ECE12: Introduction to Programming
Lecture 9

Rainer Dömer

doemer@uci.edu

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

Lecture 9: Overview

- Namespaces and Scope
  - Program introspection
  - Module namespaces
- Data Structures
  - Introduction
  - Python Sequences

Namespaces and Scope

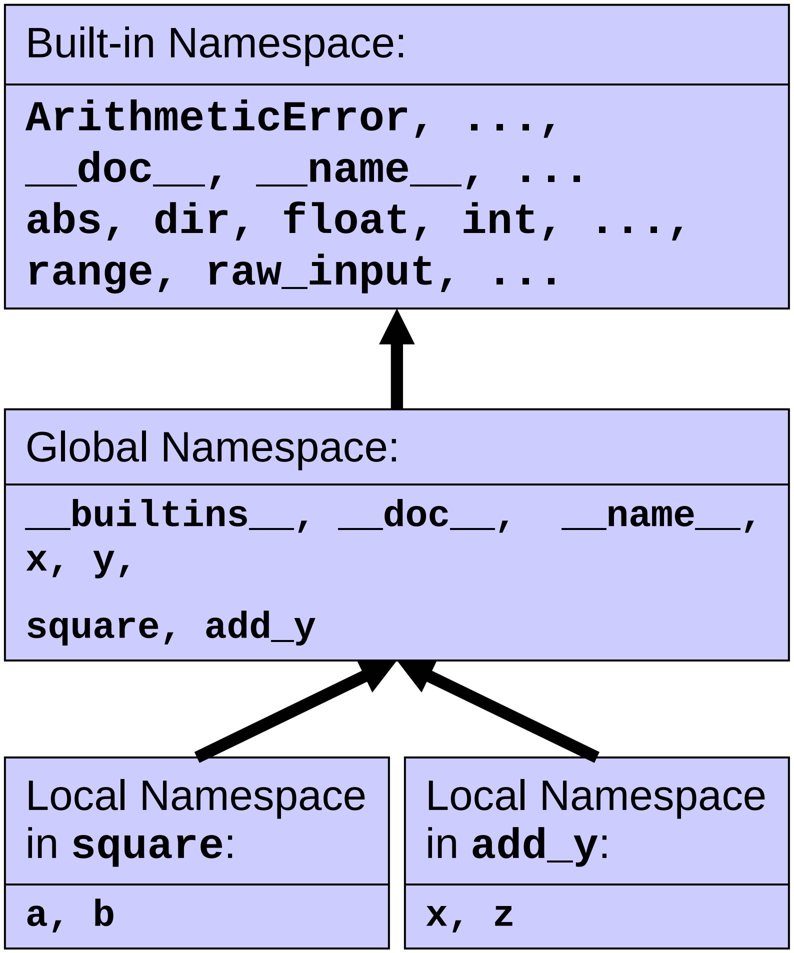
- Example:

```
# 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

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']
```

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
```