ECE12: Introduction to Programming
Review of Lectures 8-15

Rainer Dömer

doemer@uci.edu

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

Review of Lectures 8-15

- Lecture 8: Functions, namespaces, scope
- Lecture 9: Sequences, strings, lists, tuples
- Lecture 10: Sequence, dictionary operations
- Lecture 11: Object references
- Lecture 12: Functional list operations
- Lecture 13: Functional programming, recursion
- Lecture 14: Object-oriented programming
- Lecture 15: Classes, objects, access control

Random Number Generation

- Module **`random`**
  - part of Python standard library
  - provides pseudo-random number generator and associated convenience functions
  - function **`randrange()`** returns a random integer in the range specified by the arguments
    - **`randrange(end)`**
      returns a random integer between **`0`** and **`end-1`**
    - **`randrange(start, end)`**
      returns a random integer between **`start`** and **`end-1`**
- Example:

```
import random

for i in range(10):
    print random.randrange(10),

for i in range(20):
    print random.randrange(1,7),
```

Function Arguments

- Default arguments
  - Default values for arguments can be specified with the function definition
  - Default arguments are optional in function calls
- Keyword arguments
  - Arguments for functions may be specified by keyword association
  - In this case, order of arguments does not matter in function calls
- Example:

```
def box_volume(length = 1, width = 1, height = 1):
    return length * width * height

print box_volume(4, 5, 6)
print box_volume(4, 5)
print box_volume()

print box_volume(width = 8, height = 9)
```

Namespaces and Scope

- Identifiers are used to reference objects
  - Identifier serves as a *name* for an object
  - Identifiers are stored in a *namespace*
  - Identifiers have an associated *scope*
- Namespace
  - local namespace (i.e. in a function body)
  - global namespace (aka. module namespace)
  - built-in namespace (e.g. **`raw_input()`**, etc.)
- Scope
  - program region in which an identifier is visible
  - *shadowing*: an identifier is made invisible due to a local identifier with the same name

Namespaces and Scope

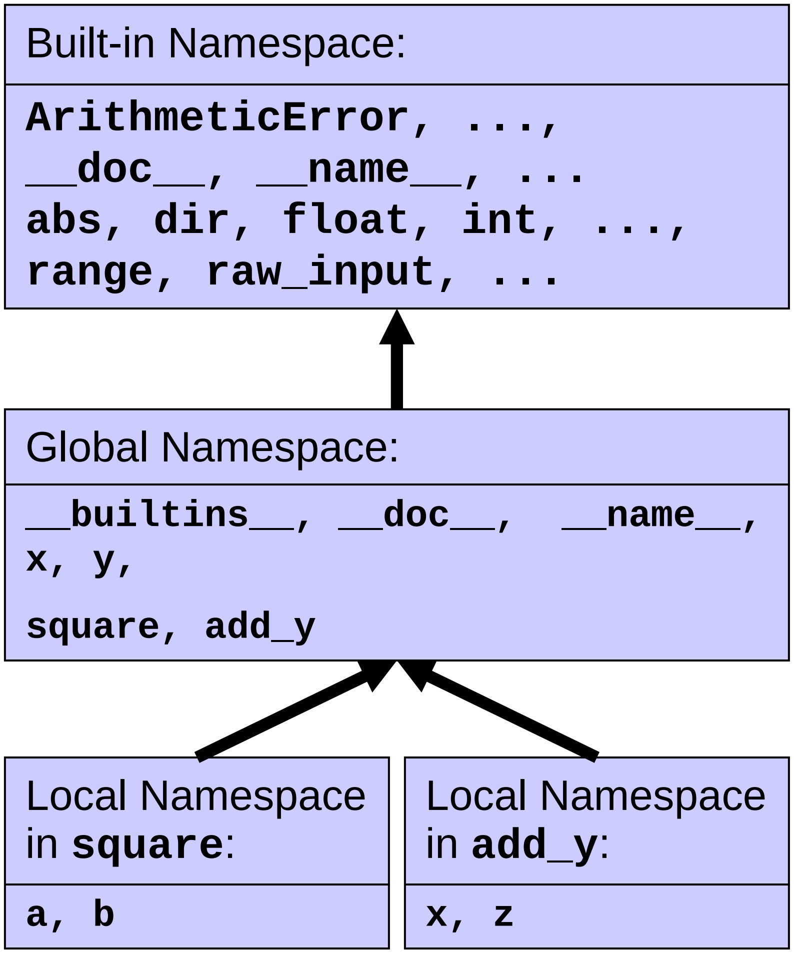
- Example:

```
# global variables
x = 10
y = 20

# function definitions
def square(a):
    b = a * a
    return b

def add_y(x)
    z = x + y
    return z

# main program
print x + y
```



Built-in Namespace:

ArithmeticError, ...,
__doc__, __name__, ...
abs, dir, float, int, ...,
range, raw_input, ...

Global Namespace:

__builtins__, __doc__, __name__,
x, y,
square, add_y

Local Namespace in **square**:

a, b

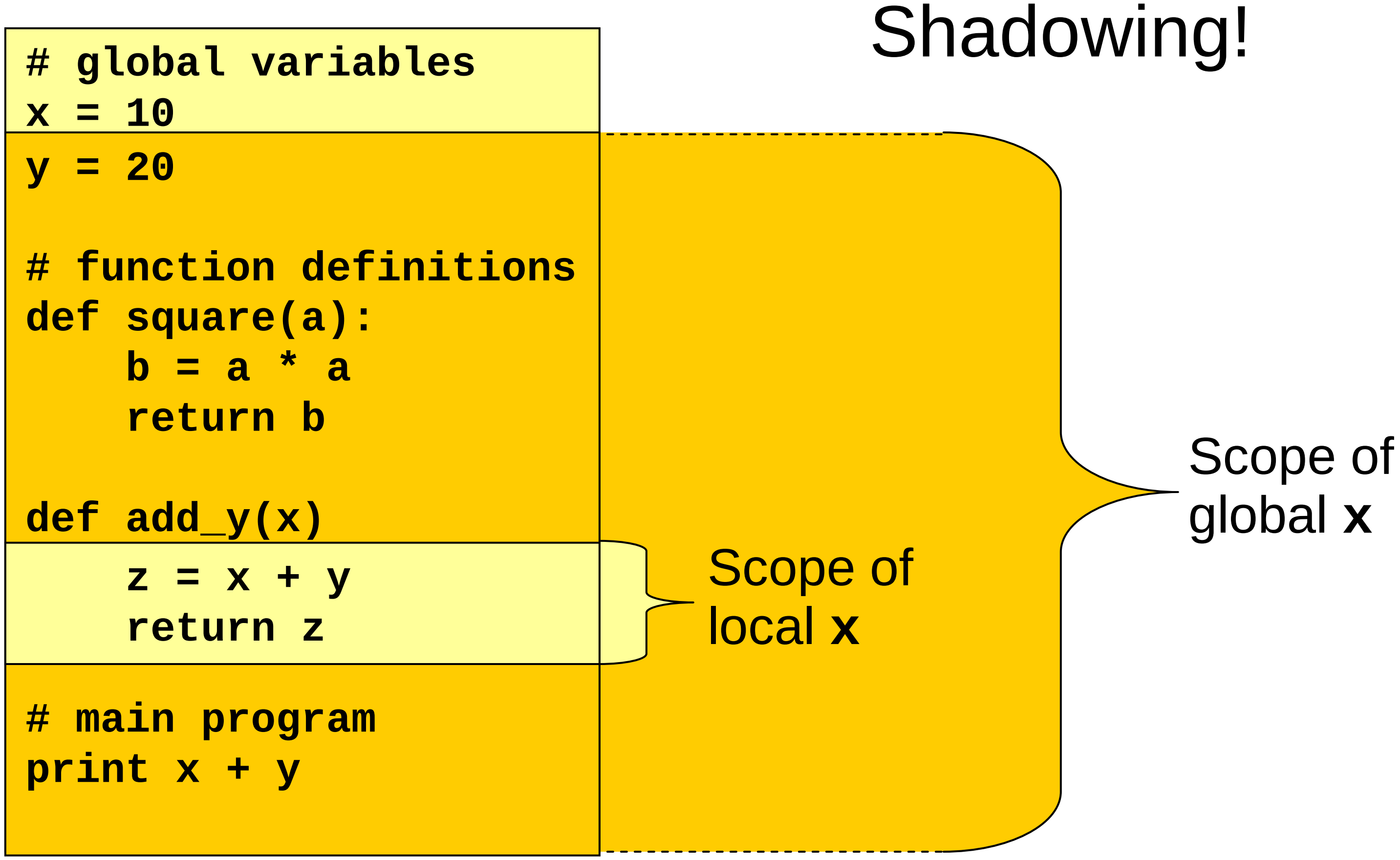
Local Namespace in **add_y**:

x, z

Namespaces and Scope

- Example:

Shadowing!



```
# global variables
x = 10
y = 20

# function definitions
def square(a):
    b = a * a
    return b

def add_y(x)
    z = x + y
    return z

# main program
print x + y
```

Scope of local **x**

Scope of global **x**

Program Introspection

- Introspection
  - – Ability to obtain information about identifiers in namespaces at program runtime
    - usually *not* available in compiled languages (e.g. C/C++)
- **`type()`** function
  - – Built-in function that returns the type of an object
  - – Examples:
    - **`type(42)`** returns **`<type ‘int’>`**
    - **`type(1.0)`** returns **`<type ‘float’>`**
- **`dir()`** function
  - – Built-in function that returns a list of all identifiers
    - in the current namespace (no argument)
    - in the specified namespace (one argument)
  - – Example:
    - **`import math ; dir(math)`** returns **`[‘acos’, ‘asin’, ‘atan’, ...]`**

Namespaces and Scope

- Interactive Example:

```
% python
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> x = 10 ; y = 20
>>> dir()
['__builtins__', '__doc__', '__name__', 'x', 'y']
>>> def f():
...     a = 5 ; b = 6
...     print a,b,x,y
...     print dir()
>>> f()
5 6 10 20
['a', 'b']
>>> def g():
...     x = 42
...     print x
...     print dir()
>>> g()
42
['x']
>>> print x
10
>>> dir()
[..., 'f', 'g', 'x', 'y']
```

Module Namespaces

- Modules have their own namespace
  - The **`import`** construct imports identifiers from a module namespace into the current namespace
- Examples:
  - insert **`math`** module into current namespace

    -
      ```
      import math
      print math.sqrt(9.0)
      ```
  - insert **`sqrt`** and **`cos`** functions from **`math`** into current namespace

    -
      ```
      from math import sqrt, cos
      print sqrt(9.0)
      ```
  - insert all (!) names from **`math`** into current namespace

    -
      ```
      from math import *
      print sqrt(9.0)
      ```
  - insert **`sqrt`** from **`math`** as **`square_root`** into current namespace

    -
      ```
      from math import sqrt as square_root
      print square_root(9.0)
      ```
  - insert **`math`** module as **`std_math_lib`** into current namespace

    -
      ```
      import math as std_math_lib
      print std_math_lib.sqrt(9.0)
      ```

Data Structures

- Introduction
  - Until now, we have used mostly single data elements of basic (non-composite) type
    - integer types
    - floating point types
  - Most programs, however, require complex data structures of composite types
    - arrays, lists, queues, stacks
    - trees, graphs
    - dictionaries
- Python provides built-in support for
  - Sequences
    - string
    - list
    - tuple
  - Mappings (aka. associative arrays or hash tables)
    - dictionary

Sequences

- Types of sequences
  - String `s = “This is a string.”`
  - List `l = [6, 3, 2, 4, 5]`
  - Tuple `t = (3, 2, 0)`
    - Lists are mutable, strings and tuples are immutable!
- Operations on sequences
  - Length
      - `len(s) = 17` `len(l) = 5` `len(t) = 3`
  - Element access (by position)
    - from the front
      - `s[0] = “T”` `l[1] = 3` `t[2] = 0`
    - from the end
      - `s[-1] = “.”` `l[-2] = 4` `t[-3] = 3`
  - Concatenation and extension
    - **+**, **+=** operators
      - `s + “XYZ” = “This is a string.XYZ”`

Sequences

- Operations on sequences (continued)
  - Iteration over sequence

    -
      ```
      sequence = [23, 45, 67]
      for item in sequence:
          print item
      ```
    - Remember: **`range()`** returns a sequence of integers!
  - Sequence packing and unpacking

    -
      ```
      vector = (42, 7, 99)
      x,y,z = vector
      ```

    -
      ```
      a = 10 ; b = 20
      a,b = b,a
      ```
  - Slicing
    - **`[start:end]`** operator
      - **`s = “This is a test string.”`**
      - **`s[0:4] = “This”`**
      - **`s[-7:-1] = “string”`**
      - **`s[:4] = “This”`**
      - **`s[-12:] = “test string.”`**

List Example

- Program **`histogram.py`**:

```
# histogram.py: print a histogram for a list of numbers
#
# author: Rainer Doemer
#
# modifications:
# 02/04/04 RD    initial version (similar to fig05_05.py)

# initialize
values = []

# input
while 1:
    s = raw_input("Enter a number or type 'q' to quit: ")
    if s == 'q':
        break;
    i = int(s)
    values += [i]

# compute and output
print "Histogram for %d values:" % len(values)
for v in values:
    print "%3d" % v, "*" * v
```

List Methods

- Additional operations on lists are available as methods
  - **`append(item)`** inserts **`item`** at the end of the list
  - **`count(elem)`** returns the number of **`elem`** in the list
  - **`extend(list)`** concatenates **`list`** to the list
  - **`index(elem)`** returns the position of the first **`elem`**
  - **`insert(index,item)`** inserts **`item`** at position **`index`**
  - **`pop()`** removes and returns last element
  - **`pop(index)`** removes and returns element at **`index`**
  - **`remove(elem)`** removes the first **`elem`** from the list
  - **`reverse()`** reverses the list contents (in place)
  - **`sort()`** sorts the list contents (in place)
- Example: **`s=[3,2,6]; s.append(4); s.sort()`**

Dictionaries

- Dictionary
  - – built-in data type in Python
  - – aka. *hash table* or *associative array*
  - – (unordered) set of key-value pairs
    - Key: immutable data type (such as integer, string, tuple)
    - Value: any data type (basic or composite)
- Dictionary operations
  - – Example: Grade book
    - **`gb = { “John”:66, “Jane”:77, “Joe”:87 }`**
    - **`print gb[“Jane”]`** (read access operation)
    - **`gb[“Jane”] = 75`** (write access operation)
    - **`gb[“Jack”] = 79`** (insertion operation)
    - **`del gb[“John”]`** (deletion operation)

Dictionary Methods

- Additional operations on dictionaries are available as methods

`clear()`	deletes all items in the dictionary
`copy()`	creates a (shallow) copy of the dictionary
`get(key)`	returns the value associated with **`key`**
`has_key(key)`	returns **`1`** if **`key`** is in the dictionary, otherwise **`0`**
`keys()`	returns a list of the keys in the dictionary
`values()`	returns a list of the values in the dictionary
`items()`	returns a list of tuples of key-value pairs
`update(dict)`	adds all key-value pairs of **`dict`** to the dictionary (overwriting same entries)

  - etc.

Dictionary Example

- Program **`dictionary.py`**:

```
# dictionary.py: simple English-German dictionary
# author: Rainer Doemer
#
# 02/05/04 RD  initial version

# initialize
dict = {"one":"eins", "two":"zwei", "three":"drei",
        "four":"vier", "five":"fuenf", "six":"sechs",
        "seven":"sieben", "eight":"acht", "nine":"neun"}

# input, compute, output
while 1:
    s = raw_input("Enter an English word or type 'q' to quit: ")
    if s == 'q':
        break;
    if (dict.has_key(s)):
        print "'%s' in English is '%s' in German." % (s,dict[s])
    else:
        print "No data for '%s'!" % s
```

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*)

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

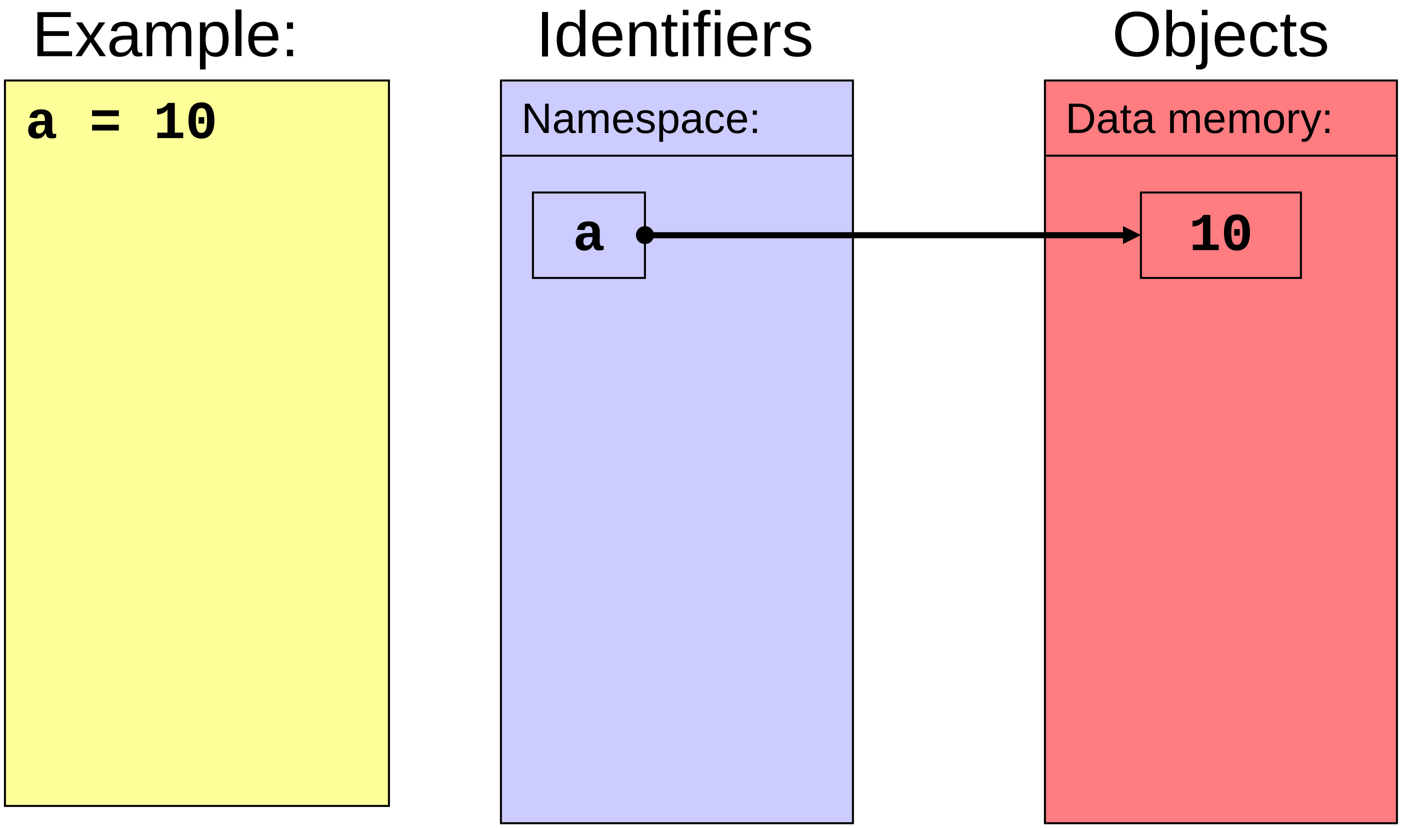
```
M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print M[0][1]
```

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:



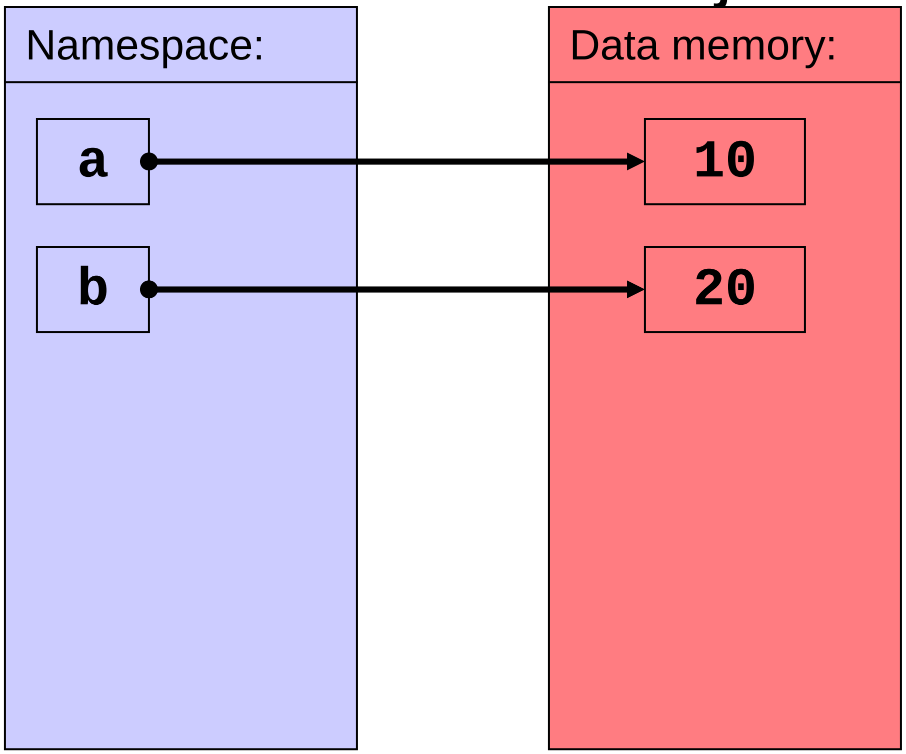
```
a = 10
```

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

Object References

- Example: Identifiers Objects

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

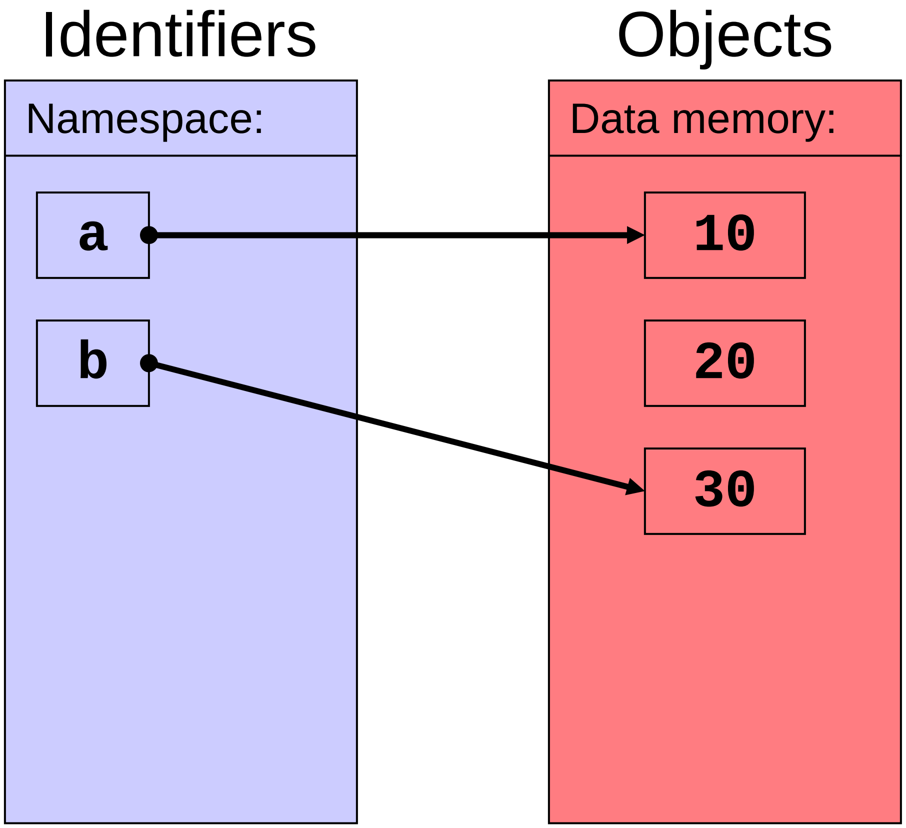


- Many identifiers and many objects may exist

Object References

- Example:

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

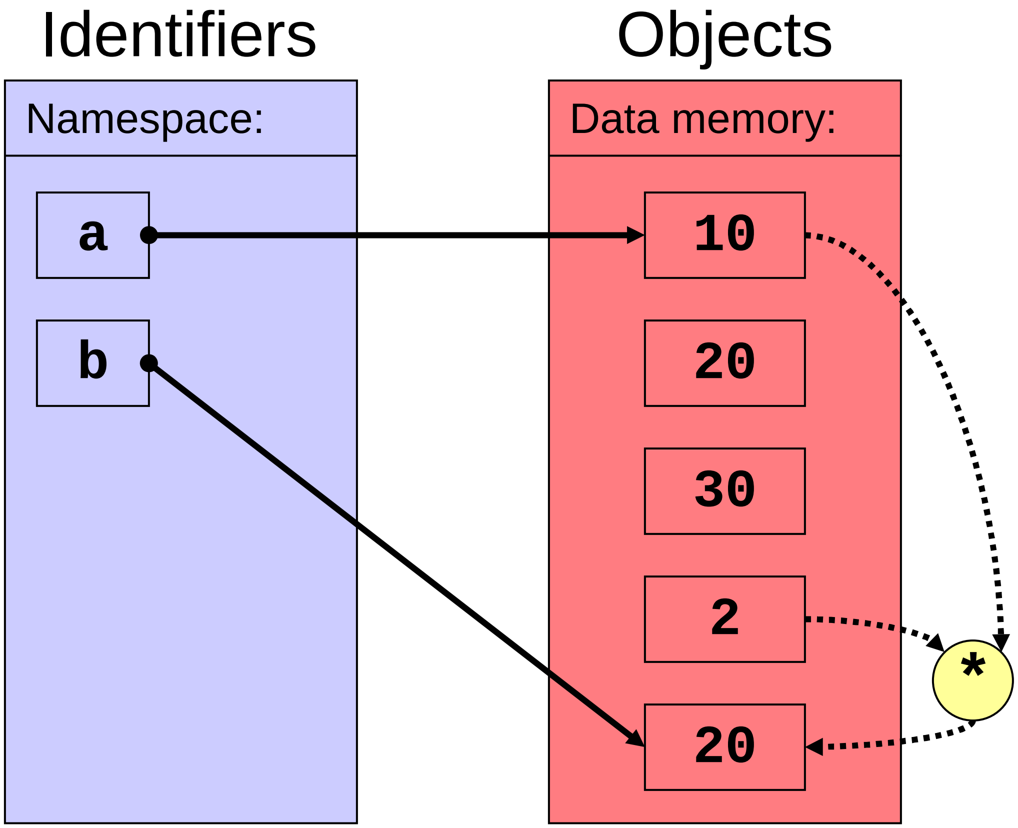


- Re-assignment to an identifer
  - only changes the reference to the new value
  - the old value is simply left alone (it is *not* overwritten!)

Object References

- Example:

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

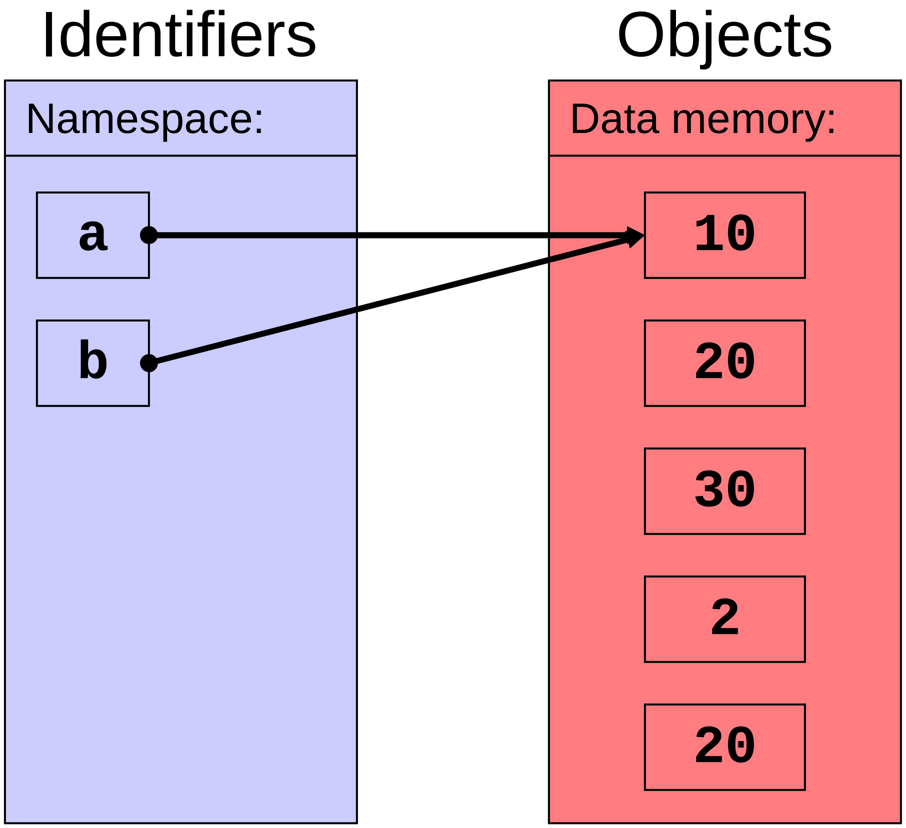


- Expression evaluation
  - uses references to access values
  - creates a new object

Object References

- Example:

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

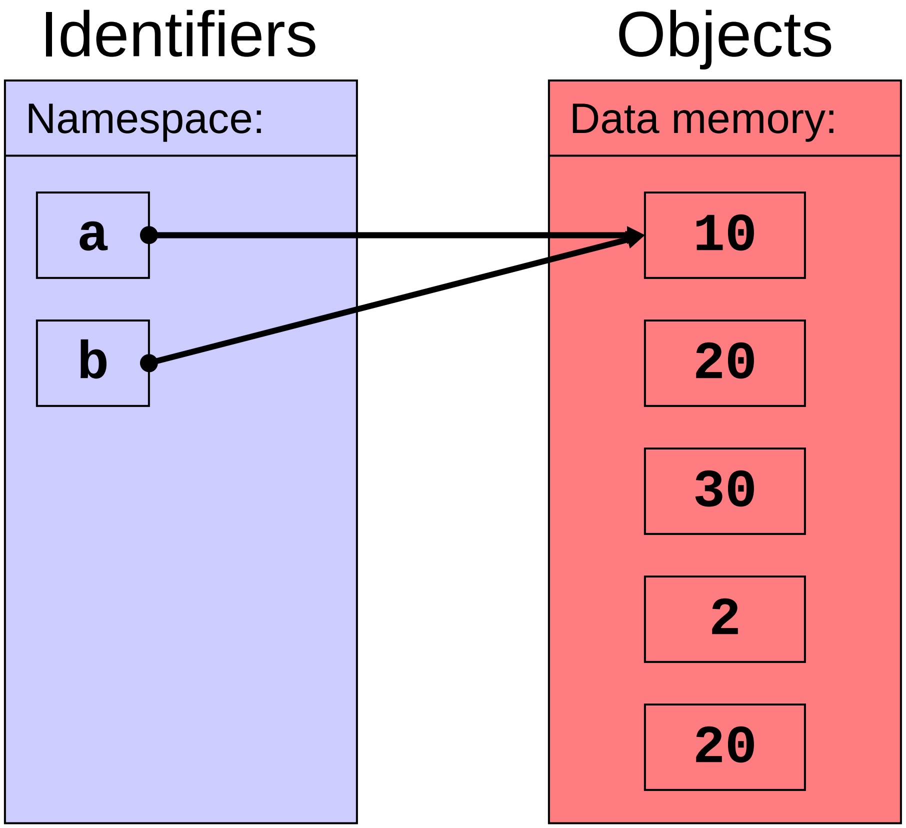


- Object assignment
  - simply (re-) assigns the reference
  - objects may be referenced by 0, 1, or many identifiers (or objects)

Object References

- Example:

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



Identifiers

Objects

Namespace:

a

b

Data memory:

10

20

30

2

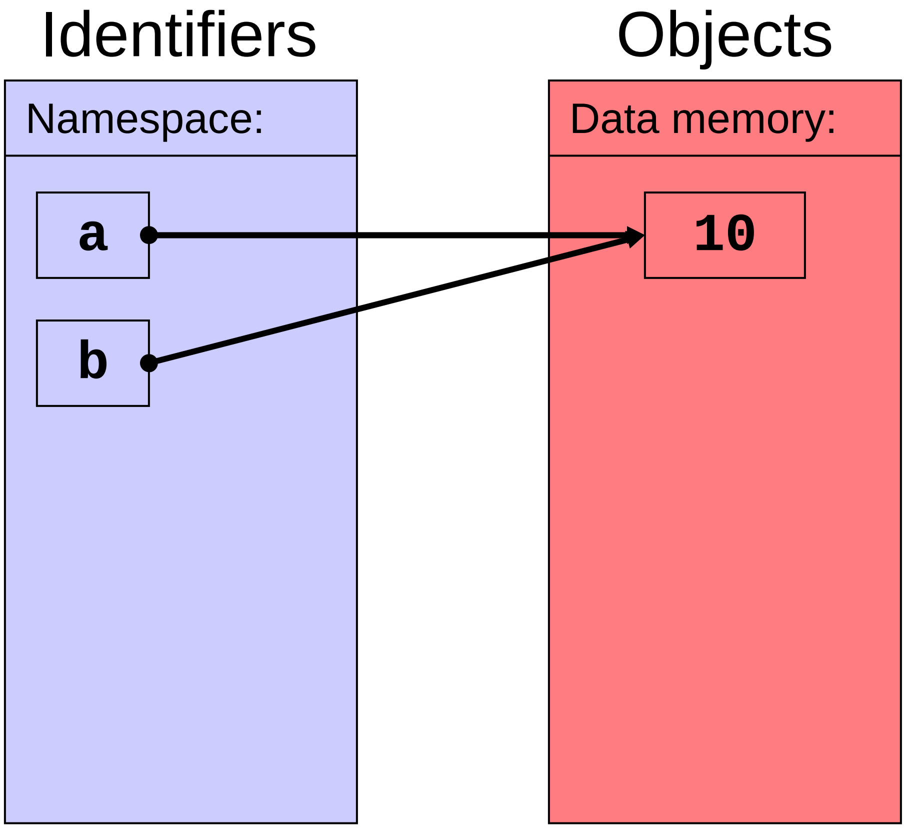
20

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

Object References

- Example:

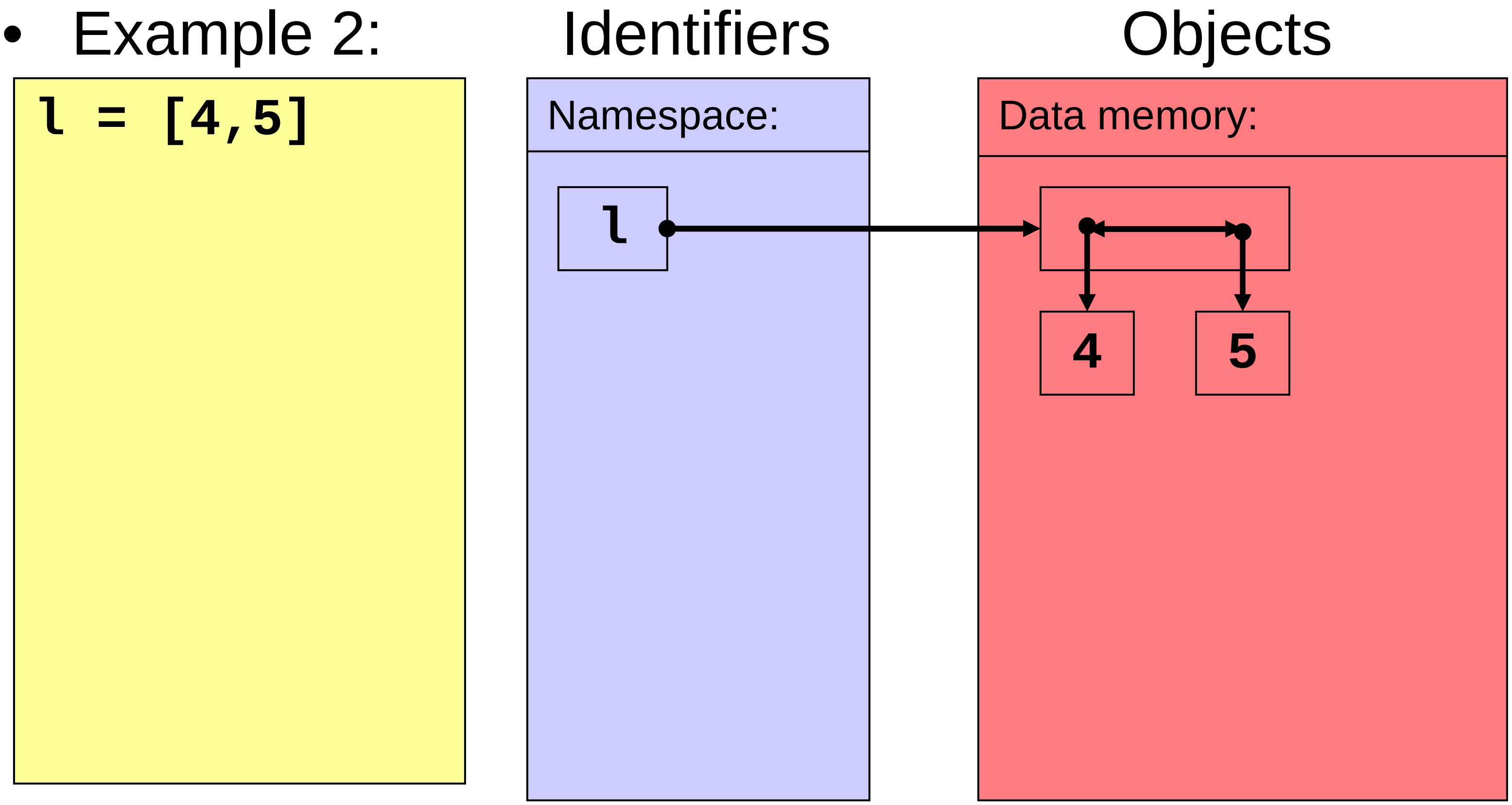
```
a = 10
b = 20
b = 30
b = a * 2
b = a
```



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

Object References: Copying Lists

- Example 2:



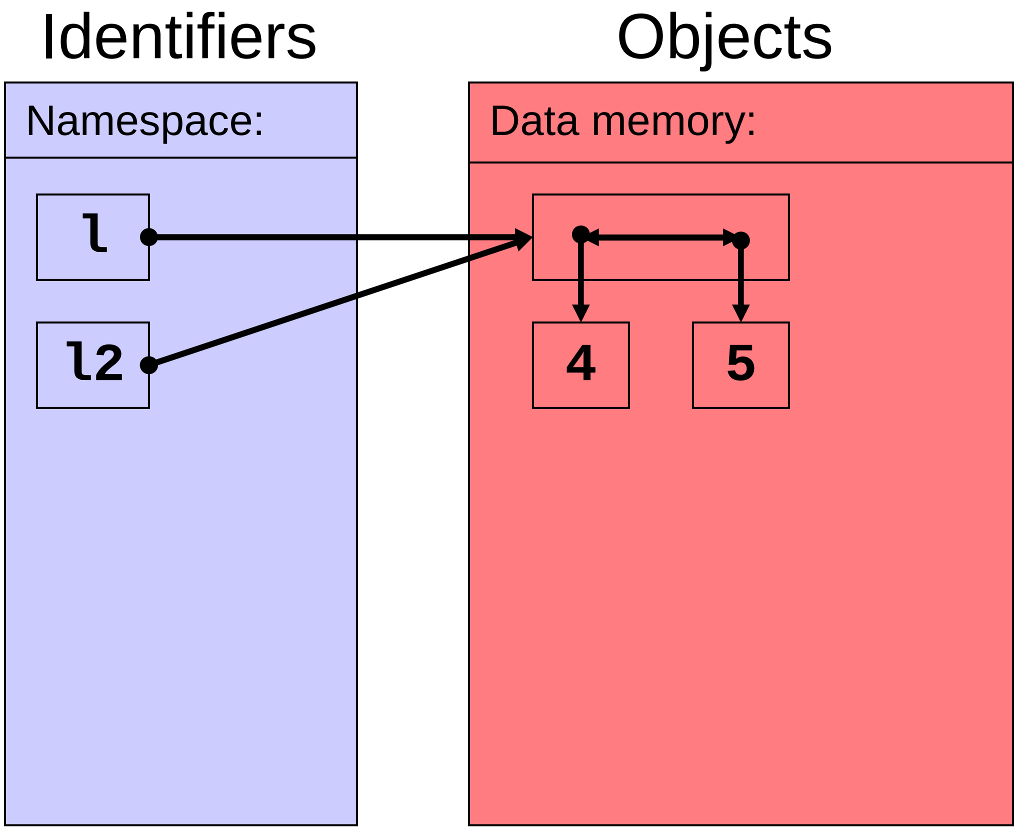
```
l = [4,5]
```

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

Object References: Copying Lists

- Example 2:

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

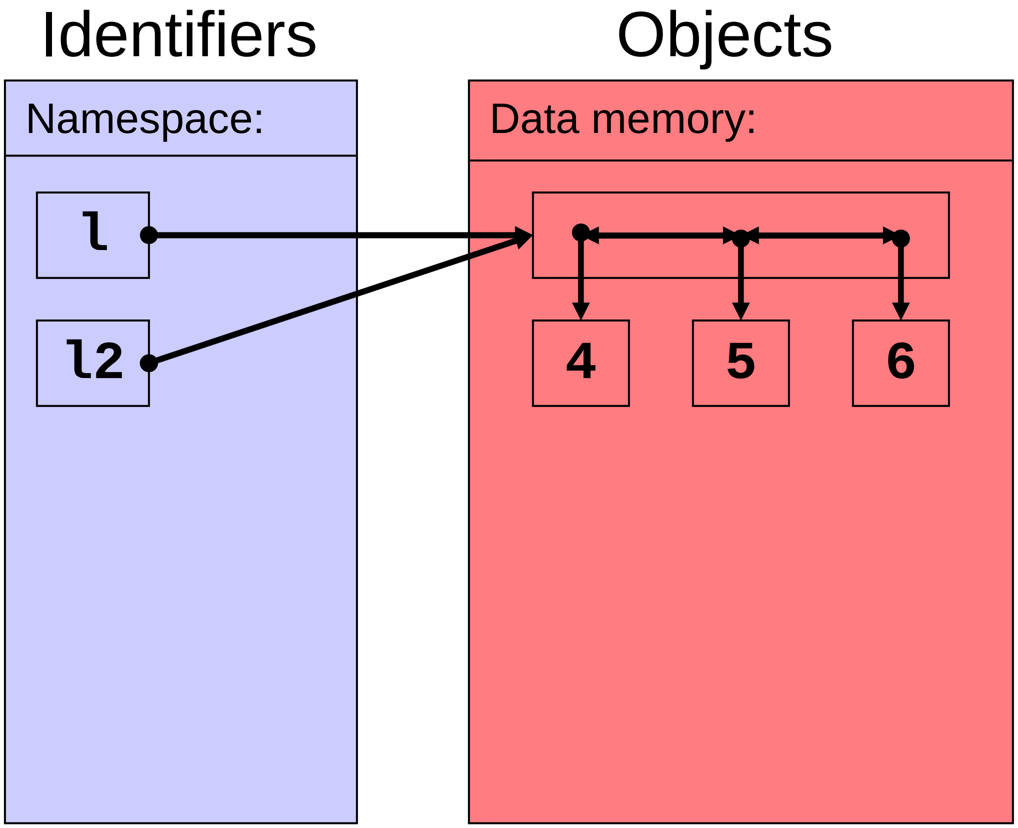


- List assignment
  - creates a new reference to the list object
  - exactly the same as standard assignment

Object References: Copying Lists

- Example 2:

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



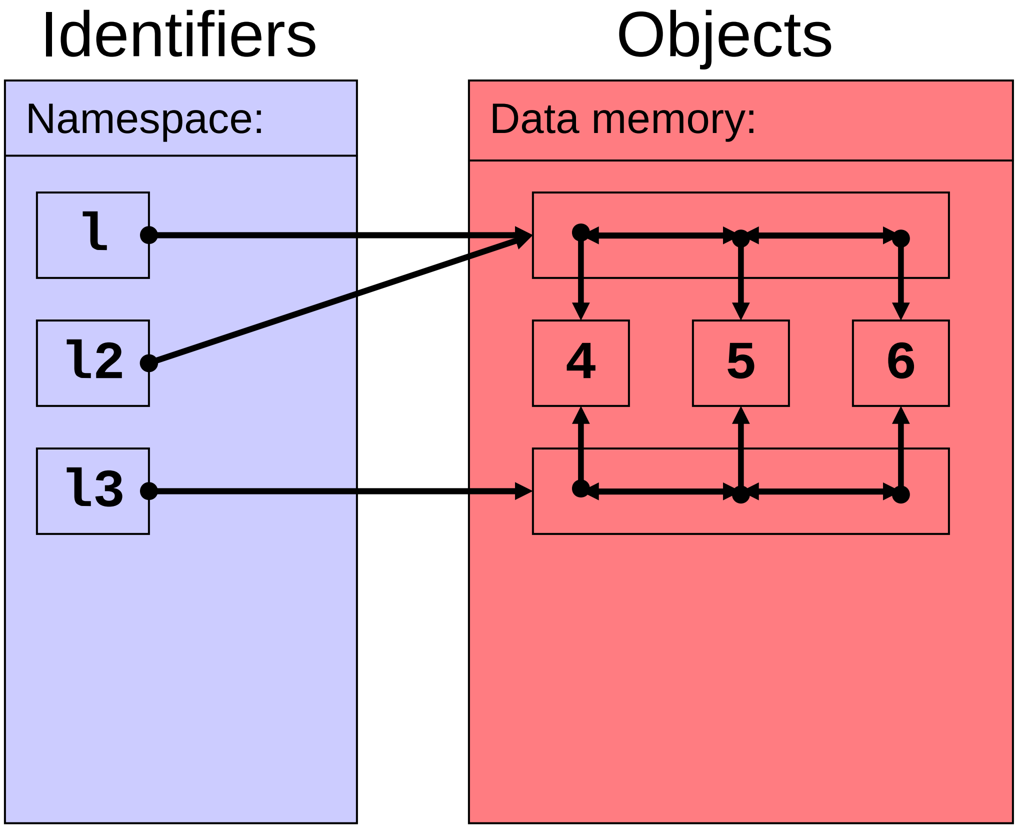
- List objects are mutable
  - appending a new list element changes the value of the list object
  - note that both **`l`** and **`l2`** are modified!

Object References: Copying Lists

- Example 2:

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

l3 = l[:]
```



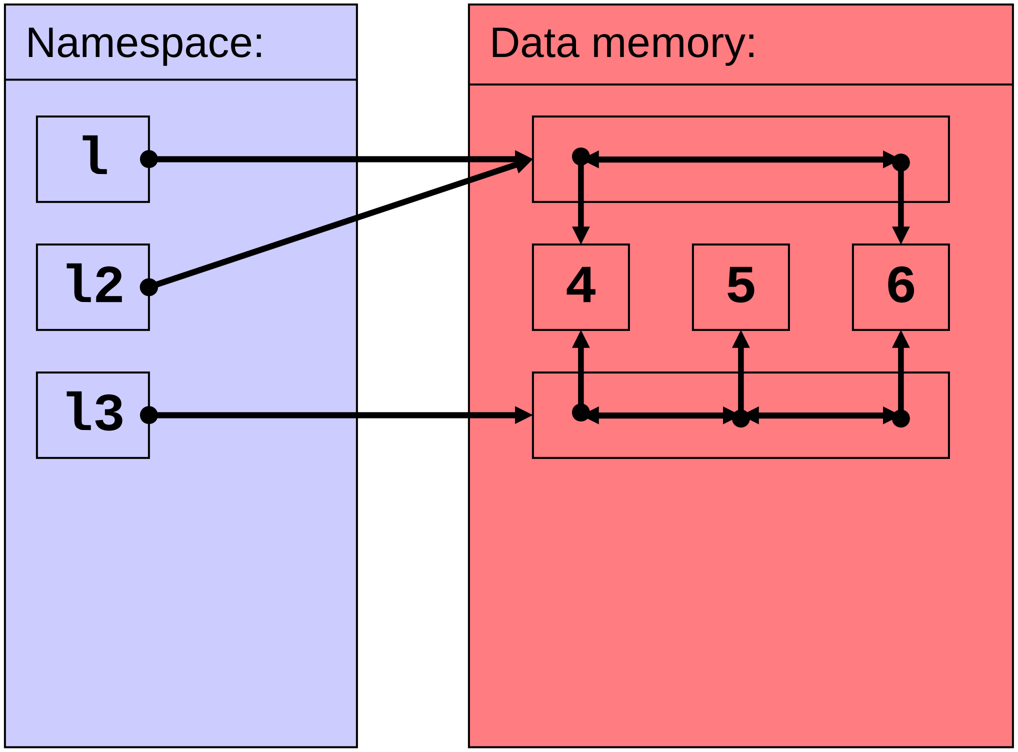
- Slicing operator creates a *shallow copy* of the list
  - list object itself is copied
  - list elements are still identical

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!

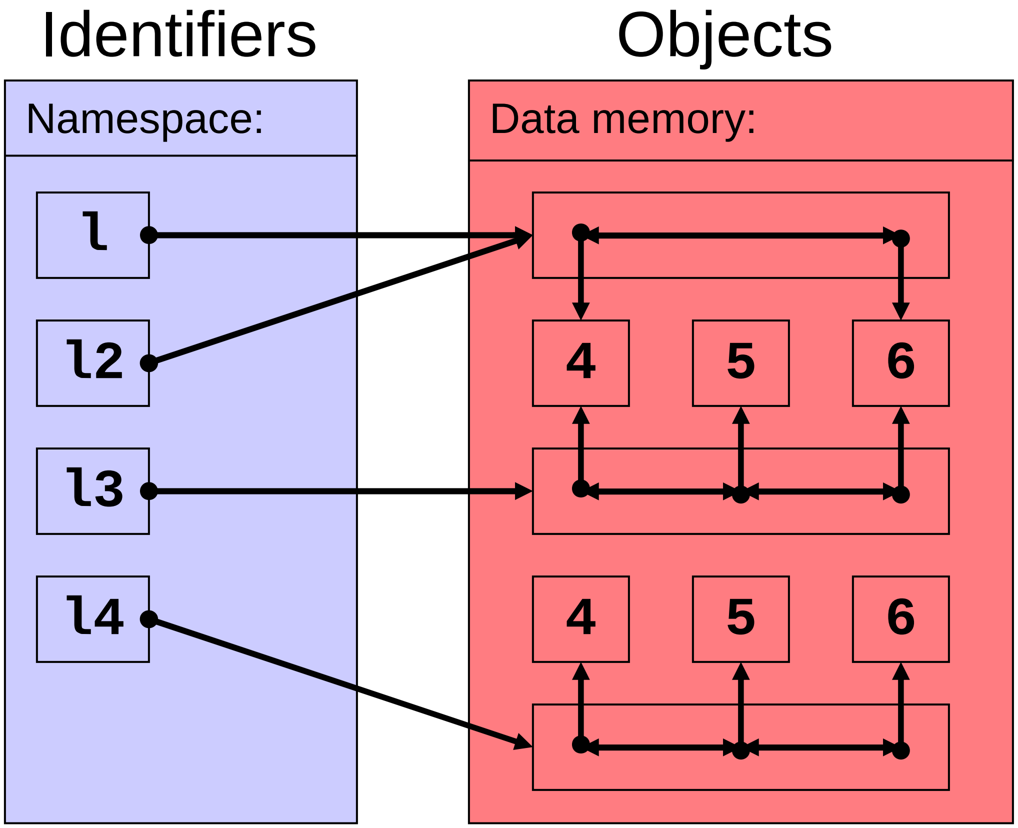
Object References: Copying Lists

- 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

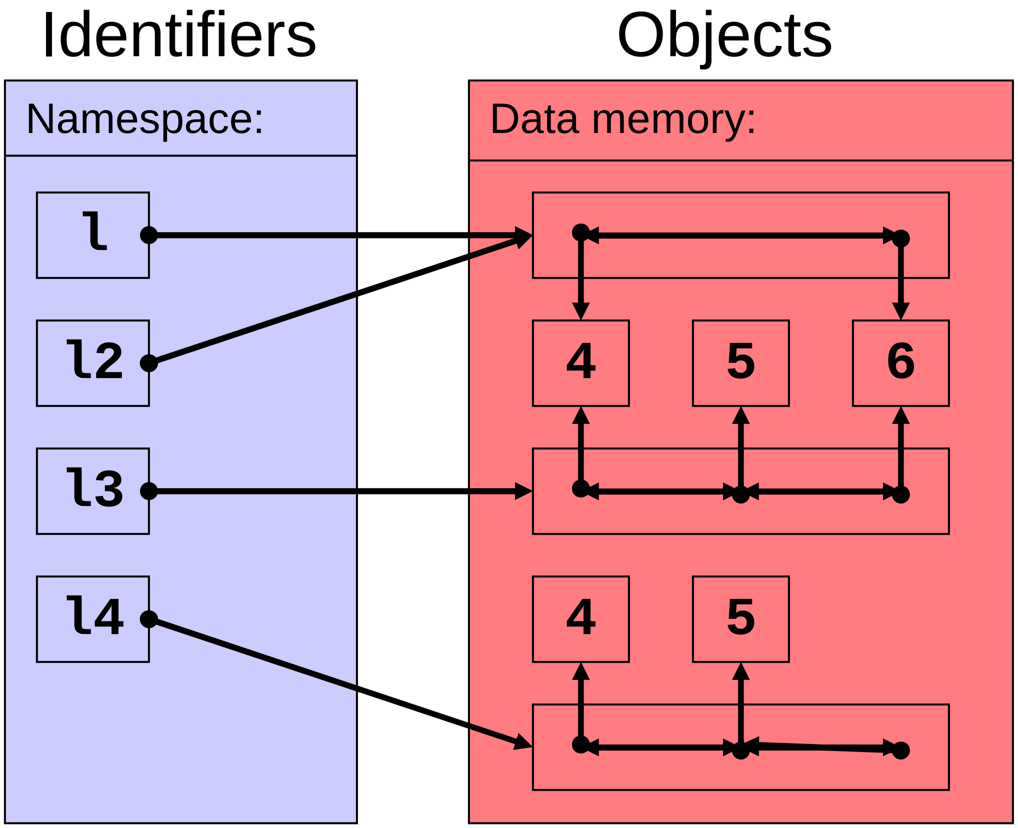
Object References: Empty Object

- 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]
```

Functional Programming

- What is Functional Programming?
  - “Function-oriented” programming
    - thinking of functions as objects
  - Using functions as objects
    - passing function objects around
    - calling function objects
    - anonymous functions
      - **`lambda`** statement (we skip this!)
  - Recursion
    - powerful programming concept
    - divide-and-conquer paradigm

Function Objects

- Interactive Example:
  - a function is an object
    - with a type
    - with a location
    - can be called
  - a function can be assigned to another identifier
  - built-in function **`apply(fct,args)`**
    - calls the function **`fct`**
    - with arguments **`args`**
    - returns return result of **`fct(args)`**

```
% python
>>> def f(a,b):
...     return 3*a + 5*b
...
>>> f(9, 3)
42
>>> type(f)
<type 'function'>
>>> print f
<function f at 0x811d704>

>>> g = f
>>> type(g)
<type 'function'>
>>> g(9, 3)
42

>>> apply(g, (9,3))
42
>>> args = (9,3)
>>> apply(g, args)
42
```

Function Objects

- Passing functions as arguments
- Example:
  Bubble Sort

Note:

Either function **CmpGreater** or **CmpSmaller** can be used in function **BubbleSort** for the comparison!

```
# bubblesort.py: Bubble Sort algorithm
#
# author: Rainer Doemer
#
# modifications:
# 02/09/04 RD    initial version

# function definitions

def CmpGreater(item1, item2):
    return item1 > item2

def CmpSmaller(item1, item2):
    return item1 < item2

def BubbleSort(list, cmp_fct=CmpGreater):
    for i in range(len(list)):
        for j in range(i+1, len(list)):
            if cmp_fct(list[i], list[j]):
                list[i],list[j] = list[j],list[i]

# initialize
items = []
...
```

Function Objects

- Example Bubble Sort, continued...

```
...
# input the sort order and a list of strings
while 1:
    s = raw_input("Sort order: (a) ascending or (b) descending? ")
    if (s == 'a'):
        comparison = CmpGreater
        break
    elif (s == 'b'):
        comparison = CmpSmaller
        break
while 1:
    s = raw_input("Enter a string or type '.' to quit: ")
    if s == '.':
        break;
    items += [s]

# compute
print "Sorting...",
BubbleSort(items, comparison)
print "Done."

# output the sorted list
print "The sorted list is:"
for item in items:
    print item
```

Functional List Operations

- Functional programming is very useful for processing lists of data
- Python provides three built-in functions
  - **`map(fct, seq)`**
  - **`filter(fct, seq)`**
  - **`reduce(fct, seq)`**
- Each function
  - takes a function **`fct`** as first argument
  - takes a sequence **`seq`** as second argument
  - applies **`fct`** to the elements of **`seq`**
  - returns the result of these operations
    - as a new list (**`map()`**, **`filter()`**)
    - as a single value (**`reduce()`**)

Functional List Operations

- **`map(fct, seq)`**
  - applies **`fct`** to each element of **`seq`**
  - returns a list of the results of these function calls
- Example:

```
% python
>>> def cube(x):
...     return x*x*x
>>> map(cube, [1,2,3,4,5,6])
[1, 8, 27, 64, 125, 216]
```

- Multiple sequences:
  - **`map(fct, seq1, seq2, ...)`**
  - **`fct`** is called with items from each sequence
  - **`fct(item1, item2, ...)`**
- Example:

```
% python
>>> def add(a, b):
...     return a + b
>>> map(add, [1,2,3,4,5], [9,8,7,6,5])
[10, 10, 10, 10, 10]
```

Functional List Operations

- **`filter(fct, seq)`**
  - applies **`fct`** to each element of **`seq`**
  - returns a list of those elements for which **`fct`** returns true
- Example:

```
% python
>>> def is_positive(x):
...     return x > 0
...
>>> def is_negative(x):
...     return x < 0
...
>>> def is_even(x):
...     return x % 2 == 0
...
>>> l = [0,-1,1,-2,2,-3,3,-4,4,-5,5]
>>> filter(is_positive, l)
[1, 2, 3, 4, 5]
>>> filter(is_negative, l)
[-1, -2, -3, -4, -5]
>>> filter(is_even, l)
[0, -2, 2, -4, 4]
```

Functional List Operations

- **`reduce(fct, seq)`**
  - reduces sequence **`seq`** to a single element
  - applies **`fct`** to “pairs” of elements of **`seq`**
    - `tmp = fct(seq[0], seq[1])`
    - `tmp = fct(tmp, seq[2])`
    - `tmp = fct(tmp, seq[3])`
    - `...`
    - `result = fct(tmp, seq[N])`
- Example:

```
% python
>>> def add(a, b):
...     return a + b
>>> def max(a, b):
...     if a > b:
...         return a
...     else:
...         return b
>>> reduce(add, [1,2,3,4,5,6,7,8,9])
45
>>> reduce(max, [3,4,2,6,9,4,3,1])
9
```

Recursion

- Recursive function
  - function that calls itself
    - directly
    - indirectly

```
def f():
    ...
    f()
    ...
```

```
def a():
    ...
    b()
    ...

def b():
    ...
    a()
    ...
```

- Concept of recursion
  - Trivial base case
    - return value defined for simple case
    - Example: **`if arg == 0: return 1`**
  - Recursion step
    - reduce the problem towards the base case
    - make a recursive function call
    - Example: **`if arg > 0: return ...fct(arg-1)...`**
- Termination of recursion
  - Converging of recursive calls to the base case
  - Recursive call must be simpler than current call

Recursion

- Example: Factorial function (*!*)
  - The factorial of a non-negative integer is
    - $n! = n * (n-1) * (n-2) * (n-3) * \ldots * 1$
  - This can be written as
    - $n! = n * ((n-1) * ((n-2) * ((n-3) * (\ldots * 1))))$
  - Recursive definition:
    - $1! = 1$
    - $n! = n * (n-1)!$
  - Example computation:

$$
\begin{aligned}
5! &= 5 * 4! \\
&= 5 * (4 * 3!) \\
&= 5 * (4 * (3 * 2!)) \\
&= 5 * (4 * (3 * (2 * 1!))) \\
&= 5 * (4 * (3 * (2 * 1))) \\
&= 5 * (4 * (3 * 2)) \\
&= 5 * (4 * 6) \\
&= 5 * 24 \\
&= 120
\end{aligned}
$$

Recursion

- Example: Factorial function (*!*)
  - Recursive implementation:
    - Base case: $n = 1 : n! = 1$
    - Recursion step: $n > 1 : n! = n * (n-1)!$

```
% python
>>> def factorial(n):
...     if n == 1:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> factorial(5)
120
>>> for i in range(1, 10):
...     print factorial(i),
...
1 2 6 24 120 720 5040 40320 362880
```

Recursion

- Example 3: Directory tree traversal
  - Tree traversal is one of the most important examples for use of recursion
    - Graph theory
    - Depth-First Search (DFS) algorithm
  - In Unix, files are organized in a hierarchy called a directory tree

files:	single file	(base case)
directories:	list of files	(recursion step)

  - Required tools

Python module **`os`**:	provides operating system functions
`os.path.isdir(name)`:	check if **`name`** is a directory (or file)
`os.listdir(name)`:	get list of files in directory **`name`**
`os.chdir(name)`:	change current directory to **`name`**

Recursion

- Example 3: Directory tree traversal
  - Recursive implementation:

```
# filetree.py: recursively list all files in a directory tree
# author: Rainer Doemer
# 02/13/04 RD   initial version

import os       # use operating system module

# function definition
def list_file(name):
    if os.path.isdir(name):     # is name a directory?
        print "Dir: ", name     # print a directory name
        files = os.listdir(name)# obtain directory entries
        os.chdir(name)          # enter the directory
        for file in files:      # handle each file in directory
            list_file(file)     # recursion!
        os.chdir("..")          # leave the directory
    else:
        print "File:", name     # base case: print a file name

# function call
list_file(".")  # start listing from the current directory
```

Object-Oriented Programming

- Introduction
  - Before: *Structured Programming*
    - Literals, identifiers, types, expressions
    - Statements, control flow, functions
    - Procedural programming, *action-oriented*
  - Now: *Object-Oriented Programming (OOP)*
    - Classes
    - Objects
- Background
  - The real world is composed of objects
    - people, animals, plants, cars, planes, buildings, ...
  - An object can be seen as an abstraction of its components
    - we see objects on a screen (not a bunch of pixels)
    - we see a beach (rather than grains of sand)
    - we see a forest (rather than trees)
    - we see buildings (rather than bricks)
  - A class is like a blue-print for an object

Object-Oriented Programming

- Concepts and Terminology
  - Object
    - Abstraction, model of real-world object
    - Has *attributes*
      - name, size, color, weight, ...
    - Exhibits *behavior*
      - people sleep, eat, walk, talk, ...
    - Uses *communication*
      - message passing
  - Class relationship
    - Classes of objects have the same characteristics
      - Class automobile contains
        - » sports car, limousine, pick-up, truck, ...
    - *Inheritance* (multiple inheritance)
      - A convertible is a sports car with a removable roof
      - A convertible is also an automobile
    - Classes of objects are derived from existing classes and add characteristics of their own

Object-Oriented Programming

- Key concepts
  - Hierarchy
  - Encapsulation
    - Attributes: data members
    - Behavior: function members, methods
    - Interfaces: communication attributes and methods
  - Information hiding
  - Reuse
- Terminology
  - Object
    - Instance of a class
    - Instantiation: creation of an object of a class
    - Destruction: deletion of an object
  - Class:
    - Abstract data type (ADT)
      - aka. user-defined type
    - Constructor: creation of objects
    - Destructor: deletion of objects

Object-Oriented Programming

- Example: **`class Time`**
  - Program **`time1.py`** (part 1/2)

```
# time1.py: abstract data type for representation of time
#           (version 1)
# author: Rainer Doemer
# 02/17/04 RD    initial version (similar to figure 7.1)

# class definition
class Time:
    """abstract data type for representation of time"""

    def __init__(self):           # constructor
        """creates a time object initialized to 12am"""
        self.hour   = 0  # 0-23   # data members
        self.minute = 0  # 0-59
        self.second = 0  # 0-59

    def Print(self):              # method
        """prints the value of a time object"""
        print "%02d:%02d:%02d" %  \
                (self.hour,self.minute,self.second)
...
```

Object-Oriented Programming

- Example: **`class Time`**
  - Program **`time1.py`** (part 2/2)

```
...
    def PrintAMPM(self):  # method
        """prints the time in am/pm notation"""
        h = self.hour % 12
        if h == 0:
            h = 12
        if self.hour < 12:
            ampm = "am"
        else:
            ampm = "pm"
        print "%2d:%02d:%02d %s" %\
                (h,self.minute,self.second,ampm)
```

Object-Oriented Programming

- Example: **`class Time`**
  - Notes (1):
    - Class definition consists of
      - Class header (keyword **`class`**, identifier **`Time`**, colon)
      - Class body (indented block of attributes and methods)
        - » contains methods **`__init__`**, **`Print`**, and **`PrintAMPM`**
    - Documentation strings
      - Triple-quoted strings (by convention)
      - Inserted between header and body
      - Optional for modules, functions, classes, methods
      - Available in attribute **`__doc__`** for inspection
    - Class constructor **`__init__`**
      - Special method for object initialization
        - » creates and initializes attributes **`hour`**, **`minute`**, and **`second`**
      - Called implicitly whenever an object of the class is created
      - Must not return any value (**`None`**)

Object-Oriented Programming

- Example: **`class Time`**
  - Notes (2):
    - Object reference **`self`**
      - Aka. *object reference argument* or *class instance object*
      - Called **`self`** by convention
      - First (explicit!) argument of every class method
      - Implicitly supplied when a method of an object is called
      - in C++, **`self`** is called **`this`**
    - Class methods
      - functions that operate on an object
        - » **`Print`**, **`PrintAMPM`**
      - require first argument **`self`** which represents the object
      - **`self`** is used to access the attributes
        - » **`self.hour`**
        - » **`self.minute`**
        - » **`self.second`**

Object-Oriented Programming

- Example: **`class Time`**
  - Notes (3):
    - Class namespace
      - Every class has its own namespace
      - Contains class attributes and class methods (which are shared among all instances of the class)
      - Access by use of dot-operator
        - » from inside the class: through object reference **`self`**
        - » from outside the class: through class name
    - Object namespace
      - Every object has its own namespace
      - Contains object attributes and object methods
      - Is typically populated by the constructor
      - Access by use of dot-operator
        - » from inside the class: through object reference **`self`**
        - » from outside the class: through object name

Access to Object Attributes

- Direct access
  - Dot-operator (`.`)
    - from inside the class: through object reference **`self`**
      - Example:
        - » Read access: **`print self.hour`**
        - » Write access: **`self.hour = 15`**
    - from outside the class: through object name
      - Example:
        - » Read access: **`print t1.hour`**
        - » Write access: **`t1.hour = 15`**
  - Direct access from outside the class
    - violates concept of information hiding!
    - can lead to an inconsistent state of an object!
      - Example:
        - » **`t1.minute = 88 # value out of range!`**

Access to Object Attributes

- Access control
  - – Object interfaces: Get and Set methods
    - Access object data under program control
      - – Get: method to obtain data from an object
        - » Read access: **`print t1.GetHour()`**
      - – Set: method to set data in an object
        - » Write access: **`t1.SetHour(15)`**
  - – Invalid accesses can be prevented
    - Internal information is hidden!
    - Ensures consistent state of the object!
      - – Example:
        - »

```
def SetMinute(self, minute):
    if 0 <= minute <= 59:
        self.minute = minute
```

Access Control to Object Attributes

- Example: **`class Time`**
  - Program **`time2.py`** (part 1/4)

```
# time2.py: abstract data type for representation of time
#           (version 2)
# author: Rainer Doemer
# 02/19/04 RD    added access control methods
# 02/17/04 RD    initial version (similar to figure 7.1)

# class definition
class Time:
    """abstract data type for representation of time"""

    def __init__(self, hour=0, minute=0, second=0):
        """creates a time object and initializes it"""
        self.SetTime(hour, minute, second)

    def SetTime(self, hour=0, minute=0, second=0):
        """sets the time of a time object"""
        self.SetHour(hour)
        self.SetMinute(minute)
        self.SetSecond(second)
...
```

Access Control to Object Attributes

- Example: **`class Time`**
  - Program **`time2.py`** (part 2/4)

```
...
    def SetHour(self, hour=0):
        """sets the hour of a time object"""
        if (0 <= hour <= 23):
            self.__hour = hour
        else:
            raise ValueError, "Hour value out of range 0-23"

    def SetMinute(self, minute=0):
        """sets the minute of a time object"""
        if (0 <= minute <= 59):
            self.__minute = minute
        else:
            raise ValueError, "Minute value out of range 0-59"

    def SetSecond(self, second=0):
        """sets the second of a time object"""
        if (0 <= second <= 59):
            self.__second = second
        else:
            raise ValueError, "Second value out of range 0-59“
...
```

Access Control to Object Attributes

- Example: **`class Time`**
  - Program **`time2.py`** (part 3/4)

```
...
    def GetTime(self):
        """returns the time in a tuple of (h,m,s)"""
        return (self.__hour,self.__minute,self.__second)

    def GetHour(self):
        """returns the hour of the time object"""
        return self.__hour

    def GetMinute(self):
        """returns the minute of the time object"""
        return self.__minute

    def GetSecond(self):
        """returns the second of the time object"""
        return self.__second

    def GetAMPM(self):
        """returns 'am' or 'pm' in a string"""
        if self.__hour < 12:
            return "am"
        else:
            return "pm“
...
```

Access Control to Object Attributes

- Example: **`class Time`**
  - Program **`time2.py`** (part 4/4)

```
...
    def Print(self):
        """prints the value of a time object"""
        print "%02d:%02d:%02d" %   \
                (self.__hour,self.__minute,self.__second)

    def PrintAMPM(self):
        """prints the time in am/pm notation"""
        h = self.__hour % 12
        if h == 0:
            h = 12
        print "%2d:%02d:%02d %s" %\
                (h,self.__minute,self.__second,self.GetAMPM())
```

Access Control to Object Attributes

- Example: **`class Time`**
  - Notes for program **`time2.py`** (1):
    - Constructor **`__init__`** takes arguments for initialization
      - Initial time can be specified at object creation
      - Default arguments are provided for convenience
    - Method **`SetTime`**
      - allows to (re-) set the time of an existing object
      - calls individual methods for data members
    - Methods **`SetHour`**, **`SetMinute`** and **`SetSecond`**
      - if argument is valid, set the internal (!) data member
      - if argument is invalid, raise an exception
    - Internal data members **`__hour`**, **`__minute`**, **`__second`**
      - marked as *private* by 2 leading underscores (__)
      - should not be accessed from outside the class
      - are subject to *name mangling* (will be renamed)

Access Control to Object Attributes

- Example: **`class Time`**
  - Notes for program **`time2.py`** (2):
    - Raising exceptions
      - Keyword **`raise`** raises the specified exception with an argument explaining the problem
      - Unless handled by an exception handler, an exception will terminate the program execution
      - More details on exceptions follow later!
    - Method **`GetTime`**
      - returns the time values in a 3-tuple (h/m/s)
    - Methods **`GetHour`**, **`GetMinute`** and **`GetSecond`**
      - return the requested time value
    - Method **`GetAMPM`**
      - returns an AM/PM indicator
    - Methods **`Print`** and **`PrintAMPM`**
      - as before, but adjusted to use modified methods