ECE12: Introduction to Programming
Lecture 11

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 11: Overview

- Sequences
  - Multi-dimensional sequences
  - Example
- Object references
  - Object assignment
  - Object referencing
  - Copying objects
  - **None** object
  - Passing objects to functions

Multi-dimensional Sequences

- Sequences may be nested
  - Result:
    - Multi-dimensional sequences
    - Multiple-subscripted sequences
- Example: Matrix
  - two-dimensional list
  - list of lists (or, list of *rows* of list of *columns*)

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad M_{1,2} = 2$$

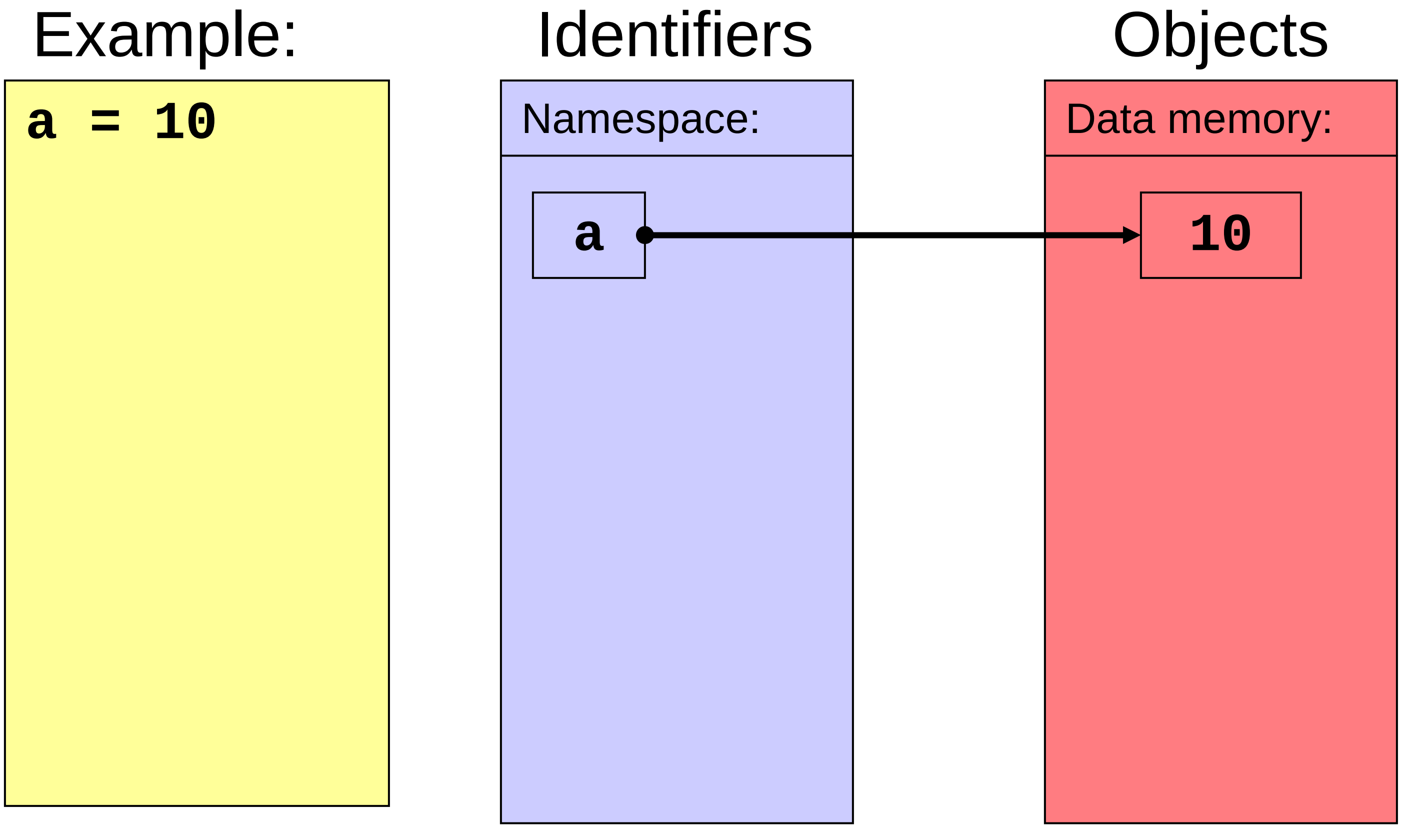
```
M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print M[0][1]
```

Object References

- Objects (revisited)
  - Objects are used to store data
  - Every object has
    - a type (e.g. integer, floating point, string, list, tuple, …)
    - a value (e.g. 42, 3.1415, “text”, [1,2,3], (4,5,6), …)
    - a size (number of bytes in the memory)
    - a location (address in the memory, aka. identity)
  - Objects are either
    - *mutable*: object value can be changed (e.g. list, dictionary)
    - *immutable*: object value cannot be changed (e.g. integer, floating point, string, tuple)
- Identifiers/variables (revisited)
  - serve as names for objects
  - are used to *reference* objects
  - are bound to objects
  - are stored in a namespace

Object References

- Example:



```
a = 10
```

Identifiers

Namespace:

a

Objects

Data memory:

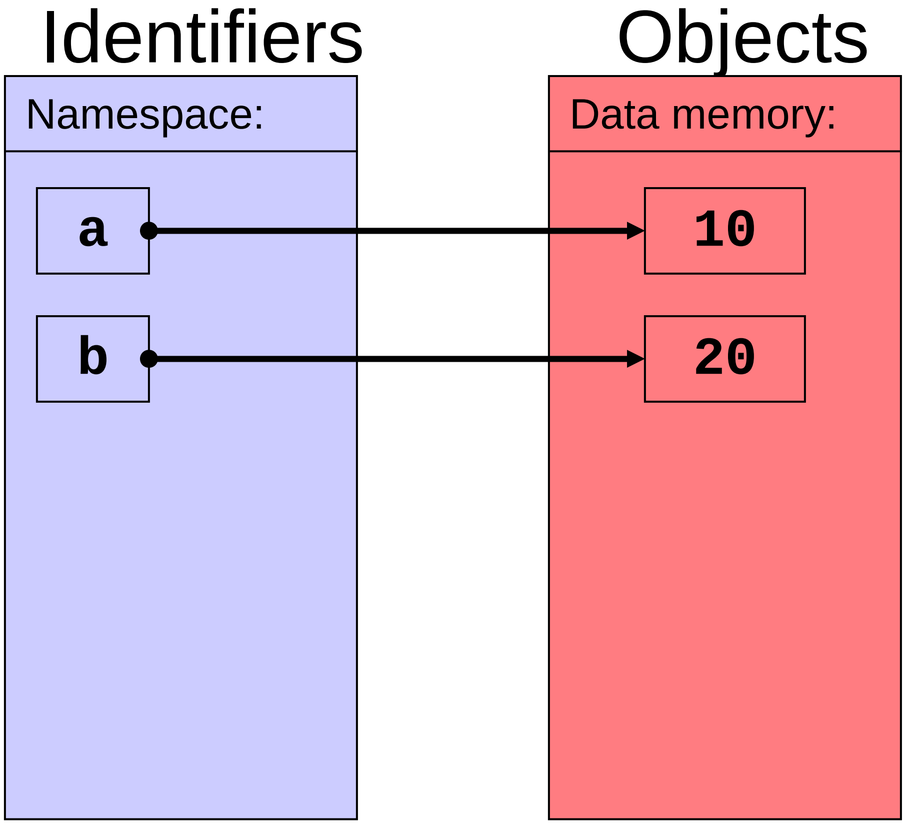
10

- Assignment operation
  - creates a *reference* from an identifier to an object
  - this reference is sometimes called a *pointer*

Object References

- Example:

```
a = 10
b = 20
```



- Many identifiers and many objects may exist

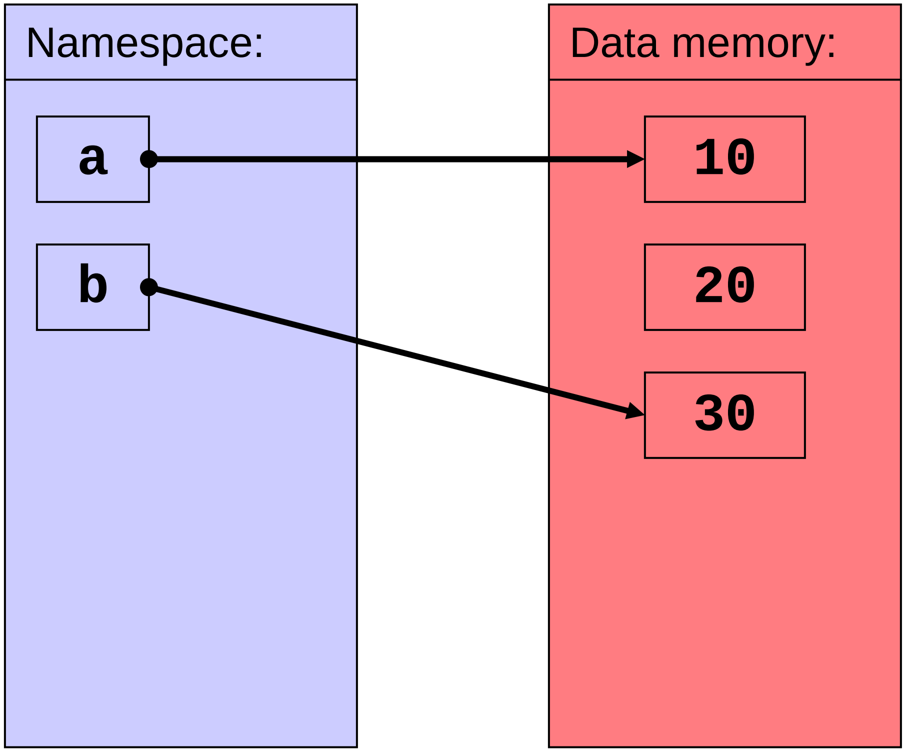
Object References

- Example:

```
a = 10
b = 20
b = 30
```

Identifiers

Objects



Namespace:

a

b

Data memory:

10

20

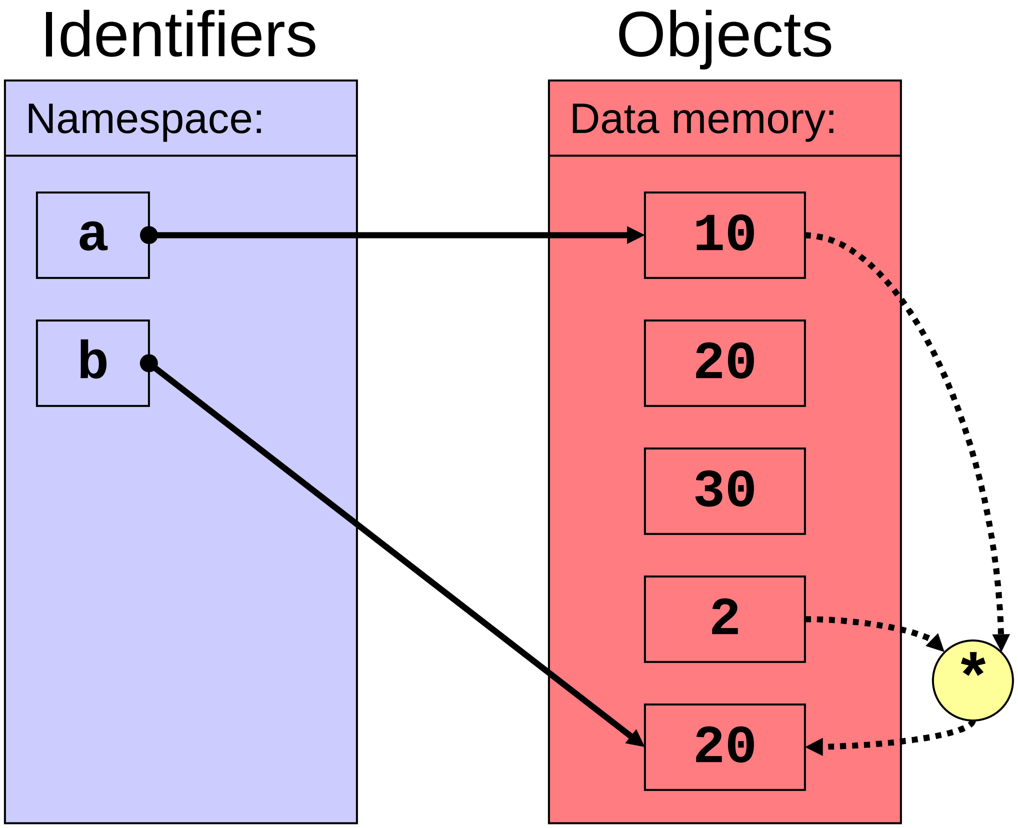
30

- Re-assignment to an identifer
  - only changes the reference to the new value
  - the old value is simply left alone (it is *not* overwritten!)

Object References

- Example:

```
a = 10
b = 20
b = 30
b = a * 2
```

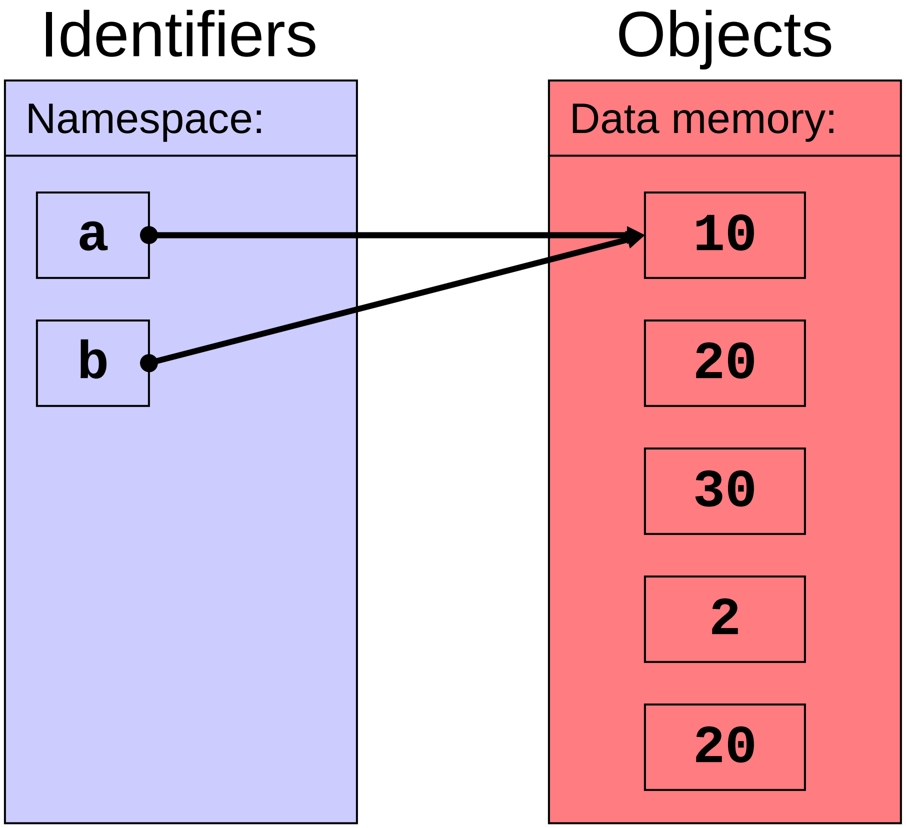


- Expression evaluation
  - uses references to access values
  - creates a new object

Object References

- Example:

```
a = 10
b = 20
b = 30
b = a * 2
b = a
```



- Object assignment
  - simply (re-) assigns the reference
  - objects may be referenced by 0, 1, or many identifiers (or objects)

