Python

Python basics :
Basics of import
Python objects :
- **mutable**
- **non-mutable**
- **variable assignment to objects**

Python object types :
- **numbers**
- **strings, boolean**
- **sets, tuples, lists, dictionaries**
- **files**

List comprehensions
Loops, conditionals

Scripting Language vs Programming Language

- Scripting languages do not require an explicit compilation step
- 'programming' languages – typically used in scenarios where the code will be around for a long time.
- 'scripting' languages – best used when you want to write something quickly and then never use it again
- Mostly these are loose and overlapping classifications.
- Scripting languages run inside another program – an interpreter, a script engine
- Scripting languages are easy to use and easy to write.
- Scripting languages today are used to build complex software – with the fast computers these days, and efficient scripting languages, that for most business operations, there is no practical speed advantage with a compiled programming language.
- Programming languages offer more control over low-level things

Some features of Python

❖ **Software Quality :**

Python code is designed to be readable, and hence reusable and maintainable. Python has deep support for more advanced software reuse mechanisms, such as object-oriented (OO) and function programming.

❖ **Developer Productivity :**

Python code is typically one-third to one-fifth the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

❖**Program Portability :**

Most Python programs run unchanged on all major computer platforms. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines.

❖**Support Libraries :**

Python comes with a large collection of prebuilt and portable functionality, known as the standard library.

❖**Component Integration :**

Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product customization and extension tool. Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA.

Program Exceution

- Python program is a text file with a '.py' extension
- When Python is instructed to run a script -
  - the code is compiled to `byte code`:
    - byte code is a lower-level, platform-independent representation
    - it is stored in a file with a '.pyc' extension
  - then it is routed to `virtual machine`
- Byte code generation is a startup speed optimization
  - when unmodifed source code is executed next time, the compilation step is ignored
- Byte code is saved in files only for files that are imported, not for the top-level files of a program that are only run as scripts - it's an import optimization.
- *To create a .pyc file for a module that is not imported, use the py_compile module as follows :*

  >>>python -m py_compile abc.py

- Python program can be written :
  - interactively

    OR
  - saved to files

Interactive Session

- To start an interactive session on linux :
  - type `python` at the prompt – displays the python prompt
  - Example : `print ('Hello everyone')`
  - to end session -> ^D (on linux), ^Z(on windows)
- When coding interactively, any number of Python commands can be typed – each is run immediately after it is entered
- The interactive session automatically prints the results of typed expressions, thus it is usually not needed to say "print" explicitly at the prompt

  Example :

>>> abc = 'hi'	variable 'abc' is assigned a value
>>> abc	expression typed – its value will be printed
'hi'	
>>> 2**10	expression typed – its value will be printed
1024	

- The interactive prompt is good for experimenting with the language and test program files on the fly

◆Multiline statements (eg, `for loops`) can be entered in this interactive mode.

Example :

```
>>> for x in 'hi' :        # press 'enter' here
...         print(x)          # indent this 'continued' statement
...                           # enter a blank line – it is the multiline statement terminator
h                             # the output
i                             # output continues followed by the prompt
```

Code Files

- Programs can be saved permanentlty by writing code in files – usually called `modules`
- Example script file :

```
# A first Python script
import sys              # Load a library module
print(sys.platform)
print(2 ** 10)          # Raise 2 to a power
x = 'hello!'
print(x * 8)            # String repetition
```

Save this to 'first.py'

- A top-level file can also be named simply as 'first', but code files that must be imported into a client must have '.py' extension
- To execute this, enter the following command :

```
python first.py
```

◆If the python script satisfies the following (on ubuntu) :

- script starts with a line that begins with the characters # ! followed by the path to the Python interpreter on your machine
  (eg : *#!/usr/bin/python*)
- script file has executable permissions

then the python script can be executed directly by simply entering the file name at the command prompt

NOTE :

On windows no special permissions are required.

A python script named as abcd.py can directly be executed on the command prompt by typing either of the following :

abcd.py

Read data from standard input

```
raw_input([prompt])
input([prompt])
```

raw_input([prompt]) :
reads one line from standard input and returns it as a string (removing the trailing newline)

Example :

```
>>> str = raw_input('### ')
### asdf jsdhfdf
>>> str
'asdf jsdhfdf'
>>>
>>>
>>> str = input('### ')
### [x*5 for x in range(2,10,2)]
>>> str
[10, 20, 30, 40]
>>>
```

input([prompt]) :
equivalent to `raw_input`
BUT
assumes that input is a valid Python expression and returns the evaluated result

In Python3, the raw_input function has been removed.
The input function itself works as raw_input

Module Imports – basics

- Every Python source code file with a '.py' extension is a module.
- Other files can access the items a module defines by importing that module.
- The contents of a module are made available to the outside world through its attributes.
- Larger programs usually take the form of multiple module files, which import tools from other module files.
- One of the modules is designated as the main or top-level file, or “script”— the file launched to start the entire program, which runs line by line
  - Below this level, it’s all about modules importing modules
- Import runs the code in a file that is being loaded as a final step – thus importing a file is another way to launch it.
- Example :

```
import first
```

runs the script

- 'import' runs only once per session (process) by default.
- After the first import, later imports do nothing – even if the source file is modified.

This is because :

  - imports are an expensive operation to repeat more than once per file, per session
  - imports must find files, compile them to byte code, and run the code

Module – attributes

- The import operation executes the imported file as the last step.
- Modules serve the role of libraries of tools
- A module is mostly just a package of variable names, known as a namespace
  - the names within that package are called attributes
    - ➔ an attribute is a variable name that is attached to a specific object (like a module)
- Importers gain access to all the names assigned at the top level of a module's file
  - These names are assigned to tools exported by the module—functions, classes, variables—that are intended to be used in other files and other programs
- Externally, a module file's names can be fetched with two Python statements : `import` and `from`, as well as the `reload` call

An Example :

Consider following python script named *myfile.py* :

```
abc = “hello everyone”
xyz = “goodbye”
```

The assignment statements create variables – the module attributes named abc and xyz

These attributes can be accessed in other components in two different ways :

a)the module can be loaded as a whole with the `import` statement

```
>>> import myfile  #Run file; load module as a whole
>>> myfile.abc     #Use its attribute name: '.' to qualify
'hello everyone'
```

a)names can be fetched (copied) out of a module with `from` statement

```
>>> from myfile import xyz     #Run file; copy its names
>>> xyz                        #Use name directly, no need to qualify
'goodbye'
```

`from` copies a modules *attributes* – they become simple *variables* in the recipient

if the `from` statement is used as follows :

```
>>> from myfile import xyz, abc
```

then both the attributes xyz and abc are accessible in the recipient :

```
>>> abc, xyz
('hello everyone', 'goodbye')
```

the valid attributes of an object can be listed via the `dir` built-in function :

```
>>> dir(someObj)    # lists the attributes of 'someObj'
```

Module and Namespaces

- Each module file is a package of variables – i.e., a namespace.
- Thus each module is a self-contained namespace :
  - one module file cannot see the names defined in another file unless it explicitly imports that other file
- Modules minimize name collisions

Python Variables

- Python variables are not required to be declared ahead of time.
- A variable is created when it is assigned a value.
- It may be assigned to any type of object
- It is replaced with its value when it shows up in an expression.

Variables naming conventions

- Names with two leading and trailing underscores (e.g., __name__) generally have special meaning to the Python interpreter, thus avoid this pattern for your own names
- Class names usually start with an upper case alphbet
- Module names start with a lower case alphabet

Consult PEP 8 – a complete style guide

Python Object

- Python objects can be :
  - **Mutable :**
    - ➔ these can be changed in-place
    - ➔ that is, these objects' values can be changed any time
  - **Immutable :**
    - ➔ the original object can not be changed through any means

Variables, Objects and References

- Variable names are not declared before being used – but they must be initialized
- When a variable is assigned to an object, it *references* that object
- Thus the statement `a = 3` does the following :
  - create an object to represent the value `3`
  - create the variable `a`, if it does not yet exist
  - link the variable `a` to the new object `3` –: `a` is a reference to `3`

- Conceptually each time a new value is generated in script by running an expression, Python creates a new object (a chunk of memory) to represent that value.
- As an optimization, Python internally caches and reuses certain kinds of unchangeable objects, such as small integers (-5 to 256) and strings (each 0 is not really a new piece of memory)

- An object is a chunk of memory used to hold :
  - the actual object values
  - a *type designator* used to mark the type of the object
  - a *reference counter* used to determine when it's OK to reclaim the object
- Thus the ***type*** is associated with the object and not with the variable :
  - a variable can be assigned to different objects of different types at different times
  - Example :

```
>>> x = 10
>>> type(x)
<type 'int'>
>>> x = 'abcd'
>>> type(x)
<type 'str'>
>>> x = [1, 2, 'abc', True]
>>> type(x)
<type 'list'>
```

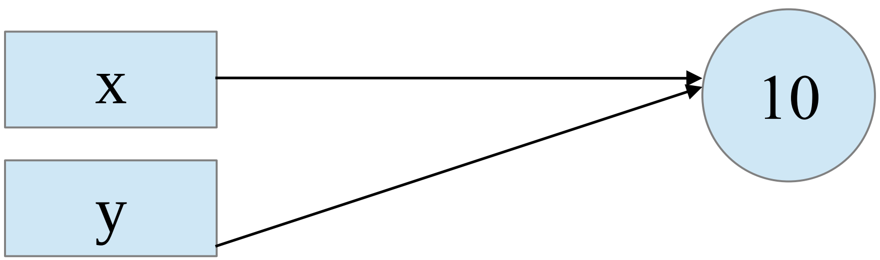
- ◆Whenever a name is assigned to a new object, the space held by the prior object is reclaimed if it is not referenced by any other name or object – this is known as garbage collection

```
>>> x = 10
>>> x = 'abcd'                    # reclaim object 10
>>> x = [1, 2, 'abc', True]       # reclaim object 'abcd'
```

- ◆When a variable is assigned to a pre-assigned variable :

```
>>> x = 10
>>> y = x
```

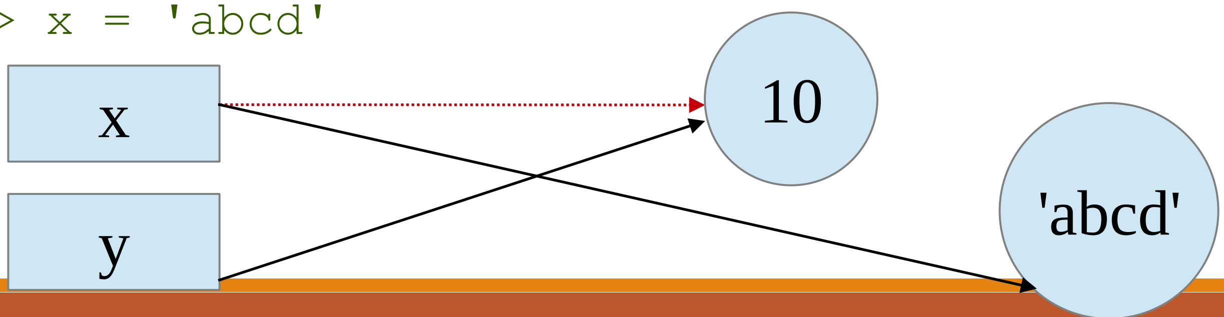
both 'x' and 'y' reference the same object




```
>>> y == x
True
>>> y is x
True
```

if now 'x' references some other object :

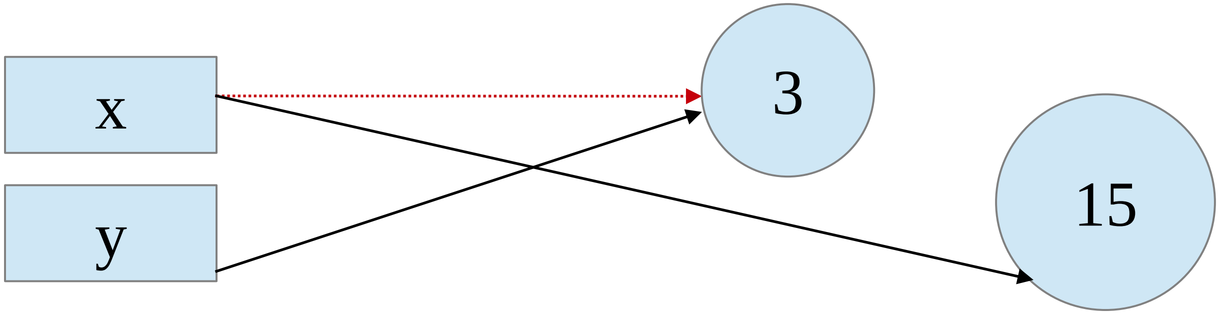
```
>>> x = 'abcd'
```




```
>>> y == x
False
>>> y is x
False
```

in the following case :

```
>>> x = 3
>>> y = x
>>> x = x + 12
```



'x' now refers to a new integer object with value 15

```
>>> x = [1, 2, 'a', 'abc', True]
>>> y = [1, 2, 'a', 'abc', True]
>>> y == x
True
>>> y is x
False
```

◆For mutable objects :

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1[0] = 'abc'
>>> L1
['abc', 2, 3]
>>> L2
['abc', 2, 3]
>>> L1 = [11, 12]     # now L1 and L2 reference different objects
```

```
>>> L1 == L2
True
>>> L1 is L2
True
```

```
>>> L1 = ['a', 1, 2]
>>> L2 = L1[:]
```

OR

```
>>> import copy
>>> L2 = copy.copy(L1)
```

```
>>> L1 == L2
True
>>> L1 is L2
False
```

Object Types

Python Object Types

- Objects can be :
  - built-in OR core data types – ***provided by python***
  - created by programmers using Python classes or external language tools such as C extension libraries
- Following are the built-in types :
  - Numbers
  - Strings
  - Lists
  - Dictionaries
  - Tuples
  - Files
  - Sets
  - Other core types : *Booleans, types, None*
  - Program unit types : *Functions, modules, classes*

Numbers

- Numbers could be :
  - **int, long :** these do not have a fractional part
  - **float :** these have a fractional part
  - **complex :** have imaginary parts
  - **decimal.Decimal :** floating-point numbers with user-definable precision
  - **numbers.Rational :** have numerator and denominator – an abstract base class to the concrete class **fractions.Fraction**

  The **fractions** module provides support for rational number arithmetic

- Numbers support basic mathematical operations (** - for exponentiation)
- Long numbers handled appropriately (eg : 2**100)
- Some useful numeric modules are shipped with Python, example : the `math` module and the random module

```
>>> import math
>>> math.pi
3.141592653589793
```

INTEGERS :

- There are 2 integer types in Python 2.X :
  - normal (often 32 bits) - *implemented using long in C - at least 32 bits of precision (sys.maxint is always set to the maximum plain integer value for the current platform, the minimum value is -sys.maxint - 1)*

  AND
  - long (unlimited precision)
  - An integer may end in an l or L to force it to become a long integer
- In Python 3.X there is only one single integer type :
  - it automatically supports the unlimited precision
- Integers may be coded in decimal, binary, octal or hexadecimal literals :
  - hexadecimals start with a leading 0x or 0X, hex digits may be coded in lower- or uppercase
  - octal literals start with a leading 0o or 0O (zero and lower or uppercase letter 'o')
  - binary literals begin with a leading 0b or 0B
  - built-in functions `hex(I)`, `oct(I)`, and `bin(I)` convert an integer to its representation string in these three bases
  - `int(str, base)` converts a runtime string to an integer per a given base

FLOATING POINT NUMBERS :

- Floating point numbers are usually implemented using double in C
- Information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in sys.float_info

The constructors int(), long(), float(), and complex() can be used to produce numbers of a specific type

Example :

```
>>> x = 100
>>> float(x)
100.0
>>> long(x)
100L
```

```
>>> x = 100L
>>> type(x)
<type 'long'>
>>> x
100L
>>> int(x)
100
>>> float(x)
100.0
```

```
>>> x = 10.101
>>> type(x)
<type 'float'>
>>> int(x)
10
>>> long(x)
10L
```

The Decimal Type :

- Python 2.4 introduced a new core numeric type : the decimal object
- Syntactically, decimals are created by calling a function within an imported module, rather than running a literal expression
- Functionally, decimals are like floating-point numbers, but they have a fixed number of decimal points.

Hence, decimals are *fixed-precision floating-point values*

- Example :
  - ➔using decimals, a floating-point value can be created that always retains just two decimal digits
  - ➔it can be specified how to round or truncate the extra decimal digits beyond the object's cutoff
- decimal incurs a performance penalty compared to normal floating point numbers

```
>>> from decimal import Decimal
>>> d = Decimal('1.21')      # creates a decimal object
                             # using a Decimal constructor function
>>> d
Decimal('1.21')
```

```
>>> x = 1.1
>>> type(x)
<type 'float'>
>>> y = 2.2
>>> x+y
3.3000000000000003
>>> from decimal import *
>>> a = Decimal(1.1)
>>> b = Decimal(2.2)
>>> a+b
Decimal('3.300000000000000266453525910')
>>> getcontext().prec=2
>>> a+b
Decimal('3.3')
>>> getcontext().prec=20
>>> a+b
Decimal('3.3000000000000002665')
```

Read [Decimal fixed point vs floating point] for more details

The Fraction Type :

- Python 2.6 introduced the `Fraction` type
- This implements the *rational number* object
- Fractions also do not map as closely to computer hardware as floating-point numbers.

Hence there performance is not as good

```
>>> from fractions import Fraction
>>> X = Fraction(1, 3)
>>> Y= Fraction(4, 6)
>>> X
Fraction(1, 3)
>>> Y
Fraction(2, 3)
>>> print Y
2/3
```

FRACTIONS (contd) :

◆The fractions module provides support for rational number arithmetic

Examples of creating fractions :

```
>>> import fractions as f
>>> f.Fraction(1,2) # creating from numerator & denominator
Fraction(1, 2)
>>> f.Fraction(1.5) # creating from float-type
Fraction(3, 2)
>>> f.Fraction('1.5') # creating from string-type
Fraction(3, 2)
>>> f.Fraction('2/3') # creating from string-type
Fraction(2, 3)

