

Organization of Programming Languages

CS320/520N

Lecture 12

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Scripting Languages: Python

- Designed by Guido van Rossum in the early 1990s.
 - Current development done by the Python Software Foundation.
 - Python facilitates multiple programming paradigms:
 - imperative programming.
 - object oriented programming.
 - functional programming.
- ⇒ **multi-paradigm programming language.**

Python: Important Features

- Python = **object-oriented** **interpreted** “**scripting**” language:
 - **Object oriented:**
 - modules, classes, exceptions.
 - dynamically typed, automatic garbage collection.
 - **Interpreted**, interactive:
 - rapid edit-test-debug cycle.
 - **Extensible:**
 - can add new functionality by writing modules in C/C++.
 - **Standard library:**
 - extensive, with hundreds of modules for various services such as regular expressions, TCP/IP sessions, etc.

Scripting Languages

- **Scripts vs. Programs:**
 - interpreted vs. compiled
 - one script = a program
 - many {*.c, *.h} files = a program
- **Higher-level “glue” language:**
 - glue together larger program/library components, potentially written in different programming languages.
 - orchestrate larger-grained computations.
 - vs. programming fine-grained computations.

The Python Interpreter

Running the interpreter

```
[razvan@texas ~]$ python
```

```
Python 2.5.1 (r251:54863, Mar 7 2008, 03:39:23)
```

```
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> □
```

The Python Interpreter: Help()

```
>>> help()
```

Welcome to Python 2.5! This is the online help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://www.python.org/doc/tut/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

```
help>
```

The Python Interpreter: Keywords

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

The Python Interpreter: Keywords

help> lambda

5.11 Lambdas

```
lambda_form ::= "lambda" [parameter_list[1]]: expression[2]
old_lambda_form ::= "lambda" [parameter_list[3]]: old_expression[4]
```

Download entire grammar as text.[5]

Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `lambda arguments: expression` yields a function object.

The unnamed object behaves like a function object defined with

```
def name(arguments):
    return expression
```

See section 7.6[6] for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

The Python Interpreter: Modules

help> modules

commands:

execute shell commands via `os.popen()` and return status, output.

compiler:

package for parsing and compiling Python source code.

gzip:

functions that read and write gzipped files.

HTMLParser:

a parser for HTML and XHTML (defines a class `HTMLParser`).

math:

access to the mathematical functions defined by the C standard.

exceptions:

Python's standard exception class hierarchy.

The Python Interpreter: Modules

help> modules

os:

OS routines for Mac, NT, or Posix depending on what system we're on.

re:

support for regular expressions (RE).

string:

a collection of string operations (most are no longer used).

sys:

access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter:

sys.argv: command line arguments.

sys.stdin, *sys.stdout*, *sys.stderr*: standard input, output, error file objects.

threading:

thread modules emulating a subset of Java's threading model.

The Python Interpreter: Integer Precision

```
>>> def fib(n):
...     a, b, i = 0, 1, 0
...     while i < n:
...         a,b,i = b, a+b, i+1
...     return a
..
>>> fib(10)
55
>>> fib(100)
354224848179261915075L
>>> fib(1000)
4346655768693745643568852767504062580256466051737178040248172908953655541
4905189040387984007925516929592259308032263477520968962323987332247116164
996440906533187938298969649928516003704476137795166849228875L
```

Built-in Types: Basic

- **plain integers**, implemented using long in C – int().
 - decimal, octal and hex literals.
- **long integers**, with unlimited precision – long().
 - append “L”.
- **floating point numbers**, implemented using double in C – float().
- **complex numbers**, real and imaginary as double in C – complex().
 - 10+5j, 1j
- **boolean values**, True and False – bool().
 - False is: None, False, 0, 0L, 0.0, 0j, “”, (), [], {},
 - user defined class defines methods nonzero() or len().
- **strings – str(), type, function, ...**
 - “Hello world”, ‘Hello world’

Built-in Types: Composite

- **lists:**
 - [], [1, 1, 2, 3], [1, “hello”, 2+5j]
- **tuples:**
 - (), (1,), (1, 1, 2, 3), (1, “hello, 2+5j)
- **dictionaries:**
 - {“john”: 12, “elaine”: 24}
- **files**