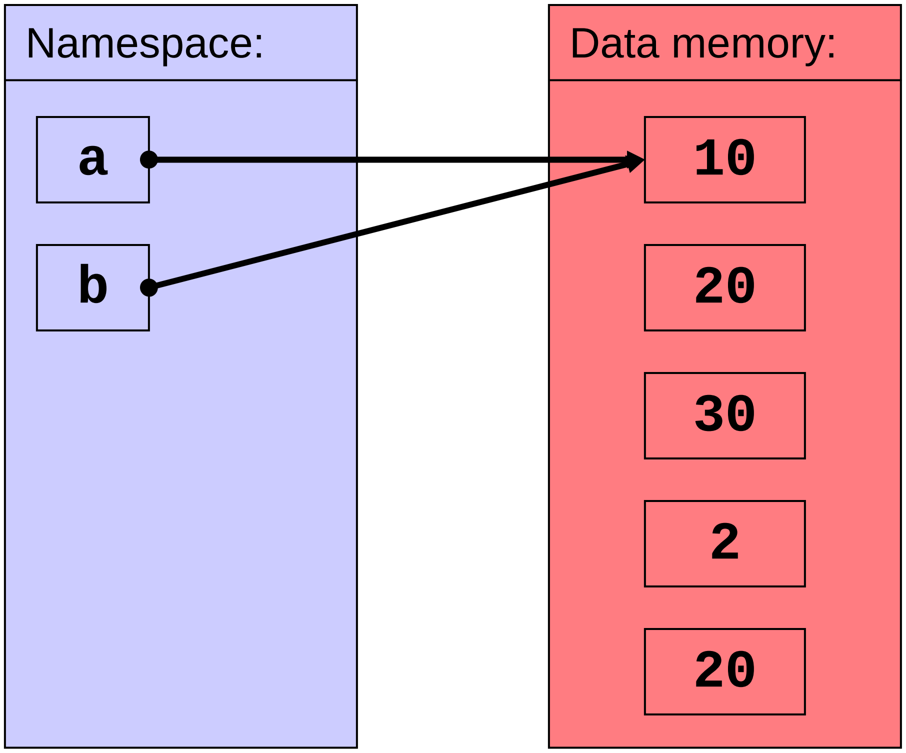
Object References

- Example:

```
a = 10
b = 20
b = 30
b = a * 2
b = a
```

Identifiers

Objects

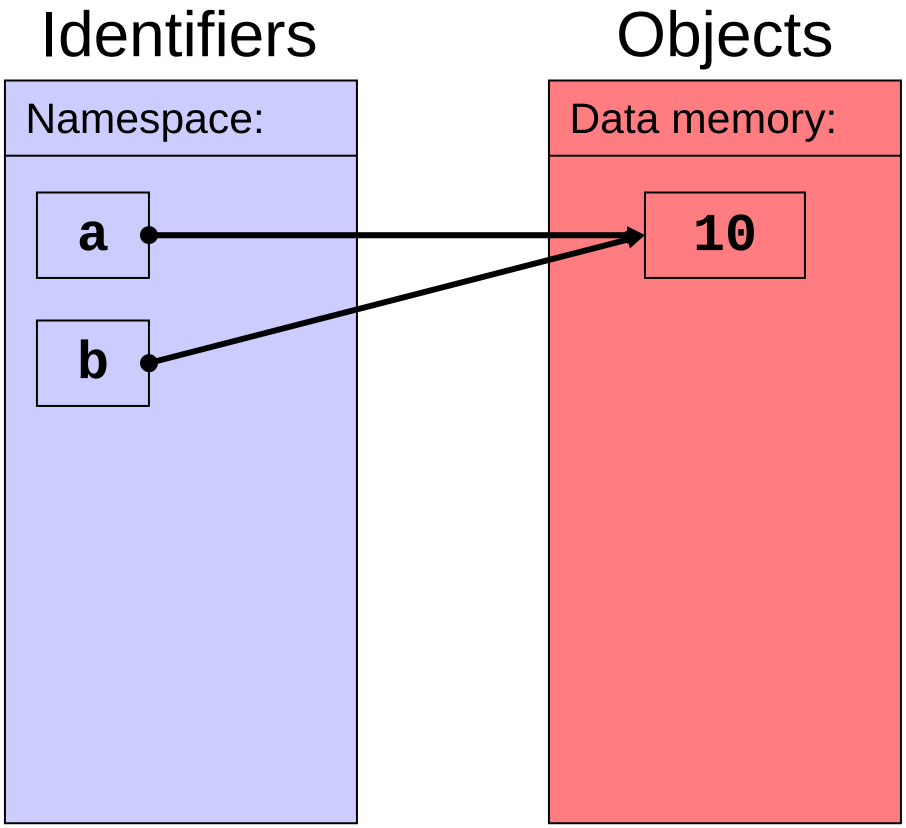


- Reference count
  - number of references to an object
  - if reference count is zero, an object cannot be accessed any more

Object References

- Example:

```
a = 10
b = 20
b = 30
b = a * 2
b = a
```



Identifiers

Objects

Namespace:

Data memory:

a

b

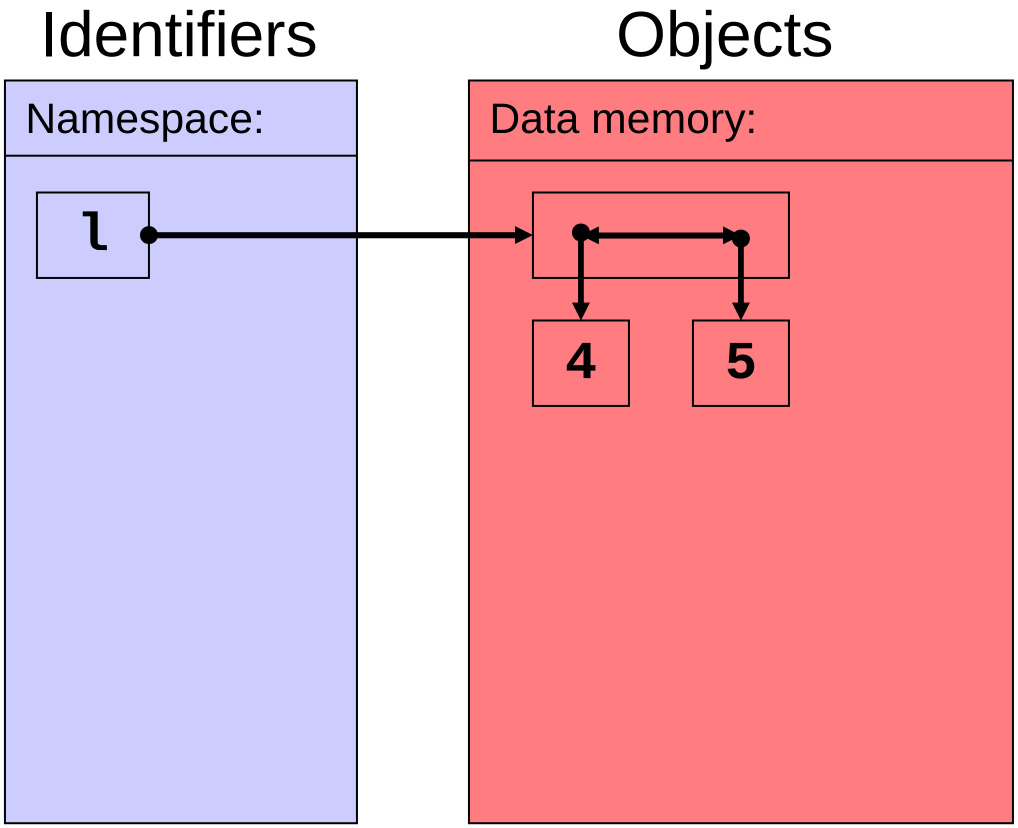
10

- Garbage collection
  - frees memory occupied by un-referenced objects
  - automatic in Python (at unspecified times)

Object References: Copying Lists

- Example 2:

```
l = [4,5]
```

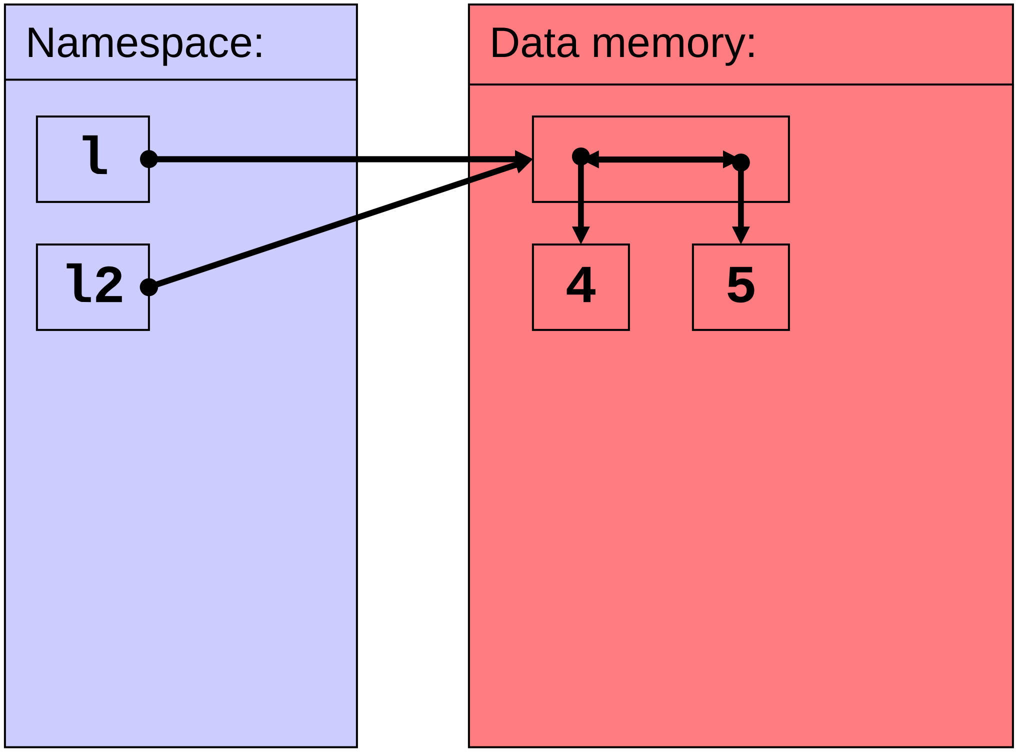


- List object
  - composite object (composed of child objects)
  - contains references to child objects

