Your reservation was successfully created!

Your reference number is 538c8d056ee08

Pasted from <<https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00>>

* + Meisam Hejazi Nia
  + Resources to be reserved  
    [Austin Room 3.232 \_\_\_ (Must have 6 person min present at checkout)](https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00)   
    [(More Resources)](https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00)

Pasted from <<https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00>>

* + Meisam Hejazi Nia
  + Resources to be reserved  
    [Austin Room 3.232 \_\_\_ (Must have 6 person min present at checkout)](https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00)   
    [(More Resources)](https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00)

Pasted from <<https://libservices.utdallas.edu/room/Web/reservation.php?rid=2&sid=1&rd=2014-06-03&sd=2014-06-03%2010:00:00&ed=2014-06-03%2010:30:00>>

**MS ACCESS: ABS FUNCTION**

Learn how to use the Access **Abs function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Abs function** returns the absolute value of a number.

**SYNTAX**

The syntax for the Microsoft Access **Abs function** is:

Abs ( number )

**PARAMETERS OR ARGUMENTS**

*number* is a numeric value.

**APPLIES TO**

The **Abs function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **Abs (-210.67)** | would return 210.67 |
| **Abs (-2.9)** | would return 2.9 |
| **Abs (-3)** | would return 3 |

**EXAMPLE (IN VBA CODE)**

The **Abs function** can be used in VBA code in Microsoft Access. For example:

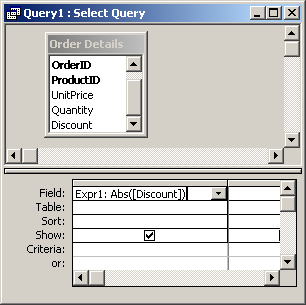
Dim LNumber As Double

LNumber = Abs (-210.6)

In this example, the variable called LNumber would now contain the value of 210.6.

**EXAMPLE (IN SQL/QUERIES)**

You can also use the **Abs function** in a query in Microsoft Access.



Pasted from <<http://www.techonthenet.com/access/functions/numeric/abs.php>>

**MS ACCESS: AND FUNCTION**

Learn how to use the Access **And function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **And function** returns TRUE if all conditions are TRUE. It returns FALSE if any of the conditions are FALSE.

**SYNTAX**

The syntax for the Microsoft Access **And function** is:

condition1 And condition2 [... And condition\_n] )

**PARAMETERS OR ARGUMENTS**

*condition1, condition2, ... condition\_n* are expressions that you want to test that can either be TRUE or FALSE.

**APPLIES TO**

The **And function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **And function** with this syntax can only be used in VBA code in Microsoft Access.

Let's look at an example that combines the **AND function** with the [**IF Statement**](http://www.techonthenet.com/access/functions/advanced/if_then.php) in VBA:

If LWebsite = "TechOnTheNet.com" And LCount > 250 Then  
 LResult = "High"  
Else  
 LResult = "Low"  
End If

This would set the LResult variable to the string value "High" if both LWebsite was "TechOnTheNet.com" and LCount > 250. Otherwise, it would set the LResult variable to the string value "Low".

You can use the **AND function** with the [**OR function**](http://www.techonthenet.com/access/functions/advanced/or.php) in VBA, for example:

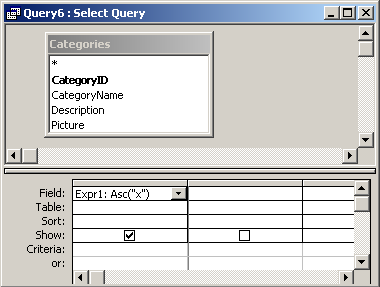
If (LWebsite = "TechOnTheNet.com" Or LWebsite = "CheckYourMath.com") And LCount >=1000 Then  
 LTotal = 5  
Else  
 LTotal = 25  
End If

This would set the LTotal variable to the value 5 if LWebsite was either "TechOnTheNet.com" or "CheckYourMath.com" and LCount >=1000. Otherwise, it would set the LTotal variable to the value 25.

Pasted from <<http://www.techonthenet.com/access/functions/advanced/and.php>>

**EXAMPLE (IN SQL/QUERIES)**

You can also use the **Asc function** in a query in Microsoft Access.



[**View a listing of the ascii values**](http://www.techonthenet.com/ascii/chart.php).

C:\Users\MeisamHe\AppData\Local\Temp\msohtmlclip1\02\clip_image003.png

Pasted from <<http://www.techonthenet.com/access/functions/string/asc.php>>

**MS ACCESS: ATN FUNCTION**

Learn how to use the Access **Atn function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Atn function** returns the arctangent of a number.

**SYNTAX**

The syntax for the Microsoft Access **Atn function** is:

Atn ( number )

**PARAMETERS OR ARGUMENTS**

*number* is a numeric expression.

**APPLIES TO**

The **Atn function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **Atn (2)** | would return 1.10714871779409 |
| **Atn (2.51)** | would return 1.19166451926354 |
| **Atn (-3.25)** | would return -1.27229739520872 |

**EXAMPLE (IN VBA CODE)**

The **Atn function** can be used in VBA code in Microsoft Access. For example:

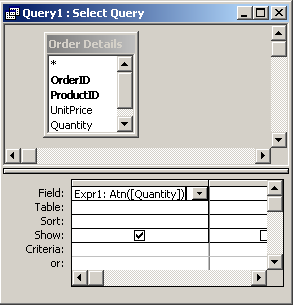
Dim LNumber As Double

LNumber = Atn (210)

In this example, the variable called LNumber would now contain the value of 1.56603445802574.

**EXAMPLE (IN SQL/QUERIES)**

You can also use the **Atn function** in a query in Microsoft Access.



Pasted from <<http://www.techonthenet.com/access/functions/numeric/atn.php>>

**MS ACCESS: AVG FUNCTION**

Learn how to use the Access **Avg function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Avg function** returns the average (arithmetic mean) of a set of numeric values in a select query.

**SYNTAX**

The syntax for the Microsoft Access **Avg function** is:

Avg ( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is a numeric value. It can be a field or a formula that evaluates to a numeric expression.

**APPLIES TO**

The **Avg function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN SQL/QUERIES)**

You can use the **Avg function** in a query by clicking on the Totals button in the toolbar (This is the button with the summation symbol). The **Avg function** is used in conjunction with the **Group By**clause.

This query would return the average UnitPrice by ProductName.

Pasted from <<http://www.techonthenet.com/access/functions/numeric/avg.php>>

**MS ACCESS: CASE STATEMENT**

Learn how to use the Access **Case statement** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Case statement** can only be used in VBA code. It has the functionality of an [**IF-THEN-ELSE statement**](http://www.techonthenet.com/access/functions/advanced/if_then.php).

**SYNTAX**

The syntax for the Microsoft Access **Case statement** is:

Select Case *test\_expression*

Case condition\_1  
 result\_1  
 Case condition\_2  
 result\_2  
 ...  
 Case condition\_n  
 result\_n

Case Else  
 result\_else

End Select

**PARAMETERS OR ARGUMENTS**

*test\_expression* is a string or numeric value. It is the value that you are comparing to the list of conditions. (ie: condition\_1, condition\_2, ... condition\_n)

*condition\_1* to *condition\_n* are evaluated in the order listed. Once a *condition* is found to be true, the **Case statement** will execute the corresponding code and not evaluate the conditions any further.

*result\_1* to *result\_n* is the code that is executed once a *condition* is found to be true.

**NOTE**

If no *condition* is met, then the *Else* portion of the **Case statement** will be executed. It is important to note that the *Else* portion is optional.

**APPLIES TO**

The **Case function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **Case statement** can only be used in VBA code in Microsoft Access.

Let's look at a simple example:

Select Case [Region]  
 Case "N"  
 [RegionName] = "North"  
 Case "S"  
 [RegionName] = "South"  
 Case "E"  
 [RegionName] = "East"  
 Case "W"  
 [RegionName] = "West"  
End Select

You can also use the **To keyword** to specify a range of values. For example:

Select Case LNumber  
 Case 1 To 10  
 [RegionName] = "North"  
 Case 11 To 20  
 [RegionName] = "South"  
 Case 21 To 30  
 [RegionName] = "East"  
 Case Else  
 [RegionName] = "West"  
End Select

You can also **comma delimit** values. For example:

Select Case LNumber  
 Case 1, 2  
 [RegionName] = "North"  
 Case 3, 4, 5  
 [RegionName] = "South"  
 Case 6  
 [RegionName] = "East"  
 Case 7, 11  
 [RegionName] = "West"  
End Select

And finally, you can also use the **Is keyword** to compare values. For example:

Select Case LNumber  
 Case Is < 100  
 [RegionName] = "North"  
 Case Is < 200  
 [RegionName] = "South"  
 Case Is < 300  
 [RegionName] = "East"  
 Case Else  
 [RegionName] = "West"  
End Select

Pasted from <<http://www.techonthenet.com/access/functions/advanced/case.php>>

**MS ACCESS: CBOOL FUNCTION**

Learn how to use the Access **CBool function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CBool function** converts a value to a boolean.

**SYNTAX**

The syntax for the Microsoft Access **CBool function** is:

CBool( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to a boolean.

**APPLIES TO**

The **CBool function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CBOOL function** can be used in VBA code in Microsoft Access. For example:

Dim LCompare as Boolean

LCompare = CBool(1=2)

In this example CBool(1=2) would return false. The variable LCompare would now contain the value false.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/cbool.php>>

**MS ACCESS: CBYTE FUNCTION**

Learn how to use the Access **CByte function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CByte function** converts a value to a byte (ie: number between 0 and 255).

**SYNTAX**

The syntax for the Microsoft Access **CByte function** is:

CByte( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to a byte.

**APPLIES TO**

The **CByte function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CBYTE function** can be used in VBA code in Microsoft Access. For example:

Dim LCompare as Byte

LCompare = CByte(12)

In this example, the variable LCompare would now contain the byte value 12.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/cbyte.php>>

**MS ACCESS: CCUR FUNCTION**

Learn how to use the Access **CCur function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CCur function** converts a value to currency.

**SYNTAX**

The syntax for the Microsoft Access **CCur function** is:

CCur( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to currency.

**APPLIES TO**

The **CCur function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CCur function** can be used in VBA code in Microsoft Access. For example:

Dim LValue As Double  
Dim LCurValue As Currency

LValue = 123.4567812  
LCurValue = CCur(LValue)

In this example, the CCur function would return the value 123.4568. The variable called LCurValue would now contain 123.4568.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/ccur.php>>

**MS ACCESS: CDATE FUNCTION**

Learn how to use the Access **CDate function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CDate function** converts a value to a date.

**SYNTAX**

The syntax for the Microsoft Access **CDate function** is:

CDate( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to a date.

**APPLIES TO**

The **CDate function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CDate function** can be used in VBA code in Microsoft Access. For example:

Dim LstrDate As String  
Dim LDate As Date

LstrDate = "Apr 6, 2003"  
LDate = CDate(LstrDate)

The function called "CDate" will convert any value to a date as long as the expression is a valid date.

In this example, the variable LDate would now contain the value 4/6/2003.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/cdate.php>>

**MS ACCESS: CDBL FUNCTION**

Learn how to use the Access **CDbl function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CDbl function** converts a value to a double.

**SYNTAX**

The syntax for the Microsoft Access **CDbl function** is:

CDbl( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to a double.

**APPLIES TO**

The **CDbl function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CDbl function** can be used in VBA code in Microsoft Access. For example:

Dim LDouble As Double

LDouble = CDbl(8.45 \* 0.005 \* 0.01)

The variable called LDouble would now contain the value of 0.0004225.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/cdbl.php>>

**S ACCESS: CDEC FUNCTION**

Learn how to use the Access **CDec function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CDec function** converts a value to a decimal number.

**SYNTAX**

The syntax for the Microsoft Access **CDec function** is:

CDec( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to a decimal number.

**APPLIES TO**

The **CDec function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CDec function** can be used in VBA code in Microsoft Access. For example:

Dim LDecimal As Double

LDecimal = CDec(8.45 \* 0.005 \* 0.01)

The variable called LDecimal would now contain the value of 0.0004225.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/cdec.php>>

**MS ACCESS: CHDIR STATEMENT**

Learn how to use the Access **ChDir function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **ChDir statement** allows you to change the current directory or folder.

**SYNTAX**

The syntax for the Microsoft Access **ChDir statement** is:

ChDir path

**PARAMETERS OR ARGUMENTS**

*path* is the path that you'd like to set the current directory to.

**NOTE**

The **ChDir statement** lets you change the current directory on the current drive. If you need to change drives, try using the [**ChDrive statement**](http://www.techonthenet.com/access/functions/file/chdrive.php) first.

**APPLIES TO**

The **ChDir statement** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **ChDir "C:\instructions"** | would set the current directory to C:\instructions |
| **ChDir "C:\Documents\Supplies"** | would set the current directory to C:\Documents\Supplies |

**EXAMPLE (IN VBA CODE)**

The **ChDir statement** can be used in VBA code in Microsoft Access. For example:

ChDir "C:\instructions"

In this example, the current directory would now be changed to C:\instructions.

Pasted from <<http://www.techonthenet.com/access/functions/file/chdir.php>>

**MS ACCESS: CHDRIVE STATEMENT**

Learn how to use the Access **ChDrive statement** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **ChDrive statement** allows you to change the current drive.

**SYNTAX**

The syntax for the Microsoft Access **ChDrive statement** is:

ChDrive drive

**PARAMETERS OR ARGUMENTS**

*drive* is the drive that you wish to change to.

