ECE12: Introduction to Programming
Lecture 8

Rainer Dömer

doemer@uci.edu

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

Lecture 8: Overview

- Random number generation
  - Module **`random`**
  - Function **`randrange()`**
  - Example: **`dice.py`**
- Function Arguments
  - Default arguments
  - Keyword arguments
- Namespaces and Scope
  - Local, global and built-in namespaces
  - Local and global scope
  - Shadowing
  - Program introspection
  - Module namespaces

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),
```

Example: Roll the Dice

- Program **`dice.py`**, part 1

```
# dice.py: roll the dice
#
# author: Rainer Doemer
#
# modifications:
# 01/31/04 RD    initial version, based on fig04_07.py

# import modules
import random

# function definition
def roll_die():
    return random.randrange(1, 7)

# initialization
num1 = 0
num2 = 0
num3 = 0
num4 = 0
num5 = 0
num6 = 0
...
```

Example: Roll the Dice

- Program **`dice.py`**, part 2

```
...

# compute
for i in range(6000):
    face = roll_die()
    if   face == 1: num1 += 1
    elif face == 2: num2 += 1
    elif face == 3: num3 += 1
    elif face == 4: num4 += 1
    elif face == 5: num5 += 1
    elif face == 6: num6 += 1

# output
print "Rolled %4d times a 1." % num1
print "Rolled %4d times a 2." % num2
print "Rolled %4d times a 3." % num3
print "Rolled %4d times a 4." % num4
print "Rolled %4d times a 5." % num5
print "Rolled %4d times a 6." % num6
```

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)
```

Example: Roll the Dice

- Modified example:

```
...
# function definition
def roll_die(sides = 6, verbose = 0):
    face = random.randrange(1, 1+sides)
    if verbose:
        print "Rolled a %d" % face
    return face
...

# default call (same as before)
face = roll_die()
...

# use die with 10 faces
face = roll_die(10)
...

# print every roll (be verbose)
face = roll_die(verbose = 1)
...
```

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

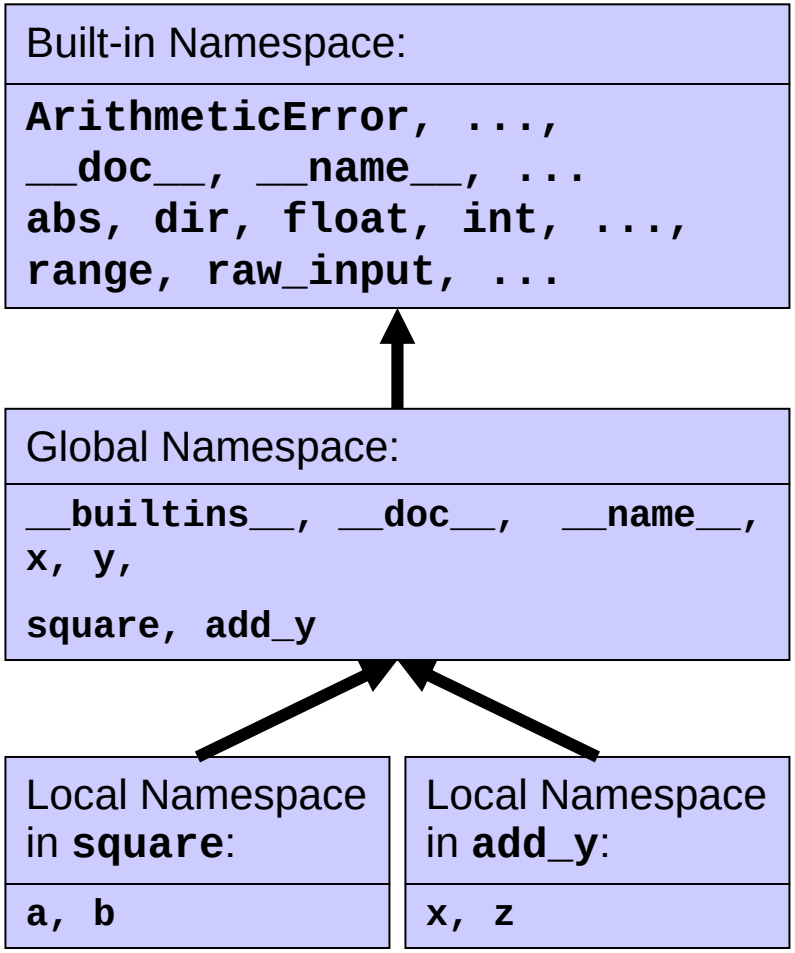
- Example:

```
# 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
```



Namespaces and Scope

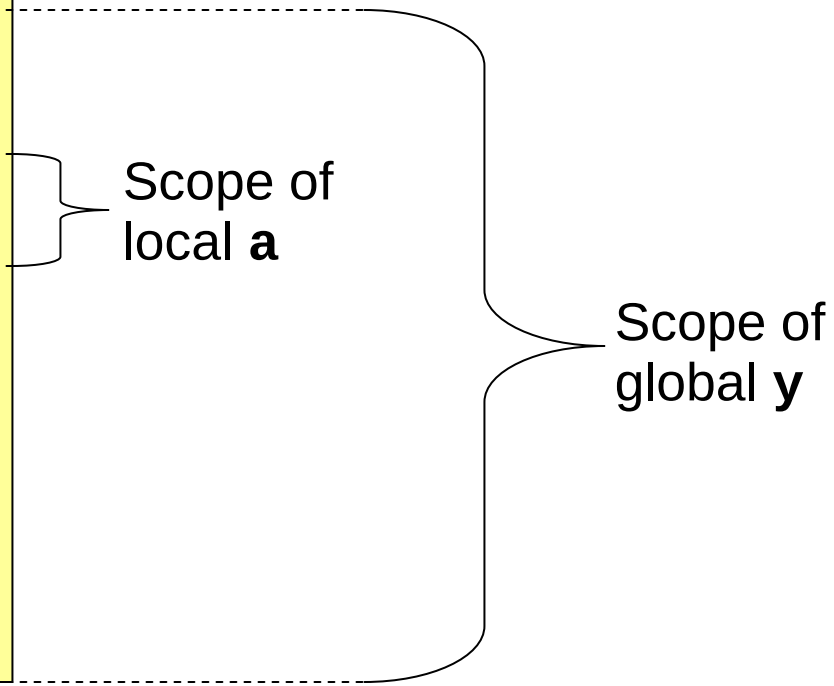
- Example:

```
# 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
```



Namespaces and Scope

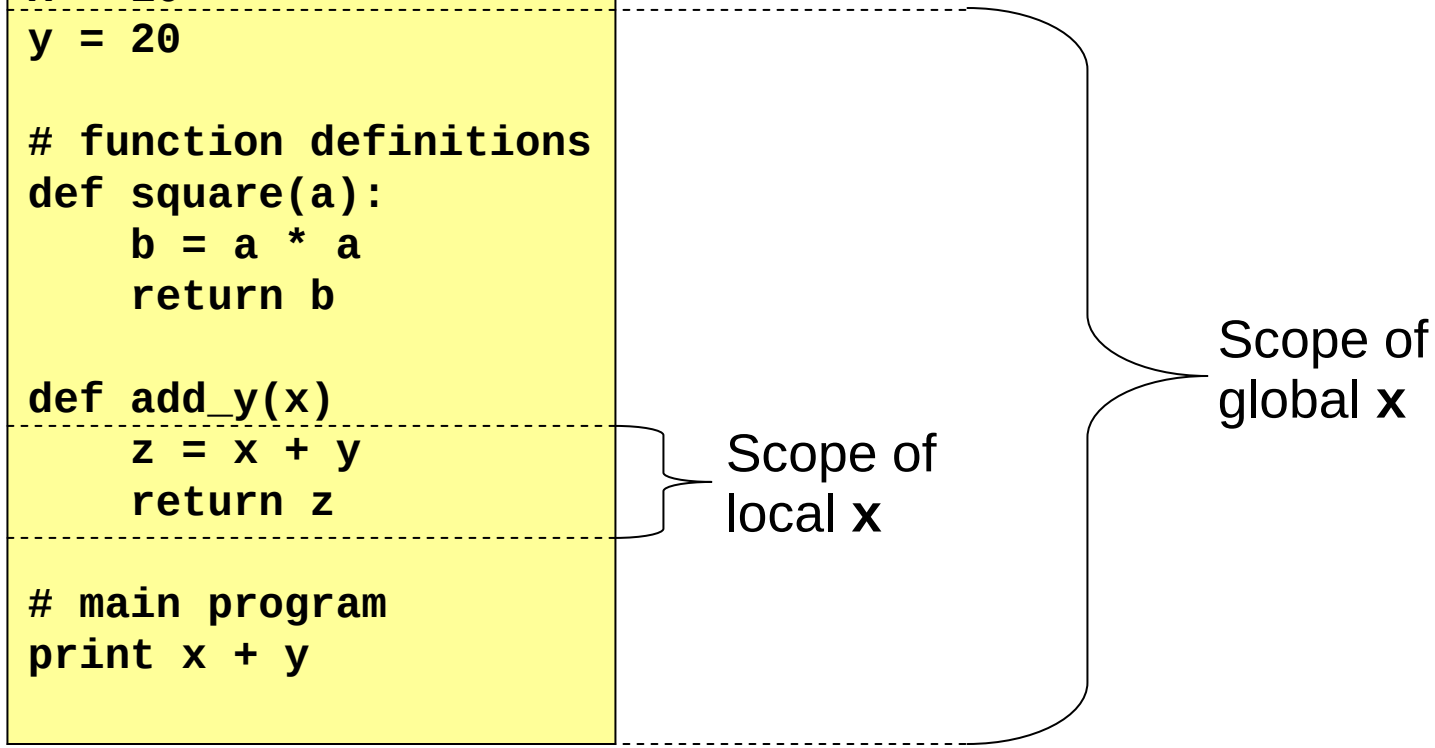
- Example:

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

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

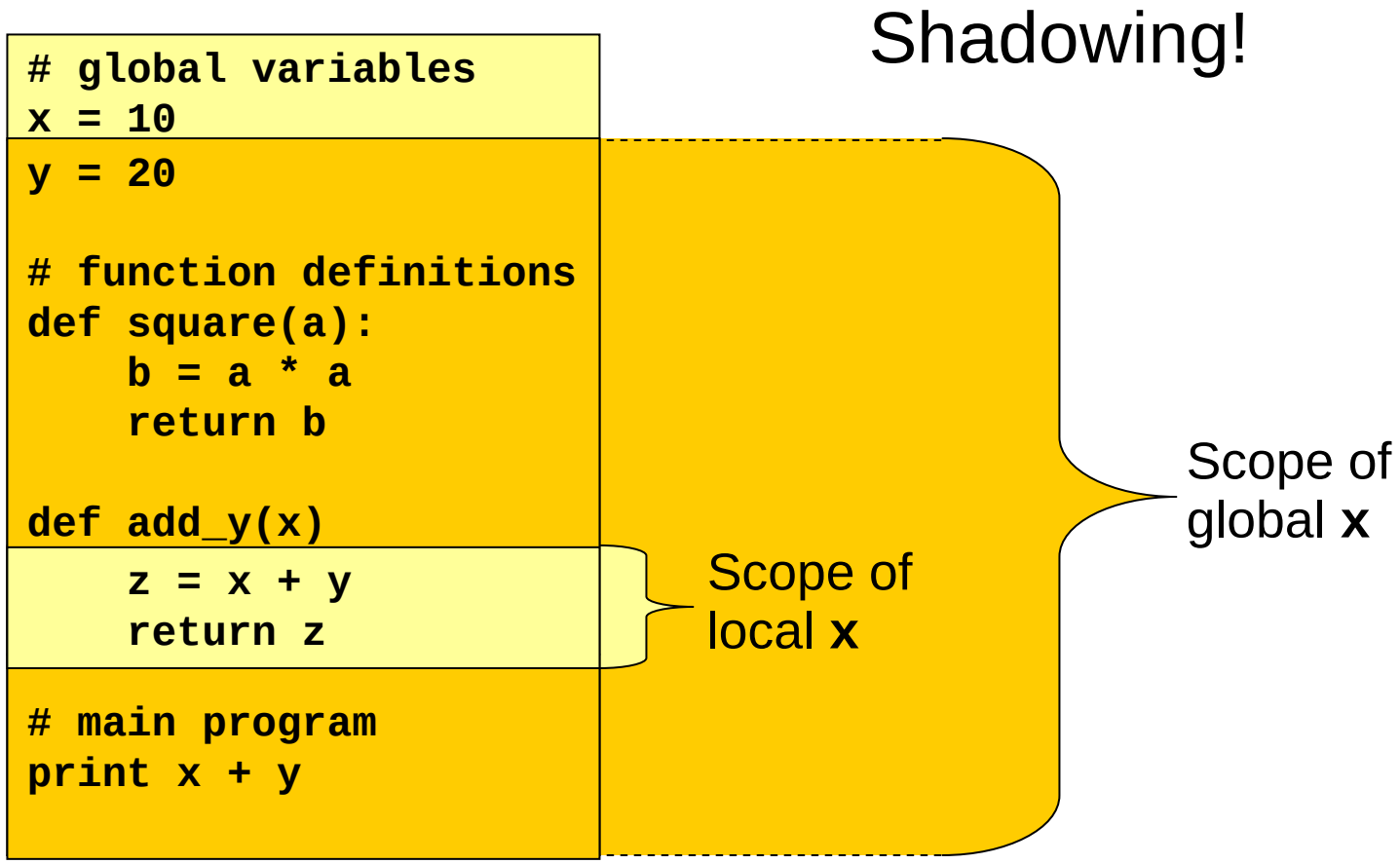
def add_y(x)
    z = x + y
    return z

# main program
print x + y
```



Namespaces and Scope

- Example:



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', ...]`**

Program Introspection

- Interactive Example:

```
% python
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> type(__name__)
<type 'str'>
>>> print __name__
__main__
>>> type(__doc__)
<type 'NoneType'>
>>> type(__builtins__)
<type 'module'>
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', ...
'__doc__', '__name__', ...
'abs', ..., 'dir', ..., 'range', 'raw_input', ...]
>>> print __builtins__.__name__
__builtin__
>>> print __builtins__.__doc__
Built-in functions, exceptions, and other objects.
>>> type(__builtins__.range)
<type 'builtin_function_or_method'>
```

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)
      ```
      ```

Module Namespaces

- Interactive Examples:

```
% python
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math‘]
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos',
'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh',
'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf',
'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh']
```

```
% python
>>> from math import sqrt, cos
>>> dir()
['__builtins__', '__doc__', '__name__', 'cos',
'sqrt']
>>> from math import pi as circle_constant
>>> dir(math)
['__builtins__', '__file__', '__name__', 'cos',
'sqrt', 'circle_constant']
```