Python Programming Language

By: Priyanka Pradhan

Books include:

- *Learning Python* by Mark Lutz
- *Python Essential Reference* by David Beazley
- *Python Cookbook*, ed. by Martelli, Ravenscroft and Ascher
- (online at http://code.activestate.com/recipes/langs/python/)
- http://wiki.python.org/moin/PythonBooks

4 Major Versions of Python

- "Python" or "CPython" is written in C/C++
  - Version 2.7 came out in mid-2010
  - Version 3.1.2 came out in early 2010

- "Jython" is written in Java for the JVM
- "IronPython" is written in C# for the .Net environment

Contd…

- Created in 1989 by Guido Van Rossum
- Python 1.0 released in 1994
- Python 2.0 released in 2000
- Python 3.0 released in 2008
- Python 2.7 is the recommended version
- 3.0 adoption will take a few years

Development Environments

IDE

1. PyDev with Eclipse
2. Komodo
3. Emacs
4. Vim
5. TextMate
6. Gedit
7. Idle
8. PIDA (Linux)(VIM Based)
9. NotePad++ (Windows)
10. Pycharm

Web Frameworks

- Django
- Flask
- Pylons
- TurboGears
- Zope
- Grok

Introduction

- Multi-purpose (Web, GUI, Scripting, etc.)
- Object Oriented
- Interpreted
- Strongly typed and Dynamically typed
- Focus on readability and productivity

Python features

- no compiling or linking
- rapid development cycle
- no type declarations
- simpler, shorter, more flexible
- automatic memory management
- garbage collection
- high-level data types and operations

Contd..

- fast development
- object-oriented programming
- code structuring and reuse, C++
- embedding and extending in C
- mixed language systems
- classes, modules, exceptions,multithreading
- "programming-in-the-large" support

Uses of Python

- shell tools
  - system admin tools, command line programs
- extension-language work
- rapid prototyping and development
- language-based modules
  - instead of special-purpose parsers
- graphical user interfaces
- database access
- distributed programming
- Internet scripting

Who Uses Python

- Google
- PBS
- NASA
- Library of Congress
- the ONION
- ...the list goes on...

Python structure

- modules: Python source files or C extensions
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

Indentation

- Most languages don't care about indentation
- Most humans do
- We tend to group similar things together

```
/* Bogus C code */
if (foo)
    if (bar)
        baz(foo, bar);
else
    qux();
```

Hello World

- Open a terminal window and type “python”
- If on Windows open a Python IDE like IDLE
- At the prompt type ‘hello world!’

```
>>> 'hello world!'
'hello world!'
```

Python Overview

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

The print Statement

- Elements separated by commas print with a space between them
- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print 'hello'
hello
>>> print 'hello', 'there'
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Everything is an object

- Everything means everything, including functions and classes (more on this later!)
- Data type is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Basic operations

- Assignment:
  - `size = 40`
  - `a = b = c = 3`
- Numbers
  - integer, float
  - complex numbers: `1j+3, abs(z)`
- Strings
  - `'hello world', 'it\'s hot'`
  - `"bye world"`

String operations

- concatenate with + or neighbours
  - `word = 'Help' + x`
  - `word = 'Help' 'a'`
- subscripting of strings
  - `'Hello'[2]` → 'l'
  - slice: `'Hello'[1:2]` → 'el'
  - `word[-1]` → last character
  - `len(word)` → 5
  - immutable: cannot assign to subscript