Object References: Copying Lists

- Example 2: Identifiers Objects

```
l = [4,5]
l2 = l
```

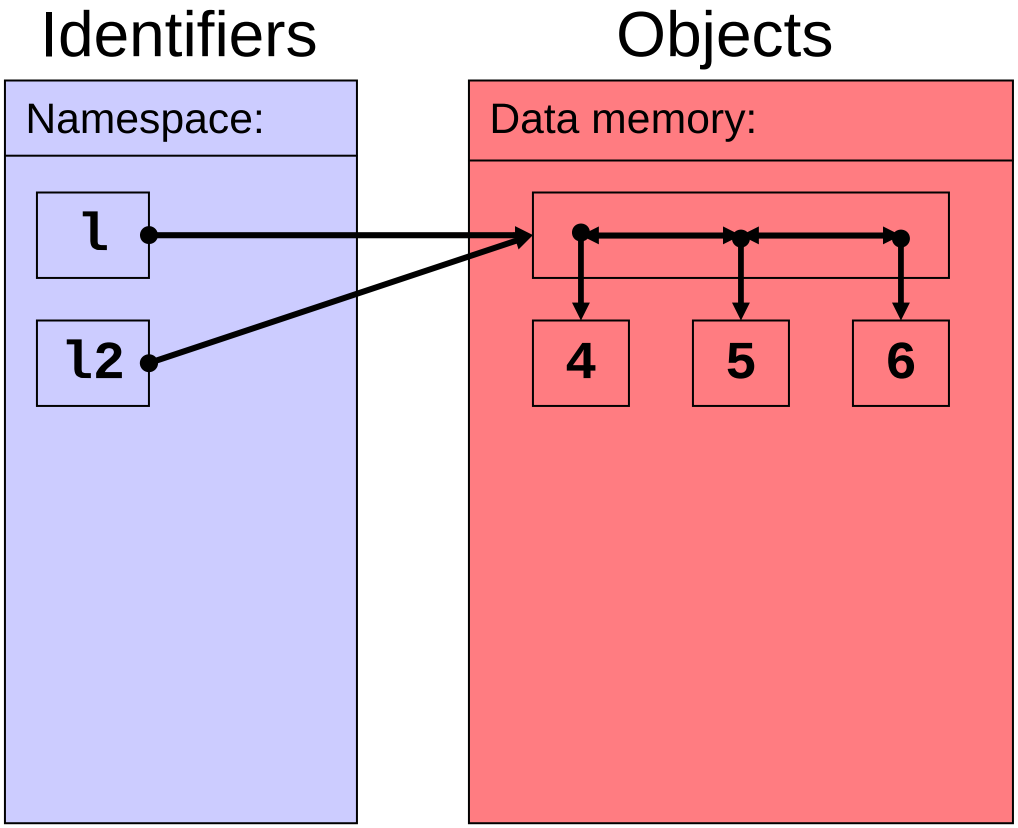


- List assignment
  - creates a new reference to the list object
  - exactly the same as standard assignment

Object References: Copying Lists

- Example 2:

```
l = [4,5]
l2 = l
l.append(6)
```



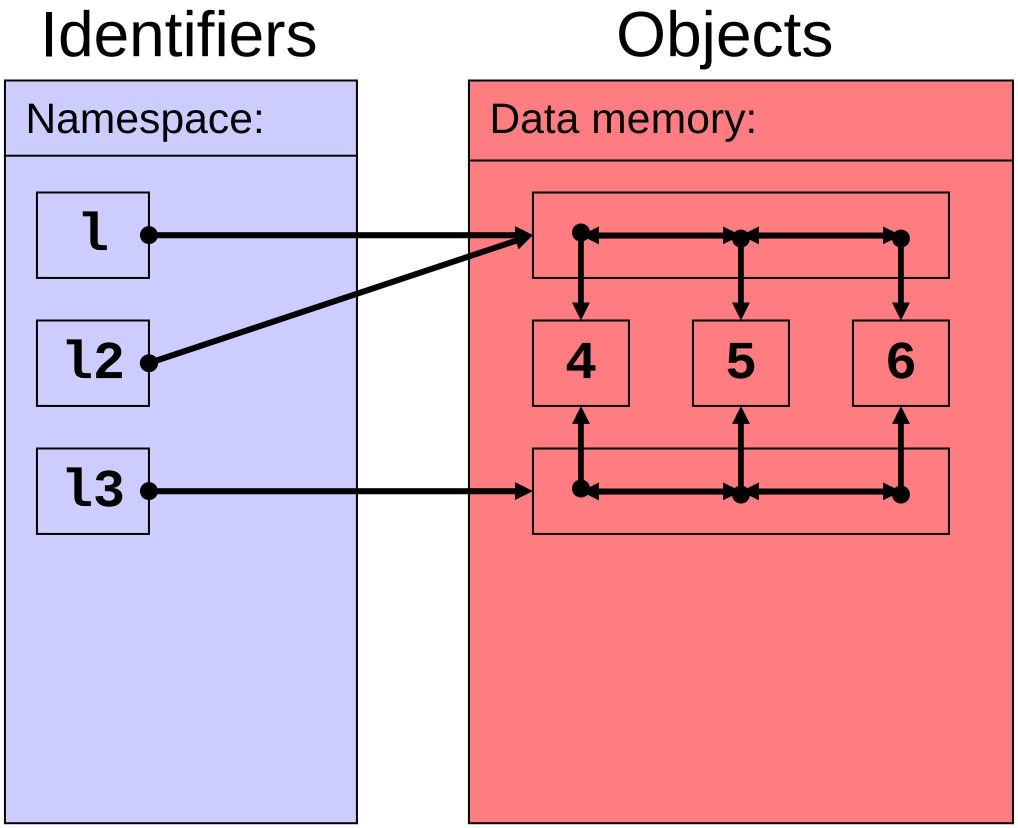
- List objects are mutable
  - appending a new list element changes the value of the list object
  - note that both **`l`** and **`l2`** are modified!

