Welcome to GKTCS

IT Training .Consultancy .
Software Development. Staffing

Surendra Panpaliya

Director, GKTCS Innovations Pvt. Ltd, Pune.

16 + Years of Experience (MCA, PGDCS, BSc. [Electronics] , CCNA)

- Founder,GKTCS Innovations Pvt. Ltd. Pune [Nov 2009 – Till date]
- 500 + Corporate Training for HP, IBM, Cisco,Wipro, Samsung etc.
- **Skills**
  - ❑ **Python, Perl,Jython, Django, Android ,**
  - ❑**Ruby, Rail, Cake PHP, LAMP**
  - ❑ Data Communication & Networking, CCNA
  - ❑UNIX /Linux Shell Scripting, System Programming
  - ❑ CA Siteminder, Autosys, SSO, Service Desk, Service Delivery
- Author of 4 Books
- National Paper Presentation Awards at BARC Mumbai

Agenda

Day	Module	Topics
Day 3	Module 8	Object Oriented Programming
	Module 9	File Handling
	Module 10	Exception Handling

Module 8: Object Oriented Programming Concepts

- Introduction to object oriented concepts
- Classes and Objects
- The “self” keyword
- Methods and Attributes
- Constructor and Destructor
- Instance and static member
- Class Inheritance
- Super keyword

Object Oriented Programming in Python: Defining Classes

It's all objects…

- Everything in Python is really an object.
  - We've seen hints of this already…

```
"hello".upper()
list3.append('a')
dict2.keys()
```

  - These look like Java or C++ method calls.
  - New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
  - The *class* also stores some data items that are shared by all the instances of this class.
  - *Instances* are objects that are created which follow the definition given inside of the class.
- Python doesn't use separate class interface definitions as in some languages. You just define the class and then use it.

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block.
  - There must be a special first argument **`self`** in *all* method definitions which gets bound to the calling instance
  - There is usually a special method called **`__init__`** in most classes
  - We'll talk about both later…

A simple class definition: *student*

```
class student:
  """A class representing a
  student."""
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Creating and Deleting Instances

Instantiating Objects

- There is no “new” keyword as in Java.
- Merely use the class name with () notation and assign the result to a variable.
- `__init__` serves as a constructor for the class. Usually does some initialization work.
- The arguments passed to the class name are given to its `__init__()` method.
  - So, the __init__ method for student is passed “Bob” and 21 here and the new class instance is bound to b:

```
b = student(“Bob”, 21)
```

Constructor: __init__

- An `__init__` method can take any number of arguments.
  - Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.

- However, the first argument `self` in the definition of __init__ is special…

Self

- The first argument of every method is a reference to the current instance of the class.
  - By convention, we name this argument ***self***.

- In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.
  - Similar to the keyword *this* in Java or C++.
  - But Python uses *self* more often than Java uses *this*.

Self

- Although you must specify *self* explicitly when *defining* the method, you don't include it when *calling* the method.
- Python passes it for you automatically.

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
    self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Deleting instances: No Need to "free"

- When you are done with an object, you don't have to delete or free it explicitly.
  - Python has automatic garbage collection.
  - Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
  - Generally works well, few memory leaks.
  - There's also no "destructor" method for classes.

Access to Attributes and Methods

Definition of student

```
class student:
  """A class representing a
  student."""
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Traditional Syntax for Access

```
>>> f = student ("Bob Smith", 23)

>>> f.full_name     # Access an attribute.
"Bob Smith"

>>> f.get_age()      # Access a method.
23
```

Accessing unknown members

- Problem: Occasionally the name of an attribute or method of a class is only given at run time…
- Solution: `getattr(object_instance, string)`
  - **`string`** is a string which contains the name of an attribute or method of a class
  - **`getattr(object_instance, string)`** returns a reference to that attribute or method

getattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)

>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
 <method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_age")()    # We can call this.
23

>>> getattr(f, "get_birthday")
      # Raises AttributeError - No method exists.
```

hasattr(object_instance,string)

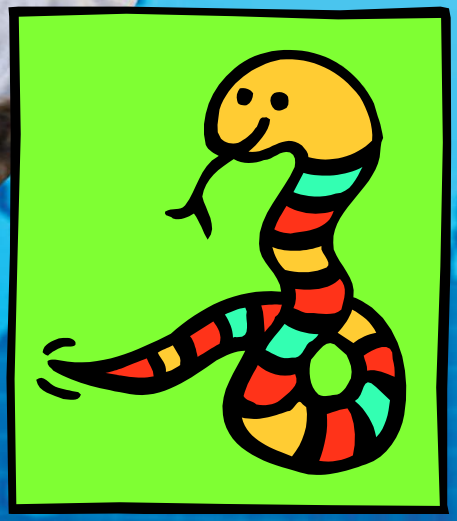
```
>>> f = student("Bob Smith", 23)

>>> hasattr(f, "full_name")
True

>>> hasattr(f, "get_age")
True

>>> hasattr(f, "get_birthday")
False
```

Attributes



Two Kinds of Attributes

- The non-method data stored by objects are called attributes.

- *Data* attributes
  - Variable owned by a *particular instance* of a class.
  - Each instance has its own value for it.
  - These are the most common kind of attribute.

- *Class* attributes
  - Owned by the *class as a whole*.
  - *All instances of the class share the same value for it.*
  - Called “static” variables in some languages.
  - Good for
    - class-wide constants
    - building counter of how many instances of the class have been made

Data Attributes

- Data attributes are created and initialized by an `__init__()` method.
  - Simply assigning to a name creates the attribute.
  - Inside the class, refer to data attributes using **`self`**
    - for example, **`self.full_name`**

```
class teacher:
   "A class representing teachers."
   def __init__(self,n):
       self.full_name = n
   def print_name(self):
       print self.full_name
```

Class Attributes

- Because all instances of a class share one copy of a class attribute:
  - when *any* instance changes it, the value is changed for *all* instances.
- Class attributes are defined
  - *within* a class definition
  - *outside* of any method

- Since there is one of these attributes *per class* and not one *per instance*, they are accessed using a different notation:
  - Access class attributes using `self.__class__.name` notation.

```
class sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Data vs. Class Attributes

```
class counter:
   overall_total = 0
         # class attribute
   def __init__(self):
       self.my_total = 0
         # data attribute
   def increment(self):
      counter.overall_total = \
      counter.overall_total + 1
      self.my_total = \
      self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Inheritance

Inheritance

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

Multiple Inheritance

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

Private Variables

- Any identifier of the form __spam
- textually replaced with _classname__spam

Defining own class

```
class Stack:
    def _ _init_ _(self, data):
        self._data = list(data)
    def push(self, item):
        self._data.append(item)
    def pop(self):
        item = self._data[-1]
        del self._data[-1]
        return item
```

Execution

```
>>> thingsToDo = Stack(['write to mom',
   'invite friend over', 'wash the kid'])
>>> thingsToDo.push('do the dishes')
>>> print thingsToDo.pop()
do the dishes
>>> print thingsToDo.pop()
wash the kid
```

UserList

```
from UserList import UserList    # subclass the UserList
    class
class Stack(UserList):
    push = UserList.append
    def pop(self):
        item = self[-1] # uses _ _getitem_ _
        del self[-1]
        return item
```

Execution

```
>>> thingsToDo = Stack(['write to mom', 'invite friend
    over', 'wash the kid'])
 >>> print thingsToDo
 ['write to mom', 'invite friend over', 'wash the kid']
>>> thingsToDo.pop()
'wash the kid'
>>> thingsToDo.push('change the oil')
>>> for chore in thingsToDo:
...         print chore
```

What is an object?

- Objects are collections of data and functions that operate on that data.
- These are bound together so that you can pass an object from one part of your program and they automatically get access to not only the data *attributes* but the *operations* that are available too.
- This combining of data and function is the very essence of Object Oriented Programming and is known as *encapsulation*.

What is a Class?

- Data has various types so objects can have different types.
- These collections of objects with identical characteristics are collectively known as a *class*.
- We can define classes and create *instances* of them, which are the actual objects.
- We can store references to these objects in variables in our programs.

What are polymorphism and inheritance?

- If we have two objects of different classes but which support the same set of messages but with their own corresponding methods.
- We can collect these objects together and treat them identically in our program but the objects will behave differently.
- This ability to behave differently to the same input messages is known as *polymorphism*.

Inheritance

- Inheritance is often used as a mechanism to implement polymorphism.
- A class can *inherit* both attributes and operations from a *parent* or *super* class.
- A new class which is identical to another class in most respects does not need to re-implement all the methods of the existing class,
- it can inherit those capabilities and then *override* those that it wants to do differently

Using a trivial class and made up attributes

```
>>> class null: # a do nothing much class
        ... pass # do nothing statement
        ...
>>> a=null()        # a is an object created by null class.

>>> b=null()        # b is another object

>>> a.c=2           # give object a an attribute c with value 2

 >>> b.d=4          # same kind of deal

>>> a.c+b.d         # add the value attributes and print
 6
```

Methods

A class method is a function that knows its object.

```
>>> class rectangle:
        ... def area(self):
                ... return self.width*self.height
...
>>> a=rectangle()
>>> a.width=2
>>> a.height=3
>>> a.area()
 6
```

Constructor methods

```
# geometry module: constructorex.py
class rectangle:   # rectangle class
    # make a rectangle using top left and bottom right
    coordinates
    def __init__(self,tl,br):
            self.tl=tl self.br=br
            self.width=abs(tl.x-br.x) # width
            self.height=abs(tl.y-br.y) # height
    def area(self): # gets area of rectangle
            return self.width*self.height
```

Constructor methods cont..

```
class coordinate: # coordinate class
    def __init__(self,x,y):
    # make a coordinate object with a
    #       reference (self), an x and a y
            self.x=x
            self.y=y
    def distance(self,another):
                                    # distance between 2 coordinates
    import math
    xdist=abs(self.x-another.x)
    ydist=abs(self.y-another.y)
 return math.sqrt(xdist**2+ydist**2)
                                    # pythagoras theorem
```

Constructorex Package

- Constructorex Package contains 2 classes, coordinate and rectangle. The following commands import this package, construct 2 coordinates and a rectangle and then calculate the area of the rectangle and the distance between the 2 coordinates:

```
>>> a=constructorex.coordinate(2,3)
>>> b=constructorex.coordinate(5,7)
>>> c=constructorex.rectangle(a,b)
>>> c.area()
12
>>> a.distance(b)
5.0
```

Class data attributes

Case Study : Washing Machine Factory

If each washing machine manufactured from a production line has its own unique serial number, how does the factory know which serial number to give to the next washing machine off the production line ?

Solution:

- For this we use a **class attribute**. Here is a class which simulates a washing machine, with class attribute: no_made.

washing.py class

- # washing module file: washing.py

```
class machine:
        no_made=0
        def __init__(self):
            machine.no_made+=1
            self.serial=machine.no_made
        def spin(self):
            print “wheeeeeeeeeeeeeeeeeeeeeee!!"
        def wash(self):
            print "slosh slosh slosh slosh slosh“
        def label(self):
            print "washing machine: %d" % self.serial
```

washing.py class

- The following commands were used to test this class:

```
>>> import washing
>>> a=washing.machine()
 >>> a.wash()
slosh slosh slosh slosh slosh
>>> a.spin()
wheeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee!!
>>> a.serial
 1
>>> a.no_made
 1
>>> b=washing.machine()
 >>> washing.machine.no_made
2
```

washing.py class

```
>>> c=washing.machine()
>>> a.no_made
3
>>> b.serial
2
>>> c.serial
3
>>> c.label()
washing machine: 3
>>> a.label()
washing machine: 1
```

Notice that only one copy of the class attribute: no_made exists, but every object has its own serial number.

Inheritance

- Classes can also inherit from superclasses or parent classes. Again both data and functional attributes are inherited.
- Inheritance allows you to have a general class and to create a number of specialised versions of it.
- The child or specialised classes reuse code within the generalised parent class, and add some of their own, to override attributes within the parent, and by adding new ones.
- EX. Here we have a **Suprex Deluxe washing machine** which does all that our generalised washing machine does, but it has a model attribute, a tumble dry cycle and **overides the label** method within its parent.

suprex.py

```
#  file: suprex.py
from washing import machine
class deluxe(machine):
                    # deluxe subclasses parent class machine
          model="Suprex Deluxe" # adds an attribute

          def tumble_dry(self): # adds a method
                    print "tumble tumble chug tumble tumble chug"
           def label(self): # overrides a method in parent class
                    print "Model: %s Serial No: %d" %
                                        (deluxe.model,self.serial)
```

suprex.py

- Here is a test run, with comments after # symbols
- >>> import suprex
- >>> a=suprex.deluxe() # make a suprex deluxe using parent constructor
- >>> b=suprex.machine() # suprex module can construct object of parent class
- >>> a.tumble_dry() # a suprex can tumble dry tumble tumble chug tumble tumble chug
- >>> a.wash() # suprex knows how to wash from parent slosh slosh slosh slosh slosh
- >>> b.wash() # so can an ordinary machine slosh slosh slosh slosh slosh

suprex.py

```
>>> b.tumble_dry() # ordinary machines can't tumble dry Traceback (most
    recent call last): File "<interactive input>", line 1, in ? AttributeError:
    machine instance has no attribute 'tumble_dry'
>>> a.serial # a got this object attribute from parent class
 1
>>> b.serial # can the parent also count instances of children ?
2
>>> a.label() # suprex has its own method for this
Model: Suprex Deluxe Serial No. 1
 >>> b.label() # ordinary machines have other code
 washing machine:  2
 >>> b.no_made # no_made attribute is accessible through both classes
 2
>>> a.no_made
 2
```

Polymorphism

- This word means something having many forms
- You can override built in names in Python, e.g. by defining your own len() function and localising the override to the scope where this is needed.
- You can override class methods by subclassing if this is useful.
- Python classes also allow you to define methods with special names e.g: __add__(), __del__(), so that you can define what happens when you use + and - operators between your objects.
- Many Python operators can be overriden for class objects.
- In the following example we use the __getitem__ method to override what happens when we index an object:

use of __getitem__ to intercept indexing operations

```
>>> class mystring:
        ... def __getitem__(self,index):
                ... import string
                ... capital=string.upper(self.contents[index])                    ...
    return capital
        ...

>>> a=mystring()
>>> a.contents="abcdefghijklmnopqrstuvwxyz"
>>> a[0] # __getitem__ method overides indexing operator
'A‘
 >>> a[25]
'Z'
>>> a.contents[25] # a.contents was and still is lower case
 'z‘
```

use of __repr__ to intercept print operations

```
>>> class printmachine(suprex.deluxe):
... def __repr__(self):
        ... return "Instance of model: %s Serial Number:                %d" %
    (self.model,self.serial)
...

>>> a=printmachine()
>>> print a
Instance of model: Suprex Deluxe Serial Number: 4
```

Controlling access to class and object attributes

- Up to a point people won't go into houses where they're not welcome.
- If the nature of your project is such that the security needs of your classes go beyond the assumption that unintended forms of access is someone else's problem,
- Python does allow you to code methods called __getattr__ and __setattr__ to intercept read and write access to your class attributes.
- These methods can force consistent attribute behaviour when unknown attributes are referenced or inappropriate access is made to values which should be managed inside the class.

Controlling access to class and object attributes

- **__getattr__(*self, name)***
- Returns a value for an attibute when the name is not an instance attribute nor is it found in any of the parent classes. *name is the attribute name. This method returns the attribute value or raises an* AttributeError exception.
- **__setattr__(*self, name, value)***
- Assigns a value to an attribute. *name is the attribute name, value is the value to assign to it.* Note that if you naively do 'self.name= value' in this method, you will have an infinite recursion of __setattr__() calls.
- To access the internal dictionary of attributes, __dict__, you have to use the following:
- 'self.__dict__[name] = value'.

A class which controls access to its attributes

```
class locked_data:
        max=100 # constant
        def __init__(self,module="WPA4"):
                self.module=module
        def __getattr__(self,attrib):
             if attrib == "title":
                    return "Website Programming Applications IV“
            else: # redirect access to unknown attributes
                     return    self.module
         def __setattr__(self,attrib,value):
            if attrib in ["module","title"]:
                     self.__dict__[attrib]=value
        # List the attributes which can be written to here.
# Have to access through __dict__ to avoid infinite regression
           else:
                    raise AttributeError
```

Demonstrates

This test run demonstrates attribute read redirections and write locking-mechanisms :

```
>>> from locked import locked_data
>>> a=locked_data()
>>> a.max # constant class attribute
    100
 >>> a.title # default values
 'Website Programming Applications IV‘
 >>> a.module
     'WPA4‘
 >>> a.thing # __getattr__ returns module for
            unknown attribute
'WPA4'
```

Demonstrates

```
>>> a.max=50 # __setattr__ prevents write to class
    constant Traceback (most recent call last): File
    "<interactive input>",line 1, in ? File "locked.py", line
    15, in __setattr__
    raise AttributeError
 AttributeError
>>> a.max # a.max stays the same
100
```

Demonstrates

```
>>> a.thing=42 # can't write to non-existent attribute
 Traceback (most recent call last): File "<interactive input>", line
     1, in ?
 File "locked.py", line 15, in __setattr__ raise AttributeError
AttributeError
>>> a.title="Another" # can change module and/or title
 >>> a.module="new"
>>> a.title
'Another'
>>> a.module
 'new'
```

Subclasses

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
class ai_student(student):
```

  - Python has no 'extends' keyword like Java.
  - Multiple inheritance is supported.

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won't get executed.

- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

  - **The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.**

Definition of a class extending student

```
class student:
   "A class representing a student."

   def __init__(self,n,a):
       self.full_name = n
       self.age = a

   def get_age(self):
       return self.age
```

```
class ai_student (student):
   "A class extending student."

   def __init__(self,n,a,s):
       student.__init__(self,n,a) #Call __init__ for student
       self.section_num = s

   def get_age():    #Redefines get_age method entirely
       print "Age: " + str(self.age)
```

Extending __init__

- Same as for redefining any other method…
  - Commonly, the ancestor's **`__init__`** method is executed in addition to new commands.
  - You'll often see something like this in the **`__init__`** method of subclasses:

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class.

Super keyword

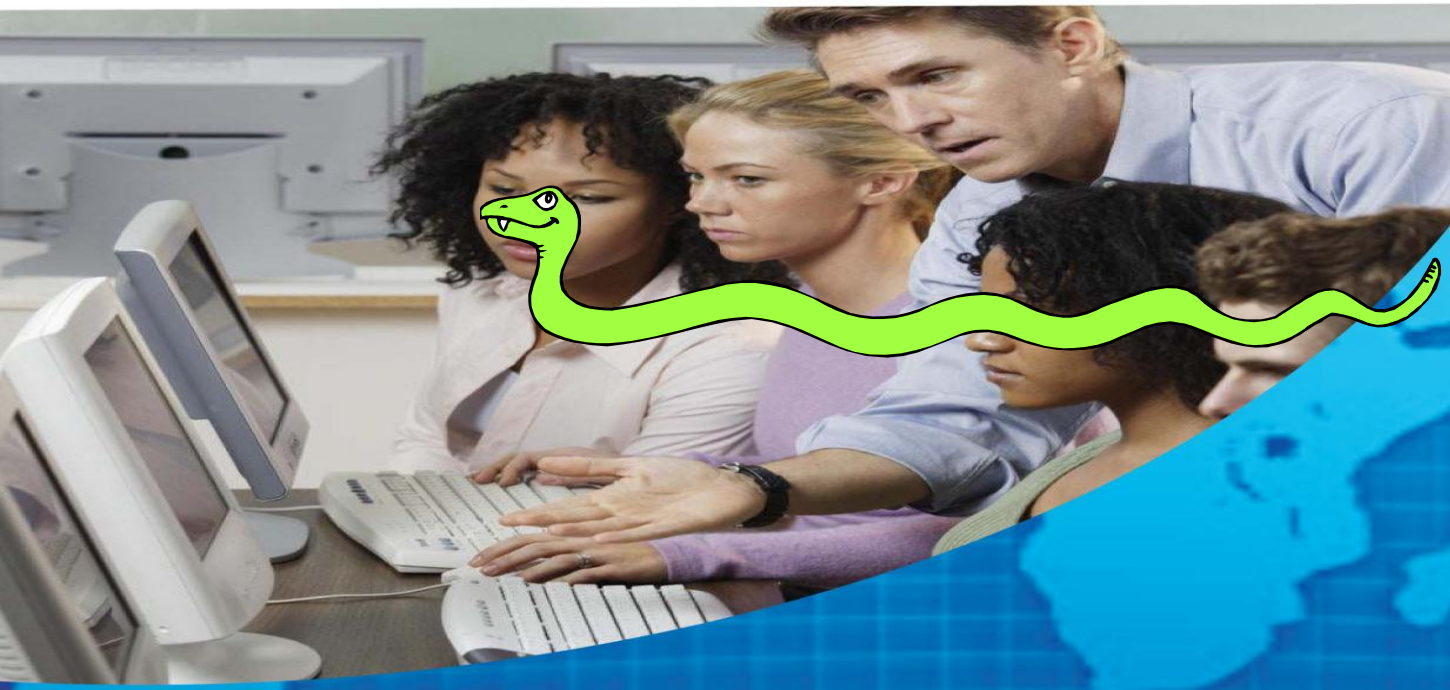
- We can use super() to distinguish between method functions with the same name defined in the superclass and extended in a subclass.
- **super(*type, variable)***
- This will do two things: locate the superclass of the given type, and it then bind the given variable to create an object of the superclass.
- This is often used to call a superclass method from within a subclass: 'super(classname ,self).method()'

Super keyword

- Here's a template that shows how a subclass __init__() method uses super() to evaluate the superclass __init__() method.

```
class Subclass( Superclass ):
    def __init__( self ):
        super(Subclass,self)__init__()
        # Subclass-specific stuff follows
```

Special Built-In Methods and Attributes



Built-In Members of Classes

- Classes contain many methods and attributes that are included by Python even if you don't define them explicitly.
  - Most of these methods define automatic functionality triggered by special operators or usage of that class.
  - The built-in attributes define information that must be stored for all classes.
- All built-in members have double underscores around their names: `__init__` `__doc__`

Special Methods

- For example, the method `__repr__` exists for all classes, and you can always redefine it.
- The definition of this method specifies how to turn an instance of the class into a string.
  - **`print f`** sometimes calls **`f.__repr__()`** to produce a string for object f.
  - If you type **`f`** at the prompt and hit ENTER, then you are also calling **`__repr__`** to determine what to display to the user as output.

Special Methods – Example

```
class student:
    ...
     def __repr__(self):
       return "I'm named " + self.full_name
    ...

>>> f = student("Bob Smith", 23)
>>> print f
I'm named Bob Smith
>>> f
"I'm named Bob Smith"
```

Special Data Items

- These attributes exist for all classes.

  `__doc__` : Variable storing the documentation string for that class.

  `__class__` : Variable which gives you a reference to the class from any instance of it.

  `__module__` : Variable which gives you a reference to the module in which the particular class is defined.

Useful:

- **`dir(x)` returns a list of all methods and attributes defined for object x**

Special Data Items – Example

```
>>> f = student("Bob Smith", 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >

>>> g = f.__class__("Tom Jones", 34)
```

Private Data and Methods

- Any attribute or method with two leading underscores in its name (but none at the end) is private. It cannot be accessed outside of that class.
  - Note:

  Names with two underscores at the beginning ***and the end*** are for built-in methods or attributes for the class.
  - Note:

  There is no 'protected' status in Python; so, subclasses would be unable to access these private data either.

File Handling

- What is File Input output?
- How to open a file
- How to close a file
- Read and write data to a file
- Pickle Module

Files

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at
  80a0960>
```

Methods of File Objects

- call f.read(*size*), which reads some quantity of data and returns it as a string. *size* is an optional numeric argument.
- When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory

```
>>> f.read()
'This is the entire file.\n'
```

readline

- f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline.

