
baabtra.com
Mentoring Partner

OOPs in

pinjut@gmail.com

sanooja jabbar t.com/baabtra

twitter.com/baabtra

in.linkedin.com/in/baabtra

Date:30/01/2015

Object Oriented Programming

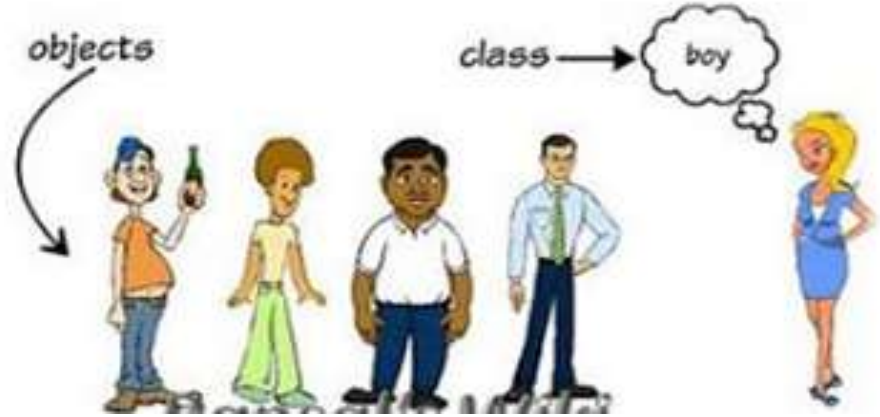
- Python is an **object-oriented programming language**.
- Python doesn't force to use the object-oriented paradigm exclusively.
- Python **also supports procedural programming** with modules and functions, so you can select the most suitable programming paradigm for each part of your program.

"Generally, the object-oriented paradigm is suitable when you want to group state (data) and behavior (code) together in handy packets of functionality."

python

objects

a software item that contains variables and methods



- Objects are the basic run-time entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program must handle.
- When a program is executed the objects interact by sending messages to one another.
- **Objects have two components:**

  Data (i.e., attributes)

  Behaviors (i.e., methods)

classes

binds the member variables and

into a single unit

Syntax:

```
class classname:
        statement(s)
```

Example:

```
class myClass:
        def hello(self):
                print "Hello"
        def sum(self):
                int_res=100+200
                return int_res
```

A method defined inside a class body will always have a mandatory first parameter, conventionally named self, that refers to the instance on which you call the method.

Creating Instance Objects

Syntax

```
anInstance=class_name()
```

Example

```
classObj=myClass()
```

Accessing Attributes

Example:

```
classObj.hello()
int_sum=classObj.sum()
print “The sum is :  ”,int_sum
```

output:

```
Hello
The sum is :  300
```

Object Oriented Design focuses on

➔ Encapsulation:

dividing the code into a public **interface**, and a private **implementation** of that interface

➔ Polymorphism:

the ability to **overload** standard operators so that they have appropriate behavior based on their context

➔ Inheritance:

the ability to create **subclasses** that contain specializations of their parents

inheritance

child class acquires the properties of the parent

ass

➔ Simple Inheritance

Syntax

```
derived_class(base_class)
```

➔ Multiple Inheritance

Syntax

```
derived_class(base_class[,base_class1][,base_class2][....])
```

Example for Simple Inheritance:

```
class myClass:
   def sum(self):
                        int_res=100+200
                        return int_res

class newClass(myClass):

   def hello(self):
            print "Maya"

classObj=newClass()

classObj.hello()

int_sum=classObj.sum()

print “The sum is :  ”,int_sum
```

output:

```
Maya
The sum is :  300
```

Example for Multiple inheritance

```
class A:
    def displayA(self):
        print "Class A"
class B:
    def displayB(self):
        print "Class B"
class c(A,B):
    def displayC(self):
        print "Class C"

objC=c()
objC.displayA()
objC.displayB()
objC.displayC()
```

Output:

```
Class A
Class B
Class C
```

polymorphism

process of using an operator or

- Uses polymorphism extensively in built-in types.
- Here we use the same indexing operator for three different data types.
- Polymorphism is most commonly used when dealing with inheritance.
- Example:

```
a = "alfa"                    # string
b = (1, 2, 3, 4)   # tuple
c = ['o', 'm', 'e', 'g', 'a']  # list
print a[2]
print b[1]
print c[3]
```

Output:

```
f
2
g
```

Two kinds of Polymorphism:

- Overloading

  - Two or more methods with different signatures

  - Python operators works for built-in Classes

- Overriding

  - Replacing an inherited method with another having the same signature

Explanation for Operator Overloading Sample Program

What actually happens is that, when you do p1 - p2, Python will call p1.__sub__(p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below

Operator	Expression	Internally
Addition	p1+p2	p1.__add__(p2)
Subtraction	p1-p2	p1.__sub__(p2)
Multiplication	p1*p2	p1.__mul__(p2)
Power	p1**p2	p1.__pow__(p2)
Division	p1/p2	p1.__truediv__(p2)

Example for Overriding

```
class Animal:
        def __init__(self, name=''):
                self.name = name
        def talk(self):
                pass
class Cat(Animal):
        def talk(self):
                print "Meow!"
class Dog(Animal):
        def talk(self):
                print "Woof!"

a = Animal()
a.talk()
c = Cat("Missy")
c.talk()
d = Dog("Rocky")
d.talk()
```

Output:

```
Meow!
Woof!
```

encapsulation

process of binding dat
member functions into a single unit

- An important concept in OOP
- Data Abstraction is achieved through Encapsulation
- Generally speaking encapsulation is the mechanism for restricting the access to some of an object's components, this means, that the internal representation of an object can't be seen from outside of the objects definition.

The following table shows the different behaviour of Public, Protected and Private Data

Name	Notation	Behaviour
name	Public	can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside
__name	Private	Can't be seen and accessed from outside

Example for Public, Private and Protected Data

```
>>> class Encapsulation():
        def __init__(self,a,b,c):
            self.public=a
            self._protected=b
            self.__private=c
>>> x=Encapsulation(11,13,17)
>>> x.public
11
>>> x._protected
13
>>> x._protected=23
>>> x._protected
23
>>> x.__private
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    x.private
AttributeError: 'Encapsulation' object has no attribute '__private'
>>>
```

special methods

classes in python can implement certain operations with special method names.

- They are also called **“Magic Methods”**
- They are not called directly, but by a specific language syntax
- This is similar to what is known as ***operator overloading*** in C++
- Contains clumsy syntax, ie., double underscore at the beginning and end
- So simplicity __init__() can be read as **“dunder init dunder”**
- So magic methods also called **“Dunder Methods”**

Some special methods are:

1) **__init__()**

default constructor, when an instance of class is created

2) **__str__()**

If we print an object then its __str__() method will get called

3) **__len__()**

Returns the length of the container

4) **__del__()**

It destructs the constructors that have created using __init__()

Example for Special Methods

```
class Book:
    def __init__(self, title, author, pages):
        print "A book is created"
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return "Title:%s , author:%s, pages:%s " % \
            (self.title, self.author, self.pages)
    def __len__(self):
        return self.pages
    def __del__(self):
        print "A book is destroyed"

book = Book("Inside Steve's Brain", "Leander Kahney", 304)
print book
print len(book)
del book
```

Output:

```
A book is created
Title:Inside Steve's Brain , author:Leander Kahney, pages:304
304
A book is destroyed
```

```
class Book:
    def __init__(self, title, author, pages):
        print "A book is created"
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return "Title:%s , author:%s, pages:%s " % \
            (self.title, self.author, self.pages)
    def __len__(self):
        return self.pages
    def __del__(self):
        print "A book is destroyed"
```

Output:
A book is created

Here we call the __init__() method. The method creates a new instance of a Book class.

```
book = Book("Inside Steve's Brain", "Leander Kahney", 304)
print book
print len(book)
del book
```

```
class Book:
    def __init__(self, title, author, pages):
        print "A book is created"
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return "Title:%s , author:%s, pages:%s " % \
            (self.title, self.author, self.pages)
    def __len__(self):
        return self.pages
    def __del__(self):
        print "A book is destroyed"

book = Book("Inside Steve's Brain", "Leander Kahney", 304)
print book
print len(book)
del book
```

Output:

```
A book is created
Title:Inside Steve's Brain , author:Leander Kahney, pages:304
```

The print keyword calls the __str__() method. This method should return an informal string representation of an object.

```
class Book:
    def __init__(self, title, author, pages):
        print "A book is created"
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return "Title:%s , author:%s, pages:%s " % \
            (self.title, self.author, self.pages)
    def __len__(self):
        return self.pages
    def __del__(self):
        print "A book is destroyed"

book = Book("Inside Steve's Brain", "Leander Kahney", 304)
print book
print len(book)
del book
```

Output:

```
A book is created
Title:Inside Steve's Brain , author:Leander Kahney, pages:304
304
```

The len() function invokes the __len__() method. In our case, we print the number of pages of your book.

```
class Book:
    def __init__(self, title, author, pages):
        print "A book is created"
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return "Title:%s , author:%s, pages:%s " % \
            (self.title, self.author, self.pages)
    def __len__(self):
        return self.pages
    def __del__(self):
        print "A book is destroyed"

book = Book("Inside Steve's Brain", "Leander Kahney", 304)
print book
print len(book)
del book
```

Output:

```
A book is created
Title:Inside Steve's Brain , author:Leander Kahney, pages:304
304
A book is destroyed
```

The del keyword deletes an object. It calls the __del__() method.

python

modules

sing import statement

- A typical Python program is made up of several source files. *Each source file corresponds to a module*, which packages program code and data for reuse.
- *Modules* are normally *independent* of each other so that other programs can *reuse the specific modules they need.*
- A module explicitly establishes dependencies upon another module by using **import** statements.

Import Statement

Syntax

```
import modname [as varname][,...]
```

Example

```
import MyModule1
import MyModule as Alias
```

Import Statement

Syntax

```
import modname [as varname][,...]
```

Example

```
import MyModule1
import MyModule as Alias
```

In the simplest and most common case, modname is an identifier, the name of a variable that Python binds to the module object when the import statement finishes

Import Statement

Syntax

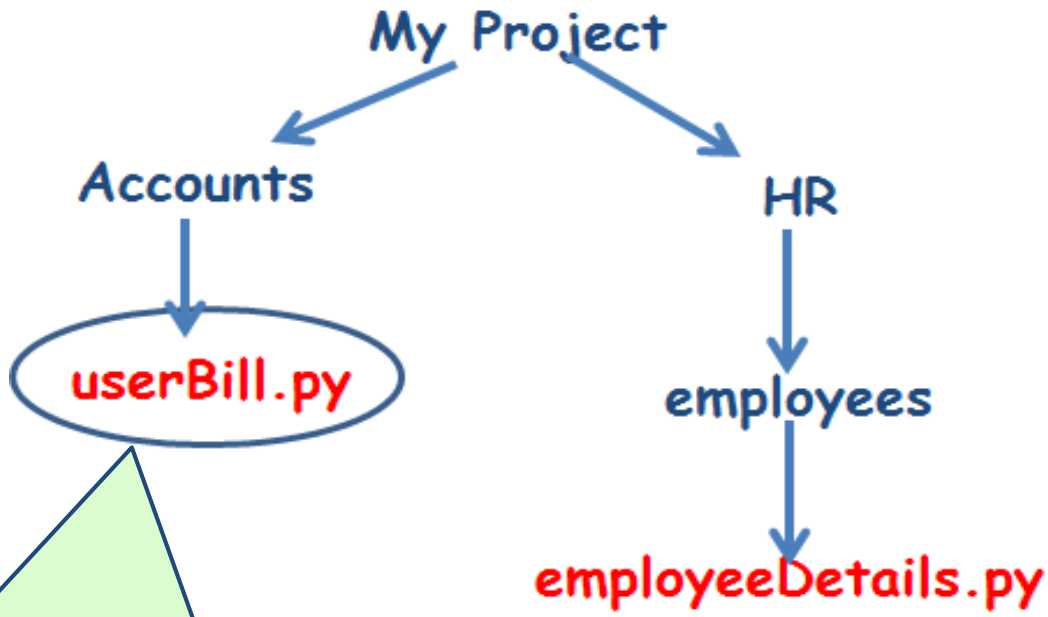
```
import modname [as varname][,...]
```

Example

```
import MyModule1
import MyModule as Alias
```

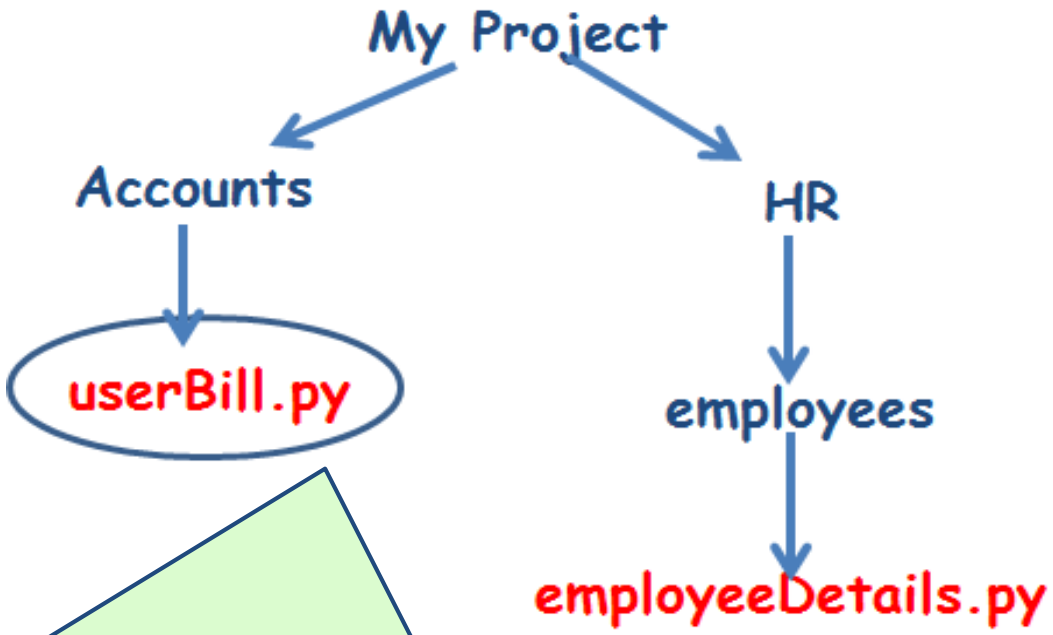
looks for the module named MyModule and binds the variable named Alias in the current scope to the module object. varname is always a simple identifier.