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```

Numbers: Floating Point

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

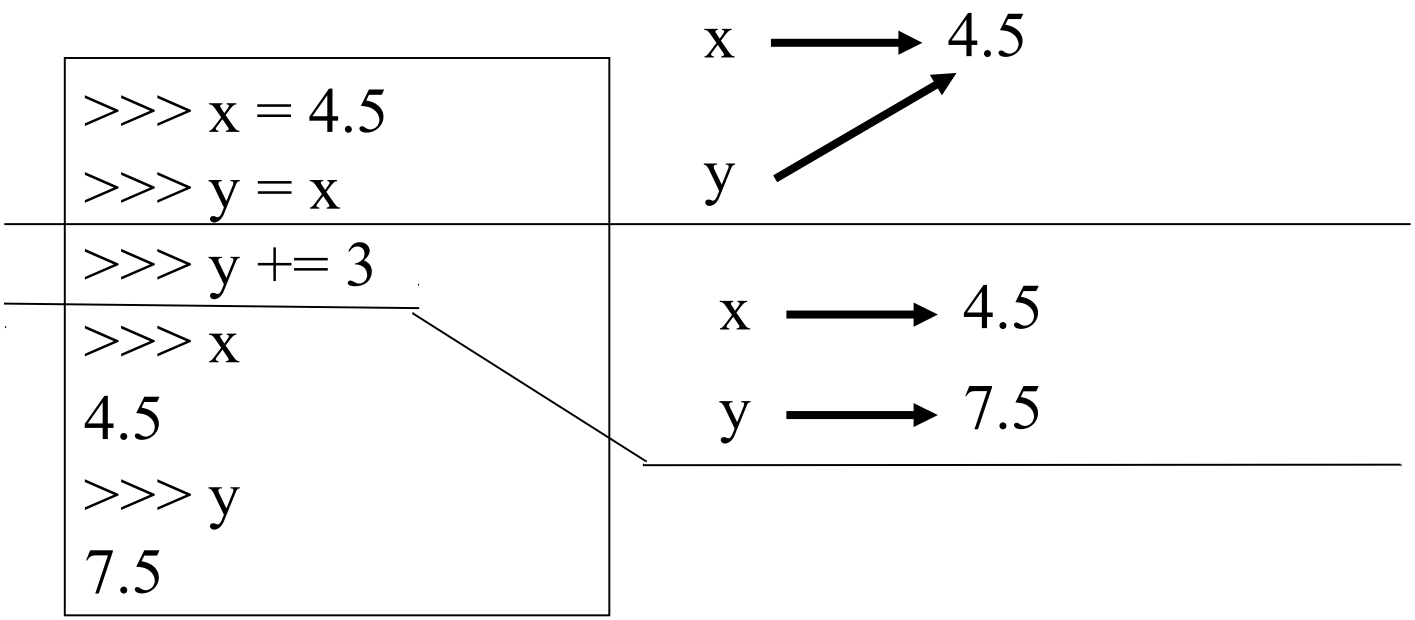
```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Numbers are *immutable*



String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)‘
>>> print s
And me too!
though I am much longer
than the others :)‘
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Do nothing

- pass does nothing
- syntactic filler

```
while 1:
    pass
```

Operators

- Arithmetic

```
a = 10              # 10
a += 1              # 11
a -= 1              # 10

b = a + 1           # 11
c = a - 1           # 9

d = a * 2           # 20
e = a / 2           # 5
f = a % 3           # 1
g = a ** 2          # 100
```

String Manipulation

```
animals = "Cats " + "Dogs "
animals += "Rabbits"
# Cats Dogs Rabbits

fruit = ', '.join(['Apple', 'Banana', 'Orange'])
# Apple, Banana, Orange

date = '%s %d %d' % ('Sept', 11, 2010)
# Sept 11 2010

name = '%(first)s %(last)s' % {
    'first': 'Nowell',
    'last': 'Strite'}
# Nowell Strite
```

Logical Comparison

```
# Logical And
a and b

# Logical Or
a or b

# Logical Negation
not a

# Compound
(a and not (b or c))
```

Identity Comparison

```
# Identity
1 is 1 == True

# Non Identity
1 is not '1' == True

# Example
bool(1) == True
bool(True) == True

1 and True == True
1 is True == False
```

Arithmetic Comparison

```
# Ordering
a > b
a >= b
a < b
a <= b

# Equality/Difference
a == b
a != b
```

Class Declaration

```
class User(object):
    pass
```

Class Attributes

- Attributes assigned at class declaration should always be immutable

```
class User(object):
    name = None
    is_staff = False
```

Class Methods

```
class User(object):
    is_staff = False

    def __init__(self, name='Anonymous'):
        self.name = name
        super(User, self).__init__()

    def is_authorized(self):
        return self.is_staff
```

Class Instantiation & Attribute Access

```
anonymous = User()
print user.name
# Anonymous

print user.is_authorized()
# False
```

Class Inheritance

```
class SuperUser(User):
    is_staff = True
```

```
nowell = SuperUser('Nowell Strite')
print user.name
# Nowell Strite
print user.is_authorized()
# True
```

Imports

```
# Imports the datetime module into the
# current namespace
import datetime
datetime.date.today()
datetime.timedelta(days=1)

# Imports datetime and addes date and
# timedelta into the current namespace
from datetime import date, timedelta
date.today()
timedelta(days=1)
```

Error Handling

```
import datetime
import random

day = random.choice(['Eleventh', 11])
try:
    date = 'September ' + day
except TypeError:
    date = datetime.date(2010, 9, day)
else:
    date += ' 2010'
finally:
    print date
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

List

- lists can be heterogeneous
  - `a = ['spam', 'eggs', 100, 1234, 2*2]`
- Lists can be indexed and sliced:
  - `a[0]` → spam
  - `a[:2]` → ['spam', 'eggs']
- Lists can be manipulated
  - `a[2] = a[2] + 23`
  - `a[0:2] = [1,12]`
  - `a[0:0] = []`
  - `len(a)` → 5

List methods

- `append(x)`
- `extend(L)`
  - append all items in list (like Tcl lappend)
- `insert(i,x)`
- `remove(x)`
- `pop([i]), pop()`
  - create stack (FIFO), or queue (LIFO) → pop(0)
- `index(x)`
  - return the index for value *x*