>>> import decimal as d
>>> deci = d.Decimal(1.5)
>>> f.Fraction(deci)  # creating from decimal-type
Fraction(3, 2)
```

OPERATORS

x if y else z : Ternary selection (x is evaluated only if y is true)
x or y : Logical OR (y is evaluated only if x is false)
x and y : Logical AND (y is evaluated only if x is true)
not x : Logical negation
x in y, x not in y : Membership (for iterables, sets)
x is y, x is not y : Object identity tests
x < y, x <= y, x > y, x >= y
x == y, x != y : Value equality operators
x | y : Bitwise OR, set union
x ^ y : Bitwise XOR, set symmetric difference
x & y : Bitwise AND, set intersection
x << y, x >> y : Shift x left or right by y bits
x * y : Multiplication, repetition
x % y : Remainder, format;
x / y, x // y : Division: true and floor
~x : Bitwise NOT (inversion)
x ** y : Power (exponentiation)

Sets

- Python 2.4 introduced a new collection type, the `set`
- It is an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory. **A set itself is a mutable object**
- Sets are essentially like valueless dictionaries – the items behave much like a dictionary's keys
- To make a set object, pass in a sequence or other iterable object to the built-in set function :

```
>>> S1 = set('abcde')
>>> S1
set(['a', 'c', 'b', 'e', 'd'])
>>> S1.add(1)  # object S1 modified
>>> S1
set(['a', 1, 'c', 'b', 'e', 'd'])
```

- Following set operations are defined :

x – y	*difference*
x | y	*union*
x & y	*intersection*
x ^ y	*symmetric difference (XOR)*
x > y, x < y	*superset, subset*

- These expressions are also available as *methods* of set objects
  `x.intersection(y)`
  `x.symmetric_difference(y)`
- As iterable containers, sets can also be used in operations such as `len`, `for` loops, and list comprehensions
- Because they are unordered, they don't support sequence operations like indexing and slicing
- Since Python 2.7, the following can also be used :

```
>>> S1 = {1,2,3,4}              # S1 is a set
>>> S2 = set([1, 2, 3, 4])      # S2 is a set
```

- Sets can only contain *immutable* types – thus lists, sets and dictionaries cannot be embedded in sets, tuples can be
- Set comprehensions are available since Python 2.7
  - they are coded in curly braces
  - they run a loop and collect the result of an expression on each iteration
  - a loop variable gives access to the current iteration value for use in the collection expression

```
>>> {x ** 2 for x in [1, 2, 3, 4]}
set([16, 1, 4, 9])
```

Strings

- Strings are used to record :
  - textual information (eg : name)

    OR
  - arbitrary collections of bytes (eg : an image file's contents)
- **Python does not support a character type**; these are treated as strings of length one
- Strings are an example of Python ***sequence***.
- Sequence
  - it is a positionally ordered collection of other objects
  - they maintain a left-to-right order among the items they contain
  - their items are stored and fetched by their relative positions.
  - Strings are sequences of one-character strings
  - other more general sequence types include lists and tuples

- ◆Textual data in Python is handled with str objects
- ◆String literals are written in a variety of ways :
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """"Three double quotes""”

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal

- ◆Normal strings in Python are stored internally as 8-bit ASCII
- ◆Unicode strings are stored as 16-bit Unicode
  - This allows for a more varied set of characters, including special characters from most languages in the world
- ◆raw strings use the prefix 'r'

<u>Example</u> :

```
raw_str = r'this is a raw string'
uni_str = u'this is a unicode string'
```

- ◆**In Python 3 strings are unicode strings**

String Features

◆**Sequence operations :**

➢operations that assume a positional ordering among items

Example :

```
>>> S = 'abcd'
>>> S[0]
    'a'
>>> S[1]
    'b'
```

➢index backword from the end :

➔ positive indexes (starting at 0) count from the left

➔ negative indexes (starting at -1) count back from the right

```
>>> S[-1]
    'd'
```

➢string length can be obtained using built-in function `len`

Example :

```
>>> len(S)
    4
```

- Slicing

  Example :

```
>>> S[1:3]
    'bc'
```

Slicing :

general form is X[i:j], ie, string starting at X[i] and goind upto X[j-1]
the left bound defaults to 0
the right bound defaults to the length of the sequence being sliced

- Concatenation with the '+' sign :

```
>>> S + 'xyz'
    'abcdxyz'          # S does not change
```

- Repetition with the '*' sign :

```
>>> S * 3
    abcdabcdabcd
```

◆Immutability :

- no operation causes a change to the original string object
- a string object once created is immutable
- operations only create new strings holding the operation result

```
>>> S = 'abcd'
>>> S + '_xyz'
   'abcd_xyz'
>>> S
   'abcd'
>>> S[1] = 'w'          #reports error
>>> S = 'z' + S[1:]     #expressions can
                        #make new objects
>>> S
   'zbcd'
```

*here the original string object has not changed, but now **S** is referring to a new object*

◆Type-Specific Methods :

- Strings have operations of their own - *methods*

Example :

```
>>> S = 'abcd'
>>> S.find('bc')
   1
>>> S.replace('bc', 'HELLO')
   'aHELLOd'
>>> S
   'abcd'
```

- Some other string methods are : split, upper, isalpha, rstrip
- Strings support formatting :
  - ➔ as an expression

```
>>> '%s, eggs, and %s' % ('spam', 'SPAM!')
   'spam, eggs, and SPAM!'
```

  - ➔ as a string method call

```
>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!')
          'spam, eggs, and SPAM!'
```

Lists

- Python list object is the most general sequence provided by the language
- Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size
- They are mutable – lists can be modified in place by assignment to offsets as well as a variety of list method calls
- They provide a very flexible tool for representing arbitrary collections – lists of files in a folder, employees in a company etc
- A List can be arbitrarily nested
- Technically, lists contain zero or more references to other objects
- List items can be fetched by using a list offset

List Features

◆ **Sequence operations :**

➢ lists support all the sequence operations – the results of these operations are lists

Example :

```
>>> L = [1, 'abcd', '10', 20]
>>> L[0]
    1
>>> L[1]
    'abcd'
>>> L[:-1]     #list slicing
    [1, 'abcd', '10']
>>> L + [4, 5, 6]
    [1, 'abcd', '10', 20, 4, 5, 6]
>>> L * 2
    [1, 'abcd', '10', 20, 1, 'abcd', '10', 20]
```

In all these examples, no operation tries to modify the list object itself

- **Type-Specific Methods :**
  - Pythin lists have no fixed type constraint – they can be containing objects of all different types
  - They have no fixed size – they can grow and shrink on demand
  - Lists have operations of their own - *methods*

Example :

```
>>> L = [1, 'abcd', '10', 20]
>>> L.append(1.2)
>>> L
   [1, 'abcd', '10', 20, 1.2]
>>> L.pop(2)
   '10'
>>> L
   [1, 'abcd', 20, 1.2]
```

  - Some other list methods are : sort, reverse, count, index, extend

◆**Bounds Checking :**

➢Python still doesn't allow us to reference items that are not present – indexing off the end of the list is an error :

```
>>> L = [1, 'abcd', '10', 20]
>>> L[99]     # error reported
```

◆**Nesting :**

➢A list can nest Python's core data types in any combination and to any depth.

Example : a list can contain a dictionary, which can contain another list and so on ...

Application : lists can be used to create multi-dimensional arrays

```
>>> L = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]
>>> L
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> L[1]
[4, 5, 6]
>>> L[1][2]
6
```

◆Comprehensions :

- ➢List comprehensions are a way to build a new list by applying an expression to each item in a sequence (or in any iterable), one at at time, from left to right
- ➢They are close relatives to for loops
- ➢List comprehensions are coded in square brackets and are composed of an expression and a looping construct that share a variable name

Some Simple Examples :

```
>>> L = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> col2 = [row[1] for row in L]
>>> col2
    [2, 5, 8]     # L is unchanged
>>> [row[1] + 1 for row in L]
    [3, 6, 9]
>>> [L[i][i] for i in [0, 1, 2]]    # a matrix diagonal
    [1, 5, 9]
>>> [c * 2 for c in 'square']
    ['ss', 'qq', 'uu', 'aa', 'rr', 'ee']
```

Dictionaries

- Dictionaries are NOT sequences – they are known as *mappings*
- These collections store objects by key instead of by relative position
- They don't maintain any reliable left-to-right order
- They are mutable – may be changed in place and can grow and shrink on demand
- Dictionaries are coded in curly braces
- They consist of a series of "key: value" pairs.
- Each key can have just one associated value, but that value can be a collection of multiple objects
- A given value can be stored under any number of keys
- Dictionaries are useful when it is required to associate a set of values with keys
- Dictionary items are accessed by *keys* (not by offset)
- A Dictionary object is an unorderd collection of arbitrary objects
- A Dictionary is of variable-length, heterogeneous, and arbitrarily nestable

Dictionary Features

◆ **Mapping operations :**

- a dictionary associates a set of values with keys
- it is indexed by keys

Example :

```
>>> D = {'food': 'egg', 'no': 4, 'color': 'white'}
>>> D['food']
'egg'
>>> D['no'] += 1
>>> D
{'food': 'egg', 'color': 'white', 'no': 5}
>>> D['aa'] = 100      #new key-value can be created
   #via assignment
>>> D
{'food': 'egg', 'color': 'white', 'aa': 100, 'no': 5}
```

◆**Keys :**

- accessing non-existent key is an error
- keys must be of an immutable data type such as strings, numbers, or tuples
- the dictionary `in` membership expression can be used to test for presence of a key

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> 'f' in D
False
>>> if 'f' not in D
   print 'wrong key'
```

- the dictionary `get` method can be used :

```
>>> D.get('b')          # returns '2'
>>> D.get('f')          # returns empty string
```

- `try` can be used to check for the same

◆**Type-Specific Methods :**

- Some of the type specific methods are : pop, update, copy, clear

◆Dictionary comprehensoin example :

```
>>> d = {'id4': 'qqq', 'id2': 'aaa', 'id3': 'qwe',
    'id1': 'aaa'} # id, name
>>> d1 = {'id4': 'qqq@q', 'id1': 'aaa@a'} # id, email
>>> newD = {d[x]:d1[x] for x in d if x in d1}
>>> newD
{'aaa': 'aaa@a', 'qqq': 'qqq@q'} # name, email
```

Tuples

- The tuple object is like a list that cannot be changed – they are immutable sequences
- They are coded in parenthesis and support arbitrary types, arbitrary nesting, and the usual sequence operations
- They are ordered collections of arbitrary objects
- They can be accessed by offset
- They are fixed-length, heterogeneous, and arbitrarily nestable
- Following are examples of defining tuple objects :

```
>>> T1 = (1, 2, 'qwe')
>>> T2 = 1, 2, 'qwe'
>>> T3 = tuple(x)   # where 'x' is some iterable object
```

Tuple Features

◆**Tuple object :**

➢following is a tuple object

```
>>> T = (1, 'abcd', '10', 20)
>>> T[1]
'abcd'
>>> T = ('asdf',)        # a tuple containing a single item
>>> T + (4, 5, 6)        # concatenation
('asdf', 4, 5, 6)
 >>> T = ('asdf',)
>>> T * 3
('asdf', 'asdf', 'asdf')
>>> T = ('asdf', 11, 22, 33)
>>> T = (2,) + T[1:]
>>> T
(2, 11, 22, 33)
```

◆**Tuple object is immutable :**

```
>>> T = ('asdf', 11, 22, 33)
>>> T[0] = 'q'          # reports error
```

- following is a tuple object

```
>>> T = (1, 'abcd', '10', 20)
```

◆**Type-specific methods :**

- some of the type specific methods of tuples are : index, count

The 'None' Object

- This is special object always considered to be *false*
- It serves like a empty placeholder
- It is also the default return value of functions that don't exit by running into a return statement with a result value

Assignments

◆Normal simple assignment :

```
day = "Monday"
```

◆Tuple and list unpacking assignment :

```
day, no = 'Mon', int(10)  # Tuple assignment (positional)
[d, no] = ['Mon', int(10)]  # List assignment (positional)
```

- When there is a tuple or list on the left side of the '=' the objects on the right side are paired with targets on the left by position and assignment happens from left to right
- Internally a tuple of the items on the right is made – hence this is called tuple-unpacking assignment

```
>>> var1, var2 = 10, 90
>>> var2, var1 = var1, var2
>>> var1, var2
(90, 10)
```

- A temporary tuple is created that saves the original values of the variables on the right while the statement runs – thus unpacking assignments are also a way to swap two variables' values without creating any temporary variable

- Sequence assignment :

```
a, b, c, d = 'part'   # Sequence assignment
```

  - any sequence of names can be assigned to any sequence of values – the items will be assigned one at a time by position

- Extended sequence unpacking :# Python 3.X (only)

```
a, *b = 'part'
```

  - matches `'a'` with the first character in the string on the right and `'b'` with the rest : **a** is assigned 'p', and **b** is assigned ['a', 'r', 't']

```
a,*b,c,d = 'parts'
```

  - **a** is assigned 'p', and **b** is assigned ['a', 'r'], **c** is assigned ‘t’ and **d** is aassigned ‘s’

- Multiple target assignment :

```
var1 = var2 = 'hello'
```

- Augmented assignment :

```
var1 = 100
var1 += 20
```

◆Advanced sequence unpacking assignment :

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
>>> list(range(4))
[0, 1, 2, 3]
```

Statement Delimiters

- A statement in Python normally ends at the end of the line on which it appears
- Statements may span multiple lines if you're continuing an open syntactic pair
  - a statement can be continued on the next line if something is coded enclosed in a (), {}, or [] pair
- Statements may span multiple lines if they end in a backslash
- For string literals : triple-quoted string blocks are designed to span multiple lines

Truth Values

- All objects have an inherent Boolean true or false value
- Any nonzero number or nonempty object is true
- Zero numbers, empty objects, and the special object `None` are considered false
- Comparisons and equality tests are applied recursively to data structures
- Comparisons and equality tests return `True` or `False` (custom versions of 1 and 0)
- Boolean `and` and `or` operators return a true or false operand object
- Boolean operators stop evaluating ("short circuit") as soon as a result is known

The Control Statements

The *if* statement

- The syntax :

```
if x < y:
   print (y)
   y = x
   print (“now y equals x”)
else :
   print (“in else”)
print (“after the if block”)
```

- All compound statements follow the following pattern :
  - a header line terminated in a colon
  - the header line followed by a nested block of code indented underneath the header line
  - the end of indentation is end of block
  - indentation can be achieved via *spaces* or *tabs* – but it should be absolutely same in the entire block, the next sub block (if any) may follow its own indentation technique

- The following can also be followed :

```
if x<y: print x; y = x; print "now y equals x";
else: print “in else”
print "after if"
```

- All compound statements follow the following pattern :
  - a header line terminated in a colon
  - the header line followed by a nested block of code indented underneath the header line
  - the end of indentation is end of block
  - indentation can be achieved via *spaces* or *tabs* – but it should be absolutely same in the entire block, the next sub block (if any) may follow its own indentation technique

- The general form of `if` statement is :

```
if test1:                  # if test
    statement set#1        # Associated block
elif test2:                # Optional elifs
    statement set#2
else:                      # Optional else
    statement set#3
```

◆No *switch-case* statement available :

➢dictionaries and lists (built at runtime dynamically) can be used than hardcoded `if` logic in script

```
>>> choice = 'ccc'
>>> print ({'aaa': 1.1,                    # defining dictionary
        'sss': 2.2,
        'ccc' :  3.3,
        'wwwww' : 4.4} [choice])    # using 'choice' to index
```

the above code can replace the following :

```
>>> if choice == 'spam':
          print(1.25)
    elif choice == 'ham':
          print(1.99)
    elif choice == 'eggs':
          print(0.99)
    elif choice == 'bacon':
          print(1.10)
    else:
          print('Bad choice')
```

- The `get` method in `dictionary` can be used to achive the same

```
>>> d = {'a':1, 's':2, 'd':3}
>>> d.get('a', -1)        # if 'a' is in d, return d['a'], else -1
1
>>> d.get('q', -1)
-1
```

dictionaries can also contain functions to represent more complex branch actions

- The ternary `if/else` expression :
  - The following statement :

```
if X:
    A = Y
else:
    A = Z
```

can be written as :

```
A = Y if X else Z
```

The *while* loop

◆The general format :

```
while test:          # Loop test
    statements       # Loop body
else:                # Optional else
    statements       # Run if didn't exit loop with break
```

◆A simple example :

```
>>> x = 8
>>> while x>5 :
        print x
        x -= 1
8
7
6
```

◆break:

➢jumps out of the closest enclosing loop

```
x = 10
while x>0 :
    print x,
        x -= 1
        if x == 6 : break
print
print "out of while"
10 9 8 7
out of while

x = 10
while x>0 :
    print x,
        x -= 1
        break
print
print "out of while"
10
out of while
```

```
x = 10
while x>0 :
print x,
    x -= 1
    if x == 7 : break
print "last statement"
print "out of while"
10 last statement
9 last statement
8 out of while
```

- `continue`:
  - jumps to the top of the closest enclosing loop

```
x = 10
while x>5 :
   print x,
      x -= 1
      if x == 7 : continue
      print "last statement"
print "out of while"
```

```
10 last statement
9 last statement
8 7 last statement
6 last statement
out of while
```

For x = 8, “last statement” is not printed

- `pass` :
  - does nothing at all: it's an empty statement placeholder
  - it is used when the syntax requires a statement, but there is nothing useful to say

Example : following is an empty infinite loop

```
while True: pass
```

  - can be used to define empty functions :

```
def func1():
   pass          # Add real code here later
```

◆ loop else block:

➢ Runs if and only if the loop is exited normally, without hitting a break

Example #1 :

```
x = 10
while x>5 :
    print x,
    x -= 1
    if x == 7 : break
    print "last statement"
else :
    print "else"
print "out of while"
```