Booleans

```
>>> bool
<type 'bool'>
>>> [bool(0), bool(0.0), bool(0j),bool([]), bool(()), bool({}), bool(''), bool(None)]
[False, False, False, False, False, False, False, False]
>>> [bool(1), bool(1.0), bool(1j), bool([1]), bool((1,)), bool({'1':'one'})], bool('1')]
[True, True, True, True, True, True, True]
>>> str(True), repr(True)
('True', 'True')
>>> True & True, True & False, False & False
(True, False, False)
>>> True | True, True | False, False | False
(True, True, False)
>>> True ^ True, True ^ False, False ^ True
(False, True, False)
```

Floating Point

```
>>> float
<type 'float'>
>>> 3.14
3.1400000000000001
>>> str(3.14), repr('3.14')
(3.14, 3.1400000000000001)
>>> 3.14/2, 3.14//2
(1.5700000000000001, 1.0)
>>> 1.9999999999999999
2.0
>>> import math
>>> math.pi, math.e
(3.1415926535897931, 2.7182818284590451)
>>> help('math')
```

```
===== Python =====
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
===== C++ =====
float sum = 0.0;
for (int i = 0; i < 10; i++)
    sum += 0.1;
cout.precision(17);
cout << sum << endl;
⇒ 1.0000001192092896
```

<http://docs.python.org/tut/node16.html>

Strings

- Immutable Sequences of Characters:

```
>>> str
<type 'str'>
>>> str.upper('it')
'IT'
>>> ", "
(' ','')
>>> "functional % languages" % 'programming'
'functional programming languages'
>>> "object oriented " + "programming"
'object oriented programming'
>>> 'orient' in 'object oriented', 'orient' in 'Object Oriented'
(True, False)
>>> s = "object oriented"
```

```
>>> len(s), s.find('ct'), s.split()
(15, 4, ['object', 'oriented'])
>>> s[0], s[1], s[4:6], s[7:]
('o', 'b', 'ct', 'oriented')
>>> s[7:100]
'oriented'
>>> help('str')
```

Lists

```
>>> [] #empty list
```

```
[]
```

```
>>> x = [3.14, "hello", True]
```

```
[3.1400000000000001, 'hello', True]
```

```
>>> x + [10, [], len(x)]
```

```
[3.1400000000000001, 'hello', True, 10, [], 3]
```

```
>>> x.append(0.0)
```

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

```
>>> x
```

```
[0.0, True, 'hello', 3.1400000000000001]
```

```
>>> x * 2
```

```
[0.0, True, 'hello', 3.1400000000000001, 0.0, True, 'hello', 3.1400000000000001]
```

```
>>> x.sort()
```

```
>>> x
```

```
[0.0, True, 3.1400000000000001, 'hello']
```

```
>>> help('list')
```

Tuples

```
>>> tuple
<type 'tuple'>
>>> () # empty tuple
()
>>> (10) # not a one element tuple!
10
>>> (10,) # a one element tuple
(10,)
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> (1, 2) * 2
(1, 2, 1, 2)
>>> x = (0, 1, 'x', 'y')
```

```
>>> x[0], x[1], x[: -1]
(0, 1, (0, 1, 'x'))
>>> y = (13, 20, -13, -5, 0)
>>> temp = list(y)
>>> temp.sort()
>>> y = tuple(temp)
>>> y
(-13, -5, 0, 13, 20)
>>> help('tuple')
```

Files

```
>>> file
<type 'file'>
>>> output = open('/tmp/output.txt', 'w') # open tmp file for writing
>>> input = open('/etc/passwd', 'r') # open Unix passwd file for reading
>>> s = input.read() # read entire file into string s
>>> line = input.readline() # read next line
>>> lines = input.readlines() # read entire file into list of line strings
>>> output.write(s) # write string S into output file
>>> output.write(lines) # write all lines in 'lines'
>>> output.close()
>>> input.close()

>>> help('file')
```

Statements & Functions

- Assignment Statements
- Compound Statements
- Control Flow:
 - Conditionals
 - Loops
- Functions:
 - Defining Functions
 - Lambda Expressions
 - Documentation Strings
 - Generator Functions

Assignment Forms

- Basic form:
 - $x = 1$
 - $y = \text{'John'}$
- Tuple positional assignment:
 - $x, y = 1, \text{'John'}$
 - $x == 1, b == \text{'John'} \Rightarrow (\text{True}, \text{True})$
- List positional assignment:
 - $[x, y] = [1, \text{'John'}]$
- Multiple assignment:
 - $x = y = 10$

Compound Statements

- Python does not use block markers (e.g. ‘begin .. end’ or ‘{ ... }’) to denote a compound statements.
 - Need to indent statements at the same level in order to place them in the same block.
 - Can be annoying, until you get used to it; intelligent editors can help.
 - Example:

```
if n == 0:  
    return 1  
else:  
    return n * fact(n - 1)
```

Conditional Statements

```
if <bool_expr_1>:  
    <block_1>  
elif <bool_expr_2>:  
    <block_2>  
...  
else:  
    <block_n>
```

- There can be zero or more elif branches.
- The else branch is optional.
- “Elif “ is short for “else if” and helps in reducing indentation.