**NOTE**

The **ChDrive** lets you change the drive. If you need to change the current directory as well, try using the [**ChDir statement**](http://www.techonthenet.com/access/functions/file/chdir.php).

**APPLIES TO**

The **ChDrive statement** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **ChDrive "S"** | would set the current drive to S |
| **ChDrive "C"** | would set the current drive to C |

**EXAMPLE (IN VBA CODE)**

The **ChDrive statement** can be used in VBA code in Microsoft Access. For example:

ChDrive "S"

ChDir "S:\Games"

In this example, the current drive is set to the S drive. Then the current directory is set to S:\Games.

Pasted from <<http://www.techonthenet.com/access/functions/file/chdrive.php>>

**MS ACCESS: CHOOSE FUNCTION**

Learn how to use the Access **Choose function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Choose function** returns a value from a list of values based on a given position.

**SYNTAX**

The syntax for the Microsoft Access **Choose function** is:

Choose ( position, value1, value2, ... value\_n )

**PARAMETERS OR ARGUMENTS**

*position* is position number in the list of values to return.

*value1*, *value2*, ... *value\_n* is a list of values.

**NOTE**

If *position* is less than 1, the Choose function will return a null value.

If *position* is greater than the number of values, the Choose function will return a null value.

If *position* is a fraction (not an integer value), it will be rounded to the nearest whole number.

**APPLIES TO**

The **Choose function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **Choose(1, "Tech", "on", "the", "Net")** | would return "Tech" |
| **Choose(2, "Tech", "on", "the", "Net")** | would return "on" |
| **Choose(3, "Tech", "on", "the", "Net")** | would return "the" |
| **Choose(4, "Tech", "on", "the", "Net")** | would return "Net" |
| **Choose(5, "Tech", "on", "the", "Net")** | would return NULL |
| **Choose(3.75, "Tech", "on", "the", "Net")** | would return "the" |

**EXAMPLE (IN VBA CODE)**

The **Choose function** can be used in VBA code in Microsoft Access. For example:

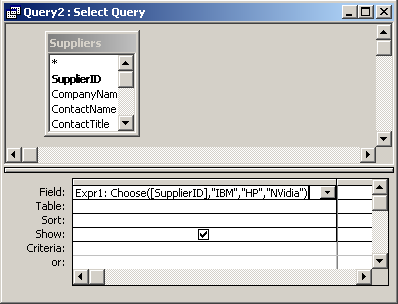
Dim LValue As String

LValue = Choose(1, "Tech", "on", "the", "Net")

In this example, the variable called LValue would contain "Tech" as a value.

**EXAMPLE (IN SQL/QUERIES)**

You can also use the **Choose function** in a query in Microsoft Access.



In this example, if the SupplierID field is 1, then the Choose function will return "IBM".

If the SupplierID field is 2, then the Choose function will return "HP".

If the SupplierID field is 3, then the Choose function will return "Nvidia".

Pasted from <<http://www.techonthenet.com/access/functions/advanced/choose.php>>

**MS ACCESS: CHR FUNCTION**

Learn how to use the Access **Chr function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Chr function** returns the character based on the NUMBER code.

**SYNTAX**

The syntax for the Microsoft Access **Chr function** is:

Chr ( number\_code )

**PARAMETERS OR ARGUMENTS**

*number\_code* is the NUMBER used to retrieve the character.

**APPLIES TO**

The **Chr function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **Chr (87)** | would return "W" |
| **Chr (105)** | would return "i" |

**EXAMPLE (IN VBA CODE)**

The **Chr function** can be used in VBA code in Microsoft Access. For example:

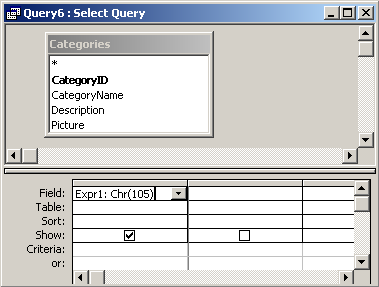
Dim LResult As String

LResult = Chr (87)

In this example, the variable called LResult would now contain the value "W".

**EXAMPLE (IN SQL/QUERIES)**

You can also use the **Chr function** in a query in Microsoft Access.



[**View a listing of the ascii values**](http://www.techonthenet.com/ascii/chart.php).

Pasted from <<http://www.techonthenet.com/access/functions/string/chr.php>>

**MS ACCESS: CINT FUNCTION**

Learn how to use the Access **CInt function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CINT function** converts a value to an integer.

**SYNTAX**

The syntax for the Microsoft Access **CInt function** is:

CInt( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to an integer.

**APPLIES TO**

The **CInt function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

Dim LValue As Integer

LValue = CInt(8.45)

In this example, the variable called LValue would now contain the value of 8.

Be careful using CInt. If you were to use the following code:

Dim LValue As Integer

LValue = CInt(8.5)

The variable LValue would still contain the value of 8. Until the fraction is greater than .5, the CInt function will not round the number up.

**If the fraction is less than or equal to .5, the result will round down.**

**If the fraction is greater than .5, the result will round up.**

We've found it useful in the past to add 0.0001 to the value before applying the CInt function to simulate the regular rounding process.

For example, CInt(8.50001) will result in a value of 9.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/cint.php>>

**MS ACCESS: CLNG FUNCTION**

Learn how to use the Access **CLng function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **CLng function** converts a value to a long integer.

**SYNTAX**

The syntax for the Microsoft Access **CLNG function** is:

CLng( expression )

**PARAMETERS OR ARGUMENTS**

*expression* is the value to convert to a long integer.

**APPLIES TO**

The **CLng function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN VBA CODE)**

The **CLng function** can be used in VBA code in Microsoft Access. For example:

Dim LValue As Long

LValue = CLng(35150.45)

In this example, the variable called LValue would now contain the value of 35150.

Be careful using CLng. If you were to use the following code:

Dim LValue As Long

LValue = CLng(35150.5)

The variable LValue would still contain the value of 35150. Until the fraction is greater than .5, the CLng function will not round the number up.

**If the fraction is less than or equal to .5, the result will round down.**

**If the fraction is greater than .5, the result will round up.**

We've found it useful in the past to add 0.0001 to the value before applying the CLng function to simulate the regular rounding process.

For example, CInt(35150.50001) will result in a value of 35151.

Pasted from <<http://www.techonthenet.com/access/functions/datatype/clng.php>>

**MS ACCESS: CONCATENATE STRINGS TOGETHER**

Learn how to use the Access **& operator** to conatentate strings together with syntax and examples.

**DESCRIPTION**

In Microsoft Access, you can concatenate multiple strings together into a single string.

**SYNTAX**

The syntax to concatenate strings using the **"&"** operator is:

string\_1 & string\_2 & string\_n

**PARAMETERS OR ARGUMENTS**

There are no parameters or arguments for the **& operator**.

**APPLIES TO**

The **Concatenation function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **"new" & "ark"** | would return "newark" |
| **"Tech on the" & " Net!"** | would return "Tech on the Net!" |
| **"This " & "is " & "a test"** | would return "This is a test" |

**EXAMPLE (IN VBA CODE)**

The "&" operator can be used to concatenate strings in VBA code. For example:

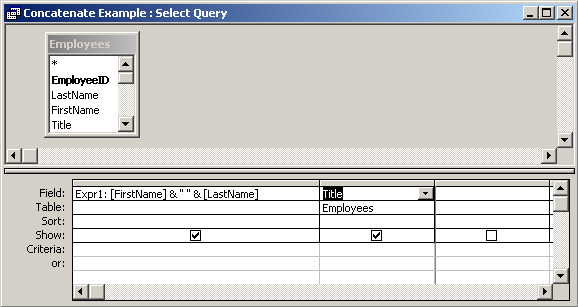
Dim LSQL as string

LSQL = "Select \* from Suppliers"  
LSQL = LSQL & " where Supplier\_ID = 2345"

**EXAMPLE (IN SQL/QUERIES)**

You can use the **"&"** operator in a query to concatenate multiple fields into a single field in your result set.

To do this, open your query in design mode. Enter your field names in the query window separated by the **"&"** symbol.



Pasted from <<http://www.techonthenet.com/access/functions/string/concat.php>>

**MS ACCESS: COS FUNCTION**

Learn how to use the Access **Cos function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Cos function** returns the cosine of an angle.

**SYNTAX**

The syntax for the Microsoft Access **Cos function** is:

Cos ( number )

**PARAMETERS OR ARGUMENTS**

*number* is a numeric expression.

**APPLIES TO**

The **Cos function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE**

|  |  |
| --- | --- |
| **Cos (2)** | would return -0.416146836547142 |
| **Cos (2.51)** | would return -0.807088180396146 |
| **Cos (-3.25)** | would return -0.994129676080546 |

**EXAMPLE (IN VBA CODE)**

The **Cos function** can be used in VBA code in Microsoft Access. For example:

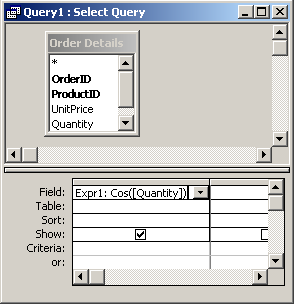
Dim LNumber As Double

LNumber = **Cos** (210)

In this example, the variable called LNumber would now contain the value of -0.883877473182372.

**EXAMPLE (IN SQL/QUERIES)**

You can also use the **Cos function** in a query in Microsoft Access.



Pasted from <<http://www.techonthenet.com/access/functions/numeric/cos.php>>

**MS ACCESS: COUNT FUNCTION**

Learn how to use the Access **Count function** with syntax and examples.

**DESCRIPTION**

The Microsoft Access **Count function** returns the number of records in a select query.

**SYNTAX**

The syntax for the Microsoft Access **Count function** is:

Count( expression )

**PARAMETERS OR ARGUMENTS**

*expression* can be a field or any string expression.

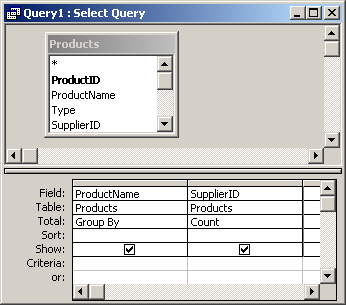
**APPLIES TO**

The **Count function** can be used in the following versions of Microsoft Access:

* + Access 2013, Access 2010, Access 2007, Access 2003, Access XP, Access 2000

**EXAMPLE (IN SQL/QUERIES)**

You can use the **Count function** in a query by clicking on the Totals button in the toolbar (This is the button with the summation symbol). The **Count function** is used in conjunction with the **Group By** clause.



This query would return the number of records for each ProductName. You can actually substitute any field with the Count function and it will return the same count result. In this case, we've used the SupplierID field to determine the number of records for each ProductName.

Pasted from <<http://www.techonthenet.com/access/functions/numeric/count.php>>

* + [Home](http://www.techonthenet.com/index.php) >

* + [MSAccess](http://www.techonthenet.com/access/index.php) >

* 1. Functions

**MS ACCESS FUNCTION NAVIGATION**

* + [**Functions - Alphabetical**](http://www.techonthenet.com/access/functions/index_alpha.php)
  + [**Functions - Category**](http://www.techonthenet.com/access/functions/index.php)

**MS ACCESS: FUNCTIONS - ALPHABETICAL**

Below is a list of the most commonly used functions in Microsoft Access. These functions can generally be used in either VBA code or SQL/Queries.

**ALPHABETICAL LISTING OF MSACCESS FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**Abs**](http://www.techonthenet.com/access/functions/numeric/abs.php)     - [**And**](http://www.techonthenet.com/access/functions/advanced/and.php)     - [**Asc**](http://www.techonthenet.com/access/functions/string/asc.php)     - [**Atn**](http://www.techonthenet.com/access/functions/numeric/atn.php)     - [**Avg**](http://www.techonthenet.com/access/functions/numeric/avg.php)     - [**Case**](http://www.techonthenet.com/access/functions/advanced/case.php)     - [**CBool**](http://www.techonthenet.com/access/functions/datatype/cbool.php)     - [**CByte**](http://www.techonthenet.com/access/functions/datatype/cbyte.php)     - [**CCur**](http://www.techonthenet.com/access/functions/datatype/ccur.php)     - [**CDate**](http://www.techonthenet.com/access/functions/datatype/cdate.php)     - [**CDbl**](http://www.techonthenet.com/access/functions/datatype/cdbl.php)     - [**CDec**](http://www.techonthenet.com/access/functions/datatype/cdec.php)     - [**ChDir**](http://www.techonthenet.com/access/functions/file/chdir.php)     - [**ChDrive**](http://www.techonthenet.com/access/functions/file/chdrive.php)     - [**Choose**](http://www.techonthenet.com/access/functions/advanced/choose.php)     - [**Chr**](http://www.techonthenet.com/access/functions/string/chr.php)     - [**CInt**](http://www.techonthenet.com/access/functions/datatype/cint.php)     - [**CLng**](http://www.techonthenet.com/access/functions/datatype/clng.php)     - [**Concat with &**](http://www.techonthenet.com/access/functions/string/concat.php)     - [**Cos**](http://www.techonthenet.com/access/functions/numeric/cos.php)     - [**Count**](http://www.techonthenet.com/access/functions/numeric/count.php)     - [**CSng**](http://www.techonthenet.com/access/functions/datatype/csng.php)     - [**CStr**](http://www.techonthenet.com/access/functions/datatype/cstr.php)     - [**CurDir**](http://www.techonthenet.com/access/functions/string/curdir.php)     - [**CurrentUser**](http://www.techonthenet.com/access/functions/advanced/currentuser.php)     - [**CVar**](http://www.techonthenet.com/access/functions/datatype/cvar.php)     - [**Date**](http://www.techonthenet.com/access/functions/date/date.php)     - [**DateAdd**](http://www.techonthenet.com/access/functions/date/dateadd.php)     - [**DateDiff**](http://www.techonthenet.com/access/functions/date/datediff.php)     - [**DatePart**](http://www.techonthenet.com/access/functions/date/datepart.php)     - [**DateSerial**](http://www.techonthenet.com/access/functions/date/dateserial.php)     - [**DateValue**](http://www.techonthenet.com/access/functions/date/datevalue.php)     - [**DAvg**](http://www.techonthenet.com/access/functions/domain/davg.php) | * + - [**Day**](http://www.techonthenet.com/access/functions/date/day.php)     - [**DCount**](http://www.techonthenet.com/access/functions/domain/dcount.php)     - [**DFirst**](http://www.techonthenet.com/access/functions/domain/dfirst.php)     - [**Dir**](http://www.techonthenet.com/access/functions/file/dir.php)     - [**DLast**](http://www.techonthenet.com/access/functions/domain/dlast.php)     - [**DLookup**](http://www.techonthenet.com/access/functions/domain/dlookup.php)     - [**DMax**](http://www.techonthenet.com/access/functions/domain/dmax.php)     - [**DMin**](http://www.techonthenet.com/access/functions/domain/dmin.php)     - [**DSum**](http://www.techonthenet.com/access/functions/domain/dsum.php)     - [**Exp**](http://www.techonthenet.com/access/functions/numeric/exp.php)     - [**FileDateTime**](http://www.techonthenet.com/access/functions/file/filedatetime.php)     - [**FileLen**](http://www.techonthenet.com/access/functions/file/filelen.php)     - [**First**](http://www.techonthenet.com/access/functions/advanced/first.php)     - [**Fix**](http://www.techonthenet.com/access/functions/numeric/fix.php)     - [**Format (with Dates)**](http://www.techonthenet.com/access/functions/date/format.php)     - [**Format (with Numbers)**](http://www.techonthenet.com/access/functions/numeric/format.php)     - [**Format (with Strings)**](http://www.techonthenet.com/access/functions/string/format.php)     - [**GetAttr**](http://www.techonthenet.com/access/functions/file/getattr.php)     - [**Hour**](http://www.techonthenet.com/access/functions/date/hour.php)     - [**IF-THEN-ELSE**](http://www.techonthenet.com/access/functions/advanced/if_then.php)     - [**iif**](http://www.techonthenet.com/access/functions/advanced/iif.php)     - [**Instr**](http://www.techonthenet.com/access/functions/string/instr.php)     - [**InstrRev**](http://www.techonthenet.com/access/functions/string/instrrev.php)     - [**Int**](http://www.techonthenet.com/access/functions/numeric/int.php)     - [**IsDate**](http://www.techonthenet.com/access/functions/advanced/isdate.php)     - [**IsNull**](http://www.techonthenet.com/access/functions/advanced/isnull.php)     - [**IsNumeric**](http://www.techonthenet.com/access/functions/advanced/isnumeric.php)     - [**Last**](http://www.techonthenet.com/access/functions/advanced/last.php)     - [**LCase**](http://www.techonthenet.com/access/functions/string/lcase.php)     - [**Left**](http://www.techonthenet.com/access/functions/string/left.php)     - [**Len**](http://www.techonthenet.com/access/functions/string/len.php)     - [**LTrim**](http://www.techonthenet.com/access/functions/string/ltrim.php)     - [**Max**](http://www.techonthenet.com/access/functions/numeric/max.php) | * + - [**Mid**](http://www.techonthenet.com/access/functions/string/mid.php)     - [**Min**](http://www.techonthenet.com/access/functions/numeric/min.php)     - [**Minute**](http://www.techonthenet.com/access/functions/date/minute.php)     - [**MkDir**](http://www.techonthenet.com/access/functions/file/mkdir.php)     - [**Month**](http://www.techonthenet.com/access/functions/date/month.php)     - [**MonthName**](http://www.techonthenet.com/access/functions/date/monthname.php)     - [**Now**](http://www.techonthenet.com/access/functions/date/now.php)     - [**Nz**](http://www.techonthenet.com/access/functions/advanced/nz.php)     - [**Or**](http://www.techonthenet.com/access/functions/advanced/or.php)     - [**Replace**](http://www.techonthenet.com/access/functions/string/replace.php)     - [**Right**](http://www.techonthenet.com/access/functions/string/right.php)     - [**Rnd (random number)**](http://www.techonthenet.com/access/functions/numeric/rnd.php)     - [**Round**](http://www.techonthenet.com/access/functions/numeric/round.php)     - [**RTrim**](http://www.techonthenet.com/access/functions/string/rtrim.php)     - [**Second**](http://www.techonthenet.com/access/functions/date/second.php)     - [**SetAttr**](http://www.techonthenet.com/access/functions/file/setattr.php)     - [**Sgn**](http://www.techonthenet.com/access/functions/numeric/sgn.php)     - [**Space**](http://www.techonthenet.com/access/functions/string/space.php)     - [**Str**](http://www.techonthenet.com/access/functions/string/str.php)     - [**StrConv**](http://www.techonthenet.com/access/functions/string/strconv.php)     - [**Sum**](http://www.techonthenet.com/access/functions/numeric/sum.php)     - [**Switch**](http://www.techonthenet.com/access/functions/advanced/switch.php)     - [**Time**](http://www.techonthenet.com/access/functions/date/time.php)     - [**TimeSerial**](http://www.techonthenet.com/access/functions/date/timeserial.php)     - [**TimeValue**](http://www.techonthenet.com/access/functions/date/timevalue.php)     - [**Trim**](http://www.techonthenet.com/access/functions/string/trim.php)     - [**UCase**](http://www.techonthenet.com/access/functions/string/ucase.php)     - [**Val**](http://www.techonthenet.com/access/functions/numeric/val.php)     - [**Weekday**](http://www.techonthenet.com/access/functions/date/weekday.php)     - [**WeekdayName**](http://www.techonthenet.com/access/functions/date/weekdayname.php)     - [**Year**](http://www.techonthenet.com/access/functions/date/year.php) |

**MISCELLANEOUS FUNCTIONS**

* + [**Test a string for an alphanumeric value**](http://www.techonthenet.com/access/functions/misc/alphanumeric.php)
  + [**Convert currency into words**](http://www.techonthenet.com/access/functions/misc/curr_to_words.php)
  + [**Creating a custom round function**](http://www.techonthenet.com/access/functions/misc/customround.php)
  + [**Retrieve the user name from Windows while in an Access database**](http://www.techonthenet.com/access/modules/username.php)

Pasted from <<http://www.techonthenet.com/access/functions/index_alpha.php>>

* + [Home](http://www.techonthenet.com/index.php) >

* 1. [MSAccess](http://www.techonthenet.com/access/index.php) >

* 1. Functions

**MS ACCESS FUNCTION NAVIGATION**

* + [**Functions - Alphabetical**](http://www.techonthenet.com/access/functions/index_alpha.php)
  + [**Functions - Category**](http://www.techonthenet.com/access/functions/index.php)

**MS ACCESS: FUNCTIONS - CATEGORY**

Below is a list of the most commonly used functions in Access. These functions can generally be used in either VBA code or SQL/Queries.

**STRING FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**Asc**](http://www.techonthenet.com/access/functions/string/asc.php)     - [**Chr**](http://www.techonthenet.com/access/functions/string/chr.php)     - [**Concat with &**](http://www.techonthenet.com/access/functions/string/concat.php)     - [**CurDir**](http://www.techonthenet.com/access/functions/string/curdir.php)     - [**Format (with Strings)**](http://www.techonthenet.com/access/functions/string/format.php)     - [**Instr**](http://www.techonthenet.com/access/functions/string/instr.php)     - [**InstrRev**](http://www.techonthenet.com/access/functions/string/instrrev.php) | * + - [**LCase**](http://www.techonthenet.com/access/functions/string/lcase.php)     - [**Left**](http://www.techonthenet.com/access/functions/string/left.php)     - [**Len**](http://www.techonthenet.com/access/functions/string/len.php)     - [**LTrim**](http://www.techonthenet.com/access/functions/string/ltrim.php)     - [**Mid**](http://www.techonthenet.com/access/functions/string/mid.php)     - [**Replace**](http://www.techonthenet.com/access/functions/string/replace.php)     - [**Right**](http://www.techonthenet.com/access/functions/string/right.php) | * + - [**RTrim**](http://www.techonthenet.com/access/functions/string/rtrim.php)     - [**Space**](http://www.techonthenet.com/access/functions/string/space.php)     - [**Str**](http://www.techonthenet.com/access/functions/string/str.php)     - [**StrConv**](http://www.techonthenet.com/access/functions/string/strconv.php)     - [**Trim**](http://www.techonthenet.com/access/functions/string/trim.php)     - [**UCase**](http://www.techonthenet.com/access/functions/string/ucase.php) |

**NUMERIC / MATHEMATICAL FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**Abs**](http://www.techonthenet.com/access/functions/numeric/abs.php)     - [**Atn**](http://www.techonthenet.com/access/functions/numeric/atn.php)     - [**Avg**](http://www.techonthenet.com/access/functions/numeric/avg.php)     - [**Cos**](http://www.techonthenet.com/access/functions/numeric/cos.php)     - [**Count**](http://www.techonthenet.com/access/functions/numeric/count.php)     - [**Exp**](http://www.techonthenet.com/access/functions/numeric/exp.php) | * + - [**Fix**](http://www.techonthenet.com/access/functions/numeric/fix.php)     - [**Format (with Numbers)**](http://www.techonthenet.com/access/functions/numeric/format.php)     - [**Int**](http://www.techonthenet.com/access/functions/numeric/int.php)     - [**Max**](http://www.techonthenet.com/access/functions/numeric/max.php)     - [**Min**](http://www.techonthenet.com/access/functions/numeric/min.php)     - [**Rnd (random number)**](http://www.techonthenet.com/access/functions/numeric/rnd.php) | * + - [**Round**](http://www.techonthenet.com/access/functions/numeric/round.php)     - [**Sgn**](http://www.techonthenet.com/access/functions/numeric/sgn.php)     - [**Sum**](http://www.techonthenet.com/access/functions/numeric/sum.php)     - [**Val**](http://www.techonthenet.com/access/functions/numeric/val.php) |

**DATE / TIME FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**Date**](http://www.techonthenet.com/access/functions/date/date.php)     - [**DateAdd**](http://www.techonthenet.com/access/functions/date/dateadd.php)     - [**DateDiff**](http://www.techonthenet.com/access/functions/date/datediff.php)     - [**DatePart**](http://www.techonthenet.com/access/functions/date/datepart.php)     - [**DateSerial**](http://www.techonthenet.com/access/functions/date/dateserial.php)     - [**DateValue**](http://www.techonthenet.com/access/functions/date/datevalue.php)     - [**Day**](http://www.techonthenet.com/access/functions/date/day.php) | * + - [**Format (with Dates)**](http://www.techonthenet.com/access/functions/date/format.php)     - [**Hour**](http://www.techonthenet.com/access/functions/date/hour.php)     - [**Minute**](http://www.techonthenet.com/access/functions/date/minute.php)     - [**Month**](http://www.techonthenet.com/access/functions/date/month.php)     - [**MonthName**](http://www.techonthenet.com/access/functions/date/monthname.php)     - [**Now**](http://www.techonthenet.com/access/functions/date/now.php)     - [**Second**](http://www.techonthenet.com/access/functions/date/second.php) | * + - [**Time**](http://www.techonthenet.com/access/functions/date/time.php)     - [**TimeSerial**](http://www.techonthenet.com/access/functions/date/timeserial.php)     - [**TimeValue**](http://www.techonthenet.com/access/functions/date/timevalue.php)     - [**Weekday**](http://www.techonthenet.com/access/functions/date/weekday.php)     - [**WeekdayName**](http://www.techonthenet.com/access/functions/date/weekdayname.php)     - [**Year**](http://www.techonthenet.com/access/functions/date/year.php) |

**ADVANCED / LOGICAL FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**And**](http://www.techonthenet.com/access/functions/advanced/and.php)     - [**Case**](http://www.techonthenet.com/access/functions/advanced/case.php)     - [**Choose**](http://www.techonthenet.com/access/functions/advanced/choose.php)     - [**CurrentUser**](http://www.techonthenet.com/access/functions/advanced/currentuser.php)     - [**First**](http://www.techonthenet.com/access/functions/advanced/first.php) | * + - [**IF-THEN-ELSE**](http://www.techonthenet.com/access/functions/advanced/if_then.php)     - [**iif**](http://www.techonthenet.com/access/functions/advanced/iif.php)     - [**IsDate**](http://www.techonthenet.com/access/functions/advanced/isdate.php)     - [**IsNull**](http://www.techonthenet.com/access/functions/advanced/isnull.php)     - [**IsNumeric**](http://www.techonthenet.com/access/functions/advanced/isnumeric.php) | * + - [**Last**](http://www.techonthenet.com/access/functions/advanced/last.php)     - [**Nz**](http://www.techonthenet.com/access/functions/advanced/nz.php)     - [**Or**](http://www.techonthenet.com/access/functions/advanced/or.php)     - [**Switch**](http://www.techonthenet.com/access/functions/advanced/switch.php) |

**DOMAIN AGGREGATE FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**DAvg**](http://www.techonthenet.com/access/functions/domain/davg.php)     - [**DCount**](http://www.techonthenet.com/access/functions/domain/dcount.php)     - [**DFirst**](http://www.techonthenet.com/access/functions/domain/dfirst.php) | * + - [**DLast**](http://www.techonthenet.com/access/functions/domain/dlast.php)     - [**DLookup**](http://www.techonthenet.com/access/functions/domain/dlookup.php)     - [**DMax**](http://www.techonthenet.com/access/functions/domain/dmax.php) | * + - [**DMin**](http://www.techonthenet.com/access/functions/domain/dmin.php)     - [**DSum**](http://www.techonthenet.com/access/functions/domain/dsum.php) |

**DATA TYPE CONVERSION FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**CBool**](http://www.techonthenet.com/access/functions/datatype/cbool.php)     - [**CByte**](http://www.techonthenet.com/access/functions/datatype/cbyte.php)     - [**CCur**](http://www.techonthenet.com/access/functions/datatype/ccur.php)     - [**CDate**](http://www.techonthenet.com/access/functions/datatype/cdate.php) | * + - [**CDbl**](http://www.techonthenet.com/access/functions/datatype/cdbl.php)     - [**CDec**](http://www.techonthenet.com/access/functions/datatype/cdec.php)     - [**CInt**](http://www.techonthenet.com/access/functions/datatype/cint.php)     - [**CLng**](http://www.techonthenet.com/access/functions/datatype/clng.php) | * + - [**CSng**](http://www.techonthenet.com/access/functions/datatype/csng.php)     - [**CStr**](http://www.techonthenet.com/access/functions/datatype/cstr.php)     - [**CVar**](http://www.techonthenet.com/access/functions/datatype/cvar.php) |

**GROUP BY FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**Avg**](http://www.techonthenet.com/access/functions/numeric/avg.php)     - [**Count**](http://www.techonthenet.com/access/functions/numeric/count.php) | * + - [**Max**](http://www.techonthenet.com/access/functions/numeric/max.php)     - [**Min**](http://www.techonthenet.com/access/functions/numeric/min.php) | * + - [**Sum**](http://www.techonthenet.com/access/functions/numeric/sum.php) |

**FILE / DIRECTORY FUNCTIONS**

|  |  |  |
| --- | --- | --- |
| * + - [**ChDir**](http://www.techonthenet.com/access/functions/file/chdir.php)     - [**ChDrive**](http://www.techonthenet.com/access/functions/file/chdrive.php)     - [**CurDir**](http://www.techonthenet.com/access/functions/string/curdir.php) | * + - [**Dir**](http://www.techonthenet.com/access/functions/file/dir.php)     - [**FileDateTime**](http://www.techonthenet.com/access/functions/file/filedatetime.php)     - [**FileLen**](http://www.techonthenet.com/access/functions/file/filelen.php) | * + - [**GetAttr**](http://www.techonthenet.com/access/functions/file/getattr.php)     - [**MkDir**](http://www.techonthenet.com/access/functions/file/mkdir.php)     - [**SetAttr**](http://www.techonthenet.com/access/functions/file/setattr.php) |

**MISCELLANEOUS FUNCTIONS**

* + [**Test a string for an alphanumeric value**](http://www.techonthenet.com/access/functions/misc/alphanumeric.php)
  + [**Convert currency into words**](http://www.techonthenet.com/access/functions/misc/curr_to_words.php)
  + [**Creating a custom round function**](http://www.techonthenet.com/access/functions/misc/customround.php)
  + [**Retrieve the user name from Windows while in an Access database**](http://www.techonthenet.com/access/modules/username.php)

Pasted from <<http://www.techonthenet.com/access/functions/index.php>>

Python Overview:

Python is a high-level, interpreted, interactive and object oriented-scripting language.

* + **Python is Interpreted**
  + **Python is Interactive**
  + **Python is Object-Oriented**
  + **Python is Beginner's Language**

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python's feature highlights include:

* + **Easy-to-learn**
  + **Easy-to-read**
  + **Easy-to-maintain**
  + **A broad standard library**
  + **Interactive Mode**
  + **Portable**
  + **Extendable**
  + **Databases**
  + **GUI Programming**
  + **Scalable**

Getting Python:

The most up-to-date and current source code, binaries, documentation, news, etc. is available at the official website of Python:

**Python Official Website :** <http://www.python.org/>

You can download the Python documentation from the following site. The documentation is available in HTML, PDF, and PostScript formats.

**Python Documentation Website :** [www.python.org/doc/](http://www.python.org/doc/)

First Python Program:

Interactive Mode Programming:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

root# python  
Python 2.5 (r25:51908, Nov 6 2007, 16:54:01)  
[GCC 4.1.2 20070925 (Red Hat 4.1.2-27)] on linux2  
Type "help", "copyright", "credits" or "license" for more info.  
>>>

Type the following text to the right of the Python prompt and press the Enter key:

>>> print "Hello, Python!";

This will produce following result:

Hello, Python!

Python Identifiers:

A Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores, and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus **Manpower** and **manpower** are two different identifiers in Python.

Here are following identifier naming convention for Python:

* + Class names start with an uppercase letter and all other identifiers with a lowercase letter.
  + Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.
  + Starting an identifier with two leading underscores indicates a strongly private identifier.
  + If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words:

The following list shows the reserved words in Python. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |

Lines and Indentation:

One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

if True:  
 print "True"  
else:  
 print "False"

However, the second block in this example will generate an error:

if True:  
 print "Answer"  
 print "True"  
else:  
 print "Answer"  
 print "False"

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

total = item\_one + \  
 item\_two + \  
 item\_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example:

days = ['Monday', 'Tuesday', 'Wednesday',  
 'Thursday', 'Friday']

Quotation in Python:

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

word = 'word'  
sentence = "This is a sentence."  
paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""

Comments in Python:

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them.

#!/usr/bin/python

# First comment  
print "Hello, Python!"; # second comment

This will produce following result:

Hello, Python!

A comment may be on the same line after a statement or expression:

name = "Madisetti" # This is again comment

You can comment multiple lines as follows:

# This is a comment.  
# This is a comment, too.  
# This is a comment, too.  
# I said that already.

Using Blank Lines:

A line containing only whitespace, possibly with a comment, is known as a blank line, and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Multiple Statements on a Single Line:

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

import sys; x = 'foo'; sys.stdout.write(x + '\n')

Multiple Statement Groups as Suites:

Groups of individual statements making up a single code block are called **suites** in Python.

Compound or complex statements, such as if, while, def, and class, are those which require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite.

Example:

if expression :   
 suite  
elif expression :   
 suite   
else :   
 suite

Python - Variable Types:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

Assigning Values to Variables:

The operand to the left of the = operator is the name of the variable, and the operand to the right of the = operator is the value stored in the variable. For example:

counter = 100 # An integer assignment  
miles = 1000.0 # A floating point  
name = "John" # A string

print counter  
print miles  
print name

Standard Data Types:

Python has five standard data types:

* + Numbers
  + String
  + List
  + Tuple
  + Dictionary

Python Numbers:

Number objects are created when you assign a value to them. For example:

var1 = 1  
var2 = 10

Python supports four different numerical types:

* + int (signed integers)
  + long (long integers [can also be represented in octal and hexadecimal])
  + float (floating point real values)
  + complex (complex numbers)

Here are some examples of numbers:

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEl | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

Python Strings:

Strings in Python are identified as a contiguous set of characters in between quotation marks.

Example:

str = 'Hello World!'

print str # Prints complete string  
print str[0] # Prints first character of the string  
print str[2:5] # Prints characters starting from 3rd to 6th  
print str[2:] # Prints string starting from 3rd character  
print str \* 2 # Prints string two times  
print str + "TEST" # Prints concatenated string

Python Lists:

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]).

#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
tinylist = [123, 'john']

print list # Prints complete list  
print list[0] # Prints first element of the list  
print list[1:3] # Prints elements starting from 2nd to 4th  
print list[2:] # Prints elements starting from 3rd element  
print tinylist \* 2 # Prints list two times  
print list + tinylist # Prints concatenated lists

Python Tuples:

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

Tuples can be thought of as **read-only** lists.

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )  
tinytuple = (123, 'john')

print tuple # Prints complete list  
print tuple[0] # Prints first element of the list  
print tuple[1:3] # Prints elements starting from 2nd to 4th  
print tuple[2:] # Prints elements starting from 3rd element  
print tinytuple \* 2 # Prints list two times  
print tuple + tinytuple # Prints concatenated lists

Python Dictionary:

Python 's dictionaries are hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs.

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}  
print dict['one'] # Prints value for 'one' key  
print dict[2] # Prints value for 2 key  
print tinydict # Prints complete dictionary  
print tinydict.keys() # Prints all the keys  
print tinydict.values() # Prints all the values

Python - Basic Operators:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | a + b will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | a - b will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | a \* b will give 200 |
| / | Division - Divides left hand operand by right hand operand | b / a will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | b % a will give 0 |
| \*\* | Exponent - Performs exponential (power) calculation on operators | a\*\*b will give 10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | 9//2 is equal to 4 and 9.0//2.0 is equal to 4.0 |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. | (a == b) is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. This is similar to != operator. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | c = a + b will assigne value of a + b into c |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / a |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | c %= a is equivalent to c = c % a |
| \*\*= | Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c = c \*\* a |
| //= | Floor Dividion and assigns a value, Performs floor division on operators and assign value to the left operand | c //= a is equivalent to c = c // a |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (a & b) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in eather operand. | (a | b) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (a ^ b) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the efect of 'flipping' bits. | (~a ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | a << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 will give 15 which is 0000 1111 |
| and | Called Logical AND operator. If both the operands are true then then condition becomes true. | (a and b) is true. |
| or | Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true. | (a or b) is true. |
| not | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | not(a && b) is false. |
| in | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here **in** results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here **not in** results in a 1 if x is not a member of sequence y. |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here **is** results in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here **is not** results in 1 if id(x) is not equal to id(y). |

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Ccomplement, unary plus and minus (method names for the last two are +@ and -@) |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND' |
| ^ | | Bitwise exclusive `OR' and regular `OR' |
| <= < > >= | Comparison operators |
| <> == != | Equality operators |
| = %= /= //= -= += |= &= >>= <<= \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| note or and | Logical operators |

The *if* statement:

The syntax of the if statement is:

if expression:  
 statement(s)

The *else* Statement:

The syntax of the *if...else* statement is:

if expression:  
 statement(s)  
else:  
 statement(s)

The *elif* Statement

The syntax of the *if...elif* statement is:

if expression1:  
 statement(s)  
elif expression2:  
 statement(s)  
elif expression3:  
 statement(s)  
else:  
 statement(s)

The Nested *if...elif...else* Construct

The syntax of the nested *if...elif...else* construct may be:

if expression1:  
 statement(s)  
 **if expression2:  
 statement(s)  
 elif expression3:  
 statement(s)  
 else  
 statement(s)**  
elif expression4:  
 statement(s)  
else:  
 statement(s)

The *while* Loop:

The syntax of the while look is:

while expression:  
 statement(s)

The Infinite Loops:

You must use caution when using while loops because of the possibility that this condition never resolves to a false value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

Single Statement Suites:

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is an example of a one-line while clause:

while expression : statement

The *for* Loop:

The syntax of the loop look is:

for iterating\_var in sequence:  
 statements(s)

Iterating by Sequence Index:

An alternative way of iterating through each item is by index offset into the sequence itself:

fruits = ['banana', 'apple', 'mango']  
for index in range(len(fruits)):  
 print 'Current fruit :', fruits[index]

print "Good bye!"

The *break* Statement:

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

for letter in **'Python'**: # First Example  
 if letter == 'h':  
 **break**  
 print 'Current Letter :', letter  
   
var = 10 # Second Example  
while var > 0:   
 print 'Current variable value :', var  
 var = var -1  
 if var == 5:  
 **break**

print "Good bye!"

The *continue* Statement:

The **continue** statement in Python returns the control to the beginning of the while loop. The **continue**statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both *while* and *for* loops.

for letter in **'Python'**: # First Example  
 if letter == 'h':  
 **continue**  
 print 'Current Letter :', letter

var = 10 # Second Example  
while var > 0:   
 print 'Current variable value :', var  
 var = var -1  
 if var == 5:  
 **continue**

print "Good bye!"

The *else* Statement Used with Loops

Python supports to have an **else** statement associated with a loop statements.

* + If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
  + If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The *pass* Statement:

The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

#!/usr/bin/python

for letter in **'Python'**:   
 if letter == 'h':  
 **pass**  
 print 'This is pass block'  
 print 'Current Letter :', letter

print "Good bye!"

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

* + Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
  + Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
  + The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
  + The code block within every function starts with a colon (:) and is indented.
  + The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

def functionname( parameters ):  
 "function\_docstring"  
 function\_suite  
 return [expression]

By default, parameters have a positional behavior, and you need to inform them in the same order that they were defined.

Example:

Here is the simplest form of a Python function. This function takes a string as input parameter and prints it on standard screen.

def printme( str ):  
 "This prints a passed string into this function"  
 print str  
 return

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function, and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

Following is the example to call printme() function:

#!/usr/bin/python

# Function definition is here  
def printme( str ):  
 "This prints a passed string into this function"  
 print str;  
 return;

# Now you can call printme function  
printme("I'm first call to user defined function!");  
printme("Again second call to the same function");

This would produce following result:

I'm first call to user defined function!  
Again second call to the same function

Python - Modules:

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.

A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes, and variables. A module can also include runnable code.

Example:

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, hello.py

def print\_func( par ):  
 print "Hello : ", par  
 return

The *import* Statement:

You can use any Python source file as a module by executing an import statement in some other Python source file. *import* has the following syntax:

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.

Example:

To import the module hello.py, you need to put the following command at the top of the script:

#!/usr/bin/python

# Import module hello  
import hello

# Now you can call defined function that module as follows  
hello.print\_func("Zara")

This would produce following result:

Hello : Zara

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

Opening and Closing Files:

The *open* Function:

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a file object which would be utilized to call other support methods associated with it.

Syntax:

file object = open(file\_name [, access\_mode][, buffering])

Here is paramters detail:

* + **file\_name:** The file\_name argument is a string value that contains the name of the file that you want to access.
  + **access\_mode:** The access\_mode determines the mode in which the file has to be opened ie. read, write append etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r)
  + **buffering:** If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. This is optional paramter.

Here is a list of the different modes of opening a file:

|  |  |
| --- | --- |
| **Modes** | **Description** |
| r | Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| rb | Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| r+ | Opens a file for both reading and writing. The file pointer will be at the beginning of the file. |
| rb+ | Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file. |
| w | Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| wb | Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| w+ | Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| wb+ | Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| a | Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| ab | Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| a+ | Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| ab+ | Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

The *file* object atrributes:

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| file.closed | Returns true if file is closed, false otherwise. |
| file.mode | Returns access mode with which file was opened. |
| file.name | Returns name of the file. |
| file.softspace | Returns false if space explicitly required with print, true otherwise. |

The *close()* Method:

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

fileObject.close();

Reading and Writing Files:

The *write()* Method:

Syntax:

fileObject.write(string);

The *read()* Method:

Syntax:

fileObject.read([count]);

File Positions:

The *tell()* method tells you the current position within the file in other words, the next read or write will occur at that many bytes from the beginning of the file:

The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Renaming and Deleting Files:

Syntax:

os.rename(current\_file\_name, new\_file\_name)

The *remove()* Method:

Syntax:

os.remove(file\_name)

Directories in Python:

The *mkdir()* Method:

You can use the *mkdir()* method of the os module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Syntax:

os.mkdir("newdir")

The *chdir()* Method:

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax:

os.chdir("newdir")

The *getcwd()* Method:

The *getcwd()* method displays the current working directory.

Syntax:

os.getcwd()

The *rmdir()* Method:

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax:

os.rmdir('dirname')

Handling an exception:

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:

Here is simple syntax of *try....except...else* blocks:

try:  
 Do you operations here;  
 ......................  
except *ExceptionI*:  
 If there is ExceptionI, then execute this block.  
except *ExceptionII*:  
 If there is ExceptionII, then execute this block.  
 ......................  
else:  
 If there is no exception then execute this block.

Here are few important points about the above mentioned syntax:

* + A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
  + You can also provide a generic except clause, which handles any exception.
  + After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
  + The else-block is a good place for code that does not need the try: block's protection.

The *except* clause with no exceptions:

You can also use the except statement with no exceptions defined as follows:

try:  
 Do you operations here;  
 ......................  
except:  
 If there is any exception, then execute this block.  
 ......................  
else:  
 If there is no exception then execute this block.

The *except* clause with multiple exceptions:

You can also use the same *except* statement to handle multiple exceptions as follows:

try:  
 Do you operations here;  
 ......................  
except(Exception1[, Exception2[,...ExceptionN]]]):  
 If there is any exception from the given exception list,   
 then execute this block.  
 ......................  
else:  
 If there is no exception then execute this block.

Standard Exceptions:

Here is a list standard Exceptions available in Python: [Standard Exceptions](http://www.tutorialspoint.com/python/standard_exceptions.htm)

The try-finally clause:

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

try:  
 Do you operations here;  
 ......................  
 Due to any exception, this may be skipped.  
finally:  
 This would always be executed.  
 ......................

Argument of an Exception:

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows:

try:  
 Do you operations here;  
 ......................  
except *ExceptionType, Argument*:  
 You can print value of Argument here...

Raising an exceptions:

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise**statement.

Syntax:

raise [Exception [, args [, traceback]]]

User-Defined Exceptions:

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here a class is created that is subclassed from*RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

class Networkerror(RuntimeError):  
 def \_\_init\_\_(self, arg):  
 self.args = arg

So once you defined above class, you can raise your exception as follows:

try:  
 raise Networkerror("Bad hostname")  
except Networkerror,e:  
 print e.args

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 access via *ClassName.\_\_doc\_\_*.
  + The *class\_suite* consists of all the component statements, defining class members, data attributes, and functions.

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

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.

Destroying Objects (Garbage Collection):

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

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

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

Class Inheritance:

Instead of starting from scratch, you can create a class by deriving it from a preexisting 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 are given after the class name:

class SubClassName (ParentClass1[, ParentClass2, ...]):  
 'Optional class documentation string'  
 class\_suite

Overriding Methods:

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

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

Base Overloading Methods:

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

|  |  |
| --- | --- |
| **SN** | **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 : *dell 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've 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:

#!/usr/bin/python

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

Data Hiding:

An object's attributes may or may not be visible outside the class definition. For these cases, you can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders:

#!/usr/bin/python

class JustCounter:  
 \_\_secretCount = 0  
   
 def count(self):  
 self.\_\_secretCount += 1  
 print self.\_\_secretCount

counter = JustCounter()  
counter.count()  
counter.count()  
print counter.\_\_secretCount

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. But 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'**.

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:

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| pattern | This is the regular expression to be matched. |
| string | This is the string which would be searched to match the pattern |
| flags | You can specifiy different flags using exclusive 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 would use *group(num)*or *groups()* function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **Match Object Methods** | **Description** |
| group(num=0) | This methods returns entire match (or specific subgroup num) |
| groups() | This method return all matching subgroups in a tuple (empty if there weren't any) |

The *search* Function

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

Here is the syntax for this function:

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

Here is the description of the parameters:

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| pattern | This is the regular expression to be matched. |
| string | This is the string which would be searched to match the pattern |
| flags | You can specifiy different flags using exclusive 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 would use *group(num)*or *groups()* function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **Match Object Methods** | **Description** |
| group(num=0) | This methods returns entire match (or specific subgroup num) |
| groups() | This method return all matching subgroups in a tuple (empty if there weren't any) |

Matching vs 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).

Search and Replace:

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

Syntax:

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

This method replace all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method would return modified string.

Regular-expression Modifiers - Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifier are specified as an optional flag. You can provide multiple modified using exclusive OR (|), as shown previously and may be represented by one of these:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| re.I | Performs case-insensitive matching. |
| 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). |
| 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). |
| re.S | Makes a period (dot) match any character, including a newline. |
| re.U | Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B. |
| 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 control characters, **(+ ? . \* ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python.

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| ^ | Matches beginning of line. |
| $ | Matches end of line. |
| . | Matches any single character except newline. Using m option allows it to match newline as well. |
| [...] | Matches any single character in brackets. |
| [^...] | Matches any single character not in brackets |
| re\* | Matches 0 or more occurrences of preceding expression. |
| re+ | Matches 0 or 1 occurrence of preceding expression. |
| re{ n} | Matches exactly n number of occurrences of preceding expression. |
| re{ n,} | Matches n or more occurrences of preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of preceding expression. |
| a| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers matched text. |
| (?imx) | Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| (?-imx) | Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| (?: re) | Groups regular expressions without remembering matched text. |
| (?imx: re) | Temporarily toggles on i, m, or x options within parentheses. |
| (?-imx: re) | Temporarily toggles off i, m, or x options within parentheses. |
| (?#...) | Comment. |
| (?= re) | Specifies position using a pattern. Doesn't have a range. |
| (?! re) | Specifies position using pattern negation. Doesn't have a range. |
| (?> re) | Matches independent pattern without backtracking. |
| \w | Matches word characters. |
| \W | Matches nonword characters. |
| \s | Matches whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches nonwhitespace. |
| \d | Matches digits. Equivalent to [0-9]. |
| \D | Matches nondigits. |
| \A | Matches beginning of string. |
| \Z | Matches end of string. If a newline exists, it matches just before newline. |
| \z | Matches end of string. |
| \G | Matches point where last match finished. |
| \b | Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets. |
| \B | Matches nonword boundaries. |
| \n, \t, etc. | Matches newlines, carriage returns, tabs, etc. |
| \1...\9 | Matches nth grouped subexpression. |
| \10 | Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code. |

Regular-expression Examples:

Literal characters:

|  |  |
| --- | --- |
| **Example** | **Description** |
| python | Match "python". |

Character classes:

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

Special Character Classes:

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

Repetition Cases:

|  |  |
| --- | --- |
| **Example** | **Description** |
| ruby? | Match "rub" or "ruby": the y is optional |
| ruby\* | Match "rub" plus 0 or more ys |
| ruby+ | Match "rub" plus 1 or more ys |
| \d{3} | Match exactly 3 digits |
| \d{3,} | Match 3 or more digits |
| \d{3,5} | Match 3, 4, or 5 digits |

Nongreedy repetition:

This matches the smallest number of repetitions:

|  |  |
| --- | --- |
| **Example** | **Description** |
| <.\*> | Greedy repetition: matches "<python>perl>" |
| <.\*?> | Nongreedy: matches "<python>" in "<python>perl>" |

Grouping with parentheses:

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

Backreferences:

This matches a previously matched group again:

|  |  |
| --- | --- |
| **Example** | **Description** |
| ([Pp])ython&\1ails | Match python&rails or Python&Rails |
| (['"])[^\1]\*\1 | Single or double-quoted string. \1 matches whatever the 1st group matched . \2 matches whatever the 2nd group matched, etc. |

Alternatives:

|  |  |
| --- | --- |
| **Example** | **Description** |
| python|perl | Match "python" or "perl" |
| rub(y|le)) | Match "ruby" or "ruble" |
| Python(!+|\?) | "Python" followed by one or more ! or one ? |

Anchors:

This need to specify match position

|  |  |
| --- | --- |
| **Example** | **Description** |
| ^Python | Match "Python" at the start of a string or internal line |
| Python$ | Match "Python" at the end of a string or line |
| \APython | Match "Python" at the start of a string |
| Python\Z | Match "Python" at the end of a string |
| \bPython\b | Match "Python" at a word boundary |
| \brub\B | \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone |
| Python(?=!) | Match "Python", if followed by an exclamation point |
| Python(?!!) | Match "Python", if not followed by an exclamation point |

Special syntax with parentheses:

|  |  |
| --- | --- |
| **Example** | **Description** |
| R(?#comment) | Matches "R". All the rest is a comment |
| R(?i)uby | Case-insensitive while matching "uby" |
| R(?i:uby) | Same as above |
| rub(?:y|le)) | Group only without creating \1 backreference |

Pasted from <<http://www.tutorialspoint.com/python/python_quick_guide.htm>>

The standard library comes with a number of modules that can be used both as modules and as command-line utilities.

The *dis* Module:

The dis module is the Python disassembler. It converts byte codes to a format that is slightly more appropriate for human consumption.

You can run the disassembler from the command line. It compiles the given script and prints the disassembled byte codes to the STDOUT. You can also use dis as a module. The **dis** function takes a class, method, function or code object as its single argument.

EXAMPLE:

#!/usr/bin/python  
import dis

def sum():  
 vara = 10  
 varb = 20

sum = vara + varb  
 print "vara + varb = %d" % sum

# Call dis function for the function.

dis.dis(sum)

This would produce the following result:

6 0 LOAD\_CONST 1 (10)  
 3 STORE\_FAST 0 (vara)

7 6 LOAD\_CONST 2 (20)  
 9 STORE\_FAST 1 (varb)

9 12 LOAD\_FAST 0 (vara)  
 15 LOAD\_FAST 1 (varb)  
 18 BINARY\_ADD  
 19 STORE\_FAST 2 (sum)

10 22 LOAD\_CONST 3 ('vara + varb = %d')  
 25 LOAD\_FAST 2 (sum)  
 28 BINARY\_MODULO  
 29 PRINT\_ITEM  
 30 PRINT\_NEWLINE  
 31 LOAD\_CONST 0 (None)  
 34 RETURN\_VALUE

The *pdb* Module

The pdb module is the standard Python debugger. It is based on the bdb debugger framework.

You can run the debugger from the command line (type n [or next] to go to the next line and help to get a list of available commands):

EXAMPLE:

Before you try to run **pdb.py**, set your path properly to Python lib directory. So let us try with above example sum.py:

$pdb.py sum.py  
> /test/sum.py(3)<module>()  
-> import dis  
(Pdb) n  
> /test/sum.py(5)<module>()  
-> def sum():  
(Pdb) n  
>/test/sum.py(14)<module>()  
-> dis.dis(sum)  
(Pdb) n  
 6 0 LOAD\_CONST 1 (10)  
 3 STORE\_FAST 0 (vara)

7 6 LOAD\_CONST 2 (20)  
 9 STORE\_FAST 1 (varb)

9 12 LOAD\_FAST 0 (vara)  
 15 LOAD\_FAST 1 (varb)  
 18 BINARY\_ADD  
 19 STORE\_FAST 2 (sum)

10 22 LOAD\_CONST 3 ('vara + varb = %d')  
 25 LOAD\_FAST 2 (sum)  
 28 BINARY\_MODULO  
 29 PRINT\_ITEM  
 30 PRINT\_NEWLINE  
 31 LOAD\_CONST 0 (None)  
 34 RETURN\_VALUE  
--Return--  
> /test/sum.py(14)<module>()->None  
-v dis.dis(sum)  
(Pdb) n  
--Return--  
> <string>(1)<module>()->None  
(Pdb)

The *profile* Module:

The profile module is the standard Python profiler. You can run the profiler from the command line:

EXAMPLE:

Let us try to profile the following program:

#!/usr/bin/python

vara = 10  
varb = 20

sum = vara + varb  
print "vara + varb = %d" % sum

Now, try running **cProfile.py** over this file *sum.py* as follows:

$cProfile.py sum.py  
vara + varb = 30  
 4 function calls in 0.000 CPU seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno  
 1 0.000 0.000 0.000 0.000 <string>:1(<module>)  
 1 0.000 0.000 0.000 0.000 sum.py:3(<module>)  
 1 0.000 0.000 0.000 0.000 {execfile}  
 1 0.000 0.000 0.000 0.000 {method ......}

The *tabnanny* Module

The tabnanny module checks Python source files for ambiguous indentation. If a file mixes tabs and spaces in a way that throws off indentation, no matter what tab size you're using, the nanny complains:

EXAMPLE:

Let us try to profile the following program:

#!/usr/bin/python

vara = 10  
varb = 20

sum = vara + varb  
print "vara + varb = %d" % sum

If you would try a correct file with tabnanny.py, then it won't complain as follows:

$tabnanny.py -v sum.py  
'sum.py': Clean bill of health.

