summary

- ▶ **Variables store references to the objects, not the actual objects**
 - ▶ When you assign a variable, a reference is copied, not the 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 than 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**



Exercise 2

- ▶ What is the output of the following code fragments?

```
name = 'Angela'
x = name
name = 'Bob'
print(name)
print (x)
print (name is x)
print (name == x)
name = 'Angela'
print(name)
print (x)
print (name is x)
print (name == x)
```

```
my_list = [1,2,3]
y = my_list
my_list = [4,5]
print(my_list)
print (y)
print (my_list is y)
print (my_list == y)
my_list = [1,2,3]
print(my_list)
print (y)
print (my_list is y)
print (my_list == y)
```