```
10 last statement
9 last statement
8 out of while
```

Example #2 :

```
x = 10
while x>5 :
    print x,
    x -= 1
    print "last statement"
else :
    print "else"
print "out of while"
```

```
10 last statement
9 last statement
8 last statement
7 last statement
6 last statement
else
out of while
```

The *for* loop

- The general format :

```
for target in object:
    statements
else:                          # Run if didn't exit loop with break
    statements
```

- Some simple examples :

```
for x in [10, 20, 30] : print x,
10 20 30

d = {'a':10, 's':20, 'd':30}
for x in d : print (x, d[x]),
('a', 10) ('s', 20) ('d', 30)

for i in range(3) :   print i,
0 1 2

for i in range(3, 10) :   print i,
3 4 5 6 7 8 9

for i in range(3, 10, 2) :print i,
3 5 7 9
```

In Python 2.x, `range` is a built-in function that returns a list.
In Python 3.x `range` is an immutable sequence type – thus call to `range` returns an iterable object and not a list

- Some more examples with *range* :

```
range(20, 10, -2)
[20, 18, 16, 14, 12]

range(20, 10, -2)[1]
18
```

In Python 3.x, the above can be executed as follows :

```
list(range(20, 10, -2))
[20, 18, 16, 14, 12]
```

◆`for` loop used to alter a list entries :

```
L = [1, 2, 3, 4]
for i in range(len(L)) :
    L[i] += 10
print L
[11, 12, 13, 14]
```

◆working on multiple lists parallely :

```
l1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
l2 = [22, 33, 44, 11]
l = zip(l1, l2, l3, ...]     # produces list (of tuples) in 2.X
                             # produces an iterable obj (of tuples)
                             # in 3.X
                             # wrap it in a list call to get a list
```

```
for (x, y, z) in l :
    print x, y, z
1 22 222
2 33 333
3 44 444
4 11 111
```

```
for x in l :
    print x
(1 22 222)
(2 33 333)
(3 44 444)
(4 11 111)
```

```
for x,y,z in l :
    print x
1
2
3
4
```

- using `for` loop with `enumerate` function :
  - the `enumerate` function returns tuples of the type *`(index, item)`* as a `generator` object
  - the *index* can be given as the second argument to `range` function (`start`) to the function that defaults to 0

```
S = 'spam'
for (offset, item) in enumerate(S):
   print(item, 'appears at offset', offset)
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

Files

- File objects are the main interface to external files on your computer
- They can be used to read and write text memos, audio clips, Excel documents etc
- There is no specific literal syntax for creating a file object
- To create a file object, the built-in `open` function is used, passing in an external filename and an optional processing mode as strings
- The file *iterator* can be used to read lines apart from the `read` method
- The data read is received as *string* and not as object
- Output files are buffered – text written may not be transferred from memory to disk immediately, closing a file or running its flush method, forces the buffered data to disk
- When `file` objects are reclaimed, Python also automatically closes the files if they are still open

File Features

◆ **The file object :**

➢ following creates a text output file :

```
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')          # Write a string of characters to it
>>> f.close()
```

➢ the file contents can be read back by reopening the file in 'r' mode

```
>>> f = open('data.txt')   # default mode is 'r'
>>> text = f.read()
>>> text
'Hello\n'
```

◆ **Type-specific methods are :**

➢ Some file functions are : read, readline, write, close, flush

◆ Writing objects to file :

➢objects must be converted to strings while writing to files :

```
>>> F = open('datafile', 'w')
>>> X, Y, Z = 43, 44, 45        # Three numbers defined
>>> F.write('%s,%s\n' % (X, Y))  # nos converted to strings
>>> L = [1, 2, 3]
>>> F.write(str(L) + '\n')
>>> D = {'a': 1, 'b': 2}
>>> F.write(str(D) + '\n')
>>> f.close()
```

➢This file data can now be read as :

```
>>> F = open('datafile')
>>> line = f.readline()
>>> line
'43,44\n'
>>> parts = line.split(',')
>>> parts
['43', '44\n']
>>> numbers = [int(P) for P in parts]
>>> numbers
[43, 44]
```

```
>>> line = f.readline()
>>> line
'[1, 2, 3]\n'
>>> eval(line)          # 'eval' is built in function, it can convert a
                        # string containing a Python expression
[1, 2, 3]
>>> line = f.readline()
>>> line
"{'a': 1, 'b': 2}\n"
>>> eval(line)
{'a': 1, 'b': 2}
```

The *`Iterable`* object and the *`Iterator`*

- An object is considered *iterable* if :
  - it is a physically stored sequence

  OR
  - an object that produces one result at a time in the context of an iteration tool like a `for` loop
- A common followed convention :
  - **iterable :** an object that supports the `iter` call
  - **iterator :** an object returned by an iterable on `iter` that supports the `next(I)` call

The Iteration Protocol -

with File Iterators as example

- Files have a method named `__next__` in 3.X (and `next` in 2.X) :
  - `__next__` raises a built-in `StopIteration` exception at end of file
  - `next` returns an empty string at end of file

use `f.next()` (in python2) OR `next(f)` (in all python)

- The `for` loop automatically calls the `__next__` method to advance to the next line on each iteration

Example :

```
for line in open('test.py'):
 print line.upper(),
```

reads the file 'test.py' line by line and prints the uppercase version of each line

the same could have been achieved as follows :

```
for line in open('test.py').readlines():
 print line.upper(),
```

The first option is better because :

`readlines()` performs poorly in terms of memory usage – it loads the entire file in memory all at once

- It is also possible to read a file line by line as follows :

```
f = open('test.py')
while True :
   line = f.readline()
   if not line : break
   print line.upper(),
```

this may run slower than the iterator-based `for` loop version, because iterators run at C language speed inside Python, whereas the while loop version runs Python byte code through the Python virtual machine

iter and *next*

◆Check the following :

```
>>> l = [11, 22, 33, 44]
>>> i = iter(l)
>>> i.next()
11
>>> i.next()
22
```

l is the itarable
i is the iterator

for files :

```
>>> f = open('test.py')
>>> f.next() # Python 2.7, use f.__next__() for Python 3
# first line in file
>>> f.next()
# second line in file
```

Thus the initial step (to acquire an interator) is not required for files, because a file object is its own iterator

◆There can be multiple iterator objects for the same iterable object, each referring to a different location

- Similarly for dictionaries :

```
>>> d = {'a' : 10, 'b' : 20, 'c' : 30, 'd' : 40}
>>> i = iter(d)
>>> i.next()
'a'
>>> d[i.next()]
30                                  # for dictionary storage order is not known
>>> d[i.next()]
20
```

List Comprehensions

List Comprehensions

- ◆Used to construct lists in a very natural, easy way, like a mathematician is used to do
- ◆Its syntax is derived from a construct in set theory notation that applies an operation to each item in a set

Example :

```
L1 = [1, 2, 3, 4]
L2 = [x+10 for x in L1]
print L2
[11, 12, 13, 14]
```

- ◆List comprehensions are written in square brackets.
- ◆They begin with an arbitrary expression that we make up, which uses a loop variable that – `(x + 10)`
- ◆This is followed by what makes the header of a for loop – it names the loop variable and an iterable object `(for x in L)`

- Technically speaking, list comprehensions are not really required – a list can always be build up manually with for loops that append results The previous example can be written as :

```
L2 = []
L1 = [1, 2, 3, 4]
for x in L1:
    L2.append(x + 10)
print L2
```

- List comprehensions are more concise to write
- List comprehensions might run much faster than manual `for` loop statements because their iterations are performed at C language speed inside the interpreter, rather than with manual Python code
- For larger data sets, there is often a major performance advantage to using this expression
- Another example :

```
f = open('test.py')
lines = f.readlines()   # readlines method loads file
                        # into a list of line strings
L = [x.rstrip() for x in lines]
```

Filter clause : *if*

◆The following example creates a list of only those lines of an input files that start with the letter 'p' :

```
L = [line.rstrip() for line in open('test.py') if line[0]
 == 'p']
```

Nested loops : *for*

◆List comprehensions can be nested to any level by use of `for` loops

Example :

```
[x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Exercises :

Create the following using comprehensions :

- A list containing all even nos lying between 100 and 200
- A list containing all uppercase versions of words contained in a given set
- A collection of unique email extensions from a set of email ids
- A list of words given in a set that start with alphabet 'p' to 'z'

DAY 2

Command line arguments
Docstrings
Functions
def, lambda
global, nonlocal, name resolution,
nested scope
return, yield - *generators*
factory functions
argument passing,
argument matching rules
returning values
functions as objects, function attributes

Understanding modules in detail
namespaces
import, from, reload statements
module filename extension
executing modules as script

Package import
the __init__.py file
– initialization code
creating a package

Programming tools like map, filter, reduce

DOCSTRINGS

- ◆Documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods
- ◆**An object's docsting is defined by including a string constant as the first statement in the object's definition**
- ◆It's specified in source code that is used, like a comment, to document a specific segment of code
- ◆Docstring should describe what the function does, not how
- ◆All functions should have a docstring
- ◆This allows the program to inspect these comments at run time, for instance as an interactive help system, or as metadata
- ◆Docstrings can be accessed by the `__doc__` attribute on objects

Example :

```
def func() :
    '''this is doc str
going on to line#2
and line#3'''
    print(func.__doc__)
```

Output :

this is doc str
going on to line#2
and line#3

docstring/docstrEg1

```
'''
this is module testing docstrings
this string can go to any no of lines
'''

def func() :
    '''this is doc str
going on to line#2
and line#3'''
'''not a docstring'''
    x = 10
    '''not even this'''

class C :
    '''this is class docstr'''
    x = 10
    def f(s) :
        '''this is docstr of f in C'''
        a = 10

print(func.__doc__)
print(C.__doc__)
```

Command line arguments

- `sys` module provides access to any command-line arguments via `sys.argv`
- This serves two purposes :
  - `sys.argv` is the list of command-line arguments
  - `len(sys.argv)` is the number of command-line arguments
  - `sys.argv[0]` is the program ie. script name

```
import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'
print 'Argument List:', str(sys.argv)
```

python jlt.py a1 12 www 1.3

```
Number of arguments: 5 arguments.
Argument List: ['jlt.py', 'a1', '12', 'www', '1.3']
```

Functions

Functions

- Functions could be :
  - built-in

  A complete list of built-in functions can be viewed at :

  https://docs.python.org/2/library/functions.html

  https://docs.python.org/3/library/functions.html

  OR
  - user defined

User defined functions

- There are two ways to make functions :
  - `def`
  - `lambda`

- There are two ways to manage scope visibility :
  - `global`
  - `nonlocal`

- There are two ways to send results back to callers :
  - `return`
  - `yield`

- `def` creates an object and assigns it to a name :
  - generates a new function object
  - this object is assigned to the function's name –:
    - this is similar to any other assignment
    - the function object may be assigned to other names :

```
def mul(x, y):
    print("in mul", x, y)

print("calling mul...")
mul(2, 4)

x = mul;
print("calling x...")
x(10, 20)
i = int ; ii = i(10) ;
print(type(ii))
```

the function name can be assigned to another object :

```
def mul(x, y):
Print("in mul", x, y)

mul(2, 4)
mul = [1,2,3,4] ;
print mul
```

- function names can be stored in a list :

```
def mul(x, y):
Print("in mul", x, y)

def abc():
  print "func abc"

def xyz() :
print "func xyz"

mul(10, 20);
abc(); xyz()  # normal function calls

l = [mul, abc, xyz];
l[0](1,2);
l[1]()
```

- `lambda` creates an object but returns it as a result *(discussed later)*
- `return` sends a result object back to the caller :

```
def mul(x, y) : return x*y
print mul(10, 20)
```

- `yield` sends a result object back to the caller, but remembers where it left off
- By default, all names assigned in a function are local to that function and exist only while the function runs
  - `global` declares module-level variables that are to be assigned
  - `nonlocal` declares enclosing function variables that are to be assigned
- Arguments are passed by assignment (object reference)
  - changing an argument name within a function does not change the corresponding name in the caller
  - changing passed-in mutable objects in place can change objects shared by the caller – this serves as a function result

```
def mul(x, y):x = x+10;print "x in func", x;y[1]= -y[1]
a = 100; l = [1, 2, 3]
mul(a, l)
print("a after func call is", a);
print l
```

- Arguments are passed by position, unless otherwise mentioned
  - function calls can also pass arguments by name with *name=value* keyword syntax
- Arguments, return values, and variables are not declared
  - nothing about a function needs to be declared ahead of time
  - arguments of any type can be passed to a function
  - a function can return any kind of object

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression

```
x = lambda a : a + 10

print(x(5))

x = lambda a, b : a * b
print(x(5, 6))

x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

def

- `def` statement creates a function object and assigns it to a name
- General format is as follows :

```
def name(arg1, arg2,... argN) :
    statements
```

- Because it's a statement, it can appear anywhere a statement can – even nested in other statements

Example :

```
if test:
    def func():   # Define func one way
    ...
else:
    def func():   # Or else define it the other way
    ...
...
func()
```

- Before calling a function, it must be made by running its `def` statement :
  - *either* by typing it interactively
  - *or* by coding it in a module file and importing the file

Polymorphism

- Argument types are not specified while defining functions
- What a function means and does depends on what is passed to it
  - the same function can generally be applied to a whole category of object types automatically
  - as long as those objects support the expected interface the function can process them

Example :

```
def test(x, y) : return x * y
print test(2, 3)           # function returns 6
print test('hello!', 2)    # function returns
                        # 'hello!hello!'
print test('hello!', 'aa') # exception raised
```

Thus any two objects that support the '*' operator in that order can be passed to this function

- This is known as polymorphism

- Python is a dynamically typed language ***(types are associated with values not variables)*** – thus polymorphism runs unrestricted
- This is a major difference between Python and statically typed languages like C++ and Java
- In Python, your code is not supposed to care about specific data types
  - if it does, it will be limited to working on just the types anticipated at the time of coding
  - it will not support other compatible object types that may be coded in the future

Python supports no form of function overloading

Local Variables

◆It is a name that is visible only to code inside the function `def` and that exists only while the function runs

Example :

```
def pow(x, y) :
   res = 1
   for i in range(y) :
         res = res*x
   print res

pow(10, 3)
print res       # error : can not access 'res' out of scope of function 'pow'
print i         # error : can not access 'i' out of scope of function 'pow'
```

Arguments are passed by assignment, so `x` and `y` are also local

◆Function's local variables won't remember values between calls

◆Names assigned inside a def do not clash with variables outside the def, even if the same names are used elsewhere

Example :

```
i = 10; res=20; j = 30 # i, j, res are global variables
print i;print res       # i and res are 10 and 20
def pow(x, y) :
    res = 1
    for i in range(y) : res = res*x
 # for creates a local variable i which is hiding the
 # global variable i defined earlier
    print res          # i and res are as per values in pow
print j            # j assigned before pow, is available
                   # in pow as well
pow(10, 3)
print i;print res   # i and res are 10 and 20
```

◆Thus variables may be assigned in three different places :

- a variable assigned inside a `def` – it is local to that function
- a variable assigned in an enclosing `def` – it is nonlocal to nested functions
- a variable assigned outside all `def`s – it is global to the entire file

- The enclosing module is a global scope :
  - each module is a global scope – a namespace in which variables created (assigned) at the top level of the module file live.
  - global variables become attributes of a module object to the outside world after imports
  - these can also be used as simple variables within the module file itself

Example :

```
# file file1.py
impVar = 90

# file test.py
import file1
print file1.impVar
```

- The term 'global' is applicable to a single file only :
  - there is no notion of a single, all-encompassing global file-based scope
  - names are partitioned into modules – import a module explicitly if the names defined in a file are to be used

- By default, all the names assigned inside a function definition are put in the local scope
  - a name in a function can be declared in a `global` statement to access the name outside the function, but only after the function has been executed

```
def func() :
    global i
    i = 10
    print "i in func", i
func()
print i          # if prev func() call is removed, this generates error
```

Name Resolution - LEGB

- With a `def` statement :
  - name assignments create or change local names
  - name references search at most four scopes :
    - ➔local - **L**
    - ➔*then* enclosing functions (if any) - **E**
    - ➔*then* global - **G**
    - ➔*then* built-in - **B**

```
#open = 10
def f1() :
      #open = [1,2,3,4]
      def f2() :
            #open = (1, 2, 3)
            print open
      f2()
f1()    # uncomment the statements one-by-one and check the output
```

- names declared in `global` and `nonlocal` (available in 3.X) statements map assigned names to enclosing module and function scopes, respectively

The `global` statement

- `global` allows to change names that live outside a `def` at the top level of a module file
- The `global` statement consists of the keyword `global`, followed by one or more names separated by commas
- All the listed names will be mapped to the enclosing module's scope when assigned or referenced within the function body

```
x = 10;y = 99
def f1() :
    global x
    x = [1,2] ; y = [20, 30, 40]
 # a local 'y' is assigned and created here
 # whereas, 'x' refers to the global name
f1()
print x, y
```

Thus `'global'` can be used as :

◆***creating a global from within a function definition :***

```
# ModA.py
def f1() :
    global x
    x = [1,2] ; y = [20, 200]
    # a local 'y' is created – availableonly within 'f1'
    # 'x' can be accessed anywhere after a call to 'f1'
f1()
print x, y # 'y' can not be accesed – error
```

```
# ModB.py
import ModA
print ModA.x
```

```
# ModC.py
from ModA import *
print x
```

```
# import causes 'f1()' to be executed as well
# thus 'ModA.x' becomes accessible
```

◆***accessing a globally declared name***

```
foo = 10
def f1() :
    foo = 'abc' # new local foo created
def f2() :
    global foo
    foo = 'xyz' # value of global foo changes
```

Example with `global` and `nonlocal`

functions/global_local.py

```
f1-->g1:: -9 g2:: abcd e1:: [1, 2] e2:: (11, 22)
g-->g1:: 100 g2:: qqq e1:: [1, 2, 111] e2:: (11, 22)
f2-->g1:: -9 g2:: abcd e1:: [1, 2] e2:: (11, 22)
g1 in main:: 10 g2 in main:: abcd
```

```
g1 = 10
g2 = 'abc'
def f() :
    g1 = -9
    global g2; g2 = 'abcd'
    e1 = [1,2]
    e2 = (11,22)
    print 'f1-->g1::', g1, 'g2::', g2, 'e1::', e1, 'e2::', e2
    def g() :
        g1 = 100; g2 = 'qqq'
        e1 = [1, 2, 111]
        #nonlocal e2      # only in 3.X
        #e2 = 'new val'   # would allow the change to be reflected
                       # in the enclosing bloack
        print 'g-->g1::', g1, 'g2::', g2, 'e1::', e1, 'e2::', e2
    g()
    print 'f2-->g1::', g1, 'g2::', g2, 'e1::', e1, 'e2::', e2

f()
print 'g1 in main::', g1, 'g2 in main::', g2
```

Cross file changes

global-scope variables of a module become attributes of the loaded module object

- Consider the following code :

 # file `file1.py`

```
impVar = 90
def f() :
    global impVar
    impVar += 1    # the value of impVar becomes unpredictable if
                   # this module is imported and its value changed
                   # over there
    print impVar
```

 # file `test.py`

```
import file1
file1.impVar = 900
file1.f()
```

- Thus as long as the importing module *reads* the value of the attribiute 'impVar' its is OK
- But when the importing modules *writes* to the attribute, problem is anticipated

Nested Scope – scope lookup

- Functions can access names in all physically enclosing def statements

Example :

```
X = 99              # global scope name - unused
def f1():
    X = 88          # Enclosing def local
    def f2():
        print X     # Reference made in nested def

f2()             # Prints 88: enclosing def local
f1()
```

- The enclosing scope lookup works even if the enclosing function has already returned

Example :

```
def f1():
    print 'in f1'; X = 88
    def f2():
        print X     # Remembers X in enclosing def scope
    return f2       # Return f2 but don't call it
action = f1()       # Make, return function
action()            # Call it now: prints 88
```

- Another example : *execute this in the* ***idle debugger*** *to see the flow*

```
def f1() :
    x = [1, 2]; print "in f1"
    def f2() :
        x = 'asdf'; print "in f2"
        def f3() :
            print "in f3"; print x
        return f3
    return f2
f = f1()
print '11111111'
f = f()
print '22222222'
f()
```

```
in f1
11111111
in f2
22222222
in f3
asdf
```

Factory functions : Closures

◆The function object remembers values in enclosing scopes regardless of whether those scopes are still present in memory – the enclosing local scope is retained as state information

Example :

```
def creator(n) :
      def action(x) :      # Make and return 'action'
         return x**n       # 'action' retains 'n' from enclosing scope
      return action

f = creator(2)             # for 'f', 'n' takes value 2
print f(3)                 # value 3 is passed to 'x', prints 3**2

g = creator(4)             # for 'g', 'n' takes value 4
print g(10)                # prints 10**4
print f(10)                # prints 10**2
```

- the function `creator` makes `action` and simply returns action without running it
- both `f` and `g` reference the generated nested function `action` but with different values of *n*
- this *n* is retained as **state information** attached to the generated `action`
- thus `f` and `g` are references to functions with different state information stored

First Class Functions

- First class objects in a language are handled uniformly throughout
  - They may be stored in data structures, passed as arguments, or used in control structures
  - A programming language is said to support first-class functions if it treats functions as first-class objects
  - Python supports the concept of First Class functions

- Properties of First Class functions :
  - A function is an instance of the Object type
  - A function can be stored in a variable
  - A function can be passed as a parameter to another function
  - A function can be returned from a function
  - A function can be stored in data structures such as lists, …

Argument Passing

- Arguments are passed by automatically assigning objects to local variable names
- Assigning to argument names inside a function does not affect the caller
- Changing a mutable object argument in a function may impact the caller
- Immutable arguments are effectively passed **by value** – *because immutable objects cannot be changed **in place***
- Mutable atguments are effectively passed **by pointer** – because mutable objects can be changed in place in the function

Example :

```
def change1(a, b) :
    a = 'abcd'           # 'a' receives an immutable object
    b[0] = 'qwerty'      # 'b' receives a list object
                     # list is mutable
                     # it can undergo an in place change
def change2(a, b) :
    a = 'qwer'
    b = 'zxc'            # 'b' now is made to refer to new
                     # object and therefore argument 'y'
                     # is not changed

x = 10
y = [1, 2, 3, 4]
change1(x, y)
print "after change 1 :::", x, y
change2(x, y)
print "after change 2 :::", x, y

after change 1 ::: 10 ['qwerty', 2, 3, 4]
after change 2 ::: 10 ['qwerty', 2, 3, 4]
```

Example : ***Avoiding mutable argument changes*** **-** *method#1*

```
def change1(a, b) :
    a = 'abcd'
    b[0] = 'qwerty'

x = 10
y = [1, 2, 3, 4]
change1(x, y[:]) # create explicit copy of mutable object
             # 'y' to the function
             # now 'b' in 'change1' refers to a
             # different object which is initially a
             # copy of 'y'

print "after change 1 :::", x, y
```

```
after change 1 ::: 10 [1, 2, 3, 4]
```

<u>Example</u> : *Avoiding mutable argument changes* – *method#2*

Define the function such that it starts with creating a copy of the argument 'b' and then works on this 'b' :

```
def change1(a, b) :
 b = b[:] # Copy input list so as not to impact caller
    a = 'abcd'
    b[0] = 'qwerty'
```

<u>Example</u> : *Avoiding mutable argument changes* – *method#3*

Do not pass a mutable object at all to the function – convert it to immutable object and pass this immutable object to the function :

```
def change1(a, b) :
    a = 'abcd'
    b[0] = 'qwerty'

x = 10 ; y = [1, 2, 3, 4]
change1(x,tuple(y))# Pass a tuple, so changes become errors
print "after change 1 :::", x, y
```

Return Values from functions

- Function can return value by use of `return` statement
- To return multiple values, function may package them in some collection, eg, a tuple
  - this feature can be used to simulate **call-by-reference** feature by returning tuples and assigning the results back to the original argument names in the caller
  - Example :

```
def changeMany(a, b) :
      a = 'abcd'
   b = 'qwerty'
   return a, b      # return multiple values in a tuple

x = 10
y = [1, 2, 3, 4]
x, y = changeMany(x, y)
print "after change 1 :::", x, y

after change 1 ::: abcd qwerty
```

Argument Matching Rules

- By default, arguments are matched by position, from left to right – The number of arguments passed must be exactly as many arguments as there are argument names in the function header

- Callers can specify which argument in the function is to receive a value by using the argument's name in the call, with the `name=value` syntax

Example :

```
def someFunc(a, b) :
      print a, b
x = 10;  y = [1, 2, 3, 4]
someFunc(b=x, a=y)
```

◆A function argument can be created with a default value – if the function is called without passing this argument then the argumnet takes this default value (a non-default argument CAN NOT follow a default argument)

Example :

```
def f(x=10) :
      print (x)
f(100)        # 'x' takes value 100
f()              # 'x' takes default value 10
```

Another example :

```
def someFunc(a, b=88, c=90) : print (a, b, c)

x = 'a'
y = 'b'
someFunc(b=x, a=y)
someFunc(y)      # 'y' recvd in 'a'
             # 'b' and 'c' take default values
someFunc(x, y)   # only 'c' takes default value

b a 90
b 88 90
a b 90
```

◆**Varargs collecting :** Functions can use special arguments preceded with `*` or `**` characters to collect an arbitrary number of possibly extra arguments – this feature is referred to as *varargs*

Example #1 : use of * *(a non-starred arg CAN NOT be added after a starred arg)*

not required in Python3

All the positional arguments are collected into a new tuple :

like any tuple object it can be indexed, stepped through with a for loop, etc

```
def func(*aaa) : print aaa    # 'aaa' is a tuple
func(1, 'www', 32)
(1, 'www', 32)
```

Python3 accepts the following :

```
def f(a1, a2, *a, a3, a4=1) : # 'a3' & 'a4' can't be filled by
                              # positional argument, must be passed
                              # using name=value
   print('a1->', a1) ; print('a2->', a2) ; print('a->', a)
   print('a3->', a3) ; print('a4->', a4)
f(1,2, 20,30, a3=90)
a1-> 1
a2-> 2
a-> (20, 30)
a3-> 90
a4-> 1
```

Example #2 : use of ** *(a double-starred arg CAN NOT be followed by any other arg)*

The arguments are collected in a dictionary object :

```
def f(**aaa) : print aaa      # 'aaa' ia a dictionary
f(z=1, s='www', c=32)
{'s': 'www', 'z': 1, 'c': 32}
```

Example #3 : use of * as well as **

combining normal arguments, the *, and the ** :

```
def func(a1,a2,*aaa,**bbb): # 'func' requires at least two args
    print a1, a2, aaa, bbb
func(1, 'www', 32)
func(1, 'www', 32, 'asd', [1,2], f1='one', f2='two')
func(1, 2)
```

◆**Varargs unpacking :** Callers can use the * syntax to unpack argument collections into separate arguments and ** syntax to unpack to form dictionary

Example #1 : use of *

The no of arguments in the collection passed *must be* the same as the no of arguments required by the function :

```
def func(a1, a2, a3, a4) : print a1, a2, a3, a4
args = [10, 2, -20, 6]; func(*args)            #'args' is a list
args = ('qwe', 66, [1, 2], 'aa'); func(*args) #'args' is a tuple
10 2 -20 6
qwe 66 [1, 2] aa
```

```
def f() :
  print('this returns multiple objects'); return 10, 'str', [1,2,3]
def g(a, b, c) :
  print('this recvs multiple objects'); print(a, b, c)
res = f() ; g(*res)
```

Example #2 : use of **

The ** syntax in a function call unpacks a dictionary of key/value pairs into separate keyword arguments

the dictionary keys *must* match the function argument names

```
def func(a1, a2, a3, a4) : print a1, a2, a3, a4
args = {'a1': 1, 'a2': 2, 'a3': 3, 'a4':4}
func(**args)
1 2 3 4
```

Example #3 : use of * as well as **

```
func(*(1, 2), **{'d': 4, 'c': 3}) # Same as func(1, 2, d=4, c=3)
func(1, *(2, 3), **{'d': 4})      # Same as func(1, 2, 3, d=4)
func(1, c=3, *(2,), **{'d': 4})   # Same as func(1, 2, c=3, d=4)
```

◆**Keyword only arguments :** In Python 3.X (but not 2.X), functions can also specify arguments that must be passed by name with keyword arguments, not by position

Example :

```
def f(a, b, *, c) :
''' '*' indicates the end of positional arguments
and the beginning of keyword-only arguments '''
      print (a)
      print (b)
      print (c)

f(10, 20, c=9)
f(10, 20, 9) # error
```

Keyword arguments and their default values work as expected

```
def f(a, b, *, c=100, d) :
   '''can not be called without specifying
   keyword-argument 'd' '''
   print (a,b,c,d)
f(10, 20, c=9, d=100)
print('******************')
f(10, 20, d=100)
```

OUTPUT :

```
10 20 9 100
****************
10 20 100 100
```

A general function call example :

```
def findMin(*args) :
    min = args[0]
    print min
    for i in args[1:] :
        if i < min :
            min = i
    print min

findMin(10, -1, 9, -30, 4, 5, -90, 2, 7)

10
-90
```

A recursive function example :

```
def fact(args) :
    if(args != 1) :
        return args * fact(args-1)
    else :
        return 1

print fact(5)
```

Functions as Objects

- Python functions are objects :
  - function objects may be assigned to other names

```
def someFunc(a, b) : print 'someFunc called'
myFunc = someFunc
myFunc(10, 'qq')
```

  - they can be passed to other functions

```
def f1(a, b) : print 'f1 called'
def f2(fun, x, y) : fun(x, y)
f2(f1, 'hello', 10)
```

  - they can be embedded in data structures

```
def f1(a) : print 'f1', a
def f2(a) : print 'f2', a
listOfFuncs = [(f1, "good morning"), (f2, "bye")]
for (f, arg) in listOfFuncs : f(arg)
```

  - they can be returned from one function to another

```
def varFunc(value) :
    def f1() : print 'f1 called'
    def f2() : print 'f2 called'
    def f3() : print 'f3 called'
    if value == 1 : return f1
    elif value == 2 : return f2
    else : return f3
f = varFunc(2) ; f()
```

Anonymous Functions : *lambdas*

- Lambda is a nameless functions
- It provides an expression form that generates function objects
- This expression form creates a function and returns the function to be called later
- It has the following syntax :

```
lambda arg1, arg2,... argN : expression using arguments
```

- The lambda's body is similar to what you'd put in a def body's return statement – the result is typed as an expression, instead of explicitly returning it
- Lambda can appear in places a def is not allowed – inside a list literal or a function call's arguments, etc

lambda inside a list :

```
l = [lambda a:a**2, lambda a,b:a*b, lambda a:a**4]
print l[0](10), l[1](2, 20), l[2](3)
```

lambda inside a dictionary :

```
key = 'a'
d = {'a': lambda x:2*x, 'b': lambda x:3*x}
print d[key](9)
```

lambda functions can reference variables from the containing scope :

```
a = [10]
l = lambda a1,a2 : a.pop()+a1+a2
print(l(1,2))  # outut is 13
```

Certain Functional Programming Tools

- Python provides certain tools that apply functions to sequences and other iterables
- This set includes tools that :
  - call functions on an iterable’s items (map)
  - filter out items based on a test function (filter)
  - apply functions to pairs of items and running results (reduce)

Mapping Functions over Iterables : *map*

- `map` helps to easily apply an operation on a collection
- It can be invoked as follows :

  `map(function, iterable, ...)`
- It applies function to every item of *iterable* (and returns a list of the results in 2.7)

  refer https://docs.python.org/2/library/functions.html#map for more details
- Example :

```
def f(a,b) : return 20+a+b
def g(a) : return a*a

print map(None, [1,2,3,4], [11, 22, 33, 44])
print map(lambda x,y:x+y, [1,2,3,4], [11, 22, 33, 44])
print map(g, [1,2,3,4])
print (list(map(f, [1,2,3,4], [10,20,30,40])))
```

Selecting Items in Iterables : *filter*

- `filter` selects an iterable's items based on a test function
- It can be invoked as follows :

  `filter(function, iterable)`

- It constructs a list from those elements of iterable for which function returns true
- Example :

```
def f(a) :
    if a%2 == 0 :
    return True
    else :
    return False

print filter((lambda x: x > 0), range(-5, 5))
print filter(f, range(-5, 5))
```

Combining Items in Iterables : *reduce*

- `reduce` processes an iterable and produces a single result (*not another iterable*)
- It applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value
- It can be invoked as follows :
  `reduce(function, iterable[, initializer]`
- The left argument of the function is an accumulated value as generated after applying the function and the right argument is the update value from the iterable
- Example :

```
def f(a,b) : return a*b

print functools.reduce((lambda x, y: x + y), [1, 2, 3, 4])
print functools.reduce(f, [1, 2, 3, 4])
```

List Comprehensions vs *map* vs a 'for' loop

```
#with a for loop
res = []
for x in 'basic':
    res.append(ord(x))
print res

#using map
print map(ord, 'basic')

#using comprehension
print [ord(x) for x in 'basic']
```

zip()

```
>>> l1 = [1,2,3,4]
>>> l2 = [11,22,33]
>>> list(zip(l1, l2))
[(1, 11), (2, 22), (3, 33)]
```

enumerate()

```
>>> l1 = [1,2,3,4]
>>> list(enumerate(l1))
[(0, 1), (1, 2), (2, 3), (3, 4)]
```

String formatting …

String format operator %

◆Similar to C printf

string % values

<u>Example</u> :

```
>>> name = 'Manoj Kumar'
>>> t = delay = 10
>>> print "Good morning Mr. %s. You are late by %d minutes!!!" % (name, delay)
Good morning Mr. Manoj Kumar. You are late by 10 minutes!!!
>>>
>>>
>>> char = 'a'
>>> no = 100
>>> fraction = 12.7
>>> print "The char is %c, number in hex is %x, float no is %f, its exponential notation is %e" %(char, no, fraction, fraction)
The char is a, number in hex is 64, float no is 12.700000, its exponential notation is 1.270000e+01
>>>
```

If format requires a single argument : values may be a single non-tuple object

```
>>> no = 10
>>> print 'string has a single arg, %d' % no
string has a single arg, 10
>>> print 'string has a single arg, %d' % (no,)
string has a single arg, 10
```

If format requires multiple arguments : values must be a tuple with exactly the number of items specified by the format string

```
>>> no = 10
>>> str = 'SOME STR'
>>> f = 12.345
>>> print 'str with multiple args, a no %d, a string %s, a float %f' % (no, str, f)
str with multiple args, a no 10, a string SOME STR, a float 12.345000
```

Values can be a single mapping object (example, a dictionary)

```
>>> print 'str with multiple args, a no %(no)d a string %(ss)s and a float %(ff)E' % {"no":100,  'ss' : 'SOME STR', 'ff' : 12.232}
str with multiple args, a no 100 a string SOME STR and a float 1.223200E+01
```

The Pythonic way
using the `str.format()` method

◆**Usage :** *`mystr.format(*args, **kwargs)`*

{} is the placeholder for the substituted variables

If no format is specified, it will insert and format as a string

The placeholder can be used with numeric position of the variables

this gives some flexibility when doing formatting

if you made a mistake in the order you can easily correct without shuffling all variables around

```
print 'Some {1} string. And here is more {0}'.format('garbage', 'non-sense')
Some non-sense string. And here is more garbage
```

Named arguments can be used

```
print "I {verb} the {object} off the {place} ".format(verb="took", object="cheese", place="table")
I took the cheese off the table
```

Same variable can be used multiple times

```
print "A {0} {0} good morning to all".format('very')
A very very good morning to all
```

Use *format* as a function

```
# defining formats
email_f = "Your email address was {email}".format

# use elsewhere
print(email_f(email="bob@example.com"))

Your email address was bob@example.com
```

MODULES

Modules – detailed understanding

- Modules :
  - are the highest-level program organization unit
  - package program code and data for reuse
  - provide self-contained namespaces that minimize variable name clashes across programs
- Modules might also correspond to extensions coded in external languages such as C, Java, or C#, and even to directories in package imports
- Modules are processed with two statements and one important function :
  - `import` : Lets a client (importer) fetch a module as a whole
  - `from` : Allows clients to fetch particular names from a module
  - `reload` (`imp.reload in 3.X`) : Provides a way to reload a module's code without stopping Python
- `import` statements are executed at runtime
- `import` runs statements in the target file one at a time to create its contents

- There is a large collection of utility modules known as the *standard library*
- This collection contains platform-independent support for common programming tasks : operating system interfaces, object persistence, text pattern matching, network and Internet scripting, GUI construction etc
- None of these tools are part of the Python language itself, but they can be used by importing the appropriate modules on any standard Python installation

The `import` statement

- It is a runtime operation that performs three distinct steps the **first time** a program imports a given file
  - ***find*** the module's file
  - ***compile*** it to byte code (if needed)
  - ***run*** the module's code to build the objects it defines

- Later imports of the same module in a program run bypass all of these steps and simply fetch the already loaded module object in memory

- This is done by storing loaded modules in a table named `sys.modules` and checking there at the start of an import operation.
  - if the module is not present, the three-step process begins

Finding file

- To locate the module file corresponding to an `import` statement :
  - a standard module search path is used
  - known file types are used

<u>Module search path</u> :

- The module search path is contained in `sys.path`, a mutable list of directory name strings
- It is concatenation of the following components :
  - ***The home directory of the program***
    - this is the directory containing your program's top-level script file
    - because this directory is searched first, its files will override modules of the same name in directories elsewhere on the path – even library modules
  - ***PYTHONPATH directories*** (if set)
    - next all directories listed in PYTHONPATH environment variable setting are searched from left to right
    - this is a list of user-defined and platform-specific names of directories that contain Python code files
    - all the directories from which import must be done can be added here
    - Python will extend the module search path to include all the directories PYTHONPATH lists

- ***Standard library directories***
  - next the directories where the standard library modules are installed are searched
- ***The contents of any .pth files*** (if present)
  - directories can be added to the module search path by simply listing them, one per line, in a text file whose name ends with a .pth suffix (for "path")
  - on Unix-like systems, this file might be located in usr/local/lib/python3.3/site-packages or /usr/local/lib/site-python
- The site-packages is home of third-party extensions

<u>Module file selection</u> :

- Fillename extensions are omitted from `import` statements
- Python chooses the first file it can find on the search path that matches the imported name
- Imports are the point of interface to a host of external components – *source code*, *multiple flavors of byte code*, *compiled extensions*, etc
- Python automatically selects any type that matches a module's name

- An `import` statement of the form `import b` might resolve to :
  - a source code file named b.py
  - a byte code file named b.pyc
  - an optimized byte code file named b.pyo (a less common format)
  - a directory named b, for package imports
  - a compiled extension module, coded in C, C++, or another language, and dynamically linked when imported (e.g., b.so on Linux)
  - a compiled built-in module coded in C and statically linked into Python
  - a ZIP file component that is automatically extracted when imported
  - an in-memory image, for frozen executables
  - a Java class, in the Jython version of Python
  - a .NET component, in the IronPython version of Python
- In case two files, say b.py and b.so are both found in the same location, Python follows a standard picking order – this order is not guaranteed to stay the same over time or across implementations

Compile file

- After finding a source code file that matches an `import` statement by traversing the module search path, Python next compiles it to byte code (if necessary)

Run

- The final step of an import operation executes the byte code of the module.
- All statements in the file are run in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object
- Future imports skip all three import steps and reuse the already loaded module in memory

Module Filenames

- The modules that need to be imported should be provided a .py extension
- The .py is technically optional for top-level files that will be run but not imported
- Module names become variable names inside a Python program (without the .py)

The `import` statement :

- The module name in the `import` statement serves two purposes :
  - it identifies an external file to be loaded
  - it becomes a variable in the script, which references the module object after the file is loaded
- The `import` statement simply lists one or more names of modules to be loaded separated by commas
- Henceforth the module name must be used to fetch its attributes

The `from` statement :

- `from` copies specific names from one file over to another scope
- Thus these names can be used directly in the script without the module name
- To import multiple names from the same module, give a comma separated list
- To import all names from a module :

```
from math import *
```

- Modules are loaded and run on the first `import` or `from`
  - later import operations simply fetch the already loaded module object
  - because top-level code in a module file is usually executed only once, it can be used to initialize variables
- `import` and `from` are implicit assignments
  - `import` assigns an entire module object to a single name
  - `from` assigns one or more names to objects of the same names in another module

mymath.py :

```
print "math loaded"; x=10; l = [1,2,3,4]
def factorial(i) :
       print x, l, "in fact" ; return i+10
```

mymain.py :

```
import mymath
print mymath.x, mymath.l, "in main#1"

mymath.x = 100;   mymath.l[0] = -90
print mymath.factorial(5)
print mymath.x, mymath.l, "in main#2"

mymath.l = [11,22]
print mymath.factorial(5)
print mymath.x, mymath.l, "in main#3"
```

```
math loaded
10 [1, 2, 3, 4] in main#1
100 [-90, 2, 3, 4] in fact
15
100 [-90, 2, 3, 4] in main#2
100 [11, 22] in fact
15
100 [11, 22] in main#3
```

mymath.py : *(same as previous example)*

```
print "math loaded"; x=10; l = [1,2,3,4]
def factorial(i) :
       print x, l, "in fact" ; return i+10
```

mymain.py :

```
from mymath import x, factorial, l
print x, l, "in main#1";print

x = 100; l[0] = -90 # Changes local x only, Changes shared mutable 'l' in place
factorial(5)
print x, l, "in main#3";print

l = [111,222]       # 'l' refers to new object
factorial(5)
print x, l, "in main#2"
```

```
math loaded
10 [1, 2, 3, 4] in main#1

10 [-90, 2, 3, 4] in fact
100 [-90, 2, 3, 4] in main#3

10 [-90, 2, 3, 4] in fact
100 [111, 222] in main#2
```

<u>Some more points</u> :

M.py

```
def func():
    ...do something...
```

N.py :

```
def func():
    ...do something else...
```

O.py

```
from M import func
from N import func # This overwrites the 'func' fetched from M
func()             # Calls N.func only
```

O.py

```
import M, N        # Get the whole modules, not their names
M.func()           # Can call both names now
N.func()           # The module names make them unique
***** OR ****
```

O.py

```
from M import func as mfunc   # Rename uniquely with "as"
from N import func as nfunc
mfunc(); nfunc()
```

Imports and Scopes

- `import` operations never give upward visibility to code in imported files – an imported file cannot see names in the importing file :
  - functions can never see names in other functions, unless they are physically enclosing
  - module code can never see names in other modules, unless they are explicitly imported

moda.py

```
X = 88
def f():
 global X    # Change this file's X
 X = 99
```

modb.py :

```
X = 11          # My X: global to this file only
import moda
moda.f()        # Gain access to names in moda
          # Sets moda.X, not this file's X
print(X, moda.X)
```

on executing `python modb.py`

```
11 99
```

Namespace Nesting

mod3.py

```
X = 3
```

mod2.py :

```
X = 2
import mod3
print(X, mod3.X)
```

mod1.py

```
X = 1
import mod2
print(X, mod2.X, mod2.mod3.X)
```

on executing `python mod1`.py

```
2 3
1 2 3
```

NOTE :

mod1 can say import mod2, and then mod2.mod3.X, but it cannot say import mod2.mod3 – this syntax invokes something called package (directory) imports

Reloading Modules

- Imports (via both import and from statements) load and run a module's code only the first time the module is imported in a process
- Later imports use the already loaded module object without reloading or rerunning the file's code
- The `reload` function forces an already loaded module's code to be reloaded and rerun
- Assignments in the file's new code change the existing module object in place
- `reload`:
  - is a function, not a statement
  - is passed an existing module object, not a new name
  - lives in the module `imp` in Python 3.X and must be imported itself

PACKAGES

Package Imports

- An `import` or `from` can name a directory path
- A directory of Python code is called a ***package*** – such imports are called ***package imports***
- Packages help to organize modules and resolve import ambiguities when multiple program files of the same name are installed on a single machine

Coding requirements for package imports

- Consider that the following packages need to be created :
  - **cmplx** containing module **cmplx.py**
  - **pack1.pack11** containing module **mod1.py**
  - **pack2** containing module **mod2.py**
  - **pack2.pack21.pack211** containing module **mod211.py**

- For this create the following directory structure :

```
testpack -------------------------------
     |             |              |
    cmplx         pack1          pack2--------------
    |             |                       |         |
cmplx.py          pack11                mod2.py   pack21
                  |                                 |
                  mod1.py                           mod211.py
```

- Each sub-directory named within the path of a package import statement must contain a file named `__init__.py`, or else package imports will fail
  - these files can contain Python code (just like normal module files)
  - their names are special because their code is run automatically the first time a Python program imports a directory, and thus serves as a hook for performing initialization steps required by the package
  - these files can also be completely empty

- The directory where these package directories have been created (*testpack* in this example) must be an automatic path component – the home, libraries, or site-packages directories or be given in PYTHONPATH or .pth file settings package/

cmplx.py :

```
print 'cmplx loaded'
def add(c1, c2) :
    r = c1[0] + c2[0]
    i = c1[1] + c2[1]
    return r, i
```

mod1.py :

```
print 'mod1 loaded'
def mod1f1() :
    print 'mod1 f1'
def mod1f2() :
    print 'mod1 f2'
```

mod2.py :

```
print 'mod2 loaded'
def mod2f1() :
    print 'mod2 f1'
def mod2f2() :
    print 'mod2 f2'
```

mod211.py :

```
print 'mod211 loaded'
def mod211f1() :
    print 'mod211 f1'
```

packagetest.py :

```
import cmplx.cmplx
import pack1.pack11.mod1
import pack2.mod2
import pack2.pack21.pack211.mod211


cmp1 = (1, -1)
print cmp1[0]
cmp2 = (10, 2)


print cmplx.cmplx.add(cmp1, cmp2)
pack1.pack11.mod1.mod1f1()
pack1.pack11.mod1.mod1f2()
print pack1.pack11.pack11var, 'pack11var'
pack2.mod2.mod2f1()
pack2.mod2.mod2f2()
pack2.pack21.pack211.mod211.mod211f1()
```

pack1/pack11/__init__.py :

```
pack11var=10
```

An example with `from` for the same package structure :

packFrom.py :

```
from pack1.pack11.mod1 import mod1f1
mod1f1()
```

The __init__.py file

optional from Python 3.3 onwards

- This file plays the following role :
  - ***Package initialization :***
    - The first time a program imports through a directory, it automatically runs all the code in the directory's `__init__.py` file
    - thus these files contain code to initialize the state required by files in a package

  Example :

  to create required data files, open connections to databases
  - ***Module usability declarations :***
    - the presence of these files in a directory indicates that this directory is a Python package
  - ***Module namespace initialization :***
    - these files provide a namespace for module objects created for directories, which would otherwise have no real associated module file
    - thus after *import* `pack1.pack11.mod1` works, a module object is returned whose namespace contains all names assigned to *pack11/__init__.py*

➢ *`from *` statement behavior :*

- by default this statement imports all attributes of the specified module file object
- it also loads names defined by assignments in the directory's `__init__.py` file
- it loads any submodules explicitly imported by code in this file, eg, the statement *`from submodule import X`* in a directory's `__init__.py` makes the name `X` available in that directory's namespace

<u>Example</u> :

```
pack
    m1.py
        from pack import *
        print 'm1 imported-->', subm1.x
    __init__.py
        from subpack import *
        print "package init", subm1.x
    subpack
        subm1.py
            print "subpack module"
            x = 10
        __init__.py
            __all__ = ['subm1']
```

```
testpack.py
import pack.m1
```

OUTPUT :

```
subpack module
package init 10
m1 imported--> 10
```

➔ this file can contain `__all__ list` to define what is exported when a directory is imported with the `from *` statement form – this is taken to be the list of submodule names that should be automatically imported when from * is used on the package (directory) name

only modules listed here will be imported, rest will not be imported; if list empty or undefined, nothing is imported

<u>Example</u> :

```
pack1
    mod1.py -> print 'mod1 imported'
    mod2.py -> print 'mod2 imported'
    mod3.py -> print 'mod3 imported'
    __init__.py
        __all__ = ['mod1', 'mod2']
```

```
testpack.py
from pack1 import *
```

```
OUTPUT :
mod1 imported
mod2 imported
```

More on modules ...

- ***You are always executing in a module :***
  - there's no way to write code that doesn't live in some module.
  - even code typed at the interactive prompt really goes in a built-in module called `__main__`
- ***Minimize module coupling: global variables***
  - like functions, modules work best if they're written to be closed boxes
  - modules should be as independent of global variables used within other modules as possible
  - the only things a module should share with the outside world are the tools it uses, and the tools it defines
- ***Maximize module cohesion: unified purpose***
  - let all the components of a module share a general purpose
  - then you're less likely to depend on external names

◆***Modules should rarely change other modules' variables :***

- it's perfectly OK to use globals defined in another module (that's how clients import services)
- but changing globals in another module is often a symptom of a design problem
- try to communicate results through devices such as function arguments and return values, not cross-module changes
- otherwise your globals' values become dependent on the order of arbitrarily remote assignments in other files, and your modules become harder to understand and reuse

Data hiding in modules

- In Python, data hiding in modules is a convention, not a syntactical constraint – there is no constraint imposed by the language, the programmars take up the responsibility
- In case of `from *` imports :
  - all names are copied out to the importer, **except** for the names which start with an underscore (e.g., _X)
  - Example :

modb.py :

```
print 'modb loaded'
modbX = 10
_modbX = 100
__modbX = 1000
def fb() :
    print 'f in modb'
```

moda.py :

```
import modb
print modb.modbX, modb._modbX, modb.__modbX
modb loaded
 10 100 100
```

moda.py :

```
from modb import *
print modbX
print _modbX        # error
print __modbX       # error
```

- A hiding effect for the `from *` import statement can also be achieved by assigning a list of variable name strings to the variable `__all__` at the top level of the module
  - this again only effects the `from *` imports and **does not effect** the `import mymod` statement
  - Example :

moda.py :

```
from modb import *
print modbX          # error
fb()
print _modbX
print __modbX
```

modb.py :

```
__all__ = ['_modbX', '__modbX', 'fb']
print 'modb loaded'
modbX = 10
_modbX = 100
__modbX = 1000
def fb() :
    print 'f in modb'
```

Executing modules as scripts

- `__name__` is a special built-in variable which evaluates to :
  - the name of the current module
  - `__main__` :- when module is run directly as script

- To know whether the script is executed directly as top-level script or imported as module, use the following :

```
def tool1() : pass
def tool2(a, b) : pass

if __name__ == "__main__" :
   print 'executing script as top-level script'
   tool1()
      tool2('abc', 1000)
else :
      print 'script imported'
```

DAY 3

Python Classes

defining classes, class object, instance object
empty class
inheritance – single, multiple
super
magic functions : *new, init, del, str, add etc*
instance methods / data
class methods / data
@staticmethod, @classmethod
abstract class
metaclass
isinstance, issubclass
decorators
properties

CLASSES

CLASSES

Classes provide with the following features :

- ***Multiple instances :***
  - they are factories for generating one or more objects
  - every time a class is called, a new object is generated with a distinct namespace
  - each object generated from a class has access to the class’s attributes and gets a namespace of its own data that varies per object
    - this is similar to the per-call state retention of closure functions
    - but this is explicit and natural in classes
- ***Customization via inheritance :***
  - a class can be extended by defining its attributes outside the class itself in new software components coded as subclasses
  - classes can build up namespace hierarchies which define names to be used by objects created from classes in the hierarchy

- ***Operator overloading :***
  - classes can define objects that respond to the sorts of operations that work on built-in types
  - eg, objects made with classes can be sliced, concatenated, indexed, and so on
  - classes can intercept and implement any built-in type operation

In the Python object model, classes and the instances generated from them are two distinct object types :

- *Classes :*
  - serve as instance factories
  - their attributes provide behavior – data and functions
  - this behavior is inherited by all the instances generated from them
  - **a class inherits attributes from all classes above it in the inheritance tree**
- *Instances :*
  - represent the concrete items in a program’s domain
  - their attributes record data that varies per specific object
  - **an instance inherits attributes from its class**

Python classes are truly dynamic in nature :

- they are created at run time
- they can be modified after creation
- classes themselves are objects

Defining a class

- The `class` starts with a header line that lists the class name, followed by a body of one or more nested statements
- If the nested statements are `def`s, they define functions that implement the behavior that the class means to export
- Functions inside a class are usually called *methods*
  - they're coded with normal `def`s
  - they support everything that functions do (they can return values, yield items on request, etc)
  - in a method function, the first argument automatically receives an implied instance object when called – the subject of the call

Example :

```
class emp :
    def initdata(self, val) : self.data = val
    def disp(self) : print self.data


emp1 = emp();emp1.initdata("aaa")
emp2 = emp();emp2.initdata(1234)
emp1.disp();emp2.disp()
```

- With respect to the last example :
  - within a method, `self` (the name given to the leftmost argument by convention – could have used some other name), automatically refers to the instance being processed
  - the assignments to `self.data` store values in the instances' namespaces, not the class's
  - different object types have been passed to the `data` member in each instance
  - as with everything else in Python, there are no declarations for instance attributes (sometimes called *members*) – they spring into existence the first time they are assigned values
    - had `display` been called before calling `setdata`, an undefined name error would have been generated
  - instance attributes can be assigned values outside the class also as follows :

```
emp1.data = 'dep1'
```

  - or a new instance attribute can be created as follows :

```
emp1.newattr = 'someval'
```

A more complete example :

```
class x :
        varX = 123
    def disp(self) :
        print self.x

o = x()
o.x = 10            # statement1
o.disp()            # statement2
```

This class defines two attribute references :
x.varX – returning an integer
x.disp – returning a function object

'statement2' can not be executed before 'statement1'

Defining object of a class :

```
obj = ClassX()
```

This creates the object and executes the class constructor, the `__init__` function

Arguments can be passed while defining / creating objects, provided a matching constructor is available

Here *ClassX()* creates the object and *obj* is a reference to that object

◆A class definition executes the class

src.py

```
class X :
    print 'class statement'
```

python src.py

```
class statement
```

◆A class definition can be placed in a branch of an if statement, or inside a function

◆Class definitions, like function definitions (*def* statements) must be executed before they have any effect

```
a=10
if a>1 :
     class X : print 'class X statement'
else :
     class Y : print 'class Y statement'

X()          # creates class X instance object
# Y() # generates error : name 'Y' is not defined
```

Output :

```
class X statement
```

- When a class definition is entered, a new namespace is created, and used as the local scope
  - all assignments to local variables go into this new namespace
  - function definitions bind the name of the new function to this namespace
- When a class definition ends :
  - *class object* is created (not object of class instance)
  - the original local scope (the one in effect just before the class definition was entered) is reinstated
  - the class object is bound here to the class name given in the class definition header
- Every class instance has a built–in attribute, `__class__`, which is the object's class

```
class C : pass
o = C()
print o.__class__
```

```
__main__.C
```

Class Object – *provide default behavior*

- ***The `class` statement creates a class object and assigns it a name***
  - like the function `def` statement, the `class` statement is an executable statement
  - when reached and run, it generates a new class object and assigns it to the name in the class header
  - like `def`s, `class` statements run when the files they are coded in are first imported
- ***Assignments inside class statements make class attributes***
  - like in module files, top-level assignments within a `class` statement (not nested in a `def`) generate attributes in a class object
  - the `class` statement defines a local scope that is the attribute namespace of the class object – like a module's global scope
  - after running a `class` statement, class attributes are accessed by name qualification : *object.name*
- ***Class attributes provide object state and behavior***
  - attributes of a class object record state information and behavior to be shared by all instances created from the class
  - function `def` statements nested inside a class generate methods, which process instances

- ***Class objects support two kinds of operations :***
  - attribute references
    - use the standard *obj.name* syntax

  and

  - instantiation -
    - uses function notation
    - creates a new instance of the class

Instance Objects – *are concrete objects*

- ***Calling a class object like a function makes a new instance object***
  - each time a class is called, it creates and returns a new instance object
  - instances represent concrete items in your program's domain
- ***Each instance object inherits class attributes and gets its own namespace***
  - instance objects created from classes are new namespaces
  - they start out empty but inherit attributes that live in the class objects from which they were generated
- ***Assignments to attributes of `self` in methods make per-instance attributes***
  - inside a class's method functions, the first argument (called `self` by convention) references the instance object being processed
  - assignments to attributes of `self` create or change data in the instance, not the class

- ***Instance objects suport only one operation :***
  - attribute reference – these can be
    - data attributes
    - methods
- ***Data attributes are referenced as :***
  - `instance.attr` outside the class

  and
  - `self.attr` within the class

Data attributes *vs* Class attributes

- Data attributes : are variables owned by a specific instance of a class
  - they are created on an per instance basis
- Class attributes : are variables owned by the class itself
  - they are available before any instance of a class are created

Example :

```
class C :
    classAttr = 'this is class attr'
    def __init__(self) :
        self.dataAttr = 'this is data attr'
```

Class – special attributes

- Following are spcially named class attributes that Python will invoke automatically :
  - `__init__` is run when a new instance object is created : **Constructor**
  - `__add__` is run when a class instance appears in a + expression
  - `__str__` is run when an object is printed (when it's converted to its print string by the str built-in function or its Python internals equivalent)

Example :

```
class emp :
    def __init__(self, val1, val2) :
        print 'init called'
        self.m1 = val1; self.m2 = val2
    def __add__(self, oth) :
        print 'add called'
        return emp(self.m1+oth.m1, self.m2+oth.m2)
    def __str__(self) :
        print 'str called'
        return 'emp is %s %s' % (self.m1, self.m2)
    def disp(selfie) : print selfie.m1, selfie.m2
empObj1 = emp('abc', 12); empObj2 = emp('qwe', -12)
print 'empObj1 ::', empObj1
newemp = empObj1 + empObj2
print 'newemp ::', newemp
```

The `__init__` function : constructor

- This is like any other function except that it is invoked automatically whenever an instance of the class is created
- It can be created with arguments having default values
- `__init__` method can not return a value
- It can be invoked explicitly (as shown in the following code) – but it is not done :

```
class C :
    def __init__(self) : print 'in init'
o = C()     # automatic invocation
print '$$$$$$$$'
o.__init__()      # explicit invocation
```

The `__call__` function :

- This function is called when the function call operator is applied to an instance of a class

```
class X :
   def __init__(s) : print ('init called')
   def __call__(s) : print ('call called')
   def f(s) : print ('f called')

print ('111111')
x = X()
print ('222222')
x()
print ('333333')
x.f()
print ('444444')
```

Output :

```
111111
init called
222222
call called
333333
f called
444444
```

The `__class__` attribute

◆ `__class__` is a built–in attribute available to every class instance

◆ It is a reference to the class just as `self` is a reference to an instance

◆ Using this attribute, an instance can access the class namespace dictionary (more on this later)

◆ Ecample :

```
class C :
  x = 'class attr'
o = C()
```

Now : ***`o.__class__.__dict__`*** is same as ***`C.__dict__`***
And : ***`o.__class__.x`*** is the class attribute ***`C.x`***

A class with no attributes

- Following defines a class with no attributes :

```
class record : pass
```

- Attributes can be attached to this class by assigning names to it completely outside of the original `class` statement :

```
record.name = 'abc'
record.age = 35
print record.name
abc
```

- When instances to such a class are created :
  - these instances begin their lives as completely empty namespace objects
  - because they remember the class from which they were made, they will obtain the attributes we attached to the class by inheritance
  - Thus :

```
rec1 = record(); rec2 = record
print rec1.name, rec2.name
abc abc
```

A complete code :

```
class emp : pass

emp.name = 'aaa'
emp.age = 11

emp1 = emp()
emp2 = emp()
print 'emp1 name :::', emp1.name
print 'emp2 name :::', emp2.name

emp1.name = 'newname'
print emp.name, emp1.name, emp2.name

emp.name = 'xxx'
print emp.name, emp1.name, emp2.name
```

```
emp1 name ::: aaa
emp2 name ::: aaa
aaa newname aaa
xxx newname xxx
```

Another example :

```
class C :
    cAttr = 'class attr'
    def __init__(myself, a, b) : # constructor
        myself.objAttr1 = a
        myself.objAttr2 = b
        print 'constr'
    def disp(myself) :
        print 'obj attr1 ::: ', myself.objAttr1
        print 'obj attr2 ::: ', myself.objAttr2
        #print cAttr
        #error - tries to access a 'cAttr' from local scope
        myself.cAttr = 'my attr'
        print 'class attr ::: ', C.cAttr
        print 'obj attr ::: ', myself.cAttr

print 'creating class instance...'
obj = C(12, [1,2,3,4])

print 'calling disp func on instance...'
obj.disp()
```

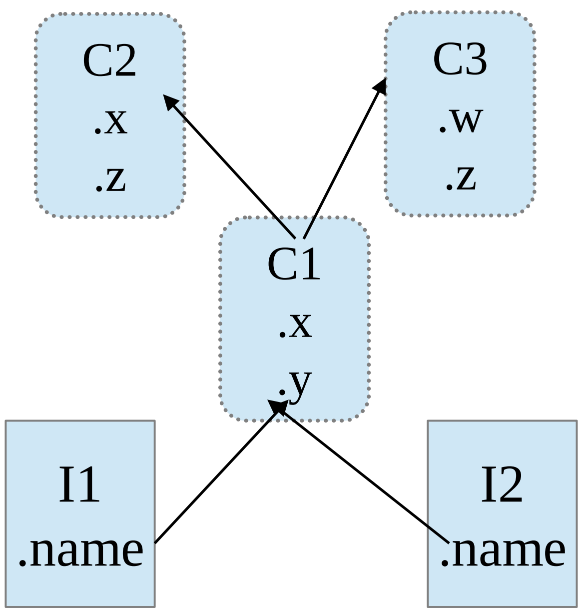
```
creating class instance...
constr
calling disp func on instance...
obj attr1 :::  12
obj attr2 :::  [1, 2, 3, 4]
class attr :::  class attr
obj attr :::  my attr
```

Attribute Inheritance Search

- *object.attribute* calls for a lookup for the first appearance of attribute that can be found
- The following rule is followed :

Find the first occurrence of attribute by looking in object, then in all classes above it, from bottom to top and left to right

Example :



- this tree links together three class objects and two instance objects into an inheritance search tree
- invoking I2.w, invokes inheritance
- the linked objects in the tree are searched in the following order, stopping when found first : I2, C1, C2, C3
- thus :
  - I1.x and I2.x both find x in C1 and stop because C1 is lower than C2
  - I1.y and I2.y both find y in C1 because that's the only place y appears
  - I1.z and I2.z both find z in C2 because C2 is further to the left than C3
  - I2.name finds name in I2 without climbing tree at all

```
class c2par :
    w='c2par w'

class c2(c2par) :
    x='c2 x'
    z='c2 z'

class c3 :
    w='c3 w'
    z='c3 z'

class c1(c2, c3) :
    x = 'c1 x'
    y = 'c1 y'

i1 = c1()
print i1.w
```

Output :

```
c2par w
```

```
class c2 :
    x='c2 x'
    z='c2 z'

class c3 :
    w='c3 w'
    z='c3 z'

class c1(c2, c3) :
    x = 'c1 x'
    y = 'c1 y'

i1 = c1()
print i1.w

c1.a = 90
print i1.a
```

Output :

```
c3 w
90
```

A word about Python namespaces

- Python classes and instances of classes each have their own distinct namespaces represented by pre-defined attributes :
  - `MyClass.__dict__`

  and
  - `instance_of_MyClass.__dict__`

- When an attribute is accessed from an instance of a class, it first looks at its instance namespace

- If it finds the attribute, it returns the associated value

- If not, it then looks in the class namespace and returns the attribute (if it's present, throwing an error otherwise)

- The instance namespace takes supremacy over the class namespace: if there is an attribute with the same name in both, the instance namespace will be checked first and its value returned.

Python stores class and instance attributes in separate dictionaries

- Consider the following code :

```
class C :
     a = 12
     def f(self) :
            self.b = 'aaa'

print C.__dict__   # prints class dictionary
obj = C()
print obj.__dict__# prints instance dictionary - empty
obj.f()
print obj.__dict__# prints instance dictionary -
                                      # contains attribute 'b'
```

```
{'a': 12, '__module__': '__main__', '__doc__': None, 'f':
 <function f at 0xb724a144>}
{}
{'b': 'aaa'}
```

`obj.__class__.__dict__` is same as `C.__dict__`

Instance attribute hides class attribute :

```
class C :
    attr = 'class attr'
    def f1(self) :
        attr = '10'                                  # local scope
        self.attr = 'obj attr'        # hides class attribute

print C.attr
obj = C()
obj.f1()

print '############'
print 'C.attr ::: ', C.attr
print 'obj.attr :::', obj.attr
print 'accessing class attribute for object', obj.__class__.attr
```

```
class attr
############
C.attr :::  class attr
obj.attr ::: obj attr
accessing class attribute for object
class attr
```

Class attribute is shared by all instances :

```
class Pet :
  kind = 'dog'                # class variable shared by all instances
  def __init__(self, name) :
    self.name=name     # instance variable unique to each instance

pet1 = Pet('mypet')
pet2 = Pet('yourpet')
print 'pet1 kind ::: ', pet1.kind
print 'pet2 kind ::: ', pet2.kind
print 'pet1 name ::: ', pet1.name
print 'pet2 name ::: ', pet2.name

Pet.kind = 'cat'
print 'changed Pet.kind to cat'
print 'pet1 kind ::: ', pet1.kind
print 'pet2 kind ::: ', pet2.kind
```

```
pet1 kind :::  dog
pet2 kind :::  dog
pet1 name :::  mypet
pet2 name :::  yourpet
changed Pet.kind to cat
pet1 kind :::  cat
pet2 kind :::  cat
```

When *instance.attr* is invoked :

- python first searches for *attr* in *instance*
- if found, returns this
- if not found, searches for *attr* in *class*

Default behavior for attribute access is to *get*, *set*, or *delete* the attribute from object's dictionary.
Example : `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses

Verify the following by reading the __*dict*__ of Pet, pet1 and pet2

`Pet.kind`

changes class attribute

`pet1.__class__.kind`

changes class attribute

`pet1.kind = 'newkind'`

creates data attribute `kind` for `pet1`

Dealing with shared class attribute that is mutable in nature :

```
class Pet :
    kind = 'dog'     # class attribute shared by all instances
    tricks = []      # class attribute which is a mutable object
    def __init__(self, name) : self.name = name
               # instance attribute unique to each instance
    def changePet(self, newPet) : self.kind = newPet
    def addTricks(self, tricks) : print 'in addTricks', self.tricks
        self.tricks.append(tricks)
        #self.tricks = [tricks]
        #self.tricks += [tricks]
        #self.tricks = self.tricks + [tricks]
pet1 = Pet('mypet');pet2 = Pet('yourpet')
print 'pet1 kind:::', pet1.kind; print 'pet2 kind:::', pet2.kind
print 'pet1 name:::', pet1.name; print 'pet2 name:::', pet2.name

Pet.kind = 'cat';print 'changed Pet.kind to cat'
print 'pet1 kind:::', pet1.kind; print 'pet2 kind:::', pet2.kind
#changing pet1 kind does not change pet2 kind
pet1.changePet('parrot'); print 'changed pet1.kind to parrot'
print 'pet1 kind:::', pet1.kind; print 'pet2 kind:::', pet2.kind

pet1.addTricks('imitate'); pet1.addTricks('laugh')
pet2.addTricks('roll over')
print 'pet1 tricks:',pet1.tricks;print 'pet2 tricks:',pet2.tricks
print Pet.__dict__;print pet1.__dict__;print pet2.__dict__
```

Uncomment these and execute with one statement at a time

Verify the following by reading the __*dict*__ of Pet, pet1 and pet2

```
self.tricks.append(tricks)
```

Modify *self.tricks*

tricks attribute NOT available in instance

thus access *tricks* of class

There is still no *tricks* created for instance, beacuse there is no statement as *self.tricks* = ...

Append happen to class *tricks*

```
self.tricks = [tricks]
```

Assign (create) *self.tricks* as a new list with a single entry

Thus *tricks* is created for instance

Class *tricks* remains empty

`self.tricks += [tricks]`

Read *self.tricks*

tricks attribute NOT available in instance

thus access *tricks* of class

Now execute += operation on this *tricks*

This does an augmented assignment :

tricks(instance) = *tricks*(class)

so both are references to same object

the + operation is performed

both references show updated list

Class *tricks* show added trick

Instance *trick* show added trick

`self.tricks = self.tricks + [tricks]`

Assign (create) *self.tricks* as sum of *self.tricks* (class *tricks* first time and instance *tricks* after that) and a list containing single entry

Thus *tricks* is created for instance

Class *tricks* remains empty

The correct design of the class should use an instance variable :

```
class Dog:

    def __init__(self, name):
        self.name = name
              # creates a new empty list for each dog
        self.tricks = []

    def add_trick(self, trick):
              self.tricks.append(trick)
```

A class attribute can refer to a function defined outside the scope of the class :

```
def outsideF(self, a, b) :
    print 'outer func::', a, b

class C :
    f = outsideF
    def g(self) :
        print 'in g'

obj = C()
obj.f(1, [1,2])
obj.f(2, [1,2,3])
print C.__dict__
```

A method may invoke other methods of the same object :

```
class C :
    def f(self) :
        print 'in f'
    def g(self) :
        print 'in g'
        print 'calling f...'
        self.f()

obj = C()
obj.g()
```

```
in g
calling f...
in f
```

Assigning one class instance to another :

```
class C :
    def __init__(self, myname) : self.myname = myname
    def f(self) : self.x = 'aaa'
    def __del__(self) :
              print 'destr called for ', self.myname

o2 = C('o2'); o1 = C('o1')
o1.f()
print 'assigning............'
o2 = o1       # invokes destr for the obj referred by o2
o2.x = 'bbb'
print o1.x #prints 'bbb' bec o1 and o2 refer to same object
print o1, o2 # prints same address
print o1 is o2 # TRUE
```

Common operaator overloading methods

Method	Implements	Called for
__init__	Constructor	Object creation: X = Class(args)
__del__	Destructor	Object reclamation of X
__add__	Operator +	X + Y, X += Y if no __iadd__
__or__	Operator | (bitwise OR)	X | Y, X |= Y if no __ior__
__repr__, __str__	Printing, conversions	print(X), repr(X), str(X)
__len__	Length	len(X), truth tests if no __bool__
__bool__	Boolean tests	bool(X), truth tests
__lt__, __gt__, __le__, __ge__, __eq__, __ne__	Comparisons	X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y
__getitem__	Indexing, slicing, iteration	X[key], X[i:j], for loops and other iterations if no __iter__
__setitem__	Index and slice assignment	X[key] = value, X[i:j] = iterable
__delitem__	Index and slice deletion	del X[key], del X[i:j]

Inheritance

Class Trees

- Each class statement generates a new class object
- Each time a class is called, it generates a new instance object
- Instances are automatically linked to the classes from which they are created
- Classes are automatically linked to their superclasses according to the way the superclasses are listed in parentheses in a class header line – the left-to-right order there gives the order in the tree
- Code segment for the above statements :

```
class C2: ...                       # Make class objects
class C3: ...
class C1(C2, C3): ...     # Linked to superclasses
                                         #  (in this order)
I1 = C1()                           # Make instance objects
I2 = C1()                           # Linked to their classes
```

A simple inheritance example :

```
class emp :
    def __init__(selfie, val) : selfie.data = val
    def disp(selfie) : print selfie.data

class specialEmp(emp) :
    def disp(self) : print 'i am special emp', self.data

emp1 = emp("aaa")
emp1.disp()

emp2 = specialEmp(1234)
emp2.disp()
```

Classes are themselves attributes of the module in which they are defined

A more complete example :

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name; self.job = job; self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)
class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Amit Gupta')
    sue = Person('Neeta Sharma', job='dev', pay=100000)
    print(bob); print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10); print(sue)
    tom = Manager('Navven Kumar Jain', 50000)
    tom.giveRaise(.10); print(tom.lastName()); print(tom)

    for emp in (bob, sue, tom) :
        print emp.lastName()
```

Another example :

```
class base :
    bx = 90
    def __init__(self) :
        print 'base'
        self.bx = 9
        self.by = 900

class der(base) :
    def __init__(self) :
                print 'd'
                #base.__init__(self)

bObj = base()
print 'bObj:::', bObj.bx, bObj.by

dObj = der()
print 'dObj:::', dObj.bx, dObj.by #err bec dObj.by can not be accessed
```

uncomment this and check output

```
base
bObj::: 9 900
d
dObj::: 90
```

Followed by the error mesg

A simple example to highlight certain points :

```
class GrandP :
    def f(self) : print 'f in GrandP'
class Par(GrandP) :
    def g(self) : print 'g in Par'
class Ch(Par) :
    def h(self) : print 'h in Ch'

chObj = Ch()
chObj.h(); chObj.g(); chObj.f()

print 'dict GrandP:::'
print GrandP.__dict__
print
print 'dict Par:::'
print Par.__dict__
print
print 'dict Ch:::'
print Ch.__dict__
```

```
h in Ch
g in Par
f in GrandP
dict GrandP:::
{'__module__': '__main__', '__doc__': None, 'f': <function f at 0xb72aa33c>}

dict Par:::
{'__module__': '__main__', '__doc__': None, 'g': <function g at 0xb72aa6bc>}

dict Ch:::
{'h': <function h at 0xb72aaa3c>, '__module__': '__main__', '__doc__': None}
```

- Execution of a derived class definition proceeds the same as for a base class

- When the class object is constructed, the base class is remembered
  - this is used for resolving attribute references
  - if a requested attribute is not found in the class, the search proceeds to look in the base class
  - this rule is applied recursively if the base class itself is derived from some other class

- *DerivedClassName()* creates a new instance of the class just as in case of an independent class

- Method references are resolved as follows :
  - the corresponding class attribute is searched, working the chain of base classes if necessary
  - the method reference is valid if this yields a function object

- Derived classes may override methods of their base classes
- Methods have no special privileges when calling other methods of the same object :
  - a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it (*For C++ programmers: all methods in Python are effectively virtual*)

```
class Base :
        def f(self) : print 'in f of Base'
def g(self) :
                print 'in g of Base'
                print 'calling f...'
                self.f()
class Der(Base) :
        def f(self) : print 'in f of Der'
objDer = Der(); objDer.g()

in g of Base
calling f...
in f of Der
```

- An overriding method in a derived class may want to extend (rather than replace) the base class method of the same name
  - This can be done as *BaseClassName.methodname(self, arguments)*

```
class base :
  def f1(s, a, b) : print('f1 base')
  def f2(s, a, b) : print('f2 base')
  def f3(s, a, b) : print('f3 base')
  def f4(s, a, b) : print('f4 base')

class der(base) :
  def f2(s, a) : print('f2 der') # overrides base f2 with 2 args
  def f3(s, a, b) : print('f3 der') #overrides base f3
  def f4(s,a) : print('f4 der'); super(der, s).f4(a, a) #extends defn of base f4
  #def f4(s,a): print('f4 der'); base.f4(s, a, a) --> same as previous statement

d = der()

print('f1------->')
d.f1(1,2)
print('f2 2 args CAN NOT BE CALLED ------->')
#d.f2(1,2) error to call this
print('f2 1 arg ------->')
d.f2(1)
print('f3------->')
d.f3(1,2)
print('f4------->')
d.f4(1)
```

`__init__` *with inheritance*

- `__init__` methods are optional
- if `__init__` is defined, the ancestor's `__init__` must be called explicitly (if the ancestor defines one)
- each subclass can have its own set of arguments to `__init__`, as long as it calls the ancestor with the correct arguments

```
class Base :
    def __init__(self, a) :
        print 'base init', a*5

class Der(Base) :
    def __init__(self, a, b) :
        print 'der init, calling base init...'
        Base.__init__(self, a+b)

o = Der('str1', 'str2###')
```

Special Python functions that work with inheritance

- `isinstance()` :
  - used to check an instance's type
  - `isinstance(obj, X)` will be `True` only if `obj.__class__` is `X` or some class derived from `X`

- `issubclass()` :
  - used to check class inheritance
  - `issubclass(Y, X)` will be `True` if `Y` is some subclass of `X`

```
class Base : pass
class Der(Base) : pass
class DD(Der) : pass

objDer = Der()
print isinstance(objDer, Der)
print issubclass(Der, Base)
print issubclass(DD, Base)

True
True
True
```

Private, protected and public in Python

Public :

- All member variables and methods are public by default in Python

Protected :

- The only way to have protected members in Python is via convention
- This is done by prefixing the member name with a single underscore
  - its ONLY a convention that says : 'use this attr only in subclasses'
  - it is still accessible everywhere
  - so it is not really a protected member

Private :

- Any member name prefixed with at least two underscores and suffixed with at most one underscore is modified (`name mangling`) by Python
- So a member name __*mem* or __*mem*_ or ___*mem* ... is stored in the class or instance dictionary as _*class*__*mem* or _*class*__*mem*_ ...
- As a result of this name mangling, class users do not see any member by the name __*mem* ...
- But such a member is actually still accessible as _*class*__*mem*

An example :

```
class C :
    a = 'public class attr'
    _a = 'protected class attr'
    __a = 'private class attr'
    def __init__(self) :
        print 'obj constr'
        self.x = 'public instance attr'
        self._x = 'protected instance attr'
        self.__x = 'private instance attr'
        self.__x_ = 'aaaaaaa'


o = C()
print C.__dict__
print
print o.__dict__
print o.x
```

Classes and Python Operators

- ***Methods named with double underscores (__X__) are special :***
  - operator overloading can be implemented in classes by providing specially named methods to intercept operations
  - The Python language defines a fixed and unchangeable mapping from each of these operations to a specially named method
- ***Such methods are called automatically when instances appear in built-in operations :***
  - eg, if an instance object inherits an `__add__` method, that method is called whenever the object appears in a + expression
  - method's return value becomes the result of the corresponding expression
- ***Classes may override most built-in type operations :***
  - there are dozens of special operator overloading method names for intercepting and implementing nearly every operation available for built-in types
  - this includes expressions, but also basic operations like printing and object creation

- ***There are no defaults for operator overloading methods :***
  - if a class does not define or inherit an operator overloading method, then the corresponding operation is not supported for the class’s instances
- ***Operators allow classes to integrate with Python’s object model :***
  - by overloading type operations, the user-defined objects we implement with classes can act just like built-ins, and so provide consistency as well as compatibility with expected interfaces

@classmethod AND @staticmethod

- *@classmethod* is similar to a static method in C++
@
 is similar to a function – as if declared outside the class

- *classmethod* receives class object reference as first argument
staticmethod receives none

- *classmethod* can access class attributes via this argument
staticmethod can not do so

An example :

```
class MyDate:
    day = 0;    month = 0;  year = 0
    someStr = 'this is some class string'
    def __init__(self, day=0, month=0, year=0):
        self.day = day; self.month = month;     self.year = year
    def __str__(self) :
        return str(self.day)+'/'+str(self.month)+'/'+str(self.year)
    def f(self) :
        print 'just some function in class'
    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        date1 = cls(day, month, year)
        cls.f(date1);       print cls.someStr
        return date1
    @staticmethod
    def dateMillenium(day, month) :
   #print someStr  ERROR-can't access class attribute in staticmethod
        return MyDate(day, month, 2000)
    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999
```

```
class DateTime(MyDate) :
    def __str__(self) :
        return      "{0}-{1}-{2}-00:00:00PM".format(self.month,      self.day,
self.year)

date1 = MyDate(2, 10,2011)
date2 = MyDate.from_string('11-09-2012')
print MyDate.is_date_valid('33-09-1111')
date3 = MyDate.dateMillenium(14, 7)

print '##############'
print date1, isinstance(date1, MyDate)       # True
print date2, isinstance(date2, MyDate)       # True
print date3, isinstance(date3, MyDate)       # True

datetime1 = DateTime(1,1,1231)
datetime2 = DateTime.from_string('12-10-2001')
datetime3 = DateTime.dateMillenium(2, 2)
print '##############'
print   datetime1,   isinstance(datetime1,   MyDate),   isinstance(datetime1,
DateTime)      # True True
print   datetime2,   isinstance(datetime2,   MyDate),   isinstance(datetime2,
DateTime)      # True True
print   datetime3,   isinstance(datetime3,   MyDate),   isinstance(datetime3,
DateTime)      # True False
```

With respect to the previous code :

- the *classmethod* `from_string` behaves as a Factory Method :
  - it is invoked on the class `MyDate` directly – an instance is not required for invoking the *classmethod*

  it could have been invoked on an instance as well, with no side effects

  - it is used here to create an instance with a different set of arguments

  - date string parsing has been implemented in one place and it's reusable now

  - this solution fits OOP paradigm far better – the `MyDate` instance creation is very much a functionality encapsulated within the class

  - if `MyDate` class is subclassed, all children will have `from_string` defined

- the *staticmethod* `is_date_valid`:
  - performs a function that is logically bound to the `MyDate` class

  - it is invoked on the class `MyDate` directly – an instance is not required for invoking the *classmethod*

  it could have been invoked on an instance as well, with no side effects

  - it is just a function, called syntactically like a method, but without access to the object and it's internals (fields and another methods) – a classmethod does have this access

  - the *staticmethod* is also available in subclasses

- the *staticmethod* `dateMillenium`:
  - is also used as a Factory to create `MyDate` objects from otherwise unacceptable parameters

- the difference between *classmethod* Factory implementation and *staticmethod* Factory implementation is :
  - when `MyDate` is subclassed and the *staticmethod* is invoked as a factory (`dateMillenium`), a plain `MyDate` object is created

  - it has no properties of the `DateTime` subclass

  `datetime3` is only an instance of base class `MyDate`

  `datetime1` and `datetime2` are instances of base class `MyDate` as well as derived class `DateTime`

  - so *classmethod* ensures that the class is not hard-coded but rather learnt – `cls` can be any subclass

  the resulting object would rightly be an instance of `cls`

Thus :

@classmethod : when this method is called, the class is passed as first argument instead of the instance (as we normally do with methods). Hence the class and its properties can be used inside that method rather than a particular instance

@staticmethod : when this method is called, neither the class nor its instance is passed to it (as we normally do with methods). Hence a function can be put inside a class but you can't access the instance of that class

Object class

- This class was added in version 2.2
- This returns a new featureless object
- `object` is a base for all *new style classes*
- It has the methods that are common to all instances of new style classes

New style classes

- A class which inherits from `object` is a *new style class*
- This includes all built-in types like `list` and `dict`
- Only *new-style classes* can use Python's newer, versatile features like __slots__, descriptors, properties, and __getattribute__()
- *New-style classes* had been introduced since Python 2.2
- **A more complete definition :**

A *new-style class* is derived, either directly or indirectly, from a built-in type

This was not possible before Python 2.2

Built-in types include types such as:

- `int`
- `list`
- `tuple`
- `dict`
- `str`

The base class for all new-style classes is called `object`

- Some examples of *new style classes* :

```
class NewStyleUserDefinedCl(object) : pass
class DerivedFromBuiltInType(list) : pass
class IndirectlyDerivedFromType(DerivedFromBuiltInType) :
      pass
```

- *New-style classes* provide the following features :
  - **Properties :** attributes that are defined by `get/set` methods
  - Static methods and class methods
  - **Descriptors :** a protocol to define the behavior of attribute access through objects
  - Overriding the low level constructor `__new__`
  - Metaclasses
  - Method Resolution Order (MRO)

- In Python 3, the classes are automatically of type *new-style*
  - no inheritance from `object` is required

Upto Python 2.1 the concept of *class* was unrelated to the concept of *type*
Look at the following code with *old-style classes* :

```
class A : pass
class B : pass

a = A()
b = B()
print type(A), type(B)
print type(A) == type(B)

print type(a), type(b)
print type(a) == type(b)
```

The type of instance `a` *and that of* `b` *is same*

Output :

```
<type 'classobj'> <type 'classobj'>
True
<type 'instance'> <type 'instance'>
True
```

All subclasses of `A` and `B` will also be of type `classobj`

Writing the same code with *new-style classes*, `class A` and `class B` instances are recognized as different types :

```
class A(object) : pass
class B(object) : pass
a = A()
b = B()
print type(A), type(B)
print type(A) == type(B)

print type(a), type(b)
print type(a) == type(b)
```

The type of instance `a` and that of `b` are now different

Also type of `A` and that of `B` are type, which is the same as that of `int`, `dict` etc

Output :

```
<type 'type'> <type 'type'>
True
<class '__main__.A'> <class '__main__.B'>
False
```

All subclasses of `A` and `B` will also be of type `type`

- *New-style class* is the recommended way to create a class in modern Python
- A "Classic Class" or "old-style class" is a class as it existed in Python 2.1 and before
  - They have been retained for backwards compatibility
- Python hasn't yet settled on a specific official choice for the terminology

– *from https://wiki.python.org/moin/NewClassVsClassicClass*

Method Resolution order

- For classic *old-style classes* the method resolution order (MRO) is depth first and then left to right
- MRO changed with Python 2.3 but the new algorithm is used only with *new-style classes*

For the following code with *old style classes* :

```
class A:
    def whoAmI(self):
        print("I am a A obj")
class B(A):
    def whoAmI(self):
        print("I am a B obj")
class C(A):
    def whoAmI(self):
        print("I am a C obj")
class D(B,C):
    def whoAmI(self):
        print("I am a D obj")

d1 = D()
d1.whoAmI()
```

Output : `I am a D obj`

Now comment D's `whoAmI`
Output : `I am a B obj`

Now comment B's `whoAmI`
Output : `I am a A obj`

Now comment A's `whoAmI`
Output : `I am a C obj`

Finallly comment C's `whoAmI`
Output : `AttributeError`

MRO is DBAC

Now run the code with *new-style classes* :

```
class A (object):
    def whoAmI(self):
        print("I am a A obj")
class B(A):
    def whoAmI(self):
        print("I am a B obj")
class C(A):
    def whoAmI(self):
        print("I am a C obj")
class D(B,C):
    def whoAmI(self):
        print("I am a D obj")

d1 = D()
d1.whoAmI()
```

Output : `I am a D obj`

Now comment D's `whoAmI`

Output : `I am a B obj`

Now comment B's `whoAmI`

Output : `I am a C obj`

Now comment C's `whoAmI`

Output : `I am a A obj`

Finallly comment A's `whoAmI`

Output : `AttributeError`

refer the following to understand new MRO algorithm :
https://www.python.org/download/releases/2.3/mro/

MRO is DBCA

- MRO of a class can be viewed as :
  - *`__mro__` attribute*

    Example : `MyClass.__mro__`

    returns a tuple

    OR

  - *`mro()` method*

    Example : `MyClass.mro()`

    returns a list

super()

- *super(type[, object-or-type])* - python2
super([type[, object-or-type]]) - python3
- It returns a proxy object that delegates method calls to a parent or sibling class of type.
- This is useful for accessing inherited methods that have been overridden in a class
- **It only works for *new-style classes***
- **If the second argument is omitted, the super object returned is unbound**

Usage :

```
class A(object):
   def fA1(self) : print 'fA1'
class B(A) :
   def fA1(self) :
      print 'fA1 in B calling fA1 in A...'
      return super(B)
b = B()
o = b.fA1() ; print type(o)
```

Output :

```
fA1 in B calling fA1 in A...
<type 'super'>
```

With two arguments :

- The second argument can be :
  - instance of first argument
    - a bound method is returned
  - sub-class of first argument
    - an unbound method is returned

- The method dispatch is done in such a way that

```
super(cls, instance-or-subclass).method(*args, **kw)
```

corresponds more or less to

```
right-method-in-the-MRO-applied-to(instance-or-subclass,      *args,
**kw)
```

```python
class A(object) :
    def f(s) : print 'fA'
class B(A) :
    def f(s) : print 'fB(A)'
class C(A) :
    def f(s) : print 'fC(A)'
class D(B, C) :
    def f(s) : print 'fD(B,C)'
d = D()
# BOUND method returned
print 'super(D,d):::',; super(D,d).f()
print 'super(B,d):::',; super(B,d).f()
print 'super(C,d):::',; super(C,d).f()
# AttributeError: 'super' object has no attribute 'f'
# print 'super(A,d):::',; super(A,d).f()

print '**********************'
# UNBOUND method returned
print 'super(D,D):::',; super(D,D).f(d)
print 'super(B,D):::',; super(B,D).f(d)
print 'super(C,D):::',; super(C,D).f(d)

# AttributeError: 'super' object has no attribute 'f'
# print 'super(A,D):::',; super(A,D).f(d)
```

Output :

```
super(D,d)::: fB(A)
super(B,d)::: fC(A)
super(C,d)::: fA
**********************
super(D,D)::: fB(A)
super(B,D)::: fC(A)
super(C,D)::: fA
```

- **If the second argument is an object, `isinstance(obj, type)` must be true**
- A simple usage with two arguments is :

Output :

```
fA1 in B calling fA1 in A...
fA1
fA2 in B calling fA2 in A...
fA2 str1str2 [1, 2, 3]
```

Here 2nd argument is an **instance**

```
class A(object):
    def fA1(self) : print 'fA1'
    def fA2(self, a, b) : print 'fA2', a, b
class B(A) :
    def fA1(self) :
        print 'fA1 in B calling fA1 in A...'
        super(B, self).fA1()
    def fA2(self, a, b, c) :
        print 'fA2 in B calling fA2 in A...'
        super(B, self).fA2(a+b, c)

b = B() ;    b.fA1() ;  b.fA2('str1', 'str2', [1,2,3])
```

A code without super():

```
class Person (object):
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last
    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
            #invoking base class __init__()
        Person.__init__(self,first, last)
        self.staffnumber = staffnum
    def GetEmployee(self): #invoking base method 'Name()'
        return self.Name() + ", " +  self.staffnumber

x = Person("Amit", "Sharma")
y = Employee("Sunita", "Bajaj", "1007")

print(x.Name())
print(y.GetEmployee())
```

Same code re-written with `super()` :

```
class Person (object):
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last
    def __str__(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        super(Employee, self).__init__(first, last)
        #super().__init__(first, last) - Python 3 can use this
        self.staffnumber = staffnum
    def __str__(self):
        return super (Employee, self).__str__() + ", " +
self.staffnumber

x = Person("Amit", "Sharma")
y = Employee("Sunita", "Bajaj", "1007")

print(x)
print(y)
```

With multiple inheritance :

```
class First(object):
  def __init__(self):
    print "first"
    super(First, self).__init__()
class Second(object):
  def __init__(self):
    print "second"
    super(Second, self).__init__()
class Third(First, Second):
  def __init__(self):
    print "third"
    super(Third, self).__init__()
o = Third()
print Third.mro()
```

`super()` call in *`Third`* dispatches to *`First.__init__`*

Then the `super()` call in *`First`* dispatches to *`Second.__init__`*

This in turns dispatches to *`object.__init__`*

Output :

```
third
first
second
[<class '__main__.Third'>, <class '__main__.First'>, <class
'__main__.Second'>, <type 'object'>]
```

Thus :

- It cannot be said in advance where the `super()` will dispatch, unless the whole hierarchy is known
- Hence the methods in the heirarchical chain must have compatible signatures

Else the chain breaks **(the following gives error)** :

```
class First(object):
  def __init__(self, a):
    print "first"
    #no arg passed to __init__(), 'object' class expects 0 arg
    super(First, self).__init__()
class Second(object):
  def __init__(self, a):
    print "second"
    #no arg passed to __init__(), 'object' class expects 0 arg
    super(Second, self).__init__()
class Third(First, Second):
  def __init__(self, a):
    print "third"
    #1 arg passed to __init__(), 'First' and 'Second' expects 1 arg
    super(Third, self).__init__(a)
o = Third('aaa')
```

Reason for failure :

- MRO of `Third` is :

`Third, First, Second, object`

- Thus `super().__init__()` of `First` calls `super().__init__()` of `Second` which expects an argument

- But no argument was passed to `super().__init__()` of `First` because the superclass of `First` is `object`

Thus methods in the heirarchical chain must have compatible signatures

A possible solution :

```
class Dummy(object) :
    def __init__(self, *args) : pass

class First(Dummy):
  def __init__(self, a):
    print "first"
    super(First, self).__init__(a)

class Second(Dummy):
  def __init__(self, a):
    print "second"
    super(Second, self).__init__(a)

class Third(First, Second):
  def __init__(self, a):
    print "third"
    super(Third, self).__init__(a)

o = Third([1,2])
```

Abstract classes

- Abstract base classes (ABCs) enforce that derived classes implement particular methods from the base class

- Import `abc` module to create abstract classes

- An abstract class is one for which :
  - the `__metaclass__` class attribute is assigned to `abc.ABCMeta`
  - it has atleast one method which is `@abstractmethod` or at least one property that is `abstractproperty`

- An abstract class instance can not be created

- An abstract method can have an implementation that can be invoked via the `super()` mechanism from the class that overrides it

Python3 :

```
from abc import *

class myAbsCl (metaclass=ABCMeta):
  @abstractmethod
  def f(s) :
    print ('f in abs cl')

class concrete(myAbsCl) :
  def f(s) :
    print ('f in concrete class')
    myAbsCl.f(s)
    super(concrete, s).f()

o = concrete()
o.f()
```

Example with abstract method (Python2) :

```
from abc import ABCMeta, abstractmethod

class myAbsCl (object):
  __metaclass__ = ABCMeta
  @abstractmethod
  def f(s) :
    print 'f in abs cl'

class concrete(myAbsCl) :
  def f(s) :
    print 'f in concrete class'
    myAbsCl.f(s)#invoking base class 'f' using base class name
    super(concrete, s).f()

o = concrete()
o.f()

#o = myAbsCl() -- error
```

Example with abstract property (Python2) :

```
from abc import ABCMeta,  abstractproperty

class base (object) : #abstract class
    __metaclass__ = ABCMeta
    def getX(s) :
        print 'base get'
        return s._x
    def setX(s, v) :
        print 'base set'
        s._x = v
    x = abstractproperty(getX, setX)
class der(base) : #concrete class
    def getX(s) :
        print 'der get'
        return super(der, s).getX()
    def setX(s, v) :
        print 'der set'
        super(der, s).setX(v)
    x = property(getX, setX)
o = der()
o.x = 'prop'
print o.x
#o = base() -- error
```

Python3 :

```
from abc import *

class base (metaclass=ABCMeta) :
    def getX(s) :
        print ('base get')
        return s._x
    def setX(s, v) :
        print ('base set')
        s._x = v
    x = abstractproperty(getX, setX)
class der(base) : #concrete class
    def getX(s) :
        print ('der get')
        return super(der, s).getX()
    def setX(s, v) :
        print ('der set')
        super(der, s).setX(v)
    x = property(getX, setX)
o = der()
o.x = 'prop'
print (o.x)
#o = base() -- error
```

Pythonic approach to abstractness :

- This explicit declaration provided by `abc` module may be considered not very pythonic
- An abstract method can be as well declared by raising a `NotImplementedError`:

```
class Animal:
    def say_something(self):
        raise NotImplementedError()
```

OR

- A class could follow some naming conventions e.g. prefixing a class name with Base or Abstract

Metaprogramming is about code that manipulates code

The main features :

- metaclasses
- decorators
- descriptors – not in scope

Metaprogramming is used extensively in frameworks and libraries

Metaclasses

- A metaclass is defined as "the class of a class"
  - i.e., any class whose instances are themselves classes, is a metaclass

- Thus a class is an instance of its metaclass

- When we create a class, the interpreter calls the metaclass to create it

Example :

```
class C (object) : pass
print type(C)
```

OR

```
print C.__class__
```

Output :

```
<type 'type'>
```

- Metaclass is used to construct classes

Which class is metaclass of a class :

- If either a class or one of its bases has a `__metaclass__` attribute, it's taken as the metaclass
- ***Else `type` is the metaclass***
- Subclasses inherit the metaclass from their base class

Normallly a metaclass must inherit from `type`

```
class MyMetaClass(type) : pass
```

`

Making a class from a metaclass

```
# Python 3
class C(metaclass = NewMeta) : pass
# Python 2
class C :
      __metaclass__ = NewMeta
```

A class is an instance of its metaclass :

When a class is defined as follows

```
>>> class ClassOne(object) :
...      foo = 'some str'
...
```

the class is created as

```
>>> ClassOne = type('One', (object, ), {'foo':'some str'})
```

BUT

When a class is defined as follows

```
>>> class MyMetaClass(type) : pass
...
>>> class ClassTwo (object):
...      __metaclass__ = MyMetaClass
...      foo = 'abc'
...
```

the class is created as

```
>>> ClassTwo = MyMetaClass('Two',(object,),{'foo' : 'abc'})
...

>>> type (ClassOne)
<type 'type'>
>>> type (ClassTwo)
<class '__main__.MyMetaClass'>
```

Metaclass's `__call__`:

- `__call__` is called when the already-created class is "called" to instantiate a new object

unlike `__new__` and `__init__` which were invoked at class creation time

```python
class MyMeta(type):
    def __new__(meta, name, bases, dct):
        print "Allocating mem for ", name
        return super(MyMeta, meta).__new__(meta, name, bases, dct)
    def __init__(cls, name, bases, dct):
        print "Initializing ", name
        super(MyMeta, cls).__init__(name, bases, dct)
    def __call__(cls, *args, **kwds) :
        print "Call ...", cls
        return super(MyMeta, cls).__call__(*args, **kwds)
class MyKlass(object):
    __metaclass__ = MyMeta
    def foo(self, param): pass
    def __new__(cls) :
        print "new of class"
        return super(MyKlass, cls).__new__(cls)
    def __init__(self):
        print 'init of class'
print 'create obj...'
o1 = MyKlass() ; o1.foo('a')
o2 = MyKlass()
```

Output :

```
Allocating mem for MyKlass
Initializing MyKlass
create obj...
Call        ...        <class
'__main__.MyKlass'>
new of class
init of class
Call        ...        <class
'__main__.MyKlass'>
new of class
init of class
```

Subclasses inherit the metaclass from their base class
Thus with multiple inheritance, the bases must have the same metaclass :

```
>>> class Meta1(type):pass
...
>>> class Meta2(type):pass
...
>>> class Base1 :
...     __metaclass__ = Meta1
...
>>> class Base2 :
...     __metaclass__ = Meta2
...
>>> class Foobar(Base1, Base2):pass
...

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases
```

But the following works (using the *leaf*-most as the metaclass) :

```
class Meta(type) : pass
class SubMeta(Meta) : pass

class Base1(object) :
    __metaclass__ = SubMeta
class Base2(object) :
    __metaclass__ = Meta

class Foo(Base1, Base2) : pass

print type(Foo)        `
```

```
<class '__main__.SubMeta'>
```

Exercises

- Create a Python class that behaves as a final class, i.e., it can not be subclassed

Decorators

- A decorator is a function that takes another function as argument
- It extends the behavior of the latter function without explicitly modifying it

Following code shows a function taking another function as argument :

```
def befAftDecoration (func) :
    def wrap() :
        print 'hi your function will execute after I am done ...'
        func()
        print 'ok i am sorry for interrupting you !!!'
    return wrap
def funA() :
    print 'func A is executing'
def funB() :
    print 'func B is executing'
print '---------------'
f = befAftDecoration(funA)
f()
print '---------------'
f = befAftDecoration(funB)
f()
```

Output :

```
---------------
hi your function will execute after I am
done ...
func A is executing
ok i am sorry for interrupting you !!!
---------------
hi your function will execute after I am
done ...
func B is executing
ok i am sorry for interrupting you !!!
```

Thus 'befAftDecoration' function wraps a function – modifying its behavior

The same code could have been written as :

```
def befAftDecoration (func) :
    def wrap() :
        print 'hi your function will execute after I am done
 ...'
        func()
        print 'ok i am sorry for interrupting you !!!'
    return wrap

@befAftDecoration
def funA() :
    print 'func A is executing'

@befAftDecoration
def funB() :
    print 'func B is executing'

funA()
funB()
```

Another example :

```
import time

def timing_function(some_function):
    """
    Outputs the time a function takes to execute
    """
    def wrapper():
        t1 = time.time()
        some_function()
        t2 = time.time()
        return "Time it took to run the function: " +

  str((t2-t1)) + "\n"
    return wrapper
@timing_function
def my_function():
    num_list = []
    for x in (range(0,10000)):
        num_list.append(x)
    print "\nSum of all the numbers: "
                                       +str((sum(num_list)))
print my_function()
```

We LOSE some information doing it this way ...

- the wrapped (decorated) function loses its name
- the wrapped (decorated) function loses its help

Execute the following and see :

```
def decorate(function):
    """
    I am a decorator
    """
    def changeBehavior():
            """
        I change behavior of others
        """
        print 'Good Morning ...'
        function()
    return changeBehavior
@decorate
def myFunction():
    """
    This is a function that does nothing
    """
    print 'how\'s everyone'
myFunction()
print myFunction
help(myFunction)
```

SOLUTION :

```
from functools import wraps
def decorate(function):
    """
    I am a decorator
    """
        @wraps(function)
    def changeBehavior():
        print 'Good Morning ...'
        function()
    return changeBehavior
@decorate
def myFunction():
    """
    This is a function that does nothing
    """
    print 'how\'s everyone'
myFunction()
print myFunction
help(myFunction)
```

EXAMPLE : Function to be decorated is argumented ...

```
def decorate(function):
    """
    I am a decorator
    """
    def changeBehavior(*args, **kwargs):
        print 'Good Morning ...'
        function(*args, **kwargs)
    return changeBehavior

@decorate
def myFunction(a, b, c) :
    """
    This is a function that does nothing
    """
    print 'how\'s everyone'
    print 'my args are :::', a, b, c


myFunction(100, 'str', [1,2])
```

EXAMPLE : Function decorated with MULTIPLE decorators :

```
def decorateOne(function):
    def changeBehavior1() :
        print 'invoked from decorator One ...'
        function()
        print 'deco ONE over'
    return changeBehavior1
def decorateTwo(function):
    def changeBehavior1() :
        print 'invoked from decorator Two ...'
        function()
        print 'deco TWO over'
    return changeBehavior1

@decorateOne
@decorateTwo
def sayHello() : print 'GOOD MORNING'

sayHello()
```

Output :

```
invoked from decorator One ...
invoked from decorator Two ...
GOOD MORNING
deco TWO over
deco ONE over
```

Some uses of decorators ...

- debugging code can be isolated to a single location

Example (prints function name before invoking it) :

```
def debug(function):
    def changeBehavior1() :
        print function.__name__, 'invoked...'
        function()
    return changeBehavior1

@debug
def sayHello() : print 'GOOD MORNING'
@debug
def sayBye() : print 'Good Bye'

sayHello()
sayBye()
```

- some kind of logging can be implemented

Decorating methods (functions defined in a class) :

```
def decorateOne(function):
    print '*********', function
    def changeBehavior1(self) : #class method args recvd here
        print 'invoked from decorator One ...'
        function(self) #method invoked with 'self'
        print 'deco ONE over'
    return changeBehavior1

class MyClass (object) :
    @decorateOne
    def meth1(self) :
        print 'in meth'

o = MyClass()
o.meth1()
```

Output :

```
*********  <function  meth1  at
0xb7295a74>
invoked from decorator One ...
in meth
deco ONE over
```

Decorating ALL methods of a class :

```
def debug(func) :
    def wrap(self) :
        print 'dec ON ...'
        func(self)
        print 'dec OVER ...'
    return wrap
def debugmethods(cls) :
    for key, val in vars(cls).items() :
        if callable(val) :
            setattr(cls, key, debug(val))
    return cls
@debugmethods
class C :
    def m1(self) : print 'method m1'
    def m2(self) : print 'method m2'
        @classmethod
    def m3(cls) : print 'class method'
    @staticmethod
    def m4() : print 'static method'
o = C()
o.m1()   ;    o.m2()    ; C.m3()    ;   C.m4()
```

vars() return the __dict__ attribute

Output :

```
dec ON ...
method m1
dec OVER ...
dec ON ...
method m2
dec OVER …
class method
static method
```

only instance methods can be decorated

Another example :

```
import time

def timing_function(some_function):
    """
    Outputs the time a function takes to execute
    """
    def wrapper():
        t1 = time.time()
        some_function()
        t2 = time.time()
        return "Time it took to run the function: " +

  str((t2-t1)) + "\n"
    return wrapper
@timing_function
def my_function():
    num_list = []
    for x in (range(0,10000)):
        num_list.append(x)
    print "\nSum of all the numbers: "
                                        +str((sum(num_list)))
print my_function()
```

Properties :

- Properties are based on Descriptor protocol (Python descriptors are a way to create managed attributes)
- They are implemented by using `property()`
- Purpose of this function is to create a *property* of a class.
- A property looks and acts like an ordinary instance (data) attribute, except that you provide methods that control access to the attribute
- There are three kinds of attribute access: read, write, and delete – any or all of these can be defined for the *property*
- `property(fget=None, fset=None, fdel=None, doc=None)`

where:

  - `fget` – attribute get method
  - `fset` – attribute set method
  - `fdel` – attribute delete method
  - `doc` – docstring

A simple example :

```
class C(object):
    def __init__(self): self.__x = None
    def getx(self):
        print 'GETTER...' ; return self.__x
    def setx(self, value):
        print 'SETTER...' ; self.__x = value
    def delx(self):
        print 'DELETER...' ; del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
o = C()
print o.x      #invokes getx() - prints None
o.x = 'new val'       #invokes setx()
o.__x = 'QQQQQQqqnew val'    #defines a new __x
#o.__dict__ will now have _C__X corresponding to self.__x (o.x)
#AND __x defined just above
o1 = C() ; o1.x=100
print o.x, o1.x #both 'o' and 'o1' have independent 'x'
del o.x ; print o1.x #delete 'o.x' doesn't effect 'o1.x'
#o.__dict__ still has _C__X - but there is no property to access this for 'o'
o.x = 999      #creates the property 'x' again
print C.x      #prints that this is a property object
#'x' is stored as property object in C.__dict__
C.x = 1 ; print C.x    #now 'x' is stored in C.__dict__ as int
o.x = 88       #defines a 'x' in o.__dict__
#'x' is no more a property, its a data attribute
```

Creating properties by using `@property` decorator :

```
class Person (object) :
    def __init__(self) :
        self.__first= '' ;  self.__last= ''
    @property
    def firstName(self) :
        print 'firstName getter' ; return self.__first
    @firstName.setter
    def firstName(self, value) :
        print 'firstName setter' ; self.__first= value
    @firstName.deleter
    def firstName(self) :
        print 'firstName deleter' ; del self.__first
    @property
    def lastName(self) :
        print 'lastName getter' ; return self.__last
    @lastName.setter
    def lastName(self, value) :
        print 'lastName setter' ;self.__last= value
    @lastName.deleter
    def lastName(self) :
        print 'lastName deleter' ; del self.__last
p1 = Person()
p1.firstName = 'qqqqqqqqq'
```