```
>>> f.readline()
'This is the first line of the file.\n'
```

readlines

- f.readlines()
- >>> for line in f:
  - print line

leftovers

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5) # Go to the 6th byte in the
   file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte
   before the end
>>> f.read(1)
```

Pickle Module

pickle and cPickle

- Purpose Python object serialization
- The pickle module implements an algorithm for turning an arbitrary Python object into a series of bytes. This process is also called serializing" the object.
- The byte stream representing the object can then be transmitted or stored, and later reconstructed to create a new object with the same characteristics.
- The cPickle module implements the same algorithm, in C instead of Python.
- It is many times faster than the Python implementation, but does not allow the user to subclass from Pickle.

pickle and cPickle

- **Encoding and Decoding Data in Strings**

```
try:
        import cPickle as pickle
except:
        import pickle
import pprint
data = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'DATA:',
pprint.pprint(data)
data_string = pickle.dumps(data)
print 'PICKLE:', data_string
```

pickle_unpickle.py

```
try:
    import cPickle as pickle
except:
    import pickle
import pprint
data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)
data1_string = pickle.dumps(data1)
data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)
print 'SAME?:', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```

pickle_unpickle.py

```
try:
    import cPickle as pickle
except:
    import pickle
import pprint
data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)
data1_string = pickle.dumps(data1)
data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)
print 'SAME?:', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```

Exception Handling

Exception Handling

- What is an Exception?
- Run time Exceptions
- try … except statements
- Multiple except statements
- Clean up statement (finally)
- Raised exceptions
- User defined exceptions

Try block

```
while True:
    try:
        x = int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print "Oops! That was no valid number. Try again..."
```

Multiple except

```
import sys
try:
    f = open('myfile.txt'); s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer.“
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Argument to Exception

```
try:
    raise Exception('spam', 'eggs')
    except Exception, inst:
        print type(inst)
        print inst.args
        print inst
        x, y = inst
        print 'x =', x
        print 'y =', y
```

Else block

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

Else block…why

- The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement.

The details

```
def this_fails():
    x = 1/0
    ...^D
>>> try:
... this_fails()
... except ZeroDivisionError, detail:
... print 'Handling run-time error:', detail
```

Raising Exceptions

- >>> try:
- ... raise NameError, 'HiThere'
- ... except NameError:
- ... print 'An exception flew by!'
- ... raise
- ...

Raising Exceptions

- The **raise statement does two things: it creates an exception object, and immediately leaves the expected**
- program execution sequence to search the enclosing **try statements for a matching except clause. The effect**
- of a **raise statement is to either divert execution in a matching except suite, or to stop the program because**

Raising Exceptions

- no matching **except suite was found to handle the exception.**
- The Exception object created by **raise can contain a message string that provides a meaningful error**
- message. In addition to the string, it is relatively simple to attach additional attributes to the exception.

Raising Exceptions

- Here are the two forms for the **raise satement.**
- **raise exceptionClass , value**
- **raise exception**
- The first form of the **raise statement uses an exception class name. The optional parameter is the additional**
- value that will be contained in the exception. Generally, this is a string with a message, however any object can be provided

Raising Exceptions

- The second form of the raise statement uses an object constructor to create the Exception object.
- **raise ValueError("oh dear me")**
- Here's a variation on the second form in which additional attributes are provided for the exception.
- ex= MyNewError("oh dear me")
- ex.myCode= 42
- ex.myType= "O+"
- **raise ex**

User-defined Exceptions

- import exceptions
- class Expletive(exceptions.Exception):
  - def __init__(self):
    - return
  - def __str__(self):
    - print "","An Expletive occured!"
- def main():
  - raise Expletive
- if __name__=="__main__": try: main() except ImportError: print "Unable to import something..." except Exception, e: raise e

Finally block

- try:
- ... raise KeyboardInterrupt
- ... finally:
- ... print 'Goodbye, world!'
- ...

Tying them together

- def divide(x, y):

```
    try:
        result = x / y
    except ZeroDivisionError:
        print "division by zero!"
    else:
        print "result is", result
    finally:
        print "executing finally clause”
```

Contact Us on:

G K T C S Innovations Pvt. Ltd.

IT Training, Consultancy, Software Development, Staffing
#11,4th Floor,Sneh Deep, Near Warje Flyover Bridge,
Warje-Malwadi, Pune -411058, Maharashtra, India.
Mobile: +91- 9975072320, 8308761477
Email : surendra@gktcs.com
Gmail: surendra.panpaliya@gmail.com
Web: www.gktcs.com