Pasted from <<http://www.tutorialspoint.com/python/python_tools_utilities.htm>>

Python Tutorial

Python is a general-purpose interpreted, interactive, object-oriented and high-level programming language. Python was created by Guido van Rossum in the late eighties and early nineties. Like Perl, Python source code is also now available under the GNU General Public License (GPL).

Audience

This tutorial has been designed for software programmers with a need to understand the Python programming language starting from scratch. This tutorial will give you enough understanding on Python programming language from where you can take yourself to a higher level of expertise.

Prerequisites

Before proceeding with this tutorial you should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages will help you in understanding the Python programming concepts and move fast on the learning track.

Execute Python Programs

For most of the examples given in this tutorial you will find **Try it** option, so just make use of it and enjoy your learning.

Try following example using **Try it** option available at the top right corner of the below sample code box:

#!/usr/bin/python

print "Hello, Python!";

1 - Python Quick Reference Guide

A quick Python reference guide for Python Programmers.

[Python Quick Reference Guide](http://www.tutorialspoint.com/python/python_quick_guide.htm)

2 - Useful Python Resources

A collection of *Python* Sites, Books and Articles is given at this page.

[Useful Python Resources](http://www.tutorialspoint.com/python/python_useful_resources.htm)

3 - Python Tools and Utilities

Here you will find a useful Python's tools and utilities which you will require very frequently while doing Python programming.

[Python Tools and Utilities](http://www.tutorialspoint.com/python/python_tools_utilities.htm)

4 - Python Tutorial in PDF

Download a quick Python tutorial in PDF format.

[Python Tutorial in PDF](http://www.tutorialspoint.com/python/python_pdf_version.htm)

Pasted from <<http://www.tutorialspoint.com/python/index.htm>>

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python was designed to be highly readable which uses English keywords frequently where as other languages use punctuation and it has fewer syntactical constructions than other languages.

* + **Python is Interpreted:** This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it. This is similar to PERL and PHP.
  + **Python is Interactive:** This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
  + **Python is Object-Oriented:** This means that Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
  + **Python is Beginner's Language:** Python is a great language for the beginner programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python:

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features:

Python's feature highlights include:

* + **Easy-to-learn:** Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.
  + **Easy-to-read:** Python code is much more clearly defined and visible to the eyes.
  + **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain.
  + **A broad standard library:** One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows and Macintosh.
  + **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.
  + **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
  + **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
  + **Databases:** Python provides interfaces to all major commercial databases.
  + **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh and the X Window system of Unix.
  + **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below:

* + Support for functional and structured programming methods as well as OOP.
  + It can be used as a scripting language or can be compiled to byte-code for building large applications.
  + Very high-level dynamic data types and supports dynamic type checking.
  + Supports automatic garbage collection.
  + It can be easily integrated with C, C++, COM, ActiveX, CORBA and Java.

Pasted from <<http://www.tutorialspoint.com/python/python_overview.htm>>

Try it Option Online

You really do not need to set up your own environment to start learning Python programming language. Reason is very simple, we already have set up Python Programming environment online, so that you can execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try following example using **Try it** option available at the top right corner of the below sample code box:

#!/usr/bin/python

print "Hello, Python!";

For most of the examples given in this tutorial, you will find **Try it** option, so just make use of it and enjoy your learning.

Local Environment Setup

If you are still willing to set up your environment, let's understand how to set up our Python environment. Python is available on a wide variety of platforms including Linux and Mac OS X. Try opening a terminal window and type "python" to find out if its already installed and which version you have if it is installed.

* + Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
  + Win 9x/NT/2000
  + Macintosh (Intel, PPC, 68K)
  + OS/2
  + DOS (multiple versions)
  + PalmOS
  + Nokia mobile phones
  + Windows CE
  + Acorn/RISC OS
  + BeOS
  + Amiga
  + VMS/OpenVMS
  + QNX
  + VxWorks
  + Psion
  + Python has also been ported to the Java and .NET virtual machines

Getting Python:

The most up-to-date and current source code, binaries, documentation, news, etc. is available at the official website of Python:

**Python Official Website :** <http://www.python.org/>

You can download Python documentation from the following site. The documentation is available in HTML, PDF and PostScript formats.

**Python Documentation Website :** [www.python.org/doc/](http://www.python.org/doc/)

Install Python:

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms:

Unix & Linux Installation:

Here are the simple steps to install Python on Unix/Linux machine.

* + Open a Web browser and go to <http://www.python.org/download/>
  + Follow the link to download zipped source code available for Unix/Linux.
  + Download and extract files.
  + Editing the *Modules/Setup* file if you want to customize some options.
  + **run** ./configure script
  + make
  + make install

This will install python in a standard location */usr/local/bin* and its libraries are installed in*/usr/local/lib/pythonXX* where XX is the version of Python that you are using.

Windows Installation:

Here are the steps to install Python on Windows machine.

* + Open a Web browser and go to <http://www.python.org/download/>
  + Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you are going to install.
  + To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Just save the installer file to your local machine and then run it to find out if your machine supports MSI.
  + Run the downloaded file by double-clicking it in Windows Explorer. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you're ready to roll!

Macintosh Installation:

Recent Macs come with Python installed, but it may be several years out of date. See <http://www.python.org/download/mac/> for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available."

Jack Jansen maintains it and you can have full access to the entire documentation at his Web site -**Jack Jansen Website :** <http://www.cwi.nl/~jack/macpython.html>

Just go to this link and you will find complete installation detail for Mac OS installation.

Setting up PATH:

Programs and other executable files can live in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. These variables contain information available to the command shell and other programs.

The **path** variable is named PATH in Unix or Path in Windows (Unix is case-sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Setting path at Unix/Linux:

To add the Python directory to the path for a particular session in Unix:

* + **In the csh shell:** type   
    setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
  + **In the bash shell (Linux):** type   
    export PATH="$PATH:/usr/local/bin/python" and press Enter.
  + **In the sh or ksh shell:** type   
    PATH="$PATH:/usr/local/bin/python" and press Enter.

**Note:** /usr/local/bin/python is the path of the Python directory

Setting path at Windows:

To add the Python directory to the path for a particular session in Windows:

* + **At the command prompt :** type   
    path %path%;C:\Python and press Enter.

**Note:** C:\Python is the path of the Python directory

Python Environment Variables:

Here are important environment variables, which can be recognized by Python:

|  |  |
| --- | --- |
| **Variable** | **Description** |
| PYTHONPATH | Has a role similar to PATH. This variable tells the Python interpreter where to locate the module files you import into a program. PYTHONPATH should include the Python source library directory and the directories containing your Python source code. PYTHONPATH is sometimes preset by the Python installer. |
| PYTHONSTARTUP | Contains the path of an initialization file containing Python source code that is executed every time you start the interpreter (similar to the Unix .profile or .login file). This file, often named .pythonrc.py in Unix, usually contains commands that load utilities or modify PYTHONPATH. |
| PYTHONCASEOK | Used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it. |
| PYTHONHOME | An alternative module search path. It's usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy. |

Running Python:

There are three different ways to start Python:

(1) Interactive Interpreter:

You can enter **python** and start coding right away in the interactive interpreter by starting it from the command line. You can do this from Unix, DOS or any other system, which provides you a command-line interpreter or shell window.

$python # Unix/Linux

or

python% # Unix/Linux

or

C:>python # Windows/DOS

Here is the list of all the available command line options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| -d | provide debug output |
| -O | generate optimized bytecode (resulting in .pyo files) |
| -S | do not run import site to look for Python paths on startup |
| -v | verbose output (detailed trace on import statements) |
| -X | disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6 |
| -c cmd | run Python script sent in as cmd string |
| file | run Python script from given file |

(2) Script from the Command-line:

A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

$python script.py # Unix/Linux

or

python% script.py # Unix/Linux

or

C:>python script.py # Windows/DOS

**Note:** Be sure the file permission mode allows execution.

(3) Integrated Development Environment

You can run Python from a graphical user interface (GUI) environment as well. All you need is a GUI application on your system that supports Python.

* + **Unix:** IDLE is the very first Unix IDE for Python.
  + **Windows:** PythonWin is the first Windows interface for Python and is an IDE with a GUI.
  + **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

Before proceeding to next chapter, make sure your environment is properly set up and working perfectly fine. If you are not able to set up the environment properly, then you can take help from your system admin.

All the examples given in subsequent chapters have been executed with Python 2.4.3 version available on CentOS flavor of Linux.

Pasted from <<http://www.tutorialspoint.com/python/python_environment.htm>>

The Python language has many similarities to Perl, C and Java. However, there are some definite differences between the languages. This chapter is designed to quickly get you up to speed on the syntax that is expected in Python.

First Python Program:

INTERACTIVE MODE PROGRAMMING:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

$ python  
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)  
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>>

Type the following text to the right of the Python prompt and press the Enter key:

>>> print "Hello, Python!";

If you are running new version of Python, then you would need to use print statement with parenthesis like **print ("Hello, Python!");**. However at Python version 2.4.3, this will produce following result:

Hello, Python!

**SCRIPT MODE PROGRAMMING:**

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. All python files will have extension **.py**. So put the following source code in a test.py file.

print "Hello, Python!";

Here, I assumed that you have Python interpreter set in PATH variable. Now, try to run this program as follows:

$ python test.py

This will produce the following result:

Hello, Python!

Let's try another way to execute a Python script. Below is the modified test.py file:

#!/usr/bin/python

print "Hello, Python!";

Here, I assumed that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows:

$ chmod +x test.py # This is to make file executable  
$./test.py

This will produce the following result:

Hello, Python!

Python Identifiers:

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $ and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are following identifier naming convention for Python:

* + Class names start with an uppercase letter and all other identifiers with a lowercase letter.
  + Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.
  + Starting an identifier with two leading underscores indicates a strongly private identifier.
  + If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words:

The following list shows the reserved words in Python. These reserved words may not be used as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |

Lines and Indentation:

One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

if True:  
 print "True"  
else:  
 print "False"

However, the second block in this example will generate an error:

if True:  
 print "Answer"  
 print "True"  
else:  
 print "Answer"  
 print "False"

Thus, in Python all the continous lines indented with similar number of spaces would form a block. Following is the example having various statement blocks:

**Note:** Don't try to understand logic or different functions used. Just make sure you undertood various blocks even if they are without braces.

#!/usr/bin/python

import sys

try:  
 # open file stream  
 file = open(file\_name, "w")  
except IOError:  
 print "There was an error writing to", file\_name  
 sys.exit()  
print "Enter '", file\_finish,  
print "' When finished"  
while file\_text != file\_finish:  
 file\_text = raw\_input("Enter text: ")  
 if file\_text == file\_finish:  
 # close the file  
 file.close  
 break  
 file.write(file\_text)  
 file.write("\n")  
file.close()  
file\_name = raw\_input("Enter filename: ")  
if len(file\_name) == 0:  
 print "Next time please enter something"  
 sys.exit()  
try:  
 file = open(file\_name, "r")  
except IOError:  
 print "There was an error reading file"  
 sys.exit()  
file\_text = file.read()  
file.close()  
print file\_text

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

total = item\_one + \  
 item\_two + \  
 item\_three

Statements contained within the [], {} or () brackets do not need to use the line continuation character. For example:

days = ['Monday', 'Tuesday', 'Wednesday',  
 'Thursday', 'Friday']

Quotation in Python:

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

word = 'word'  
sentence = "This is a sentence."  
paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""

Comments in Python:

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.

#!/usr/bin/python

# First comment  
print "Hello, Python!"; # second comment

This will produce the following result:

Hello, Python!

A comment may be on the same line after a statement or expression:

name = "Madisetti" # This is again comment

You can comment multiple lines as follows:

# This is a comment.  
# This is a comment, too.  
# This is a comment, too.  
# I said that already.

Using Blank Lines:

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User:

The following line of the program displays the prompt, Press the enter key to exit and waits for the user to press the Enter key:

#!/usr/bin/python

raw\_input("\n\nPress the enter key to exit.")