Conditional Statements: Example

```
name = 'John'
...
if name == 'Mary':
    sex = 'female'
elif name == 'John':
    sex = 'male'
elif name == 'Alex':
    sex = 'unisex'
elif name == 'Paris':
    sex = 'unisex'
else
    sex = 'unknown'
```

Conditional Statements

- There is no C-like ‘switch’ statement in Python.
- Same behavior can be achieved using:
 - if ... elif ... elif sequences.

- dictionaries:

```
name = 'John'
dict = {'Mary': 'female', 'John': 'male',
        'Alex': 'unisex', 'Paris': 'unisex'}
if name in dict:
    print dict[name]
else:
    print 'unknown'
```

While Loops

```
x = 'university'  
while x:  
    print x,  
    x = x[1:]
```

```
a, b = 1, 1  
while b <= 23:  
    print a,  
    a, b = b, a + b
```

For Loops

```
sum = 0
```

```
for x in [1, 2, 3, 4]
```

```
    sum += x
```

```
sum
```

```
D = {1:'a', 2:'b', 3:'c', 4:'d'}
```

```
for x, y in D.iteritems():
```

```
    print x, y
```

Functions

```
def mul(x,y):  
    return x * y
```

`mul(2, 5) => ?`

`mul(math.pi, 2.0) => ?`

`mul([1, 2, 3], 2) => ?`

`mul(('a', 'b'), 3) => ?`

`mul('ou', 5) => ?`

Parameter Correspondence

```
def f(a, b, c): print a, b, c
```

```
f(1, 2, 3) => 1 2 3
```

```
f(b=4, c= 8, a = 2) => 2 4 8
```

```
def f(*args): print args
```

```
f("one argument") => ('one argument')
```

```
f(1, 2, 3) => (1, 2, 3)
```

```
def f(**args): print args
```

```
f(a=2, b=4, c=8) => {'a':2, 'b':4, 'c':8}
```

Lambda Expressions

- Scheme:

```
>(define (make-adder (num)
  (lambda (x)
    (+ x num))))
```

- Python:

```
>>> def make_adder(num):
...     return lambda x: x + num
...
>>> f = make_incrementor(10)
>>> f(9)
19
```

Lambda Expressions

```
>>> formula = lambda x, y: x *x + x*y + y*y
>>> formula
<function <lambda> at 0x2b3f213ac230>
>>> apply(formula, (2,3))
19
>>> map(lambda x: 2*x, [1, 2, 3])
[2, 4, 6]
>>> filter(lambda x: x>0, [1, -1, 2, -2, 3, 4, -3, -4])
[1, 2, 3, 4]
>>> reduce(lambda x,y:x*y, [1, 2, 3, 4, 5])
120
>>> def fact(n): return reduce (lambda x, y: x*y, range(1, n+1))
...
>>> fact(5)
120
```

Generator Functions

```
def fib(): # generate Fibonacci series
    a, b = 0, 1
    while 1:
        yield b
        a, b = b, a+b
```

```
>>> it = fib()
>>> it.next() => 1
>>> it.next() => 1
>>> it.next() => 2
```

Errors & Exceptions

- Syntax errors:

```
while True print `True`  
File "<stdin>", line 1  
    while True print `True`  
                        ^
```

SyntaxError: invalid syntax

- Exceptions: errors detected during execution:

```
1 / 0
```

```
Traceback ( most recent call last):
```

```
File "<stdin>", line 1, in ?
```

```
ZeroDivisionError: integer division or modulo  
by zero
```

Handling Exceptions

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()),
            'lines'
        f.close()
```

Modules

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.
- A **module** is a file containing Python definitions and statements.
- The file name is the module name with the suffix **.py** appended.
- Within a module, the module's name (as a string) is available as the value of the global variable **__name__**

fibonacci.py

```
# Fibonacci numbers module
def fib(n):    # write Fibonacci series up
    to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
```

```
>>> import fibo
```

```
>>> fibo.fib(100)
```

```
1 1 2 3 5 8 13 21 34 55 89
```