Object References: Copying Lists

- Example 2:

```
l = [4,5]
l2 = l
l.append(6)

l3 = l[:]
```



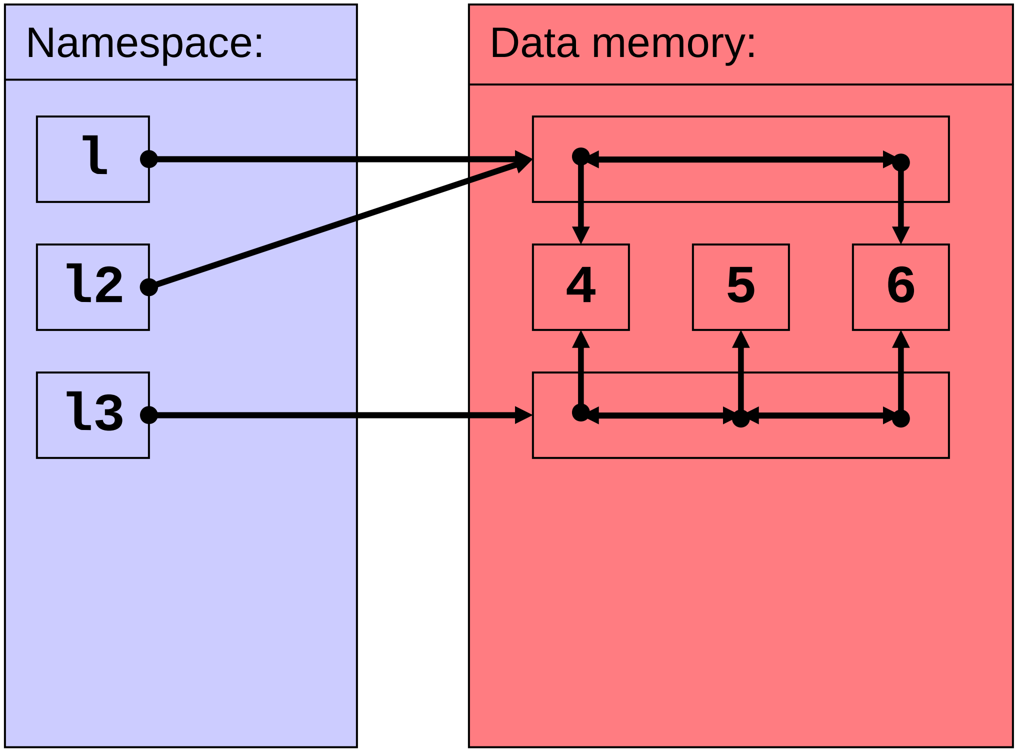
- Slicing operator creates a *shallow copy* of the list
  - list object itself is copied
  - list elements are still identical

Object References: Copying Lists

- Example 2: Identifiers Objects

```
l = [4,5]
l2 = l
l.append(6)

l3 = l[:]
del l[1]
```



- Deleting a list element
  - modifies the list object (because it is mutable)
  - note that shallow copy **l3** is not affected by this deletion!