Contd…

- `count(x)`
  - how many times x appears in list
- `sort()`
  - sort items in place
- `reverse()`
  - reverse list

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

- The method **append** modifies the list and returns **None**
- List addition (+) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Strings share many features with lists

```
>>> smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles[0]
'C'
>>> smiles[1]
'('
>>> smiles[-1]
'O'
>>> smiles[1:5]
'(=N)'
>>> smiles[10:-4]
'C(=O)'
```

String Methods: find, split

```
smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles.find("(O)")
15
>>> smiles.find(".")
9
>>> smiles.find(".", 10)
-1
>>> smiles.split(".")
['C(=N)(N)N', 'C(=O)(O)O']
>>>
```

String operators: in, not in

```
if "Br" in “Brother”:
    print "contains brother“

email_address = “clin”
if "@" not in email_address:
    email_address += "@brandeis.edu“
```

String Method: “strip”, “rstrip”, “lstrip” are ways to remove whitespace or selected characters

```
>>> line = " # This is a comment line \n"
>>> line.strip()
'# This is a comment line'
>>> line.rstrip()
' # This is a comment line'
>>> line.rstrip("\n")
' # This is a comment line '
>>>
```

More String methods

email.startswith(“c") endswith(“u”)
True/False

>>> "%s@brandeis.edu" % "clin"
'clin@brandeis.edu'

>>> names = [“Ben", “Chen", “Yaqin"]
>>> ", ".join(names)
‘Ben, Chen, Yaqin‘

>>> “chen".upper()
‘CHEN'

“\” is for special characters

\n -> newline

\t -> tab

\\ -> backslash

...

But Windows uses backslash for directories!

filename = "M:\nickel_project\reactive.smi" # DANGER!

filename = "M:\\nickel_project\\reactive.smi" # Better!

filename = "M:/nickel_project/reactive.smi" # Usually works

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Tuples and sequences

- lists, strings, **tuples**: examples of *sequence* type
- tuple = values separated by commas

```
>>> t = 123, 543, 'bar'
>>> t[0]
123
>>> t
(123, 543, 'bar')
```

Contd…

- Tuples may be nested

```
>>> u = t, (1,2)
>>> u
((123, 542, 'bar'), (1,2))
```

- Empty tuples: ()

```
>>> empty = ()
>>> len(empty)
0
```

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Contd..

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

- use has_key() to check for existence

```
>>> tel.has_key('foo')
0
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Dictionary Methods

```
person = {'name': 'Nowell', 'gender': 'Male'}

person['name']
person.get('name', 'Anonymous')
# 'Nowell Strite'

person.keys()
# ['name', 'gender']

person.values()
# ['Nowell', 'Male']

person.items()
# [['name', 'Nowell'], ['gender', 'Male']]
```

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Contd…

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

Modules

- collection of functions and variables, typically in scripts
- definitions can be imported
- file name is module name + .py
- e.g., create module `fibo.py`

```
def fib(n): # write Fib. series up to n
    ...
def fib2(n): # return Fib. series up to n
```

Contd…

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as *module.globalname*
- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print “enter your name?"
name = raw_input("> ")

print "When were you born?"
birthyear = int(raw_input("> "))

print "Hi %s! You are %d years old!" % (name, 2017 - birthyear)
```

```
~: python input.py
What's your name?
> Michael
What year were you born?
>1980
Hi Michael! You are 31 years old!
```

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- and and or return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

No Braces

- Python uses ***indentation*** instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This **forces** the programmer to use proper indentation since the indenting is part of the program!

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print ‘y = ‘,
print math.sin(y)
```

In file ifstatement.py

```
>>> import ifstatement
y =  0.999911860107
>>>
```

In interpreter

While Loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop Else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

```
~: python whileelse2.py
1
```

whileelse2.py

For Loops

forloop1.py

```
for x in [1,7,13,2] :
    print x
```

```
~: python forloop1.py
1
7
13
2
```

forloop2.py

```
for x in range(5) :
    print x
```

```
~: python forloop2.py
0
1
2
3
4
```

range(N) generates a list of numbers [0,1, …, n-1]

For Loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):
    print x
    break
else :
    print 'i got here'
```

elseforloop.py

```
~: python elseforloop.py
1
```

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

funcasparam.py

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Higher-Order Functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

Higher-Order Functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

Higher-Order Functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):
    return (x + y)
```

```
>>> from highorder import *
>>> lst = ['h','e','l','l','o']
>>> reduce(plus,lst)
'hello'
```

highorder.py

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

funcinfunc.py

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

funcreturnfunc.py

```
~: python funcreturnfunc.py
<function bar at 0x612b0>
5
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace