Intermediate Python

Genesis InSoft Limited

A name you can trust

1-7-1072/A, Opp. Saptagiri Theatre, RTC * Roads, Hyderabad - 500 020

Python

Genesis Computers (A unit of Genesis InSoft Limited) started its operations on March 16th 1992 in Hyderabad, India primarily as a centre for advanced software education, development and consultancy.

Training is imparted through lectures supplemented with on-line demonstrations using audio visual aids. Seminars, workshops and demonstrations are organized periodically to keep the participants abreast of the latest technologies. Genesis InSoft Ltd. is also involved in software development and consultancy.

We have implemented projects/training in technologies ranging from client server applications to web based. Skilled in PowerBuilder, Windows C SDK, VC++, C++, C, Visual Basic, Java, J2EE, XML, WML, HTML, UML, Java Script, MS.NET (C#, VB.NET and ASP.NET, ADO.NET etc), PHP, Jhoomla, Zend Framework, JQuery, ExtJS, PERL, Python, TCL/TK etc. using Oracle, MySql and SQL Server as backend databases.

Genesis has earned a reputation of being in forefront on Technology and is ranked amongst the top training institutes in Hyderabad city. The highlight of Genesis is that new emerging technologies are absorbed quickly and applied in its areas of operation.

We have on our faculty a team of highly qualified and trained professionals who have worked both in India and abroad. So far we have trained about 52,000+ students who were mostly engineers and experienced computer professionals themselves.

Tapadia (MS, USA), the founder of Genesis Computers, has about 32+ years experience in software industry. He worked for OMC computers, Intergraph India Private Ltd., Intergraph Corporation (USA), and was a consultant to D.E. Shaw India Software Private Ltd and iLabs Limited. He has more than 30 years of teaching experience, and has conducted training for the corporates like ADP, APSRTC, ARM, B.H.E.L, B2B Software Technologies, Cambridge Technology Enterprises Private Limited, CellExchange India Private Limited, Citicorp Overseas Software Limited, CMC Centre (Gachibowli, Posnett Bhavan), CommVault Systems (India) Private Limited, Convergys Information Management (India) Private Limited, D.E. Shaw India Software Private Limited, D.R.D.L, Dell EMC, Bangalore (behalf of DevelopIntelligence, USA), Deloitee Consulting India Private Limited, ELICO Limited, eSymbiosis ITES India Private Limited, Everypath Private Limited, Gold Stone Software, HCL Consulting (Chennai), iLabs Limited, Infotech Enterprises, Intelligroup Asia Private Limited, Intergraph India Private Limited, Invensys Development Centre India Private Limited, Ivy Comptech, JP Systems (India) Limited, Juno Online Services Development Private Limited, Malpani Soft Private Limited, Mars Telecom Systems Private Limited, Mentor Graphics India Private Limited, Motorola India Electronics Limited, NCR Corporation India Private Limited, Netrovert Software Private Limited, Nokia India Private Limited, Optima Software Services, Oracle India Private Limited, Polaris Software Lab Limited, Qualcomm India Private Limited (Chennai, Hyderabad, Bangalore), Qualcomm China, Quantum Softech Limited, R.C.I, Renaissance Infotech, Satyam Computers, Satyam GE, Satyam Learning Centre, SIS Software (India) Private Limited, Sriven Computers, Teradata - NCR, Tanla Solutions Limited, Timmins Training Consulting Sdn. Bhd, Kuala Lumpur, Vazir Sultan Tobacco, Verizon, Virtusa India Private Limited, Wings Business Systems Private Limited, Wipro Systems, Xilinx India Technology Services Private Limited, Xilinx Ireland, Xilinx Inc (San Jose).

Genesis InSoft Limited
1-7-1072/A, RTC * Roads, Hyderabad - 500 020

rtapadia@genesisinsoft.com
www.genesisinsoft.com

Set

A Set is an unordered collection data type that is iterable, mutable and has no duplicate elements. Python's set class represents the mathematical notion of a set. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table. Since sets are unordered, we cannot access items using indexes like we do in lists.

Set consists of various elements; the order of elements in a set is undefined. You can add and delete elements of a set, you can iterate the elements of the set.

Code 1 File: set1.py

```
colors = set()
colors.add("red")
print(colors)
colors.update(['green', 'blue', 'yellow', 'red', 'white', 'blue'])
print(colors)

for color in colors:
  print(color)

print(min(colors), max(colors), len(colors))

colors.discard('blue')
print(colors)

print(colors.pop())
print(colors)

colors.clear()
print(colors)
```

Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

In Python 3 set is a built-in class and is to be used without import.