Here, "\n\n" are being used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line:

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

import sys; x = 'foo'; sys.stdout.write(x + '\n')

Multiple Statement Groups as Suites:

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class, are those which require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite. For example:

if expression :   
 suite  
elif expression :   
 suite   
else :   
 suite

Command Line Arguments:

You may have seen, for instance, that many programs can be run so that they provide you with some basic information about how they should be run. Python enables you to do this with -h:

$ python -h  
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...  
Options and arguments (and corresponding environment variables):  
-c cmd : program passed in as string (terminates option list)  
-d : debug output from parser (also PYTHONDEBUG=x)  
-E : ignore environment variables (such as PYTHONPATH)  
-h : print this help message and exit

[ etc. ]

You can also program your script in such a way that it should accept various options. [Command Line Arguments](http://www.tutorialspoint.com/python/python_command_line_arguments.htm) is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

Pasted from <<http://www.tutorialspoint.com/python/python_basic_syntax.htm>>

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables:

Python variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example:

#!/usr/bin/python

counter = 100 # An integer assignment  
miles = 1000.0 # A floating point  
name = "John" # A string

print counter  
print miles  
print name

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles* and *name* variables, respectively. While running this program, this will produce the following result:

100  
1000.0  
John

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously. For example:

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example:

        a, b, c = 1, 2, "john"

Here, two integer objects with values 1 and 2 are assigned to variables a and b, and one string object with the value "john" is assigned to the variable c.

Standard Data Types:

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types:

* + Numbers
  + String
  + List
  + Tuple
  + Dictionary

Python Numbers:

Number data types store numeric values. They are immutable data types which means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example:

var1 = 1  
var2 = 10

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is:

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example:

del var  
del var\_a, var\_b

Python supports four different numerical types:

* + int (signed integers)
  + long (long integers [can also be represented in octal and hexadecimal])
  + float (floating point real values)
  + complex (complex numbers)

Examples:

Here are some examples of numbers:

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEl | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

* + Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
  + A complex number consists of an ordered pair of real floating-point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

Python Strings:

Strings in Python are identified as a contiguous set of characters in between quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ( [ ] and [ : ] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus ( + ) sign is the string concatenation operator and the asterisk ( \* ) is the repetition operator. For example:

#!/usr/bin/python

str = 'Hello World!'

print str # Prints complete string  
print str[0] # Prints first character of the string  
print str[2:5] # Prints characters starting from 3rd to 5th  
print str[2:] # Prints string starting from 3rd character  
print str \* 2 # Prints string two times  
print str + "TEST" # Prints concatenated string

This will produce the following result:

Hello World!  
H  
llo  
llo World!  
Hello World!Hello World!  
Hello World!TEST

Python Lists:

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ( [ ] and [ : ] ) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus ( + ) sign is the list concatenation operator, and the asterisk ( \* ) is the repetition operator. For example:

#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
tinylist = [123, 'john']

print list # Prints complete list  
print list[0] # Prints first element of the list  
print list[1:3] # Prints elements starting from 2nd till 3rd   
print list[2:] # Prints elements starting from 3rd element  
print tinylist \* 2 # Prints list two times  
print list + tinylist # Prints concatenated lists

This will produce the following result:

['abcd', 786, 2.23, 'john', 70.200000000000003]  
abcd  
[786, 2.23]  
[2.23, 'john', 70.200000000000003]  
[123, 'john', 123, 'john']  
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']

Python Tuples:

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read-only** lists. For example:

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )  
tinytuple = (123, 'john')

print tuple # Prints complete list  
print tuple[0] # Prints first element of the list  
print tuple[1:3] # Prints elements starting from 2nd till 3rd   
print tuple[2:] # Prints elements starting from 3rd element  
print tinytuple \* 2 # Prints list two times  
print tuple + tinytuple # Prints concatenated lists

This will produce the following result:

('abcd', 786, 2.23, 'john', 70.200000000000003)  
abcd  
(786, 2.23)  
(2.23, 'john', 70.200000000000003)  
(123, 'john', 123, 'john')  
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')

Following is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists:

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )  
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
tuple[2] = 1000 # Invalid syntax with tuple  
list[2] = 1000 # Valid syntax with list

Python Dictionary:

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ( { } ) and values can be assigned and accessed using square braces ( [] ). For example:

#!/usr/bin/python

dict = {}  
dict['one'] = "This is one"  
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one'] # Prints value for 'one' key  
print dict[2] # Prints value for 2 key  
print tinydict # Prints complete dictionary  
print tinydict.keys() # Prints all the keys  
print tinydict.values() # Prints all the values

This will produce the following result:

This is one  
This is two  
{'dept': 'sales', 'code': 6734, 'name': 'john'}  
['dept', 'code', 'name']  
['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion:

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| **Function** | **Description** |
| int(x [,base]) | Converts x to an integer. base specifies the base if x is a string. |
| long(x [,base] ) | Converts x to a long integer. base specifies the base if x is a string. |
| float(x) | Converts x to a floating-point number. |
| complex(real [,imag]) | Creates a complex number. |
| str(x) | Converts object x to a string representation. |
| repr(x) | Converts object x to an expression string. |
| eval(str) | Evaluates a string and returns an object. |
| tuple(s) | Converts s to a tuple. |
| list(s) | Converts s to a list. |
| set(s) | Converts s to a set. |
| dict(d) | Creates a dictionary. d must be a sequence of (key,value) tuples. |
| frozenset(s) | Converts s to a frozen set. |
| chr(x) | Converts an integer to a character. |
| unichr(x) | Converts an integer to a Unicode character. |
| ord(x) | Converts a single character to its integer value. |
| hex(x) | Converts an integer to a hexadecimal string. |
| oct(x) | Converts an integer to an octal string. |

Pasted from <<http://www.tutorialspoint.com/python/python_variable_types.htm>>

What is an operator?

Simple answer can be given using expression *4 + 5 is equal to 9*. Here, 4 and 5 are called operands and + is called operator. Python language supports the following types of operators.

* + Arithmetic Operators
  + Comparison (i.e., Relational) Operators
  + Assignment Operators
  + Logical Operators
  + Bitwise Operators
  + Membership Operators
  + Identity Operators

Let's have a look on all operators one by one.

Python Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

[ [Show Example](http://www.tutorialspoint.com/python/arithmetic_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | a + b will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | a - b will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | a \* b will give 200 |
| / | Division - Divides left hand operand by right hand operand | b / a will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | b % a will give 0 |
| \*\* | Exponent - Performs exponential (power) calculation on operators | a\*\*b will give 10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | 9//2 is equal to 4 and 9.0//2.0 is equal to 4.0 |

Python Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

[ [Show Example](http://www.tutorialspoint.com/python/comparison_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. | (a == b) is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. This is similar to != operator. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |

Python Assignment Operators:

Assume variable a holds 10 and variable b holds 20, then:

[ [Show Example](http://www.tutorialspoint.com/python/assignment_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | c = a + b will assigne value of a + b into c |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / a |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | c %= a is equivalent to c = c % a |
| \*\*= | Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c = c \*\* a |
| //= | Floor Dividion and assigns a value, Performs floor division on operators and assign value to the left operand | c //= a is equivalent to c = c // a |

Python Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

There are following Bitwise operators supported by Python language

[ [Show Example](http://www.tutorialspoint.com/python/bitwise_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (a & b) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in eather operand. | (a | b) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (a ^ b) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the efect of 'flipping' bits. | (~a ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | a << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 will give 15 which is 0000 1111 |

Python Logical Operators:

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

[ [Show Example](http://www.tutorialspoint.com/python/logical_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Called Logical AND operator. If both the operands are true then then condition becomes true. | (a and b) is true. |
| or | Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true. | (a or b) is true. |
| not | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | not(a and b) is false. |

Python Membership Operators:

In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below:

[ [Show Example](http://www.tutorialspoint.com/python/membership_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here **in** results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here **not in** results in a 1 if x is not a member of sequence y. |

Python Identity Operators:

Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

[ [Show Example](http://www.tutorialspoint.com/python/identity_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here **is** results in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here **is not** results in 1 if id(x) is not equal to id(y). |

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

[ [Show Example](http://www.tutorialspoint.com/python/operators_precedence_example.htm) ]

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Ccomplement, unary plus and minus (method names for the last two are +@ and -@) |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND' |
| ^ | | Bitwise exclusive `OR' and regular `OR' |
| <= < > >= | Comparison operators |
| <> == != | Equality operators |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

Pasted from <<http://www.tutorialspoint.com/python/python_basic_operators.htm>>

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Python programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either**zero** or **null**, then it is assumed as **false** value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| [if statements](http://www.tutorialspoint.com/python/python_if_statement.htm) | An **if statement** consists of a boolean expression followed by one or more statements. |
| [if...else statements](http://www.tutorialspoint.com/python/python_if_else.htm) | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| [nested if statements](http://www.tutorialspoint.com/python/nested_if_statements_in_python.htm) | You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |

Single Statement Suites:

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause:

#!/usr/bin/python

var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"

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

Value of expression is 100  
Good bye!

Pasted from <<http://www.tutorialspoint.com/python/python_decision_making.htm>>

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Python programming language provides following types of loops to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [while loop](http://www.tutorialspoint.com/python/python_while_loop.htm) | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [for loop](http://www.tutorialspoint.com/python/python_for_loop.htm) | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [nested loops](http://www.tutorialspoint.com/python/python_nested_loops.htm) | You can use one or more loop inside any another while, for or do..while loop. |

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| [break statement](http://www.tutorialspoint.com/python/python_break_statement.htm) | Terminates the **loop** statement and transfers execution to the statement immediately following the loop. |
| [continue statement](http://www.tutorialspoint.com/python/python_continue_statement.htm) | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| [pass statement](http://www.tutorialspoint.com/python/python_pass_statement.htm) | The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

Pasted from <<http://www.tutorialspoint.com/python/python_loops.htm>>

Number data types store numeric values. They are immutable data types, which means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example:

var1 = 1  
var2 = 10

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is:

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example:

del var  
del var\_a, var\_b

Python supports four different numerical types:

* + **int (signed integers)**: often called just integers or ints, are positive or negative whole numbers with no decimal point.
  + **long (long integers )**: or longs, are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
  + **float (floating point real values)** : or floats, represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
  + **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). a is the real part of the number, and b is the imaginary part. Complex numbers are not used much in Python programming.

Examples:

Here are some examples of numbers:

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEl | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

* + Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
  + A complex number consists of an ordered pair of real floatingpoint numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

Number Type Conversion:

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you'll need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

* + Type **int(x)**to convert x to a plain integer.
  + Type **long(x)** to convert x to a long integer.
  + Type **float(x)** to convert x to a floating-point number.
  + Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
  + Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Mathematical Functions:

Python includes following functions that perform mathematical calculations.

|  |  |
| --- | --- |
| **Function** | **Returns ( description )** |
| [abs(x)](http://www.tutorialspoint.com/python/number_abs.htm) | The absolute value of x: the (positive) distance between x and zero. |
| [ceil(x)](http://www.tutorialspoint.com/python/number_ceil.htm) | The ceiling of x: the smallest integer not less than x |
| [cmp(x, y)](http://www.tutorialspoint.com/python/number_cmp.htm) | -1 if x < y, 0 if x == y, or 1 if x > y |
| [exp(x)](http://www.tutorialspoint.com/python/number_exp.htm) | The exponential of x: ex |
| [fabs(x)](http://www.tutorialspoint.com/python/number_fabs.htm) | The absolute value of x. |
| [floor(x)](http://www.tutorialspoint.com/python/number_floor.htm) | The floor of x: the largest integer not greater than x |
| [log(x)](http://www.tutorialspoint.com/python/number_log.htm) | The natural logarithm of x, for x> 0 |
| [log10(x)](http://www.tutorialspoint.com/python/number_log10.htm) | The base-10 logarithm of x for x> 0 . |
| [max(x1, x2,...)](http://www.tutorialspoint.com/python/number_max.htm) | The largest of its arguments: the value closest to positive infinity |
| [min(x1, x2,...)](http://www.tutorialspoint.com/python/number_min.htm) | The smallest of its arguments: the value closest to negative infinity |
| [modf(x)](http://www.tutorialspoint.com/python/number_modf.htm) | The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float. |
| [pow(x, y)](http://www.tutorialspoint.com/python/number_pow.htm) | The value of x\*\*y. |
| [round(x [,n])](http://www.tutorialspoint.com/python/number_round.htm) | **x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0. |
| [sqrt(x)](http://www.tutorialspoint.com/python/number_sqrt.htm) | The square root of x for x > 0 |

Random Number Functions:

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

|  |  |
| --- | --- |
| **Function** | **Description** |
| [choice(seq)](http://www.tutorialspoint.com/python/number_choice.htm) | A random item from a list, tuple, or string. |
| [randrange ([start,] stop [,step])](http://www.tutorialspoint.com/python/number_randrange.htm) | A randomly selected element from range(start, stop, step) |
| [random()](http://www.tutorialspoint.com/python/number_random.htm) | A random float r, such that 0 is less than or equal to r and r is less than 1 |
| [seed([x])](http://www.tutorialspoint.com/python/number_seed.htm) | Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None. |
| [shuffle(lst)](http://www.tutorialspoint.com/python/number_shuffle.htm) | Randomizes the items of a list in place. Returns None. |
| [uniform(x, y)](http://www.tutorialspoint.com/python/number_uniform.htm) | A random float r, such that x is less than or equal to r and r is less than y |

Trigonometric Functions:

Python includes following functions that perform trigonometric calculations.

