1. **Classes/Objects**

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is a small introduction of Object-Oriented Programming (OOP) to help you −

Overview of OOP Terminology

* **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. Class variables are not used as frequently as instance variables are.
* **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 or 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.

**Creating Classes**

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −

class ClassName:

'Optional class documentation string'

class\_suite

* The class has a documentation string, which can be accessed via ***ClassName.\_\_doc\_\_***.
* The ***class\_suite*** consists of all the component statements defining class members, data attributes and functions.

Example

Following is an example of a simple Python class −

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

* The variable *empCount* is a class variable whose value is shared among all the instances of a in this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
* The first method *\_\_init\_\_()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
* You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

**Creating Instance Objects**

To create instances of a class, you call the class using class name and pass in whatever arguments its *\_\_init\_\_* method accepts.

This would create first object of Employee class

emp1 = Employee("Zara", 2000)

This would create second object of Employee class

emp2 = Employee("Manni", 5000)

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)

Now, putting all the concepts together −

#!/usr/bin/python3

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

print ("Name : ", self.name, ", Salary: ", self.salary)

#This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

#This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)

When the above code is executed, it produces the following result −

Name : Zara ,Salary: 2000

Name : Manni ,Salary: 5000

Total Employee 2

You can add, remove, or modify attributes of classes and objects at any time −

emp1.salary = 7000 # Add an 'salary' attribute.

emp1.name = 'xyz' # Modify 'age' attribute.

del emp1.salary # Delete 'age' attribute.

Instead of using the normal statements to access attributes, you can use the following functions −

* The **getattr(obj, name[, default])** − to access the attribute of object.
* The **hasattr(obj,name)** − to check if an attribute exists or not.
* The **setattr(obj,name,value)** − to set an attribute. If attribute does not exist, then it would be created.
* The **delattr(obj, name)** − to delete an attribute.

hasattr(emp1, 'salary') # Returns true if 'salary' attribute exists

getattr(emp1, 'salary') # Returns value of 'salary' attribute

setattr(emp1, 'salary', 7000) # Set attribute 'salary' at 7000

delattr(emp1, 'salary') # Delete attribute 'salary'

**Built-In Class Attributes**

Every Python 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.

For the above class let us try to access all these attributes −

#!/usr/bin/python3

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

print ("Name : ", self.name, ", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)

emp2 = Employee("Manni", 5000)

print ("Employee.\_\_doc\_\_:", Employee.\_\_doc\_\_)

print ("Employee.\_\_name\_\_:", Employee.\_\_name\_\_)

print ("Employee.\_\_module\_\_:", Employee.\_\_module\_\_)

print ("Employee.\_\_bases\_\_:", Employee.\_\_bases\_\_)

print ("Employee.\_\_dict\_\_:", Employee.\_\_dict\_\_ )

When the above code is executed, it produces the following result −

Employee.\_\_doc\_\_: Common base class for all employees

Employee.\_\_name\_\_: Employee

Employee.\_\_module\_\_: \_\_main\_\_

Employee.\_\_bases\_\_: (<class 'object'>,)

Employee.\_\_dict\_\_: {

'displayCount': <function Employee.displayCount at 0x0160D2B8>,

'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': 'Common base class for all employees',

'empCount': 2, '\_\_init\_\_':

<function Employee.\_\_init\_\_ at 0x0124F810>, 'displayEmployee':

<function Employee.displayEmployee at 0x0160D300>,

'\_\_weakref\_\_':

<attribute '\_\_weakref\_\_' of 'Employee' objects>, '\_\_dict\_\_':

<attribute '\_\_dict\_\_' of 'Employee' objects>

}

**Destroying Objects (Garbage Collection)**

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

a = 40 # Create object <40>

b = a # Increase ref. count of <40>

c = [b] # Increase ref. count of <40>

del a # Decrease ref. count of <40>

b = 100 # Decrease ref. count of <40>

c[0] = -1 # Decrease ref. count of <40>

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. However, a class can implement the special method *\_\_del\_\_()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

Example

This \_\_del\_\_() destructor prints the class name of an instance that is about to be destroyed −

#!/usr/bin/python3

class Point:

def \_\_init\_\_( self, x=0, y=0):

self.x = x

self.y = y

def \_\_del\_\_(self):

class\_name = self.\_\_class\_\_.\_\_name\_\_

print (class\_name, "destroyed")

pt1 = Point()

pt2 = pt1

pt3 = pt1

print (id(pt1), id(pt2), id(pt3)); # prints the ids of the obejcts

del pt1

del pt2

del pt3

When the above code is executed, it produces the following result −

3083401324 3083401324 3083401324

Point destroyed

**Note** − Ideally, you should define your classes in a separate file, then you should import them in your main program file using *import* statement.

In the above example, assuming definition of a Point class is contained in *point.py* and there is no other executable code in it.

#!/usr/bin/python3

import point

p1 = point.Point()

**Class Inheritance**

Instead of starting from a scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

Example

#!/usr/bin/python3

class Parent: # define parent class

parentAttr = 100

def \_\_init\_\_(self):

print ("Calling parent constructor")

def parentMethod(self):

print ('Calling parent method')

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class

def \_\_init\_\_(self):

print ("Calling child constructor")

def childMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.childMethod() # child calls its method

c.parentMethod() # calls parent's method

c.setAttr(200) # again call parent's method

c.getAttr() # again call parent's method

When the above code is executed, it produces the following result −

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

In a similar way, you can drive a class from multiple parent classes as follows −

class A: # define your class A

.....

class B: # define your calss B

.....

class C(A, B): # subclass of A and B

.....

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

* The **issubclass(sub, sup)** boolean function returns True, if the given subclass **sub** is indeed a subclass of the superclass **sup**.
* The **isinstance(obj, Class)** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of Class

**Overriding Methods**

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

Example

#!/usr/bin/python3

class Parent: # define parent class

def myMethod(self):

print ('Calling parent method')

class Child(Parent): # define child class

def myMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.myMethod() # child calls overridden method

When the above code is executed, it produces the following result −

Calling child method

**Base Overloading Methods**

The following table lists some generic functionality that you can override in your own classes:

|  |  |
| --- | --- |
| **S.No.** | **Method, Description & Sample Call** |
| 1 | **\_\_init\_\_ ( self [,args...] )**  Constructor (with any optional arguments)  Sample Call : *obj = className(args)* |
| 2 | **\_\_del\_\_( self )**  Destructor, deletes an object  Sample Call : *del obj* |
| 3 | **\_\_repr\_\_( self )**  Evaluatable string representation  Sample Call : *repr(obj)* |
| 4 | **\_\_str\_\_( self )**  Printable string representation  Sample Call : *str(obj)* |
| 5 | **\_\_cmp\_\_ ( self, x )**  Object comparison  Sample Call : *cmp(obj, x)* |

**Overloading Operators**

Suppose you have created a Vector class to represent two-dimensional vectors. What happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *\_\_add\_\_* method in your class to perform vector addition and then the plus operator would behave as per expectation −

Example

#!/usr/bin/python3

class Vector:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_str\_\_(self):

return 'Vector (%d, %d)' % (self.a, self.b)

def \_\_add\_\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)

print (v1 + v2)

When the above code is executed, it produces the following result −

Vector(7,8)

**Data Hiding**

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

Example

#!/usr/bin/python3

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print (self.\_\_secretCount)

counter = JustCounter()

counter.count()

counter.count()

print (counter.\_\_secretCount)

When the above code is executed, it produces the following result −

1

2

Traceback (most recent call last):

File "test.py", line 12, in <module>

print counter.\_\_secretCount

AttributeError: JustCounter instance has no attribute '\_\_secretCount'

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object.\_className\_\_attrName*. If you would replace your last line as following, then it works for you −

.........................

print (counter.\_JustCounter\_\_secretCount)

When the above code is executed, it produces the following result −

1

2

2

1. **Regular Expression**

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

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.

We would cover two important functions, which would be used to handle regular expressions. Nevertheless, a small thing first: 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'**.

Basic patterns that match single chars

|  |  |
| --- | --- |
| **S.No.** | **Expression & Matches** |
| 1 | **a, X, 9, <**  ordinary characters just match themselves exactly. |
| 2 | **. (a period)**  matches any single character except newline '\n' |
| 3 | **\w**  matches a "word" character: a letter or digit or underbar [a-zA-Z0-9\_]. |
| 4 | **\W**  matches any non-word character. |
| 5 | **\b**  boundary between word and non-word |
| 6 | **\s**  matches a single whitespace character -- space, newline, return, tab |
| 7 | **\S**  matches any non-whitespace character. |
| 8 | **\t, \n, \r**  tab, newline, return |
| 9 | **\d**  decimal digit [0-9] |
| 10 | **^**  matches start of the string |
| 11 | **$**  match the end of the string |
| 12 | **\**  inhibit the "specialness" of a character. |

**Compilation flags**

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the re module under two names, a long name such as **IGNORECASE** and a short, one-letter form such as I.

|  |  |
| --- | --- |
| **S.No.** | **Flag & Meaning** |
| 1 | **ASCII, A**  Makes several escapes like \w, \b, \s and \d match only on ASCII characters with the respective property. |
| 2 | **DOTALL, S**  Make, match any character, including newlines |
| 3 | **IGNORECASE, I**  Do case-insensitive matches |
| 4 | **LOCALE, L**  Do a locale-aware match |
| 5 | **MULTILINE, M**  Multi-line matching, affecting ^ and $ |
| 6 | **VERBOSE, X (for ‘extended’)**  Enable verbose REs, which can be organized more cleanly and understandably |

**The match Function**

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

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

Here is the description of the parameters −

|  |  |
| --- | --- |
| **S.No.** | **Parameter & Description** |
| 1 | **pattern**  This is the regular expression to be matched. |
| 2 | **string**  This is the string, which would be searched to match the pattern at the beginning of string. |
| 3 | **flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

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

|  |  |
| --- | --- |
| **S.No.** | **Match Object Method & Description** |
| 1 | **group(num = 0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

Example

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if matchObj:

print ("matchObj.group() : ", matchObj.group())

print ("matchObj.group(1) : ", matchObj.group(1))

print ("matchObj.group(2) : ", matchObj.group(2))

else:

print ("No match!!")

When the above code is executed, it produces the following result −

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

**The search Function**

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function −

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

Here is the description of the parameters −

|  |  |
| --- | --- |
| **S.No.** | **Parameter & Description** |
| 1 | **pattern**  This is the regular expression to be matched. |
| 2 | **string**  This is the string, which would be searched to match the pattern anywhere in the string. |
| 3 | **flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

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

|  |  |
| --- | --- |
| **S.No.** | **Match Object Method & Description** |
| 1 | **group(num = 0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

Example

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if searchObj:

print ("searchObj.group() : ", searchObj.group())

print ("searchObj.group(1) : ", searchObj.group(1))

print ("searchObj.group(2) : ", searchObj.group(2))

else:

print ("Nothing found!!")

When the above code is executed, it produces the following result −

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

**Matching Versus Searching**

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search**checks for a match anywhere in the string (this is what Perl does by default).

Example

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)

if matchObj:

print ("match --> matchObj.group() : ", matchObj.group())

else:

print ("No match!!")

searchObj = re.search( r'dogs', line, re.M|re.I)

if searchObj:

print ("search --> searchObj.group() : ", searchObj.group())

else:

print ("Nothing found!!")

When the above code is executed, it produces the following result −

No match!!

search --> matchObj.group() : dogs

**Search and Replace**

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

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

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* is provided. This method returns modified string.

Example

#!/usr/bin/python3

import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments

num = re.sub(r'#.\*$', "", phone)

print ("Phone Num : ", num)

# Remove anything other than digits

num = re.sub(r'\D', "", phone)

print ("Phone Num : ", num)

When the above code is executed, it produces the following result −

Phone Num : 2004-959-559

Phone Num : 2004959559

**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 (|), as shown previously and may be represented by one of these −

|  |  |
| --- | --- |
| **S.No.** | **Modifier & Description** |
| 1 | **re.I**  Performs case-insensitive matching. |
| 2 | **re.L**  Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B). |
| 3 | **re.M**  Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string). |
| 4 | **re.S**  Makes a period (dot) match any character, including a newline. |
| 5 | **re.U**  Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B. |
| 6 | **re.X**  Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker. |

**Regular Expression Patterns**

Except for the control characters, **(+ ? . \* ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

The following table lists the regular expression syntax that is available in Python −

Here is the list of regular expression syntax in Python.

Regular Expression Examples

Literal characters

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **python**  Match "python". |

**Character classes**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **[Pp]ython**  Match "Python" or "python" |
| 2 | **rub[ye]**  Match "ruby" or "rube" |
| 3 | **[aeiou]**  Match any one lowercase vowel |
| 4 | **[0-9]**  Match any digit; same as [0123456789] |
| 5 | **[a-z]**  Match any lowercase ASCII letter |
| 6 | **[A-Z]**  Match any uppercase ASCII letter |
| 7 | **[a-zA-Z0-9]**  Match any of the above |
| 8 | **[^aeiou]**  Match anything other than a lowercase vowel |
| 9 | **[^0-9]**  Match anything other than a digit |

**Special Character Classes**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **.**  Match any character except newline |
| 2 | **\d**  Match a digit: [0-9] |
| 3 | **\D**  Match a nondigit: [^0-9] |
| 4 | **\s**  Match a whitespace character: [ \t\r\n\f] |
| 5 | **\S**  Match nonwhitespace: [^ \t\r\n\f] |
| 6 | **\w**  Match a single word character: [A-Za-z0-9\_] |
| 7 | **\W**  Match a nonword character: [^A-Za-z0-9\_] |

**Repetition Cases**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **ruby?**  Match "rub" or "ruby": the y is optional |
| 2 | **ruby\***  Match "rub" plus 0 or more ys |
| 3 | **ruby+**  Match "rub" plus 1 or more ys |
| 4 | **\d{3}**  Match exactly 3 digits |
| 5 | **\d{3,}**  Match 3 or more digits |
| 6 | **\d{3,5}**  Match 3, 4, or 5 digits |

**Nongreedy repetition**

This matches the smallest number of repetitions −

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **<.\*>**  Greedy repetition: matches "<python>perl>" |
| 2 | **<.\*?>**  Nongreedy: matches "<python>" in "<python>perl>" |

**Grouping with Parentheses**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **\D\d+**  No group: + repeats \d |
| 2 | **(\D\d)+**  Grouped: + repeats \D\d pair |
| 3 | **([Pp]ython(,)?)+**  Match "Python", "Python, python, python", etc. |

**Backreferences**

This matches a previously matched group again −

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **([Pp])ython&\1ails**  Match python&pails or Python&Pails |
| 2 | **(['"])[^\1]\*\1**  Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc. |

**Alternatives**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **python|perl**  Match "python" or "perl" |
| 2 | **rub(y|le)**  Match "ruby" or "ruble" |
| 3 | **Python(!+|\?)**  "Python" followed by one or more ! or one ? |

**Anchors**

This needs to specify match position.

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **^Python**  Match "Python" at the start of a string or internal line |
| 2 | **Python$**  Match "Python" at the end of a string or line |
| 3 | **\APython**  Match "Python" at the start of a string |
| 4 | **Python\Z**  Match "Python" at the end of a string |
| 5 | **\bPython\b**  Match "Python" at a word boundary |
| 6 | **\brub\B**  \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone |
| 7 | **Python(?=!)**  Match "Python", if followed by an exclamation point. |
| 8 | **Python(?!!)**  Match "Python", if not followed by an exclamation point. |

**Special Syntax with Parentheses**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **R(?#comment)**  Matches "R". All the rest is a comment |
| 2 | **R(?i)uby**  Case-insensitive while matching "uby" |
| 3 | **R(?i:uby)**  Same as above |
| 4 | **rub(?:y|le))**  Group only without creating \1 backreference |

1. **Common Gateway Interface (CGI)**

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA.

**What is CGI?**

* The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
* The current version is CGI/1.1 and CGI/1.2 is under progress.

**Web Browsing**

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

* Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
* Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
* Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

**CGI Architecture Diagram**



**Web Server Support and Configuration**

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /var/www/cgi-bin. By convention, CGI files have extension as. **cgi,** but you can keep your files with python extension **.py** as well.

By default, the Linux server is configured to run only the scripts in the cgi-bin directory in /var/www. If you want to specify any other directory to run your CGI scripts, comment the following lines in the httpd.conf file −

<Directory "/var/www/cgi-bin">

AllowOverride None

Options ExecCGI

Order allow,deny

Allow from all

</Directory>

<Directory "/var/www/cgi-bin">

Options All

</Directory>

Here, we assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell, etc.

**First CGI Program**

Here is a simple link, which is linked to a CGI script called [hello.py](https://www.tutorialspoint.com/cgi-bin/hello.py). This file is kept in /var/www/cgi-bin directory and it has following content. Before running your CGI program, make sure you have change mode of file using **chmod 755 hello.py** UNIX command to make file executable.

#!/usr/bin/python

print "Content-type:text/html\r\n\r\n"

print '<html>'

print '<head>'

print '<title>Hello Word - First CGI Program</title>'

print '</head>'

print '<body>'

print '<h2>Hello Word! This is my first CGI program</h2>'

print '</body>'

print '</html>'

Note − First line in the script must be path to Python executable. In Linux it should be #!/usr/bin/python3

Enter following URL in yor browser

http://localhost:8080/cgi-bin/hello.py

|  |
| --- |
| Hello Word! This is my first CGI program |

This hello.py script is a simple Python script, which writes its output on STDOUT file, i.e., screen. There is one important and extra feature available which is first line to be printed **Content-type:text/html\r\n\r\n**. This line is sent back to the browser and it specifies the content type to be displayed on the browser screen.

By now you must have understood basic concept of CGI and you can write many complicated CGI programs using Python. This script can interact with any other external system also to exchange information such as RDBMS.

**HTTP Header**

The line **Content-type:text/html\r\n\r\n** is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form −

HTTP Field Name: Field Content

For Example

Content-type: text/html\r\n\r\n

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

|  |  |
| --- | --- |
| **Sr.No.** | **Header & Description** |
| 1 | **Content-type:**  A MIME string defining the format of the file being returned. Example is Content-type:text/html |
| 2 | **Expires: Date**  The date the information becomes invalid. It is used by the browser to decide when a page needs to be refreshed. A valid date string is in the format 01 Jan 1998 12:00:00 GMT. |
| 3 | **Location: URL**  The URL that is returned instead of the URL requested. You can use this field to redirect a request to any file. |
| 4 | **Last-modified: Date**  The date of last modification of the resource. |
| 5 | **Content-length: N**  The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file. |
| 6 | **Set-Cookie: String**  Set the cookie passed through the *string* |

**CGI Environment Variables**

All the CGI programs have access to the following environment variables. These variables play an important role while writing any CGI program.

|  |  |
| --- | --- |
| **Sr.No.** | **Variable Name & Description** |
| 1 | **CONTENT\_TYPE**  The data type of the content. Used when the client is sending attached content to the server. For example, file upload. |
| 2 | **CONTENT\_LENGTH**  The length of the query information. It is available only for POST requests. |
| 3 | **HTTP\_COOKIE**  Returns the set cookies in the form of key & value pair. |
| 4 | **HTTP\_USER\_AGENT**  The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser. |
| 5 | **PATH\_INFO**  The path for the CGI script. |
| 6 | **QUERY\_STRING**  The URL-encoded information that is sent with GET method request. |
| 7 | **REMOTE\_ADDR**  The IP address of the remote host making the request. This is useful logging or for authentication. |
| 8 | **REMOTE\_HOST**  The fully qualified name of the host making the request. If this information is not available, then REMOTE\_ADDR can be used to get IR address. |
| 9 | **REQUEST\_METHOD**  The method used to make the request. The most common methods are GET and POST. |
| 10 | **SCRIPT\_FILENAME**  The full path to the CGI script. |
| 11 | **SCRIPT\_NAME**  The name of the CGI script. |
| 12 | **SERVER\_NAME**  The server's hostname or IP Address |
| 13 | **SERVER\_SOFTWARE**  The name and version of the software the server is running. |

Here is small CGI program to list out all the CGI variables. Click this link to see the result [Get Environment](http://www.tutorialspoint.com/cgi-bin/get_env.py)

#!/usr/bin/python

import os

print "Content-type: text/html\r\n\r\n";

print "<font size=+1>Environment</font><\br>";

for param in os.environ.keys():

print "<b>%20s</b>: %s<\br>" % (param, os.environ[param])

**GET and POST Methods**

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.

**Passing Information using GET method**

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows −

http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation: only 1024 characters can be sent in a request string. The GET method sends information using QUERY\_STRING header and will be accessible in your CGI Program through QUERY\_STRING environment variable.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

Simple URL Example:Get Method

Here is a simple URL, which passes two values to hello\_get.py program using GET method.

[/cgi-bin/hello\_get.py?first\_name=ZARA&last\_name=ALI](https://www.tutorialspoint.com/cgi-bin/hello_get.py?first_name=ZARA&last_name=ALI)

Below is **hello\_get.py** script to handle input given by web browser. We are going to use **cgi** module, which makes it very easy to access passed information −

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

first\_name = form.getvalue('first\_name')

last\_name = form.getvalue('last\_name')

print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Hello - Second CGI Program</title>"

print "</head>"

print "<body>"

print "<h2>Hello %s %s</h2>" % (first\_name, last\_name)

print "</body>"

print "</html>"

This would generate the following result −

|  |
| --- |
| Hello ZARA ALI |

Simple FORM Example:GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script hello\_get.py to handle this input.

<form action = "/cgi-bin/hello\_get.py" method = "get">

First Name: <input type = "text" name = "first\_name"> <br />

Last Name: <input type = "text" name = "last\_name" />

<input type = "submit" value = "Submit" />

</form>

Here is the actual output of the above form, you enter First and Last Name and then click submit button to see the result.

First Name:

Last Name:

**Passing Information Using POST Method**

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

Below is same hello\_get.py script which handles GET as well as POST method.

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

first\_name = form.getvalue('first\_name')

last\_name = form.getvalue('last\_name')

print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Hello - Second CGI Program</title>"

print "</head>"

print "<body>"

print "<h2>Hello %s %s</h2>" % (first\_name, last\_name)

print "</body>"

print "</html>"

Let us take again same example as above which passes two values using HTML FORM and submit button. We use same CGI script hello\_get.py to handle this input.

<form action = "/cgi-bin/hello\_get.py" method = "post">

First Name: <input type = "text" name = "first\_name"><br />

Last Name: <input type = "text" name = "last\_name" />

<input type = "submit" value = "Submit" />

</form>

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

First Name:

Last Name:

**Passing Checkbox Data to CGI Program**

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes −

<form action = "/cgi-bin/checkbox.cgi" method = "POST" target = "\_blank">

<input type = "checkbox" name = "maths" value = "on" /> Maths

<input type = "checkbox" name = "physics" value = "on" /> Physics

<input type = "submit" value = "Select Subject" />

</form>

The result of this code is the following form −

Maths Physics

Below is checkbox.cgi script to handle input given by web browser for checkbox button.

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('maths'):

math\_flag = "ON"

else:

math\_flag = "OFF"

if form.getvalue('physics'):

physics\_flag = "ON"

else:

physics\_flag = "OFF"

print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Checkbox - Third CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> CheckBox Maths is : %s</h2>" % math\_flag

print "<h2> CheckBox Physics is : %s</h2>" % physics\_flag

print "</body>"

print "</html>"

**Passing Radio Button Data to CGI Program**

Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio buttons −

<form action = "/cgi-bin/radiobutton.py" method = "post" target = "\_blank">

<input type = "radio" name = "subject" value = "maths" /> Maths

<input type = "radio" name = "subject" value = "physics" /> Physics

<input type = "submit" value = "Select Subject" />

</form>

The result of this code is the following form −

Maths Physics

Below is radiobutton.py script to handle input given by web browser for radio button −

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('subject'):

subject = form.getvalue('subject')

else:

subject = "Not set"

print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Radio - Fourth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Selected Subject is %s</h2>" % subject

print "</body>"

print "</html>"

**Passing Text Area Data to CGI Program**

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box −

<form action = "/cgi-bin/textarea.py" method = "post" target = "\_blank">

<textarea name = "textcontent" cols = "40" rows = "4">

Type your text here...

</textarea>

<input type = "submit" value = "Submit" />

</form>

The result of this code is the following form −

Type your text here...

Below is textarea.cgi script to handle input given by web browser −

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('textcontent'):

text\_content = form.getvalue('textcontent')

else:

text\_content = "Not entered"

print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>";

print "<title>Text Area - Fifth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Entered Text Content is %s</h2>" % text\_content

print "</body>"

**Passing Drop Down Box Data to CGI Program**

Drop Down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box −

<form action = "/cgi-bin/dropdown.py" method = "post" target = "\_blank">

<select name = "dropdown">

<option value = "Maths" selected>Maths</option>

<option value = "Physics">Physics</option>

</select>

<input type = "submit" value = "Submit"/>

</form>

The result of this code is the following form −

Maths

Physics

Below is dropdown.py script to handle input given by web browser.

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('dropdown'):

subject = form.getvalue('dropdown')

else:

subject = "Not entered"

print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Dropdown Box - Sixth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Selected Subject is %s</h2>" % subject

print "</body>"

print "</html>"

**Using Cookies in CGI**

HTTP protocol is a stateless protocol. For a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. How to maintain user's session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

**How It Works?**

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields −

* **Expires** − The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
* **Domain** − The domain name of your site.
* **Path** − The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
* **Secure** − If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
* **Name=Value** − Cookies are set and retrieved in the form of key and value pairs.

**Setting up Cookies**

It is very easy to send cookies to browser. These cookies are sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. Setting the cookies is done as follows −

#!/usr/bin/python

print "Set-Cookie:UserID = XYZ;\r\n"

print "Set-Cookie:Password = XYZ123;\r\n"

print "Set-Cookie:Expires = Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n"

print "Set-Cookie:Domain = www.tutorialspoint.com;\r\n"

print "Set-Cookie:Path = /perl;\n"

print "Content-type:text/html\r\n\r\n"

...........Rest of the HTML Content....

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

It is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line **"Content-type:text/html\r\n\r\n**.

**Retrieving Cookies**

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP\_COOKIE and they will have following form −

key1 = value1;key2 = value2;key3 = value3....

Here is an example of how to retrieve cookies.

#!/usr/bin/python

# Import modules for CGI handling

from os import environ

import cgi, cgitb

if environ.has\_key('HTTP\_COOKIE'):

for cookie in map(strip, split(environ['HTTP\_COOKIE'], ';')):

(key, value ) = split(cookie, '=');

if key == "UserID":

user\_id = value

if key == "Password":

password = value

print "User ID = %s" % user\_id

print "Password = %s" % password

This produces the following result for the cookies set by above script −

User ID = XYZ

Password = XYZ123

**File Upload Example**

To upload a file, the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type creates a "Browse" button.

<html>

<body>

<form enctype = "multipart/form-data"

action = "save\_file.py" method = "post">

<p>File: <input type = "file" name = "filename" /></p>

<p><input type = "submit" value = "Upload" /></p>

</form>

</body>

</html>

The result of this code is the following form −

File:

Above example has been disabled intentionally to save people uploading file on our server, but you can try above code with your server.

Here is the script **save\_file.py** to handle file upload −

#!/usr/bin/python

import cgi, os

import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# Get filename here.

fileitem = form['filename']

# Test if the file was uploaded

if fileitem.filename:

# strip leading path from file name to avoid

# directory traversal attacks

fn = os.path.basename(fileitem.filename)

open('/tmp/' + fn, 'wb').write(fileitem.file.read())

message = 'The file "' + fn + '" was uploaded successfully'

else:

message = 'No file was uploaded'

print """\

Content-Type: text/html\n

<html>

<body>

<p>%s</p>

</body>

</html>

""" % (message,)

If you run the above script on Unix/Linux, then you need to take care of replacing file separator as follows, otherwise on your windows machine above open() statement should work fine.

fn = os.path.basename(fileitem.filename.replace("\\", "/" ))

**How To Raise a "File Download" Dialog Box?**

Sometimes, it is desired that you want to give option where a user can click a link and it will pop up a "File Download" dialogue box to the user instead of displaying actual content. This is very easy and can be achieved through HTTP header. This HTTP header is be different from the header mentioned in previous section.

For example, if you want make a **FileName** file downloadable from a given link, then its syntax is as follows −

#!/usr/bin/python

# HTTP Header

print "**Content-Type:**application/octet-stream; name = \"FileName\"\r\n";

print "**Content-Disposition:** attachment; filename = \"FileName\"\r\n\n";

# Actual File Content will go here.

fo = open("foo.txt", "rb")

str = fo.read();

print str

# Close opend file

fo.close()

1. **Database Access**

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

You can choose the right database for your application. Python Database API supports a wide range of database servers such as −

* GadFly
* mSQL
* MySQL
* PostgreSQL
* Microsoft SQL Server 2000
* Informix
* Interbase
* Oracle
* Sybase
* SQLite

Here is the list of available Python database interfaces − [Python Database Interfaces and APIs](https://wiki.python.org/moin/DatabaseInterfaces). You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following −

* Importing the API module.
* Acquiring a connection with the database.
* Issuing SQL statements and stored procedures.
* Closing the connection

Python has an in-built support for SQLite. In this section, we would learn all the concepts using MySQL. MySQLdb module, a popular interface with MySQL is not compatible with Python 3. Instead, we shall use [PyMySQL](http://www.pymysql.org/) module.

**What is PyMySQL ?**

PyMySQL is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb.

**How do I Install PyMySQL?**

Before proceeding furthur, you make sure you have PyMySQL installed on your machine. Just type the following in your Python script and execute it −

#!/usr/bin/python3

import PyMySQL

If it produces the following result, then it means MySQLdb module is not installed −

Traceback (most recent call last):

File "test.py", line 3, in <module>

Import PyMySQL

ImportError: No module named PyMySQL

The last stable release is available on PyPI and can be installed with pip −

pip install PyMySQL

Alternatively (e.g. if pip is not available), a tarball can be downloaded from [GitHub](https://github.com/PyMySQL/PyMySQL) and installed with Setuptools as follows −

$ # X.X is the desired PyMySQL version (e.g. 0.5 or 0.6).

$ curl -L https://github.com/PyMySQL/PyMySQL/tarball/pymysql-X.X | tar xz

$ cd PyMySQL\*

$ python setup.py install

$ # The folder PyMySQL\* can be safely removed now.

**Note** − Make sure you have root privilege to install the above module.

**Database Connection**

Before connecting to a MySQL database, make sure of the following points −

* You have created a database TESTDB.
* You have created a table EMPLOYEE in TESTDB.
* This table has fields FIRST\_NAME, LAST\_NAME, AGE, SEX and INCOME.
* User ID "testuser" and password "test123" are set to access TESTDB.
* Python module PyMySQL is installed properly on your machine.
* You have gone through MySQL tutorial to understand [MySQL Basics.](https://www.tutorialspoint.com/mysql/index.htm)

Example

Following is an example of connecting with MySQL database "TESTDB" −

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using *cursor()* method

cursor = db.cursor()

# execute SQL query using *execute()* method.

cursor.execute("SELECT VERSION()")

# Fetch a single row using *fetchone()* method.

data = cursor.fetchone()

print ("Database version : %s " % data)

# disconnect from server

db.close()

While running this script, it produces the following result.

Database version : 5.5.20-log

If a connection is established with the datasource, then a Connection Object is returned and saved into **db** for further use, otherwise **db** is set to None. Next, **db** object is used to create a **cursor** object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that the database connection is closed and resources are released.

**Creating Database Table**

Once a database connection is established, we are ready to create tables or records into the database tables using **execute** method of the created cursor.

Example

Let us create a Database table EMPLOYEE −

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using *cursor()* method

cursor = db.cursor()

# Drop table if it already exist using *execute()* method.

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement

sql = """CREATE TABLE EMPLOYEE (

FIRST\_NAME CHAR(20) NOT NULL,

LAST\_NAME CHAR(20),

AGE INT,

SEX CHAR(1),

INCOME FLOAT )"""

cursor.execute(sql)

# disconnect from server

db.close()

INSERT Operation

The INSERT Operation is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record in the EMPLOYEE table −

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.

sql = """INSERT INTO EMPLOYEE(FIRST\_NAME,

LAST\_NAME, AGE, SEX, INCOME)

VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""

try:

# Execute the SQL command

cursor.execute(sql)

# Commit your changes in the database

db.commit()

except:

# Rollback in case there is any error

db.rollback()

# disconnect from server

db.close()

The above example can be written as follows to create SQL queries dynamically −

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.

sql = "INSERT INTO EMPLOYEE(FIRST\_NAME, \

LAST\_NAME, AGE, SEX, INCOME) \

VALUES ('%s', '%s', '%d', '%c', '%d' )" % \

('Mac', 'Mohan', 20, 'M', 2000)

try:

# Execute the SQL command

cursor.execute(sql)

# Commit your changes in the database

db.commit()

except:

# Rollback in case there is any error

db.rollback()

# disconnect from server

db.close()

Example

The following code segment is another form of execution where you can pass parameters directly −

..................................

user\_id = "test123"

password = "password"

con.execute('insert into Login values("%s", "%s")' % \

(user\_id, password))

..................................

**READ Operation**

READ Operation on any database means to fetch some useful information from the database.

Once the database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch a single record or **fetchall()** method to fetch multiple values from a database table.

* **fetchone()** − It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
* **fetchall()** − It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
* **rowcount** − This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 −

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.

sql = "SELECT \* FROM EMPLOYEE \

WHERE INCOME > '%d'" % (1000)

try:

# Execute the SQL command

cursor.execute(sql)

# Fetch all the rows in a list of lists.

results = cursor.fetchall()

for row in results:

fname = row[0]

lname = row[1]

age = row[2]

sex = row[3]

income = row[4]

# Now print fetched result

print ("fname = %s,lname = %s,age = %d,sex = %s,income = %d" % \

(fname, lname, age, sex, income ))

except:

print ("Error: unable to fetch data")

# disconnect from server

db.close()

Output

This will produce the following result −

fname = Mac, lname = Mohan, age = 20, sex = M, income = 2000

**Update Operation**

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as **'M'**. Here, we increase the AGE of all the males by one year.

Example

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using *cursor()* method

cursor = db.cursor()

# Prepare SQL query to UPDATE required records

sql = "UPDATE EMPLOYEE SET AGE = AGE + 1

WHERE SEX = '%c'" % ('M')

try:

# Execute the SQL command

cursor.execute(sql)

# Commit your changes in the database

db.commit()

except:

# Rollback in case there is any error

db.rollback()

# disconnect from server

db.close()

**DELETE Operation**

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 −

Example

#!/usr/bin/python3

import PyMySQL

# Open database connection

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using *cursor()* method

cursor = db.cursor()

# Prepare SQL query to DELETE required records

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

# Execute the SQL command

cursor.execute(sql)

# Commit your changes in the database

db.commit()

except:

# Rollback in case there is any error

db.rollback()

# disconnect from server

db.close()

**Performing Transactions**

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties −

* **Atomicity** − Either a transaction completes or nothing happens at all.
* **Consistency** − A transaction must start in a consistent state and leave the system in a consistent state.
* **Isolation** − Intermediate results of a transaction are not visible outside the current transaction.
* **Durability** − Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is a similar example −

# Prepare SQL query to DELETE required records

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

# Execute the SQL command

cursor.execute(sql)

# Commit your changes in the database

db.commit()

except:

# Rollback in case there is any error

db.rollback()

**COMMIT Operation**

Commit is an operation, which gives a green signal to the database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call the **commit** method.

db.commit()

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use the **rollback()** method.

Here is a simple example to call the **rollback()** method.

db.rollback()

Disconnecting Database

To disconnect the Database connection, use the close() method.

db.close()

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of the DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

**Handling Errors**

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

|  |  |
| --- | --- |
| **S.No.** | **Exception & Description** |
| 1 | **Warning**  Used for non-fatal issues. Must subclass StandardError. |
| 2 | **Error**  Base class for errors. Must subclass StandardError. |
| 3 | **InterfaceError**  Used for errors in the database module, not the database itself. Must subclass Error. |
| 4 | **DatabaseError**  Used for errors in the database. Must subclass Error. |
| 5 | **DataError**  Subclass of DatabaseError that refers to errors in the data. |
| 6 | **OperationalError**  Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter. |
| 7 | **IntegrityError**  Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys. |
| 8 | **InternalError**  Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active. |
| 9 | **ProgrammingError**  Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you. |
| 10 | **NotSupportedError**  Subclass of DatabaseError that refers to trying to call unsupported functionality. |

Your Python scripts should handle these errors, but before using any of the above exceptions, make sure your MySQLdb has support for that exception. You can get more information about them by reading the DB API 2.0 specification.

1. **Networking**

Python provides two levels of access to the network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This chapter gives you an understanding on the most famous concept in Networking - Socket Programming.

**What is Sockets?**

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary −

|  |  |
| --- | --- |
| **S.No.** | **Term & Description** |
| 1 | **domain**  The family of protocols that is used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on. |
| 2 | **type**  The type of communications between the two endpoints, typically SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols. |
| 3 | **protocol**  Typically zero, this may be used to identify a variant of a protocol within a domain and type. |
| 4 | **hostname**  The identifier of a network interface −   * A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation * A string "<broadcast>", which specifies an INADDR\_BROADCAST address. * A zero-length string, which specifies INADDR\_ANY, or * An Integer, interpreted as a binary address in host byte order. |
| 5 | **port**  Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service. |

**The socket Module**

To create a socket, you must use the *socket.socket()* function available in the socket module, which has the general syntax −

s = socket.socket (socket\_family, socket\_type, protocol = 0)

Here is the description of the parameters −

* **socket\_family** − This is either AF\_UNIX or AF\_INET, as explained earlier.
* **socket\_type** − This is either SOCK\_STREAM or SOCK\_DGRAM.
* **protocol** − This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use the required functions to create your client or server program. Following is the list of functions required –

**Server Socket Methods**

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **s.bind()**  This method binds address (hostname, port number pair) to socket. |
| 2 | **s.listen()**  This method sets up and start TCP listener. |
| 3 | **s.accept()**  This passively accept TCP client connection, waiting until connection arrives (blocking). |

**Client Socket Methods**

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **s.connect()**  This method actively initiates TCP server connection. |

**General Socket Methods**

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **s.recv()**  This method receives TCP message |
| 2 | **s.send()**  This method transmits TCP message |
| 3 | **s.recvfrom()**  This method receives UDP message |
| 4 | **s.sendto()**  This method transmits UDP message |
| 5 | **s.close()**  This method closes socket |
| 6 | **socket.gethostname()**  Returns the hostname. |

**A Simple Server**

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call the **bind(hostname, port)** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

#!/usr/bin/python3 # This is server.py file

import socket

# create a socket object

serversocket = socket.socket(

socket.AF\_INET, socket.SOCK\_STREAM)

# get local machine name

host = socket.gethostname()

port = 9999

# bind to the port

serversocket.bind((host, port))

# queue up to 5 requests

serversocket.listen(5)

while True:

# establish a connection

clientsocket,addr = serversocket.accept()

print("Got a connection from %s" % str(addr))

msg = 'Thank you for connecting'+ "\r\n"

clientsocket.send(msg.encode('ascii'))

clientsocket.close()

**A Simple Client**

Let us write a very simple client program which opens a connection to a given port 12345 and a given host. It is very simple to create a socket client using the Python's *socket* module function.

The **socket.connect(hosname, port )** opens a TCP connection to *hostname*on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

Example

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

#!/usr/bin/python3 # This is client.py file

import socket

# create a socket object

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# get local machine name

host = socket.gethostname()

port = 9999

# connection to hostname on the port.

s.connect((host, port))

# Receive no more than 1024 bytes

msg = s.recv(1024)

s.close()

print (msg.decode('ascii'))

Now run this server.py in the background and then run the above client.py to see the result.

# Following would start a server in background.

$ python server.py &

# Once server is started run client as follows:

$ python client.py

Output

This would produce following result −

on server terminal

Got a connection from ('192.168.1.10', 3747)

On client terminal

Thank you for connecting

**Python Internet Modules**

A list of some important modules in Python Network/Internet programming are given below −

|  |  |  |  |
| --- | --- | --- | --- |
| **Protocol** | **Common function** | **Port No** | **Python module** |
| HTTP | Web pages | 80 | httplib, urllib, xmlrpclib |
| NNTP | Usenet news | 119 | nntplib |
| FTP | File transfers | 20 | ftplib, urllib |
| SMTP | Sending email | 25 | smtplib |
| POP3 | Fetching email | 110 | poplib |
| IMAP4 | Fetching email | 143 | imaplib |
| Telnet | Command lines | 23 | telnetlib |
| Gopher | Document transfers | 70 | gopherlib, urllib |

Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols.

1. **Sending Email**

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending an e-mail and routing e-mail between mail servers.

Python provides **smtplib** module, which defines an SMTP client session object that can be used to send mails to any Internet machine with an SMTP or ESMTP listener daemon.

Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail −

import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local\_hostname]]] )

Here is the detail of the parameters −

* **host** − This is the host running your SMTP server. You can specifiy IP address of the host or a domain name like tutorialspoint.com. This is an optional argument.
* **port** − If you are providing *host* argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
* **local\_hostname** − If your SMTP server is running on your local machine, then you can specify just *localhost* the option.

An SMTP object has an instance method called **sendmail**, which is typically used to do the work of mailing a message. It takes three parameters −

* The *sender* − A string with the address of the sender.
* The *receivers* − A list of strings, one for each recipient.
* The *message* − A message as a string formatted as specified in the various RFCs.

### Example

Here is a simple way to send one e-mail using Python script. Try it once −

#!/usr/bin/python3

import smtplib

sender = 'from@fromdomain.com'

receivers = ['to@todomain.com']

message = """From: From Person <from@fromdomain.com>

To: To Person <to@todomain.com>

Subject: SMTP e-mail test

This is a test e-mail message.

"""

try:

smtpObj = smtplib.SMTP('localhost')

smtpObj.sendmail(sender, receivers, message)

print "Successfully sent email"

except SMTPException:

print "Error: unable to send email"

Here, you have placed a basic e-mail in message, using a triple quote, taking care to format the headers correctly. An e-mail requires a **From**, **To**, and a **Subject** header, separated from the body of the e-mail with a blank line.

To send the mail you use *smtpObj* to connect to the SMTP server on the local machine. Then use the *sendmail* method along with the message, the from address, and the destination address as parameters (even though the from and to addresses are within the e-mail itself, these are not always used to route the mail).

If you are not running an SMTP server on your local machine, you can the use *smtplib* client to communicate with a remote SMTP server. Unless you are using a webmail service (such as gmail or Yahoo! Mail), your e-mail provider must have provided you with the outgoing mail server details that you can supply them, as follows −

mail = smtplib.SMTP('smtp.gmail.com', 587)

## Sending an HTML e-mail using Python

When you send a text message using Python, then all the content is treated as simple text. Even if you include HTML tags in a text message, it is displayed as simple text and HTML tags will not be formatted according to the HTML syntax. However, Python provides an option to send an HTML message as actual HTML message.

While sending an e-mail message, you can specify a Mime version, content type and the character set to send an HTML e-mail.

### Example

Following is an example to send the HTML content as an e-mail. Try it once −

#!/usr/bin/python3

import smtplib

message = """From: From Person <from@fromdomain.com>

To: To Person <to@todomain.com>

MIME-Version: 1.0

Content-type: text/html

Subject: SMTP HTML e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>

<h1>This is headline.</h1>

"""

try:

smtpObj = smtplib.SMTP('localhost')

smtpObj.sendmail(sender, receivers, message)

print "Successfully sent email"

except SMTPException:

print "Error: unable to send email"

## Sending Attachments as an E-mail

To send an e-mail with mixed content requires setting the **Content-type**header to **multipart/mixed**. Then, the text and the attachment sections can be specified within **boundaries**.

A boundary is started with two hyphens followed by a unique number, which cannot appear in the message part of the e-mail. A final boundary denoting the e-mail's final section must also end with two hyphens.

The attached files should be encoded with the **pack("m")** function to have base 64 encoding before transmission.

### Example

Following is an example, which sends a file **/tmp/test.txt** as an attachment. Try it once −

#!/usr/bin/python3

import smtplib

import base64

filename = "/tmp/test.txt"

# Read a file and encode it into base64 format

fo = open(filename, "rb")

filecontent = fo.read()

encodedcontent = base64.b64encode(filecontent) # base64

sender = 'webmaster@tutorialpoint.com'

reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"

body ="""

This is a test email to send an attachement.

"""

# Define the main headers.

part1 = """From: From Person <me@fromdomain.net>

To: To Person <amrood.admin@gmail.com>

Subject: Sending Attachement

MIME-Version: 1.0

Content-Type: multipart/mixed; boundary=%s

--%s

""" % (marker, marker)

# Define the message action

part2 = """Content-Type: text/plain

Content-Transfer-Encoding:8bit

%s

--%s

""" % (body,marker)

# Define the attachment section

part3 = """Content-Type: multipart/mixed; name=\"%s\"

Content-Transfer-Encoding:base64

Content-Disposition: attachment; filename=%s

%s

--%s--

""" %(filename, filename, encodedcontent, marker)

message = part1 + part2 + part3

try:

smtpObj = smtplib.SMTP('localhost')

smtpObj.sendmail(sender, reciever, message)

print "Successfully sent email"

except Exception:

print ("Error: unable to send email")

1. **Multithreading**

Running several threads is similar to running several different programs concurrently, but with the following benefits −

* Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
* Threads are sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context is it currently running.

* It can be pre-empted (interrupted).
* It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

There are two different kind of threads −

* kernel thread
* user thread

Kernel Threads are a part of the operating system, while the User-space threads are not implemented in the kernel.

There are two modules which support the usage of threads in Python3 −

* \_thread
* threading

The thread module has been "deprecated" for quite a long time. Users are encouraged to use the threading module instead. Hence, in Python 3, the module "thread" is not available anymore. However, it has been renamed to "\_thread" for backwards compatibilities in Python3.

**Starting a New Thread**

To spawn another thread, you need to call the following method available in the *thread* module −

\_thread.start\_new\_thread ( function, args[, kwargs] )

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of *args*. When the function returns, the thread terminates.

Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs*is an optional dictionary of keyword arguments.

Example

#!/usr/bin/python3

import \_thread

import time

# Define a function for the thread

def print\_time( threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print ("%s: %s" % ( threadName, time.ctime(time.time()) ))

# Create two threads as follows

try:

\_thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )

\_thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )

except:

print ("Error: unable to start thread")

while 1:

pass

Output

When the above code is executed, it produces the following result −

Thread-1: Fri Feb 19 09:41:39 2016

Thread-2: Fri Feb 19 09:41:41 2016

Thread-1: Fri Feb 19 09:41:41 2016

Thread-1: Fri Feb 19 09:41:43 2016

Thread-2: Fri Feb 19 09:41:45 2016

Thread-1: Fri Feb 19 09:41:45 2016

Thread-1: Fri Feb 19 09:41:47 2016

Thread-2: Fri Feb 19 09:41:49 2016

Thread-2: Fri Feb 19 09:41:53 2016

Program goes in an infinite loop. You will have to press ctrl-c to stop

Although it is very effective for low-level threading, the *thread* module is very limited compared to the newer threading module.

**The Threading Module**

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The *threading* module exposes all the methods of the *thread* module and provides some additional methods −

* **threading.activeCount()** − Returns the number of thread objects that are active.
* **threading.currentThread()** − Returns the number of thread objects in the caller's thread control.
* **threading.enumerate()** − Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows −

* **run()** − The run() method is the entry point for a thread.
* **start()** − The start() method starts a thread by calling the run method.
* **join([time])** − The join() waits for threads to terminate.
* **isAlive()** − The isAlive() method checks whether a thread is still executing.
* **getName()** − The getName() method returns the name of a thread.
* **setName()** − The setName() method sets the name of a thread.

**Creating Thread Using Threading Module**

To implement a new thread using the threading module, you have to do the following −

* Define a new subclass of the *Thread* class.
* Override the *\_\_init\_\_(self [,args])* method to add additional arguments.
* Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls the *run()* method.

Example

#!/usr/bin/python3

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print ("Starting " + self.name)

print\_time(self.name, self.counter, 5)

print ("Exiting " + self.name)

def print\_time(threadName, delay, counter):

while counter:

if exitFlag:

threadName.exit()

time.sleep(delay)

print ("%s: %s" % (threadName, time.ctime(time.time())))

counter -= 1

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

thread1.join()

thread2.join()

print ("Exiting Main Thread")

Result

When we run the above program, it produces the following result −

Starting Thread-1

Starting Thread-2

Thread-1: Fri Feb 19 10:00:21 2016

Thread-2: Fri Feb 19 10:00:22 2016

Thread-1: Fri Feb 19 10:00:22 2016

Thread-1: Fri Feb 19 10:00:23 2016

Thread-2: Fri Feb 19 10:00:24 2016

Thread-1: Fri Feb 19 10:00:24 2016

Thread-1: Fri Feb 19 10:00:25 2016

Exiting Thread-1

Thread-2: Fri Feb 19 10:00:26 2016

Thread-2: Fri Feb 19 10:00:28 2016

Thread-2: Fri Feb 19 10:00:30 2016

Exiting Thread-2

Exiting Main Thread

**Synchronizing Threads**

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force the threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

Example

#!/usr/bin/python3

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print ("Starting " + self.name)

# Get lock to synchronize threads

threadLock.acquire()

print\_time(self.name, self.counter, 3)

# Free lock to release next thread

threadLock.release()

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print ("%s: %s" % (threadName, time.ctime(time.time())))

counter -= 1

threadLock = threading.Lock()

threads = []

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

# Add threads to thread list

threads.append(thread1)

threads.append(thread2)

# Wait for all threads to complete

for t in threads:

t.join()

print ("Exiting Main Thread")

Output

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Thread-1: Fri Feb 19 10:04:14 2016

Thread-1: Fri Feb 19 10:04:15 2016

Thread-1: Fri Feb 19 10:04:16 2016

Thread-2: Fri Feb 19 10:04:18 2016

Thread-2: Fri Feb 19 10:04:20 2016

Thread-2: Fri Feb 19 10:04:22 2016

Exiting Main Thread

**Multithreaded Priority Queue**

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue −

* **get()** − The get() removes and returns an item from the queue.
* **put()** − The put adds item to a queue.
* **qsize()**− The qsize() returns the number of items that are currently in the queue.
* **empty()** − The empty( ) returns True if queue is empty; otherwise, False.
* **full()** − the full() returns True if queue is full; otherwise, False.

Example

#!/usr/bin/python3

import queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, q):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.q = q

def run(self):

print ("Starting " + self.name)

process\_data(self.name, self.q)

print ("Exiting " + self.name)

def process\_data(threadName, q):

while not exitFlag:

queueLock.acquire()

if not workQueue.empty():

data = q.get()

queueLock.release()

print ("%s processing %s" % (threadName, data))

else:

queueLock.release()

time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = queue.Queue(10)

threads = []

threadID = 1

# Create new threads

for tName in threadList:

thread = myThread(threadID, tName, workQueue)

thread.start()

threads.append(thread)

threadID += 1

# Fill the queue

queueLock.acquire()

for word in nameList:

workQueue.put(word)

queueLock.release()

# Wait for queue to empty

while not workQueue.empty():

pass

# Notify threads it's time to exit

exitFlag = 1

# Wait for all threads to complete

for t in threads:

t.join()

print ("Exiting Main Thread")

Output

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

1. **XML Processing**

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

**What is XML?**

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

XML is extremely useful for keeping track of small to medium amounts of data without requiring an SQL-based backbone.

**XML Parser Architectures and APIs**

The Python standard library provides a minimal but useful set of interfaces to work with XML.

The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

* **Simple API for XML (SAX)** − Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from the disk and the entire file is never stored in the memory.
* **Document Object Model (DOM) API** − This is a World Wide Web Consortium recommendation wherein the entire file is read into the memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously cannot process information as fast as DOM, when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on many small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other, there is no reason why you cannot use them both for large projects.

For all our XML code examples, let us use a simple XML file *movies.xml* as an input −

<collection shelf = "New Arrivals">

<movie title = "Enemy Behind">

<type>War, Thriller</type>

<format>DVD</format>

<year>2003</year>

<rating>PG</rating>

<stars>10</stars>

<description>Talk about a US-Japan war</description>

</movie>

<movie title = "Transformers">

<type>Anime, Science Fiction</type>

<format>DVD</format>

<year>1989</year>

<rating>R</rating>

<stars>8</stars>

<description>A schientific fiction</description>

</movie>

<movie title = "Trigun">

<type>Anime, Action</type>

<format>DVD</format>

<episodes>4</episodes>

<rating>PG</rating>

<stars>10</stars>

<description>Vash the Stampede!</description>

</movie>

<movie title = "Ishtar">

<type>Comedy</type>

<format>VHS</format>

<rating>PG</rating>

<stars>2</stars>

<description>Viewable boredom</description>

</movie>

</collection>

**Parsing XML with SAX APIs**

SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing xml.sax.ContentHandler.

Your *ContentHandler* handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.

The methods *startDocument* and *endDocument* are called at the start and the end of the XML file. The method *characters(text)* is passed the character data of the XML file via the parameter text.

The ContentHandler is called at the start and end of each element. If the parser is not in namespace mode, the methods *startElement(tag, attributes)*and *endElement(tag)* are called; otherwise, the corresponding methods *startElementNS* and *endElementNS* are called. Here, tag is the element tag, and attributes is an Attributes object.

Here are other important methods to understand before proceeding −

The make\_parser Method

The following method creates a new parser object and returns it. The parser object created will be of the first parser type, the system finds.

xml.sax.make\_parser( [parser\_list] )

Here are the details of the parameters −

* **parser\_list** − The optional argument consisting of a list of parsers to use which must all implement the make\_parser method.

**The parse Method**

The following method creates a SAX parser and uses it to parse a document.

xml.sax.parse( xmlfile, contenthandler[, errorhandler])

Here are the details of the parameters −

* **xmlfile** − This is the name of the XML file to read from.
* **contenthandler** − This must be a ContentHandler object.
* **errorhandler** − If specified, errorhandler must be a SAX ErrorHandler object.

**The parseString Method**

There is one more method to create a SAX parser and to parse the specified **XML string**.

xml.sax.parseString(xmlstring, contenthandler[, errorhandler])

Here are the details of the parameters −

* **xmlstring** − This is the name of the XML string to read from.
* **contenthandler** − This must be a ContentHandler object.
* **errorhandler** − If specified, errorhandler must be a SAX ErrorHandler object.

Example

#!/usr/bin/python3

import xml.sax

class MovieHandler( xml.sax.ContentHandler ):

def \_\_init\_\_(self):

self.CurrentData = ""

self.type = ""

self.format = ""

self.year = ""

self.rating = ""

self.stars = ""

self.description = ""

# Call when an element starts

def startElement(self, tag, attributes):

self.CurrentData = tag

if tag == "movie":

print ("\*\*\*\*\*Movie\*\*\*\*\*")

title = attributes["title"]

print ("Title:", title)

# Call when an elements ends

def endElement(self, tag):

if self.CurrentData == "type":

print ("Type:", self.type)

elif self.CurrentData == "format":

print ("Format:", self.format)

elif self.CurrentData == "year":

print ("Year:", self.year)

elif self.CurrentData == "rating":

print ("Rating:", self.rating)

elif self.CurrentData == "stars":

print ("Stars:", self.stars)

elif self.CurrentData == "description":

print ("Description:", self.description)

self.CurrentData = ""

# Call when a character is read

def characters(self, content):

if self.CurrentData == "type":

self.type = content

elif self.CurrentData == "format":

self.format = content

elif self.CurrentData == "year":

self.year = content

elif self.CurrentData == "rating":

self.rating = content

elif self.CurrentData == "stars":

self.stars = content

elif self.CurrentData == "description":

self.description = content

if ( \_\_name\_\_ == "\_\_main\_\_"):

# create an XMLReader

parser = xml.sax.make\_parser()

# turn off namepsaces

parser.setFeature(xml.sax.handler.feature\_namespaces, 0)

# override the default ContextHandler

Handler = MovieHandler()

parser.setContentHandler( Handler )

parser.parse("movies.xml")

Output

This would produce the following result −

\*\*\*\*\*Movie\*\*\*\*\*

Title: Enemy Behind

Type: War, Thriller

Format: DVD

Year: 2003

Rating: PG

Stars: 10

Description: Talk about a US-Japan war

\*\*\*\*\*Movie\*\*\*\*\*

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Year: 1989

Rating: R

Stars: 8

Description: A schientific fiction

\*\*\*\*\*Movie\*\*\*\*\*

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Stars: 10

Description: Vash the Stampede!

\*\*\*\*\*Movie\*\*\*\*\*

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Stars: 2

Description: Viewable boredom

For a complete detail on SAX API documentation, please refer to the standard [Python SAX APIs](https://docs.python.org/library/xml.sax.html).

**Parsing XML with DOM APIs**

The Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying the XML documents.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.

Here is the easiest way to load an XML document quickly and to create a minidom object using the xml.dom module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.

The sample phrase calls the parse( file [,parser] ) function of the minidom object to parse the XML file, designated by file into a DOM tree object.

Example

#!/usr/bin/python3

from xml.dom.minidom import parse

import xml.dom.minidom

# Open XML document using minidom parser

DOMTree = xml.dom.minidom.parse("movies.xml")

collection = DOMTree.documentElement

if collection.hasAttribute("shelf"):

print ("Root element : %s" % collection.getAttribute("shelf"))

# Get all the movies in the collection

movies = collection.getElementsByTagName("movie")

# Print detail of each movie.

for movie in movies:

print ("\*\*\*\*\*Movie\*\*\*\*\*")

if movie.hasAttribute("title"):

print ("Title: %s" % movie.getAttribute("title"))

type = movie.getElementsByTagName('type')[0]

print ("Type: %s" % type.childNodes[0].data)

format = movie.getElementsByTagName('format')[0]

print ("Format: %s" % format.childNodes[0].data)

rating = movie.getElementsByTagName('rating')[0]

print ("Rating: %s" % rating.childNodes[0].data)

description = movie.getElementsByTagName('description')[0]

print ("Description: %s" % description.childNodes[0].data)

Output

This would produce the following result −

Root element : New Arrivals

\*\*\*\*\*Movie\*\*\*\*\*

Title: Enemy Behind

Type: War, Thriller

Format: DVD

Rating: PG

Description: Talk about a US-Japan war

\*\*\*\*\*Movie\*\*\*\*\*

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Rating: R

Description: A schientific fiction

\*\*\*\*\*Movie\*\*\*\*\*

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Description: Vash the Stampede!

\*\*\*\*\*Movie\*\*\*\*\*

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Description: Viewable boredom

Python provides various options for developing graphical user interfaces (GUIs). The most important features are listed below.

* **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
* **wxPython** − This is an open-source Python interface for wxWidgets GUI toolkit. You can find a complete tutorial on WxPython [here](https://www.tutorialspoint.com/wxpython/index.htm).
* **PyQt** −This is also a Python interface for a popular cross-platform Qt GUI library. TutorialsPoint has a very good tutorial on PyQt [here](https://www.tutorialspoint.com/pyqt/index.htm).
* **JPython** − JPython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machine [http://www.jython.org](http://www.jython.org/).

There are many other interfaces available, which you can find them on the net.

1. **GUI Programming**

**Tkinter Programming**

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

* Import the *Tkinter* module.
* Create the GUI application main window.
* Add one or more of the above-mentioned widgets to the GUI application.
* Enter the main event loop to take action against each event triggered by the user.

Example

#!/usr/bin/python3

import tkinter # note that module name has changed from Tkinter in Python 2 to tkinter in Python 3

top = tkinter.Tk()

# Code to add widgets will go here...

top.mainloop()

This would create a following window −



**Tkinter Widgets**

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table –

|  |  |
| --- | --- |
| **S.No.** | **Operator & Description** |
| 1 | [**Button**](https://www.tutorialspoint.com/python3/tk_button.htm)  The Button widget is used to display the buttons in your application. |
| 2 | [**Canvas**](https://www.tutorialspoint.com/python3/tk_canvas.htm)  The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application. |
| 3 | [**Checkbutton**](https://www.tutorialspoint.com/python3/tk_checkbutton.htm)  The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time. |
| 4 | [**Entry**](https://www.tutorialspoint.com/python3/tk_entry.htm)  The Entry widget is used to display a single-line text field for accepting values from a user. |
| 5 | [**Frame**](https://www.tutorialspoint.com/python3/tk_frame.htm)  The Frame widget is used as a container widget to organize other widgets. |
| 6 | [**Label**](https://www.tutorialspoint.com/python3/tk_label.htm)  The Label widget is used to provide a single-line caption for other widgets. It can also contain images. |
| 7 | [**Listbox**](https://www.tutorialspoint.com/python3/tk_listbox.htm)  The Listbox widget is used to provide a list of options to a user. |
| 8 | [**Menubutton**](https://www.tutorialspoint.com/python3/tk_menubutton.htm)  The Menubutton widget is used to display menus in your application. |
| 9 | [**Menu**](https://www.tutorialspoint.com/python3/tk_menu.htm)  The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton. |
| 10 | [**Message**](https://www.tutorialspoint.com/python3/tk_message.htm)  The Message widget is used to display multiline text fields for accepting values from a user. |
| 11 | [**Radiobutton**](https://www.tutorialspoint.com/python3/tk_radiobutton.htm)  The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. |
| 12 | [**Scale**](https://www.tutorialspoint.com/python3/tk_scale.htm)  The Scale widget is used to provide a slider widget. |
| 13 | [**Scrollbar**](https://www.tutorialspoint.com/python3/tk_scrollbar.htm)  The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes. |
| 14 | [**Text**](https://www.tutorialspoint.com/python3/tk_text.htm)  The Text widget is used to display text in multiple lines. |
| 15 | [**Toplevel**](https://www.tutorialspoint.com/python3/tk_toplevel.htm)  The Toplevel widget is used to provide a separate window container. |
| 16 | [**Spinbox**](https://www.tutorialspoint.com/python3/tk_spinbox.htm)  The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values. |
| 17 | [**PanedWindow**](https://www.tutorialspoint.com/python3/tk_panedwindow.htm)  A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically. |
| 18 | [**LabelFrame**](https://www.tutorialspoint.com/python3/tk_labelframe.htm)  A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. |
| 19 | [**tkMessageBox**](https://www.tutorialspoint.com/python3/tk_messagebox.htm)  This module is used to display message boxes in your applications. |

**Standard attributes**

Let us look at how some of their common attributes, such as sizes, colors and fonts are specified.

* [Dimensions](https://www.tutorialspoint.com/python3/tk_dimensions.htm)
* [Colors](https://www.tutorialspoint.com/python3/tk_colors.htm)
* [Fonts](https://www.tutorialspoint.com/python3/tk_fonts.htm)
* [Anchors](https://www.tutorialspoint.com/python3/tk_anchors.htm)
* [Relief styles](https://www.tutorialspoint.com/python3/tk_relief.htm)
* [Bitmaps](https://www.tutorialspoint.com/python3/tk_bitmaps.htm)
* [Cursors](https://www.tutorialspoint.com/python3/tk_cursors.htm)

**Geometry Management**

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

* [The pack() Method](https://www.tutorialspoint.com/python3/tk_pack.htm) − This geometry manager organizes widgets in blocks before placing them in the parent widget.
* [The grid() Method](https://www.tutorialspoint.com/python3/tk_grid.htm) − This geometry manager organizes widgets in a table-like structure in the parent widget.

[The place() Method](https://www.tutorialspoint.com/python3/tk_place.htm) − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

1. **Further Extensions**

Any code that you write using any compiled language like C, C++, or Java can be integrated or imported into another Python script. This code is considered as an "extension."

A Python extension module is nothing more than a normal C library. On Unix machines, these libraries usually end in **.so** (for shared object). On Windows machines, you typically see **.dll** (for dynamically linked library).

**Pre-Requisites for Writing Extensions**

To start writing your extension, you are going to need the Python header files.

* On Unix machines, this usually requires installing a developer-specific package such as [python2.5-dev](https://packages.debian.org/etch-m68k/python2.5-dev).
* Windows users get these headers as part of the package when they use the binary Python installer.

Additionally, it is assumed that you have a good knowledge of C or C++ to write any Python Extension using C programming.

**First look at a Python Extension**

For your first look at a Python extension module, you need to group your code into four part −

* The header file *Python.h*.
* The C functions you want to expose as the interface from your module.
* A table mapping the names of your functions as Python developers see them as C functions inside the extension module.
* An initialization function.

**The Header File Python.h**

You need to include *Python.h* header file in your C source file, which gives you the access to the internal Python API used to hook your module into the interpreter.

Make sure to include Python.h before any other headers you might need. You need to follow the includes with the functions you want to call from Python.

The C Functions

The signatures of the C implementation of your functions always takes one of the following three forms −

static PyObject \*MyFunction( PyObject \*self, PyObject \*args );

static PyObject \*MyFunctionWithKeywords(PyObject \*self,

PyObject \*args,

PyObject \*kw);

static PyObject \*MyFunctionWithNoArgs( PyObject \*self );

Each one of the preceding declarations returns a Python object. There is no such thing as a *void*function in Python as there is in C. If you do not want your functions to return a value, return the C equivalent of Python's **None**value. The Python headers define a macro, Py\_RETURN\_NONE, that does this for us.

The names of your C functions can be whatever you like as they are never seen outside of the extension module. They are defined as *static* function.

Your C functions usually are named by combining the Python module and function names together, as shown here −

static PyObject \*module\_func(PyObject \*self, PyObject \*args) {

/\* Do your stuff here. \*/

Py\_RETURN\_NONE;

}

This is a Python function called *func* inside the module *module*. You will be putting pointers to your C functions into the method table for the module that usually comes next in your source code.

**The Method Mapping Table**

This method table is a simple array of PyMethodDef structures. That structure looks something like this −

struct PyMethodDef {

char \*ml\_name;

PyCFunction ml\_meth;

int ml\_flags;

char \*ml\_doc;

};

Here is the description of the members of this structure −

* **ml\_name** − This is the name of the function as the Python interpreter presents when it is used in Python programs.
* **ml\_meth** − This is the address of a function that has any one of the signatures described in the previous section.
* **ml\_flags** − This tells the interpreter which of the three signatures ml\_meth is using.
  + This flag usually has a value of METH\_VARARGS.
  + This flag can be bitwise OR'ed with METH\_KEYWORDS if you want to allow keyword arguments into your function.
  + This can also have a value of METH\_NOARGS that indicates you do not want to accept any arguments.
* **ml\_doc** − This is the docstring for the function, which could be NULL if you do not feel like writing one.

This table needs to be terminated with a sentinel that consists of NULL and 0 values for the appropriate members.

Example

For the above-defined function, we have the following method mapping table −

static PyMethodDef module\_methods[] = {

{ "func", (PyCFunction)module\_func, METH\_NOARGS, NULL },

{ NULL, NULL, 0, NULL }

};

**The Initialization Function**

The last part of your extension module is the initialization function. This function is called by the Python interpreter when the module is loaded. It is required that the function be named **init*Module***, where *Module* is the name of the module.

The initialization function needs to be exported from the library you will be building. The Python headers define PyMODINIT\_FUNC to include the appropriate incantations for that to happen for the particular environment in which we are compiling. All you have to do is use it when defining the function.

Your C initialization function generally has the following overall structure −

PyMODINIT\_FUNC initModule() {

Py\_InitModule3(func, module\_methods, "docstring...");

}

Here is the description of **Py\_InitModule3** function −

* **func** − This is the function to be exported.
* **module\_methods** − This is the mapping table name defined above.
* **docstring** − This is the comment you want to give in your extension.

Putting all this together, it looks like the following −

#include <Python.h>

static PyObject \*module\_func(PyObject \*self, PyObject \*args) {

/\* Do your stuff here. \*/

Py\_RETURN\_NONE;

}

static PyMethodDef module\_methods[] = {

{ "func", (PyCFunction)module\_func, METH\_NOARGS, NULL },

{ NULL, NULL, 0, NULL }

};

PyMODINIT\_FUNC initModule() {

Py\_InitModule3(func, module\_methods, "docstring...");

}

Example

A simple example that makes use of all the above concepts −

#include <Python.h>

static PyObject\* helloworld(PyObject\* self)

{

return Py\_BuildValue("s", "Hello, Python extensions!!");

}

static char helloworld\_docs[] =

"helloworld( ): Any message you want to put here!!\n";

static PyMethodDef helloworld\_funcs[] = {

{"helloworld", (PyCFunction)helloworld,

METH\_NOARGS, helloworld\_docs},

{NULL}

};

void inithelloworld(void)

{

Py\_InitModule3("helloworld", helloworld\_funcs,

"Extension module example!");

}

Here the *Py\_BuildValue* function is used to build a Python value. Save above code in hello.c file. We would see how to compile and install this module to be called from Python script.

**Building and Installing Extensions**

The *distutils* package makes it very easy to distribute Python modules, both pure Python and extension modules, in a standard way. Modules are distributed in the source form, built and installed via a setup script usually called *setup.py* as.

For the above module, you need to prepare the following setup.py script −

from distutils.core import setup, Extension

setup(name = 'helloworld', version = '1.0', \

ext\_modules = [Extension('helloworld', ['hello.c'])])

Now, use the following command, which would perform all needed compilation and linking steps, with the right compiler and linker commands and flags, and copies the resulting dynamic library into an appropriate directory −

$ python setup.py install

On Unix-based systems, you will most likely need to run this command as root in order to have permissions to write to the site-packages directory. This usually is not a problem on Windows.

Importing Extensions

Once you install your extensions, you would be able to import and call that extension in your Python script as follows −

Example

#!/usr/bin/python3

import helloworld

print helloworld.helloworld()

Output

This would produce the following result −

Hello, Python extensions!!

**Passing Function Parameters**

As you will most likely want to define functions that accept arguments, you can use one of the other signatures for your C functions. For example, the following function, that accepts some number of parameters, would be defined like this −

static PyObject \*module\_func(PyObject \*self, PyObject \*args) {

/\* Parse args and do something interesting here. \*/

Py\_RETURN\_NONE;

}

The method table containing an entry for the new function would look like this −

static PyMethodDef module\_methods[] = {

{ "func", (PyCFunction)module\_func, METH\_NOARGS, NULL },

{ "func", module\_func, METH\_VARARGS, NULL },

{ NULL, NULL, 0, NULL }

};

You can use the API *PyArg\_ParseTuple* function to extract the arguments from the one PyObject pointer passed into your C function.

The first argument to PyArg\_ParseTuple is the args argument. This is the object you will be *parsing*. The second argument is a format string describing the arguments as you expect them to appear. Each argument is represented by one or more characters in the format string as follows.

static PyObject \*module\_func(PyObject \*self, PyObject \*args) {

int i;

double d;

char \*s;

if (!PyArg\_ParseTuple(args, "ids", &i, &d, &s)) {

return NULL;

}

/\* Do something interesting here. \*/

Py\_RETURN\_NONE;

}

Output

Compiling the new version of your module and importing it enables you to invoke the new function with any number of arguments of any type −

module.func(1, s = "three", d = 2.0)

module.func(i = 1, d = 2.0, s = "three")

module.func(s = "three", d = 2.0, i = 1)

You can probably come up with even more variations.

**The PyArg\_ParseTuple Function**

Here is the standard signature for the **PyArg\_ParseTuple** function −

int PyArg\_ParseTuple(PyObject\* tuple,char\* format,...)

This function returns 0 for errors, and a value not equal to 0 for success. Tuple is the PyObject\* that was the C function's second argument. Here *format* is a C string that describes mandatory and optional arguments.

Here is a list of format codes for the **PyArg\_ParseTuple** function −

|  |  |  |
| --- | --- | --- |
| **Code** | **C type** | **Meaning** |
| c | char | A Python string of length 1 becomes a C char. |
| d | double | A Python float becomes a C double. |
| f | float | A Python float becomes a C float. |
| i | int | A Python int becomes a C int. |
| l | long | A Python int becomes a C long. |
| L | long long | A Python int becomes a C long long |
| O | PyObject\* | Gets non-NULL borrowed reference to Python argument. |
| s | char\* | Python string without embedded nulls to C char\*. |
| s# | char\*+int | Any Python string to C address and length. |
| t# | char\*+int | Read-only single-segment buffer to C address and length. |
| u | Py\_UNICODE\* | Python Unicode without embedded nulls to C. |
| u# | Py\_UNICODE\*+int | Any Python Unicode C address and length. |
| w# | char\*+int | Read/write single-segment buffer to C address and length. |
| z | char\* | Like s, also accepts None (sets C char\* to NULL). |
| z# | char\*+int | Like s#, also accepts None (sets C char\* to NULL). |
| (...) | as per ... | A Python sequence is treated as one argument per item. |
| | |  | The following arguments are optional. |
| : |  | Format end, followed by function name for error messages. |
| ; |  | Format end, followed by entire error message text. |

**Returning Values**

*Py\_BuildValue* takes in a format string much like *PyArg\_ParseTuple* does. Instead of passing in the addresses of the values you are building, you pass in the actual values. Here is an example showing how to implement an add function −

static PyObject \*foo\_add(PyObject \*self, PyObject \*args) {

int a;

int b;

if (!PyArg\_ParseTuple(args, "ii", &a, &b)) {

return NULL;

}

return Py\_BuildValue("i", a + b);

}

This is what it would look like if implemented in Python −

def add(a, b):

return (a + b)

You can return two values from your function as follows. This would be captured using a list in Python.

static PyObject \*foo\_add\_subtract(PyObject \*self, PyObject \*args) {

int a;

int b;

if (!PyArg\_ParseTuple(args, "ii", &a, &b)) {

return NULL;

}

return Py\_BuildValue("ii", a + b, a - b);

}

This is what it would look like if implemented in Python −

def add\_subtract(a, b):

return (a + b, a - b)

The *Py\_BuildValue* Function

Here is the standard signature for **Py\_BuildValue** function −

PyObject\* Py\_BuildValue(char\* format,...)

Here *format* is a C string that describes the Python object to build. The following arguments of *Py\_BuildValue* are C values from which the result is built. The *PyObject\** result is a new reference.

The following table lists the commonly used code strings, of which zero or more are joined into a string format.

|  |  |  |
| --- | --- | --- |
| **Code** | **C type** | **Meaning** |
| c | char | A C char becomes a Python string of length 1. |
| d | double | A C double becomes a Python float. |
| f | float | A C float becomes a Python float. |
| i | int | A C int becomes a Python int. |
| l | long | A C long becomes a Python int. |
| N | PyObject\* | Passes a Python object and steals a reference. |
| O | PyObject\* | Passes a Python object and INCREFs it as normal. |
| O& | convert+void\* | Arbitrary conversion |
| s | char\* | C 0-terminated char\* to Python string, or NULL to None. |
| s# | char\*+int | C char\* and length to Python string, or NULL to None. |
| u | Py\_UNICODE\* | C-wide, null-terminated string to Python Unicode, or NULL to None. |
| u# | Py\_UNICODE\*+int | C-wide string and length to Python Unicode, or NULL to None. |
| w# | char\*+int | Read/write single-segment buffer to C address and length. |
| z | char\* | Like s, also accepts None (sets C char\* to NULL). |
| z# | char\*+int | Like s#, also accepts None (sets C char\* to NULL). |
| (...) | as per ... | Builds Python tuple from C values. |
| [...] | as per ... | Builds Python list from C values. |
| {...} | as per ... | Builds Python dictionary from C values, alternating keys and values. |

Code {...} builds dictionaries from an even number of C values, alternately keys and values. For example, Py\_BuildValue("{issi}",23,"zig","zag",42) returns a dictionary like Python's {23:'zig','zag':42}.