Operation	Equivalent	Result
len(s)		number of elements in set *s* (cardinality)
x in s		test *x* for membership in *s*
x not in s		test *x* for non-membership in *s*
s.issubset(t)	s <= t	test whether every element in *s* is in *t*
s.issuperset(t)	s >= t	test whether every element in *t* is in *s*
s.union(t)	s | t	new set with elements from both *s* and *t*
s.intersection(t)	s & t	new set with elements common to *s* and *t*
s.difference(t)	s - t	new set with elements in *s* but not in *t*
s.symmetric_difference(t)	s ^ t	new set with elements in either *s* or *t* but not both
s.copy()		new set with a shallow copy of *s*

Code 2 File: set2.py

```
tuple1 = (1,2,3)
list1 = [2,3,4]

s1 = set(tuple1)
s2 = set(list1)
s3 = {5, 2, 7, 9, 2, 7}
print(s3, type(s3))
print(s1, id(s1))

s1.union(s2)
print(s1.union(s2))
print(s1, id(s1))
s1 = s1 | s2                    # Union alternative notation
print(s1, id(s1))
print(len(s1))

s3 = s1.intersection(s2)
print(s3)
s3 = s1 & s2                    # Intersection alternative notation
print(s3)

s3 = s1.difference(s2)
print(s3)
s3 = s2 - s1                    # Difference alternative notation
print(s3)

s1 = set(tuple1)
s3 = s1.symmetric_difference(s2)
print(s3)
s3 = s1 ^ s2                    # Symmetric difference alternative
notation
print(s3)
```

Code 3 File: set3.py

```
s1 = set([2,3])
s2 = set([2,3,4])

print(s1.issubset(s2))
print(s2.issubset(s1))

print(s1.issuperset(s2))
print(s2.issuperset(s1))

print(s1, id(s1))

s1.add(5)
s1.remove(2)    # key error if the element doesn't exist
s1.discard(2)
print(s1, id(s1))

s4 = s2.copy()
print(s4)
s1.update(s2)
print(s1, id(s1))

print(3 in s4)
print(1 in s4)
print(3 not in s4)
print(1 not in s4)

print(s1.isdisjoint(s2))  # true if null intersection

s4.pop()
print(s4)
s4.pop()
print(s4)
s1.clear()
print(s1)
```

The following table lists operations available in Set but not found in frozenset (immutable set).

Operation	Equivalent	Result
`s.update(t)`	*s* |= *t*	return set *s* with elements added from *t*
`s.intersection_update(t)`	*s* &= *t*	return set *s* keeping only elements also found in *t*
`s.difference_update(t)`	*s* -= *t*	return set *s* after removing elements found in *t*
`s.symmetric_difference_update(t)`	*s* ^= *t*	return set *s* with elements from *s* or *t* but not both
`s.add(x)`		add element *x* to set *s*
`s.remove(x)`		remove *x* from set *s*; raises `KeyError` if not present
`s.discard(x)`		removes *x* from set *s* if present
`s.pop()`		remove and return an arbitrary element from *s*; raises `KeyError` if empty
`s.clear()`		remove all elements from set *s*

Code 4 File: frozenset1.py

```
s1 = frozenset([2,3,1])
s2 = frozenset([3,4,5])
s3 = frozenset([])

s3 = s1.union(s2)
print(s3)
s3 = s1 | s2                    # Union alternative notation
print(s3)
print(len(s3))

for value in s3:
  print(value, end = " ")

print()

s3 = s1.intersection(s2)
print(s3)
s3 = s1 & s2                    # Intersection alternative notation
print(s3)

s3 = s1.difference(s2)
print(s3)
s3 = s1 - s2                    # Difference alternative notation
print(s3)

s3 = s1.symmetric_difference(s2)
print(s3)
s3 = s1 ^ s2                    # Symmetric difference alternative
notation
print(s3)
```

Code 5 File: wordFrequencySet.py

```
def wordFrequency(str):
  str = str.split()
  uniqueWords = set(str)
  print(uniqueWords)

  for word in uniqueWords:
    print("Frequency of ", word, " = ", str.count(word))

str = "hello world hello Qualcomm Training at Qualcomm"
wordFrequency(str)
```

Code 6 File: missingNumberSet.py

```
'''
Given a list containing n numbers (duplicates allowed) find the one
that is missing from the array.
'''

def missingNumber(nums):
  num_set = set(nums)
  print(num_set)
  n = len(nums) + 1
  for number in range(min(num_set), max(num_set)):
    if number not in num_set:
      return number

lst = [2, 4, 5, 6, 7, 5, 7, 2, 4, 2, 9, 3, 2, 11, 13]
print(missingNumber(lst))
```

A shallow copy creates a new object which stores the reference of the original elements.

So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

Code 7 File: shallowcopy.py

```
import copy

ol = [[1, 2, 3], [4, 5, [11, 12], 6], [7, 8, 9]]
nl = copy.copy(ol)

print("Old list:", ol)
print("New list:", nl)
print(id(ol), id(nl))

ol.append([4, 4, 4])

print("Old list:", ol)
print("New list:", nl)
print(id(ol), id(nl))

ol[1][1] = 'AA'

print("Old list:", ol)
print("New list:", nl)
```

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.

The deep copy creates independent copy of original object and all its nested objects.

Code 8 File: deepcopy.py

```
import copy

ol = [[1, 2, 3], [4, 5, [11, 12], 6], [7, 8, 9]]
nl = copy.deepcopy(ol)

print("Old list:", ol)
print("New list:", nl)
print(id(ol), id(nl))

ol.append([4, 4, 4])

print("Old list:", ol)
print("New list:", nl)

ol[1][1] = 'AA'

print("Old list:", ol)
print("New list:", nl)
```

Classes

Python is an object-oriented language, which facilitates creation of classes and objects.

Class: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods.

Data member: A class variable or instance variable that holds data associated with a class and its objects.

Function overloading: The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects (arguments) involved.

Instance variable: A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance: The transfer of the characteristics of a class to other classes that are derived from it.
Instance: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation: The creation of an instance of a class.

Method: A special kind of function that is defined in a class definition.

Object: A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading: The assignment of more than one function to a particular operator.

The class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of private data.

The simplest form of class definition looks like this:

```
class className:
  <statement-1>
  .
  .
  <statement-N>
```

The constructor for a class must be named `__init__()`. The argument self is mandatory.

The destructor is `__del()__`. Python deletes unneeded objects (built-in types or class instances) automatically to free memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple or dictionary). The object's reference count decreases when it is deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

Class instantiation uses function notation. Just pretend that the class object is a parameter less function that returns a new instance of the class.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects in a known initial state. Therefore a class may define a special method named __init__().

What can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names. The first is data attributes. These correspond to “data members” in C++. Data attributes need not be declared; like local variables, they come existence when they are first assigned to.

The second kind of attribute references understood by instance objects are methods. A method is a function that “belongs to” an object. Valid method names of an instance object depend on its

class. By definition, all attributes of a class that are (user-defined) function objects define corresponding methods of its instances.

Use __new__ when you need to control the creation of a new instance.
Use __init__ when you need to control initialization of a new instance.

#__new__ is the first step of instance creation. It is called first, and is responsible for returning a
new instance of your class.
In contrast, __init__ doesn't return anything; it's only responsible for
initializing the instance after it's been created.

If __new__() does not return an instance of cls, then the
new instance's __init__() method will not be invoked.
If you return a different instance, original __init__ is never called.

Code 9 File: class_new_init.py

```
class test(object):
  """Use __new__ when you need to control the creation of a new
instance"""
  _dict = dict()

  def __new__(cls):
    """new method to create an instance"""
    if ('key' in test._dict):
      print("Exists")
      print(type(test._dict['key']))
      return test._dict['key']
    else:
      print("New")
      return super().__new__(cls)

  def __init__(self):
    """init method to initialize the state. It is like a
     constructor"""
    print ("Init")
    test._dict['key'] = self

obj1 = test()
obj2 = test()
print(id(obj1), id(obj2))

print(obj1.__doc__)
print(obj1.__new__.__doc__)
print(obj1.__init__.__doc__)
```

Code 10 File: class init del.py

```
class test(object):
  def __init__(self):
    print ("Init")

  def __del__(self):
    print ("del", id(self))

obj1 = test()
obj2 = test()
print(id(obj1), id(obj2))
obj1 = obj2
print("one")
obj2 = ""
print("two")
```

Code 11 File: class1.py

```
class circle:
  count = 0
  def __init__(self, radius, color='Blue'):
    self.radius = radius
    self.color=color
    circle.count += 1
    print("__init__ called")

obj = circle(5, 'green')
print(obj.radius)
print(obj.color)
obj2 = circle(10)
print(obj2.color)
print(circle.count)
```

Every class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:

__dict__ : Dictionary containing the class's namespace.
__doc__ : Class documentation string or None if undefined.
__name__: Class name.
__module__: Module name in which the class is defined. This attribute is "__main__" in interactive mode.
__bases__ : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

The special thing about methods is that the object is passed as the first argument of the function. In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

Conventionally, the first argument of methods is often called self. This is nothing more than a convention: the name self has absolutely no special meaning to Python.

Code 12 File: class2.py

```
class Account:
  ''' Account class '''
  def __init__(self, name, number, balance):
    print("initializer called with 3 arguments")
    self.accName = name
    self.accnumber = number
    self.accbalance = balance

  def withdraw(self, amount):
    self.accbalance -= amount

  def deposit(self, amount):
    self.accbalance += amount

  def show(self):
    print('Account name :', self.accName)
    print('Account number :', self.accnumber)
    print('Account balance :', self.accbalance)

customer = Account('ravi', 'S101', 5000)
customer.withdraw(1000)
customer.deposit(2000)
customer.show()   # Account.show(customer)

print("Account.__doc__:", Account.__doc__)
print("Account.__name__:", Account.__name__)
print("Account.__module__:", Account.__module__)
```

Data attributes override method attributes with the same name; to avoid accidental name conflicts it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Code 13 File: class_static_method.py

```
from datetime import date

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  # A class method to create a Person object by birth year.
  # A class method receives the class as implicit first argument,
  # just like an instance method receives the instance
```

```
  @classmethod
  def fromBirthYear(cls, name, year):
    return cls(name, date.today().year - year)

  # A static method to check if a Person is adult or not.
  # A static method does not receive an implicit first argument.

  @staticmethod
  def isAdult(age):
    return age > 18

  def isChild(self):
    return self.age <= 18

person1 = Person('Subbu', 15)
person2 = Person.fromBirthYear('Satyam', 1991)
print(person1.age)
print(person2.age)
print(Person.isAdult(12))
print(person1.isChild())
```

An object's attributes may or may not be visible outside the class definition. For these cases, you can name attributes with a double underscore prefix (private), and those attributes will not be directly visible to outsiders. Python protects those members by internally changing the name to include the class name. You can access such attributes as `object._className__attrName`. In the following example we can access using `obj._employee__name`.

Code 14 File: class_access.py

```
class employee(object):
  def __private(self):
    print("private method declaration")

  def public(self):
    print("public method declaration")

  def __init__(self, name, sal):
      self.__name=name    # private attribute (double underscore)
      self._salary = sal  # protected attribute (single underscore)

  def callprivate(self):
    self.__private()

obj = employee("Ravi", 100000)
obj.public()
#obj.__private()
obj.callprivate()
obj._employee__private()
print(obj._salary)
#print(obj.__name)
print(obj._employee__name)
```

Code 15 File: module1.py (Module/class demo)

```
class textfile:
  ntfiles = 0 # count of number of textfile objects
  def __init__(self, fname):
    textfile.ntfiles += 1
    self.name = fname # name
    self.fh = open(fname) # handle for the file
    self.lines = self.fh.readlines()
    self.nlines = len(self.lines) # number of lines
    self.nwords = 0 # number of words
    self.nchars = 0 # number of characters
    self.wordcount()
    self.charcount()

  def wordcount(self):
    "finds the number of words"
    for l in self.lines:
      w = l.split()
      self.nwords += len(w)

  def charcount(self):
    "finds the number of chars"
    for l in self.lines:
      self.nchars += len(l)

  def grep(self, target):
    "prints out all lines containing target"
    for l in self.lines:
      if l.find(target) >= 0:
        print(l)
```

Code 16 File: moduledemo.py (Module/class demo)

```
import module1

f1 = module1.textfile('module1.py')
f2 = module1.textfile('moduledemo.py')
print ("the number of text files open is", module1.textfile.ntfiles)
print ("here is some information about them (name, lines, words):")
for f in [f1, f2]:
  print (f.name, f.nlines, f.nwords, f.nchars)

f1.grep('self')
```

Inheritance

The syntax for a derived class definition looks as follows:

```
class DerivedClassName(BaseClassName):
   <statement-1>
   .
   .
   <statement-N>
```

The name BaseClassName must be defined in a scope containing the derived class definition. Instead of a base class name, an expression is also allowed. This is useful when the base class is defined in another module.

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the derived class object is constructed, the base class can also be created. This is used for resolving attribute references: if a requested attribute is not found in the class, it is searched in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, ascending up the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because 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 in fact end up calling a method of a derived class that overrides it. An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call '`BaseClassName.methodname(self, arguments)`'. This is occasionally useful to clients as well.

Note: When the constructor for a derived class is called, the constructor for the base class is not automatically called. If you wish the base constructor to be invoked, you must invoke it yourself.

Code 17 File: inheritance.py

```
class employee:
  def __init__(self, id, name, dept):
    self.id = id
    self._name = name     # proj is protected (one underscore)
    self.dept = dept

  def show(self):
    print("Employee details ",self.id, self._name, self.dept)
```

```
class manager(employee):
  def __init__(self, id, name, dept, role, bonus):
    # calling parent class constructor
    employee.__init__(self, id, name, dept)
    self.role = role
    self.__bonus = bonus   # private member variable (two underscores)

  def show_bonus(self):
    print("Bonus of",self.role,"is",self.__bonus)

e = employee("e101", "Subbu", "HR")
m = manager("e202", "Ravi", "IT", "Sales", "100000")

e.show()

# only the instance itself can change the __bonus variable
# and to show the value we have created a public function show_bonus()

m.show()
m.show_bonus()
#print(m.__bonus) # since it is private it is inaccessible directly
print(m._manager__bonus)
```

Multiple Inheritance

Python supports multiple inheritance. A class definition with multiple base classes looks as follows:

```
class DerivedClassName(Base1, Base2, Base3):
   <statement-1>
   .
   .
   .
   <statement-N>
```

The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

It is clear that indiscriminate use of multiple inheritance is a maintenance nightmare. A well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class. While it is easy enough to figure out what happens in this case (the instance will have a single copy of “instance variables” or data attributes used by the common base class).

Code 18 File: MI.py

```
class base1:
```

```
   def __init__(self):
      print("base1 Constructor called")
   def m1(self):
      print("base1 m1 called")

   def m2(self):
      print("base1 m2 called")

class base2:
   def __init__(self):
      print("base2 Constructor called")

   def m1(self):
      print("base2 m1 called")

   def m3(self):
      print("base2 m3 called")

class derived(base1, base2):
   def __init__(self):
      print("derived Constructor called")

   def m2(self):
      print("derived m2 called")

   def m4(self):
      print("derived m4 called")

obj = derived()
obj.m1()
obj.m2()
obj.m3()
obj.m4()
```

Important: Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a couple of named data items. An empty class definition will do nicely.

Code 19 File: class3.py

```
class Account:
   pass

customer = Account()
customer.name="ravi"
customer.number=100
customer.balance= 6000

print(customer.name)
print(customer.number)
print(customer.balance)
```

Iterator

Iteration is the process of taking one element at a time in a row of elements. Any time you use a loop, explicit or implicit, to go over a group of items, that is iteration.

In Python everything is an object. When an object is said to be iterable, it means that you can step through (i.e. iterate) the object as a collection.

Arrays for example are iterable. You can step through them with a for loop, and go from index 0 to index n, n being the length of the array object minus 1.

Dictionaries (pairs of key/value, also called associative arrays) are also iterable. You can step through their keys.

Obviously the objects which are not collections are not iterable. A bool object for example only have one value, True or False. It is not iterable.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.

The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

An Iterable is:
- anything that can be looped over (i.e. you can loop over a string or file)
- anything that can appear on the right-side of a for-loop: `for x in iterable:`
- anything you can call with iter() that will return an Iterator: `iter(obj)`
- an object that defines `__iter__` that returns a fresh Iterator.

An Iterator is:
- an object with state that remembers where it is during iteration
- an object with a __next__ method that:
    - returns the next value in the iteration
    - updates the state to point at the next value
    - signals when it is done by raising StopIteration
- an object that is self-iterable (meaning that it has an __iter__ method that returns self).

In the following s is a str object which is immutable and is iterable. s does not have any state. t is an Iterator, t has state (it starts by pointing at the "g"). t has a __next()__ method and an _iter__() method.

At the end next() raises StopIteration to signal that iteration is complete.

```
>>> s='gen'
>>> i=iter(s)
>>> next(i)
'g'
>>> next(i)
'e'
>>> next(i)
'n'
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

What does Python interpreter think when it sees following statement?

```
for x in obj:
   print(x)
```

Looks like a job for an iterator. Let's get one. There is this obj, so let's ask.

Obj do you have your iterator? (calls `iter(obj)`, which calls `obj.__iter__()`, which hands out a new iterator `_i.`)

Let's now start iterating then (`x = _i.next().. x = _i.next() .. )`

```
>>> x=True
>>> t=iter(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not iterable
```

Python expects iterable objects in several different contexts, the most important being the for statement. These two statements are equivalent:

```
for i in iter(obj):
   print(i)

for i in obj:
   print(i)
```

Code 20 File: iterator1.py

```
s = ['a', 'b']
i = iter(s)
print(next(i))
print(next(i))

print()

class Reverse:
  "Iterator for looping over a sequence backwards"
  def __init__(self, data):
    self.data = data
    self.index = len(data)

  def __iter__(self):
    return self

  def __next__(self):
    if self.index == 0:
      raise StopIteration

    self.index = self.index - 1
    return self.data[self.index]

obj = Reverse('Genesis')
for char in obj:
  print(char)
```

Code 21 File: iterator2.py

```
class Squares(object):
  def __init__(self, start, stop):
    self.start = start
    self.stop = stop

  def __iter__(self):
    return self

  def __next__(self):
    if self.start >= self.stop:
      raise StopIteration
    result = self.start * self.start
    self.start += 1
    return result

  def current(self):
    return self.start

obj = Squares(5, 10)
objIter = iter(obj)
```

```
print(obj.current())
print(next(objIter))
print(obj.current())
for item in objIter:
  print(item)
```

Operator Overloading

Code 22 File: Complex.py

```
print(complex(2, 1))
print(2 * complex(2, 1))
print(3J + complex(2, 1))
print(4J - complex(1j))
print(complex(1, 2) + complex(3, 4))
print(complex(1, 2) * complex(3, 4))
print(complex(1, 2) - complex(3, 4))
print(complex(1, 2) / complex(3, 4))
```

When we implement the equality and other operators, we cannot assume that the other object has the same attributes as self object. There are a number of methods to get around this problem, including:

1) using the hasattr function, or
2) using the isinstance function.

The hasattr function determines if other object has the attributes we are looking for before actually using them (it tells us if an object has a specific attribute or not). What's nice about this method is that we don't have to care what type other object is. We only care whether or not it contains the attributes we need to compare. However, the drawback to this function is that you have to test for the existence of each attribute, which can be a pain if an object has many attributes.

Another solution to the problem with our operator overloading example is to use the isinstance function to make sure that other is an instance of our class type. This forces the other object to be the same type as your class, which is an advantage (as attribute check is no longer required).

Python has magic methods to define overloaded behavior of operators. The comparison operators (<, <=, >, >=, == and !=) can be overloaded by providing definition to __lt__, __le__, __gt__, __ge__, __eq__ and __ne__ magic methods.

Code 23 File: operatorDemo.py

```
import inspect

def line_number():
   '''Returns the current line number in our program'''
   return inspect.currentframe().f_back.f_lineno

class MyComplex:
  def __init__(self, rp = 1, ip = 2):
    print("C", "Constructor called")
    self.rpart = rp
    self.ipart = ip

  def __del__(self):
    print("D", "destructor called",  id(self))

  def __eq__(self, other):
    print("E", "__eq__ called")
    if (isinstance(other, MyComplex)):
      return ((self.rpart == other.rpart) and (self.ipart ==
other.ipart))
    else:
      return "NotImplemented"

  def __ne__(self, other):
    print("NE", "__ne__ called")
    # By using the == operator, the returned NotImplemented is handled
correctly.
    return not self == other

  def __add__(self, other):
    print("A", "__add__ called")
    if(hasattr(other, "rpart") and hasattr(other, "ipart")):
      return (MyComplex(self.rpart + other.rpart, self.ipart +
other.ipart))

obj1 = MyComplex(4)
print(line_number(), id(obj1), obj1.rpart, obj1.ipart)

obj2 = MyComplex(2)
print(line_number(), id(obj2), obj2.rpart, obj2.ipart)

if(obj1 == obj2):
  print(line_number(), "Equal")
else:
  print(line_number(), "Not Equal")

i = 10
print(line_number(), obj1 == i)

obj3 = obj1 + obj2
```

```
print(line_number(), id(obj3), obj3.rpart, obj3.ipart)

if(obj1 != obj2):
  print(line_number(), "Not Equal")
else:
  print(line_number(), "Equal")
```

Code 24 File: myIntOperators.py

```
class myInt:
  def __init__(self, val):
    print("__init__")
    self.num = val

  def __call__(self, val):
    print("__call__")
    self.num = val

  def __eq__(self, other):
    if self.num == other.num:
      return "both objects are equal"
    else:
      return "both objects are not equal"

  def __lt__(self, other):
    if self.num < other.num:
      return "first object lesser than other"
    else:
      return "first object not lesser than other"

  def __gt__(self, other):
    if self.num > other.num:
      return "first object greater than other"
    else:
      return "first object not greater than other"

  def __ge__(self, other):
    if self.num >= other.num:
      return "first object greater than equal to other"
    else:
      return "first object not greater than equal to other"

  def __str__(self):
    return str(str(self.num))

  def __repr__(self):
    return "myInt(" + str(self.num) + ")"

obj1 = myInt(5)     # __init__ Initialise newly created object
print(obj1.num)
obj1(20)            # __call__
print(obj1.num)
```

```
obj2 = myInt(5)
print (obj1 > obj2)
print (obj1 >= obj2)
print (obj1 < obj2)
print(obj1 == obj2)
print(obj1)
print(repr(obj1))
```

Code 25 File: get set attr.py

```
class Employee(object):
  def __init__(self, empDict, dept):
    print("__init__")
    super().__setattr__('data', dict())
    self.data = empDict
    self.department = dept

  def __getattr__(self, name):
    print("__getattr__", name)
    if name in self.data:
      return self.data[name]
    else:
      return 0

  def __setattr__(self, key, value):
    print("__setattr__", key, value)
    if key in self.data:
      self.data[key] = value
    else:
      super().__setattr__(key, value)

emp = Employee({'age': 23, 'name': 'John'}, 'HR')
print(emp.age)
print(emp.name)
print(emp.data)
print(emp.salary)

emp.salary = 50000
print(emp.salary)

#delattr(emp, 'salary')
del(emp.salary)
print(emp.salary)
```

Exceptions

Errors detected during runtime are exceptions. Most exceptions when not handled by programs result in following error messages.

```
>>> print(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero

>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 's' is not defined

>>> str='Genesis'
>>> print(str + 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in name for the exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords). The rest of the line is a detail whose interpretation depends on the exception type; its meaning is dependent on the exception type.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines.

The following example, asks the user for input until a valid integer has been entered, but allows the user to interrupt the program.

Code 26 File: exception1.py

```
while True:
   try:
      val = int(input("Please enter a number:"))
      print(val)
      break
  except ValueError:
      print("Entered data is not numeric. Try again.")
```

The try statement works as follows.

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the rest of the try clause is skipped, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized list, for example:

```
... except (ZeroDivisionError, TypeError, NameError):
... pass
```

The last except clause may omit the exception name(s), to serve as a wildcard, to catch all exceptions. It can be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well).

Code 27 File: exception2.py

```
try:
  print(1/0)
  print(val)
  str='Genesis'
  print(str + 3)

except ZeroDivisionError as z:
  print(repr(z))

except NameError as n:
  print(str(n))

except TypeError as t:
  print(repr(t))

except:
  print("Some unhandled exception occured")
  raise
```

Exceptions should be class objects. The exceptions are defined in the module exceptions. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the exceptions module.

For class exceptions, in a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the raise statement. If the exception class is derived from the standard root class BaseException, the associated value is present as the exception instance's args attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be sub-classed to define new exceptions; programmers are recommended to derive new exceptions from the Exception class and not from BaseException.

Exception Hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EnvironmentError
      |    +-- IOError
      |    +-- OSError
      |         +-- WindowsError (Windows)
      |         +-- VMSError (VMS)
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
```

```
      |    +-- UnboundLocalError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Code 28 File: UserDefinedException.py

```
class WithdrawException(Exception):
   def __init__(self, value):
      self.value = value
      self.message = "Cannot withdraw as the amount to withdraw is
         more than allowed"

class account:
   def __init__(self, name, number, balance):
      self.name = name
      self.number = number
      self.balance = balance

   def withdraw(self, amt):
      newbalance = self.balance - amt
      if(newbalance < 1000):
         raise(WithdrawException(amt))

   def deposit(self, amt):
      self.balance += amt

   def show(self):
      print("Name is " + self.name)
      print("number is ", self.number)
      print("Balance is ", self.balance)
```

```
customer = account("Cust1", 200, 4000)

try:
  customer.deposit(2000)
  customer.withdraw(8500)
  customer.show()
except WithdrawException as we:
  print(we.message, we.value)
  print(type(we))
except Exception as e:
  print(repr(e))
```

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not. When an exception occurs in the try clause and has not been handled by an except clause (or it has occurred in a except or else clause), it is re-raised after the finally clause has been executed. The finally clause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or return statement.

As you can see, the finally clause is executed in any event. The TypeError raised by dividing two strings is not handled by the except clause and therefore re-raised after the finally clause has been executed.

In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

The try, except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

Code 29 File: exception3.py

```
import sys

def func(nume, deno):
  if(deno == 0):
    raise ZeroDivisionError(nume, deno)
  return nume/deno

def main():
  try:
    result = func(10, 0)

  except ZeroDivisionError:
    print("Division by zero error")
    sys.exit(0)
  else:
    print("result is", result)
  finally:
    print("In finally block")
```

```
  print("After finally block")

main()
```

Regular Expression

A regular expression is a sequence of characters that helps match or find other strings or sets of strings, using a specialized syntax held in a pattern.

The module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as r'expression'.

Match function attempts to match RE pattern to string with optional flags.

```
re.match(pattern, string, flags=0)
```

The re.match function returns a match object on success containing information about the search and the result, None on failure. We would use group(num) or groups() function of match object to get matched expression.

group(num=0) This method returns entire match (or specific subgroup num).
groups() This method returns all matching subgroups in a tuple or empty.

match ⇒ find something at the beginning of the string and return a match object.
.
The search function searches for first occurrence of RE pattern within string with optional flags.

```
re.search(pattern, string, flags=0)
```

The re.search function returns a match object on success, None on failure. We would use group(num) or groups() function of match object to get matched expression.

group(num=0) This method returns entire match (or specific subgroup num).
groups() This method returns all matching subgroups in a tuple or empty.

search ⇒ find something anywhere in the string and return a match object

Regular-expression Modifiers - Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|):

re.I Performs case-insensitive matching.
re.L Interprets words according to the current locale.
re.M Makes $ match the end of a line and makes ^ match the start of any line.
re.S Makes a period (dot) match any character, including a newline.
re.U Interprets letters according to the Unicode character set.
re.X It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

The sub method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided. This method would return modified string.:

```
re.sub(pattern, repl, string, max=0)
```

Regular-expression patterns:

Except for control characters, (+ ? . * ^ $ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.
^ Matches beginning of line.
$ Matches end of line.
. Matches any single character except newline. Using m option allows it to match newline as well.
[a-z] Matches any single character in brackets.
[^a-z] Matches any single character not in brackets
re* Matches 0 or more occurrences of preceding expression.
re+ Matches 1 or more occurrence of preceding expression.
re? Matches 0 or 1 occurrence of preceding expression.
re{n} Matches exactly n number of occurrences of preceding expression.
re{n,} Matches n or more occurrences of preceding expression.
re{n, m} Matches at least n and at most m occurrences of preceding expression.
a|b Matches either a or b.
(re) Group regular expressions and remember matched text.

\w Matches word characters.
\W Matches nonword characters.
\s Matches whitespace. Equivalent to [\t\n\r\f].
\S Matches nonwhitespace.
\d Matches digits. Equivalent to [0-9].
\D Matches nondigits.
\A Matches beginning of string.
\Z Matches end of string. If a newline exists, it matches just before newline.
\z Matches end of string.
\b Returns a match where the specified characters are at the beginning or at the end of a word
\B Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word
\n, \t, etc. Matches newlines, carriage returns, tabs, etc.

Character classes:

[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

Special Character Classes:

ruby?	Match "rub" or "ruby"
ruby*	Match "rub" plus 0 or more y
ruby+	Match "rub" plus 1 or more y
\d{3}	Match exactly 3 digits
\d{3,}	Match 3 or more digits
\d{3,5}	Match 3, 4, or 5 digits

Special Character Classes:

.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a nondigit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match nonwhitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a nonword character: [^A-Za-z0-9_]

Example	Description
<*>	Greedy
<.*?>	Nongreedy

Code 30 File: regularexpr1.py

```
import re

data = "Python is used in Machine Learning"

result = re.search("^Python.*Learning$", data)
if (result):
   print("Match")
else:
   print("No match")

searchObj = re.search("\s", data)
```

```
print("First white-space character is located @ position:",
searchObj.start())

searchObj = re.split("\s", data)
print(searchObj)

searchObj = re.sub("\s", "_", data)
print(searchObj)

searchObj = re.search("used", data)
print(searchObj)
```

Code 31 File: regularexpr2.py

```
import re

list = ["Issue12 testing", "Bug  development", "CR22 documentation"]

for element in list:
  matchObj = re.match("([ibc]\w+)\W+(d\w+)", element, re.M|re.I)
  if matchObj:
    print(matchObj.groups())
  else:
    print("No match found for:", element)

data = "Qualcomm PythonTraining. Training starts in India at 9:30 AM
IST."
matchObj = re.match('Qualcomm\W+(\w+)\W+Training(.*)India', data,
re.M|re.I|re.S)

if matchObj:
  print("Matched string : ", matchObj.group())
  print("Matched string 1: ", matchObj.group(1))
  print("Matched string 1: ", matchObj.group(2))
else:
   print("No match result")

data = "Qualcomm Python Training. Training starts in India at 9:30 AM
IST."

searchObj = re.search('Training(.*) in (.*?) .*', data,
re.M|re.I|re.S)

if searchObj:
   print("Search string : ", searchObj.group())
   print("Search string 1 : ", searchObj.group(1))
   print("Search string 2 : ", searchObj.group(2))
else:
   print("No search result")
```

Code 32 File: regularexpr3.py

```
import re

phone = "123-456-7890"
num = re.sub('#.*$', "", phone)
print("Phone Num : ", num)

Number = "123 qualcomm 456genesis  "

# Replace the first 5 occurrences
num = re.sub('\D', "", Number, 5)
print("Num1 : ", num)

num = re.sub('\d', "", Number)
print("Num2 : ", num)
num = re.sub('\w', "A", Number)
print("Num3 : ", num)

# Replace the first 4 occurrences
num = re.sub('\W', "%", Number, 4)
print("Num4 : ", num)

num = re.sub('\s', "#", Number)
print("Num5 : ", num)

# Replace the first 2 occurrences
num = re.sub('\s', "#", Number, 2)
print("Num6 : ", num)
```

Code 33 File: regularexpr4.py

```
import re

phone = 'Telephone: 040-12345678';

searchNo = re.search('Telephone:\s+(\d{3}-\d{8}$)', phone, re.M)

if searchNo:
   print("Search string : ", searchNo.group())
   print("Search string 1 : ", searchNo.group(1))
else:
   print("No search result")

line="What is the best time to call you, time now is 10:30 AM";

searchData = num = re.sub('the.*time', "tested", line)
print("searchData : ", searchData)

searchData = num = re.sub('the.*?time', "tested", line)
print("searchData : ", searchData)
```

Re.findall() module is used when you want to iterate over the lines of the file, it will return a list of all the matches in a single step. For example, here we have a list of e-mail addresses, and we want all the e-mail addresses to be fetched out from the list, we use the re.findall method.

The re.compile() method:

```
re.compile(pattern, flags = 0)
```

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

Code 34 File: regularexpr5.py

```
import re

data = "Python is used in Machine Learning"
result = re.findall("in", data)
print(result)

mailIds = 'rtapadia@gmail.com, info2@hotmail.com,
testing@yahoomail.com'

emails = re.findall(r'[\w\.-]+@[\w\.-]+.[\w\.-]', mailIds)

print("Found emails:", emails)
for email in emails:
  print(email)

str = "Genesis   Insoft Limited   Hyderabad."
lst = re.split("\s", str)
print(lst)

# Split @ the 3rd occurrence
lst = re.split("\s", str, 2)
print(lst)

pattern = re.compile('python')
result = pattern.findall('training in python. Scripting language
python')
print(result)
result1 = pattern.findall('training in python @ qualcomm')
print(result1)
```

The Match object has properties and methods used to retrieve information about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match.
.string returns the string passed into the function
.group() returns the part of the string where there was a match

Code 35 File: regularexpr6.py

```
import re

str = "The rainS in Spain"
x = re.search(r"\bS\w+", str)
if x:
  print(x.span())
  print(x.string)
  print(x.group())

x = re.search(r"inS\b", str)
if x:
  print(x.span())
  print(x.string)
  print(x.group())
```

The following example illustrates how to process data from a file and apply regular expression to extract and format data in a meaningful way.

Code 36 File: issues.txt

```
# Type   Code -> ProjectCode  Count
Bugs 1 -> 11 [Count=25];
Bugs 1 -> 21 [Count=23];
Counts 2 -> 31 [Count=17];
Fixed 3 -> 41 [Count=12];
Open 4 -> 51 [Count=1];
Bugs 1 -> 62 [Count=50];
Counts 2 -> 72 [Count=7];
Fixed 3 -> 62 [Count=27];
Open 4 -> 62 [Count=23];
```

Code 37 File: regExpIssues.py

```
import re

file = open("issues.txt", "r")

for line in file:
  print(line, end="")

file.seek(0,0)
print()

print("Code, ProjectCode, Count")
for line in file:
  if re.search('->',line):
    print(','.join(re.findall('[0-9]+',line)))

file.close()
```