Relative Import Statement



Suppose we are here and we want to import the file employeeDetails.py which resides in a directory as shown

Relative Import Statement




```
import sys
import os
str_current_path = os.getcwd()  ## get current working directory
str_module_path = str_current_path.replace( ' Accounts',‘HR/employees /
')
sys.path.append( str_module_path )
import employeeDetails
```

Relative Import Statement

➔ **sys.path**

A list of strings that specifies the search path for modules

➔ **sys.path.append(module path)**

This statement append our module path with the existing list of sys.path, then the import statement search for the module in the module path as we specified.

__main__()

- When the Python interpreter reads a source file, it executes all of the code found in it. But Before executing the code, it will define a few special variables.
- For example, if the python interpreter is running a module (the source file) as the main program, it sets the special __name__ variable to have a value"__main__".
- If this file is being imported from another module, __name__ will be set to the module's name.

Example

File one.py

```
def func():
        print("func() in one.py")
print("top-level in one.py")
if __name__ == "__main__":
        print("one.py is being run directly")
else:
        print("one.py is being imported into another module")
```

File two.py

```
import one
print("top-level in two.py")
one.func()
if __name__ == "__main__":
    print("two.py is being run directly")
else:
        print("two.py is being imported into another module")
```

Example

File one.py

```
def func():
        print("func() in one.py")
print("top-level in one.py")
if __name__ == "__main__":
        print("one.py is being run directly")
else:
        print("one.py is being imported into another module")
```

When we run one.py

Top-level in one.py

One.py is being run directly

File two.py

```
import one
print("top-level in two.py")
one.func()
if __name__ == "__main__":
    print("two.py is being run directly")
else:
        print("two.py is being imported into another module")
```

Example

File one.py

```
def func():
        print("func() in one.py")
print("top-level in one.py")
if __name__ == "__main__":
        print("one.py is being run directly")
else:
        print("one.py is being imported into another module")
```

File two.py

```
import one
print("top-level in two.py")
one.func()
if __name__ == "__main__":
    print("two.py is being run directly")
else:
        print("two.py is being imported into another module")
```

When we run two.py

----

top-level in one.py
one.py is being imported into another module
top-level in two.py
func() in one.py
two.py is being run directly

Looking for learning more about the above topic?

Email to info@baabtra.com or Visit baabtra.com

US	UK	UAE
7002 Hana Road, Edison NJ 08817, United States of America.	90 High Street, Cherry Hinton, Cambridge, CB1 9HZ, United Kingdom.	Suite No: 51, Oasis Center, Sheikh Zayed Road, Dubai, UAE

India Centres

Emarald Mall (Big Bazar Building)
Mavoor Road, Kozhikode,
Kerala, India.
Ph: + 91 – 495 40 25 550

NC Complex, Near Bus Stand
Mukkam, Kozhikode,
Kerala, India.
Ph: + 91 – 495 40 25 550

Cafit Square IT Park,
Hilite Business Park,
Kozhikode
Kerala, India.

Email: info@baabtra.com

TBI - NITC
NIT Campus, Kozhikode.
Kerala, India.

Start up Village
Eranakulam,
Kerala, India.

Start up Village
UL CC
Kozhikode, Kerala

Follow us @ twitter.com/baabtra

Like us @ facebook.com/baabtra

Subscribe to us @ youtube.com/baabtra

Become a follower @ slideshare.net/BaabtraMentoringPartner

Connect to us @ in.linkedin.com/in/baabtra

Give a feedback @ massbaab.com/baabtra

Thanks in advance

www.baabtra.com | www.massbaab.com | www.baabte.com

Want to learn more about programming or Looking to become a good programmer?

Are you wasting time on searching so many contents online?

Do you want to learn things quickly?

Tired of spending huge amount of money to become a Software professional?

Do an online course
@ baabtra.com

We put industry standards to practice. Our structured, activity based courses are so designed to make a quick, good software professional out of anybody who holds a passion for coding.