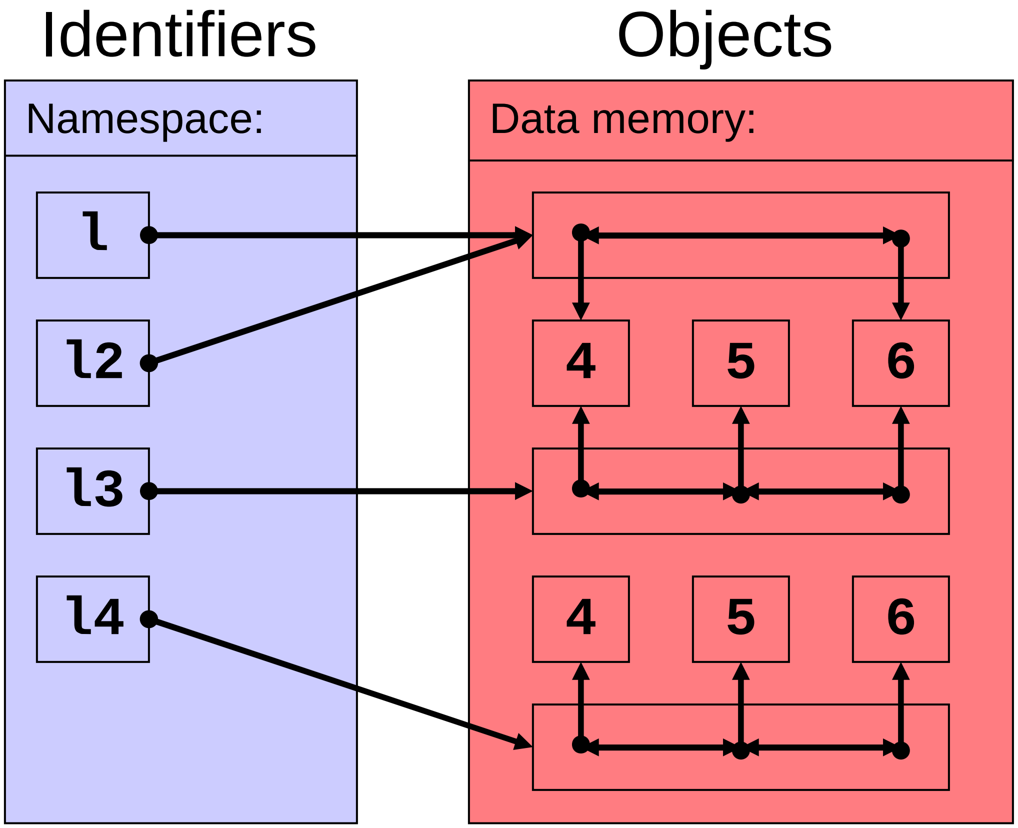
Object References: Copying Lists

- Example 2:

```
l = [4,5]
l2 = l
l.append(6)

l3 = l[:]
del l[1]

import copy
l4 = copy. \
 deepcopy(l3)
```



- Module **copy**
  - provides function **deepcopy**
  - **deepcopy** creates a copy of the entire object including its children

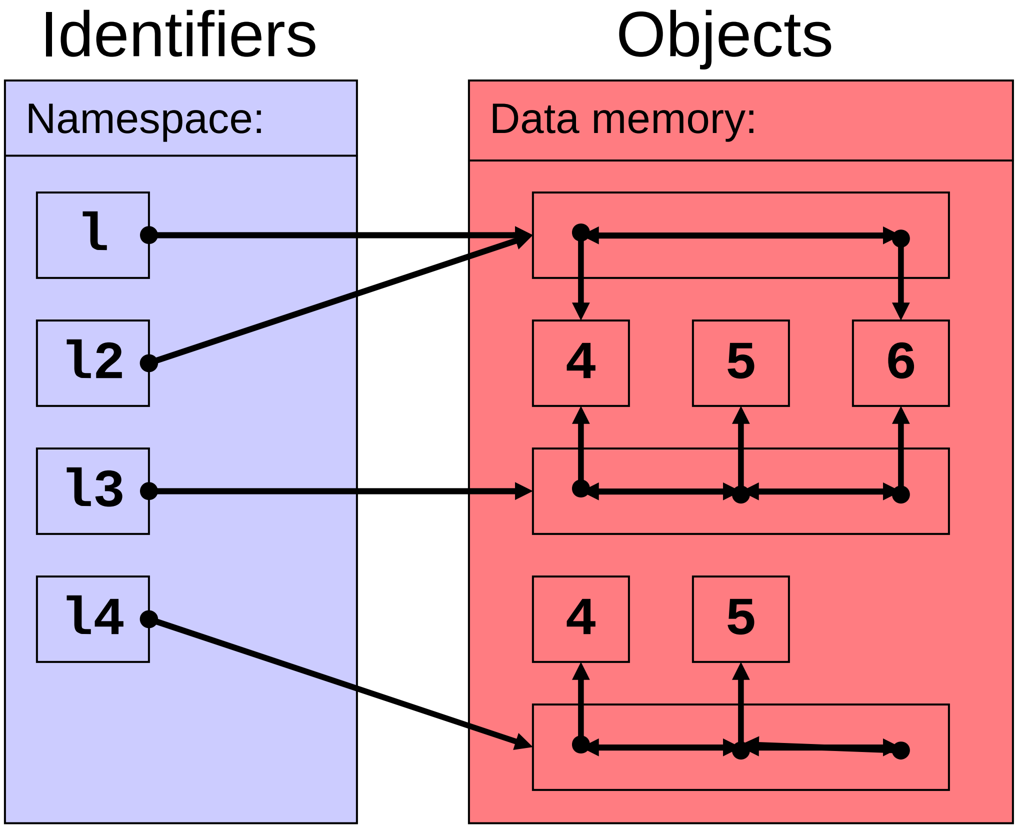
Object References: Empty Object

- Example 2:

```
l = [4,5]
l2 = l
l.append(6)

l3 = l[:]
del l[1]

import copy
l4 = copy. \
 deepcopy(l3)
l4[2] = None
```



- Object **None**
  - empty object provided by the built-in namespace
  - aka. **NIL** (Pascal) or **NULL** in C/C++

Passing Objects to Functions

- Two ways of passing arguments to functions
  - pass by value
    - a copy of the value is made
    - the function cannot modify the variable of the caller
  - pass by reference
    - a reference (pointer) to the value is passed
    - the function can modify the variable of the caller
- Python: pass by object reference
  - mutable object
    - pass by reference
  - immutable object
    - pass by value

Passing Objects to Functions

- Example:
  - passing immutable objects to functions

```
# “pass by value”

def f(x):
    print “x passed to f() is”, x
    x = 20
    print “x modified in f() to”, x

x = 10
print “Global x is”, x
f(x)
print “Global x after f() is”, x
```

```
Global x is 10
x passed to f() is 10
x modified in f() to 20
Global x after f() is 10
```

Passing Objects to Functions

- Example:
  - passing mutable objects to functions

```
# “pass by reference”

def f(l):
    print “l passed to f() is”, l
    l[0] = 2
    print “l modified in f() to”, l

l = [1,0]
print “Global l is”, l
f(l)
print “Global l after f() is”, l
```

```
Global l is [1,0]
l passed to f() is [1,0]
l modified in f() to [2,0]
Global l after f() is [2,0]
```