|  |  |
| --- | --- |
| **Function** | **Description** |
| [acos(x)](http://www.tutorialspoint.com/python/number_acos.htm) | Return the arc cosine of x, in radians. |
| [asin(x)](http://www.tutorialspoint.com/python/number_asin.htm) | Return the arc sine of x, in radians. |
| [atan(x)](http://www.tutorialspoint.com/python/number_atan.htm) | Return the arc tangent of x, in radians. |
| [atan2(y, x)](http://www.tutorialspoint.com/python/number_atan2.htm) | Return atan(y / x), in radians. |
| [cos(x)](http://www.tutorialspoint.com/python/number_cos.htm) | Return the cosine of x radians. |
| [hypot(x, y)](http://www.tutorialspoint.com/python/number_hypot.htm) | Return the Euclidean norm, sqrt(x\*x + y\*y). |
| [sin(x)](http://www.tutorialspoint.com/python/number_sin.htm) | Return the sine of x radians. |
| [tan(x)](http://www.tutorialspoint.com/python/number_tan.htm) | Return the tangent of x radians. |
| [degrees(x)](http://www.tutorialspoint.com/python/number_degrees.htm) | Converts angle x from radians to degrees. |
| [radians(x)](http://www.tutorialspoint.com/python/number_radians.htm) | Converts angle x from degrees to radians. |

Mathematical Constants:

The module also defines two mathematical constants:

|  |  |
| --- | --- |
| **Constants** | **Description** |
| pi | The mathematical constant pi. |
| e | The mathematical constant e. |

Pasted from <<http://www.tutorialspoint.com/python/python_numbers.htm>>

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

Creating strings is as simple as assigning a value to a variable. For example:

var1 = 'Hello World!'  
var2 = "Python Programming"

Accessing Values in Strings:

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. Following is a simple example:

#!/usr/bin/python

var1 = 'Hello World!'  
var2 = "Python Programming"

print "var1[0]: ", var1[0]  
print "var2[1:5]: ", var2[1:5]

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

var1[0]: H  
var2[1:5]: ytho

Updating Strings:

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. Following is a simple example:

#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'

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

Updated String :- Hello Python

Escape Characters:

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a singlequoted as well as doublequoted strings.

|  |  |  |
| --- | --- | --- |
| **Backslash**  **notation** | **Hexadecimal**  **character** | **Description** |
| \a | 0x07 | Bell or alert |
| \b | 0x08 | Backspace |
| \cx |  | Control-x |
| \C-x |  | Control-x |
| \e | 0x1b | Escape |
| \f | 0x0c | Formfeed |
| \M-\C-x |  | Meta-Control-x |
| \n | 0x0a | Newline |
| \nnn |  | Octal notation, where n is in the range 0.7 |
| \r | 0x0d | Carriage return |
| \s | 0x20 | Space |
| \t | 0x09 | Tab |
| \v | 0x0b | Vertical tab |
| \x |  | Character x |
| \xnn |  | Hexadecimal notation, where n is in the range 0.9, a.f, or A.F |

String Special Operators:

Assume string variable a holds 'Hello' and variable b holds 'Python', then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Concatenation - Adds values on either side of the operator | a + b will give HelloPython |
| \* | Repetition - Creates new strings, concatenating multiple copies of the same string | a\*2 will give -HelloHello |
| [] | Slice - Gives the character from the given index | a[1] will give **e** |
| [ : ] | Range Slice - Gives the characters from the given range | a[1:4] will give **ell** |
| in | Membership - Returns true if a character exists in the given string | **H in a** will give 1 |
| not in | Membership - Returns true if a character does not exist in the given string | **M not in a** will give 1 |
| r/R | Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark. | **print r'\n'** prints \n and **print R'\n'** prints \n |
| % | Format - Performs String formatting | See at next section |

String Formatting Operator:

One of Python's coolest features is the string format operator **%**. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example:

#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)

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

My name is Zara and weight is 21 kg!

Here is the list of complete set of symbols which can be used along with %:

|  |  |
| --- | --- |
| **Format Symbol** | **Conversion** |
| %c | character |
| %s | string conversion via str() prior to formatting |
| %i | signed decimal integer |
| %d | signed decimal integer |
| %u | unsigned decimal integer |
| %o | octal integer |
| %x | hexadecimal integer (lowercase letters) |
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e') |
| %E | exponential notation (with UPPERcase 'E') |
| %f | floating point real number |
| %g | the shorter of %f and %e |
| %G | the shorter of %f and %E |

Other supported symbols and functionality are listed in the following table:

|  |  |
| --- | --- |
| **Symbol** | **Functionality** |
| \* | argument specifies width or precision |
| - | left justification |
| + | display the sign |
| <sp> | leave a blank space before a positive number |
| # | add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used. |
| 0 | pad from left with zeros (instead of spaces) |
| % | '%%' leaves you with a single literal '%' |
| (var) | mapping variable (dictionary arguments) |
| m.n. | m is the minimum total width and n is the number of digits to display after the decimal point (if appl.) |

Triple Quotes:

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

#!/usr/bin/python

para\_str = """this is a long string that is made up of  
several lines and non-printable characters such as  
TAB ( \t ) and they will show up that way when displayed.  
NEWLINEs within the string, whether explicitly given like  
this within the brackets [ \n ], or just a NEWLINE within  
the variable assignment will also show up.  
"""  
print para\_str;

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n):

this is a long string that is made up of  
several lines and non-printable characters such as  
TAB ( ) and they will show up that way when displayed.  
NEWLINEs within the string, whether explicitly given like  
this within the brackets [  
 ], or just a NEWLINE within  
the variable assignment will also show up.

Raw strings don't treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it:

#!/usr/bin/python

print 'C:\\nowhere'

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

C:\nowhere

Now let's make use of raw string. We would put expression in **r'expression'** as follows:

#!/usr/bin/python

print r'C:\\nowhere'

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

C:\\nowhere

Unicode String:

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following:

#!/usr/bin/python

print u'Hello, world!'

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

Hello, world!

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

Built-in String Methods:

Python includes the following built-in methods to manipulate strings:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [capitalize()](http://www.tutorialspoint.com/python/string_capitalize.htm)  Capitalizes first letter of string |
| 2 | [center(width, fillchar)](http://www.tutorialspoint.com/python/string_center.htm)  Returns a space-padded string with the original string centered to a total of width columns |
| 3 | [count(str, beg= 0,end=len(string))](http://www.tutorialspoint.com/python/string_count.htm)  Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given |
| 4 | [decode(encoding='UTF-8',errors='strict')](http://www.tutorialspoint.com/python/string_decode.htm)  Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. |
| 5 | [encode(encoding='UTF-8',errors='strict')](http://www.tutorialspoint.com/python/string_encode.htm)  Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'. |
| 6 | [endswith(suffix, beg=0, end=len(string))](http://www.tutorialspoint.com/python/string_endswith.htm)  Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise |
| 7 | [expandtabs(tabsize=8)](http://www.tutorialspoint.com/python/string_expandtabs.htm)  Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided |
| 8 | [find(str, beg=0 end=len(string))](http://www.tutorialspoint.com/python/string_find.htm)  Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise |
| 9 | [index(str, beg=0, end=len(string))](http://www.tutorialspoint.com/python/string_index.htm)  Same as find(), but raises an exception if str not found |
| 10 | [isalnum()](http://www.tutorialspoint.com/python/string_isalnum.htm)  Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise |
| 11 | [isalpha()](http://www.tutorialspoint.com/python/string_isalpha.htm)  Returns true if string has at least 1 character and all characters are alphabetic and false otherwise |
| 12 | [isdigit()](http://www.tutorialspoint.com/python/string_isdigit.htm)  Returns true if string contains only digits and false otherwise |
| 13 | [islower()](http://www.tutorialspoint.com/python/string_islower.htm)  Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise |
| 14 | [isnumeric()](http://www.tutorialspoint.com/python/string_isnumeric.htm)  Returns true if a unicode string contains only numeric characters and false otherwise |
| 15 | [isspace()](http://www.tutorialspoint.com/python/string_isspace.htm)  Returns true if string contains only whitespace characters and false otherwise |
| 16 | [istitle()](http://www.tutorialspoint.com/python/string_istitle.htm)  Returns true if string is properly "titlecased" and false otherwise |
| 17 | [isupper()](http://www.tutorialspoint.com/python/string_isupper.htm)  Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise |
| 18 | [join(seq)](http://www.tutorialspoint.com/python/string_join.htm)  Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string |
| 19 | [len(string)](http://www.tutorialspoint.com/python/string_len.htm)  Returns the length of the string |
| 20 | [ljust(width[, fillchar])](http://www.tutorialspoint.com/python/string_ljust.htm)  Returns a space-padded string with the original string left-justified to a total of width columns |
| 21 | [lower()](http://www.tutorialspoint.com/python/string_lower.htm)  Converts all uppercase letters in string to lowercase |
| 22 | [lstrip()](http://www.tutorialspoint.com/python/string_lstrip.htm)  Removes all leading whitespace in string |
| 23 | [maketrans()](http://www.tutorialspoint.com/python/string_maketrans.htm)  Returns a translation table to be used in translate function. |
| 24 | [max(str)](http://www.tutorialspoint.com/python/string_max.htm)  Returns the max alphabetical character from the string str |
| 25 | [min(str)](http://www.tutorialspoint.com/python/string_min.htm)  Returns the min alphabetical character from the string str |
| 26 | [replace(old, new [, max])](http://www.tutorialspoint.com/python/string_replace.htm)  Replaces all occurrences of old in string with new or at most max occurrences if max given |
| 27 | [rfind(str, beg=0,end=len(string))](http://www.tutorialspoint.com/python/string_rfind.htm)  Same as find(), but search backwards in string |
| 28 | [rindex( str, beg=0, end=len(string))](http://www.tutorialspoint.com/python/string_rindex.htm)  Same as index(), but search backwards in string |
| 29 | [rjust(width,[, fillchar])](http://www.tutorialspoint.com/python/string_rjust.htm)  Returns a space-padded string with the original string right-justified to a total of width columns. |
| 30 | [rstrip()](http://www.tutorialspoint.com/python/string_rstrip.htm)  Removes all trailing whitespace of string |
| 31 | [split(str="", num=string.count(str))](http://www.tutorialspoint.com/python/string_split.htm)  Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given |
| 32 | [splitlines( num=string.count('\n'))](http://www.tutorialspoint.com/python/string_splitlines.htm)  Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed |
| 33 | [startswith(str, beg=0,end=len(string))](http://www.tutorialspoint.com/python/string_startswith.htm)  Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise |
| 34 | [strip([chars])](http://www.tutorialspoint.com/python/string_strip.htm)  Performs both lstrip() and rstrip() on string |
| 35 | [swapcase()](http://www.tutorialspoint.com/python/string_swapcase.htm)  Inverts case for all letters in string |
| 36 | [title()](http://www.tutorialspoint.com/python/string_title.htm)  Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase |
| 37 | [translate(table, deletechars="")](http://www.tutorialspoint.com/python/string_translate.htm)  Translates string according to translation table str(256 chars), removing those in the del string |
| 38 | [upper()](http://www.tutorialspoint.com/python/string_upper.htm)  Converts lowercase letters in string to uppercase |
| 39 | [zfill (width)](http://www.tutorialspoint.com/python/string_zfill.htm)  Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero) |
| 40 | [isdecimal()](http://www.tutorialspoint.com/python/string_isdecimal.htm)  Returns true if a unicode string contains only decimal characters and false otherwise |

Pasted from <<http://www.tutorialspoint.com/python/python_strings.htm>>

he most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists:

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Good thing about a list is that items in a list need not all have the same type.

Creating a list is as simple as putting different comma-separated values between squere brackets. For example:

list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5 ];  
list3 = ["a", "b", "c", "d"];

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists:

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. Following is a simple example:

#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]  
print "list2[1:5]: ", list2[1:5]

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

list1[0]: physics  
list2[1:5]: [2, 3, 4, 5]

Updating Lists:

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. Following is a simple example:

#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "  
print list[2];  
list[2] = 2001;  
print "New value available at index 2 : "  
print list[2];

**Note:** append() method is discussed in subsequent section.

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

Value available at index 2 :  
1997  
New value available at index 2 :  
2001

Delete List Elements:

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. Following is a simple example:

#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

print list1;  
del list1[2];  
print "After deleting value at index 2 : "  
print list1;

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

['physics', 'chemistry', 1997, 2000]  
After deleting value at index 2 :  
['physics', 'chemistry', 2000]

**Note:** remove() method is discussed in subsequent section.

Basic List Operations:

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

Indexing, Slicing, and Matrixes:

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input:

L = ['spam', 'Spam', 'SPAM!']

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

Built-in List Functions & Methods:

Python includes the following list functions:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [cmp(list1, list2)](http://www.tutorialspoint.com/python/list_cmp.htm)  Compares elements of both lists. |
| 2 | [len(list)](http://www.tutorialspoint.com/python/list_len.htm)  Gives the total length of the list. |
| 3 | [max(list)](http://www.tutorialspoint.com/python/list_max.htm)  Returns item from the list with max value. |
| 4 | [min(list)](http://www.tutorialspoint.com/python/list_min.htm)  Returns item from the list with min value. |
| 5 | [list(seq)](http://www.tutorialspoint.com/python/list_list.htm)  Converts a tuple into list. |

Python includes following list methods

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [list.append(obj)](http://www.tutorialspoint.com/python/list_append.htm)  Appends object obj to list |
| 2 | [list.count(obj)](http://www.tutorialspoint.com/python/list_count.htm)  Returns count of how many times obj occurs in list |
| 3 | [list.extend(seq)](http://www.tutorialspoint.com/python/list_extend.htm)  Appends the contents of seq to list |
| 4 | [list.index(obj)](http://www.tutorialspoint.com/python/list_index.htm)  Returns the lowest index in list that obj appears |
| 5 | [list.insert(index, obj)](http://www.tutorialspoint.com/python/list_insert.htm)  Inserts object obj into list at offset index |
| 6 | [list.pop(obj=list[-1])](http://www.tutorialspoint.com/python/list_pop.htm)  Removes and returns last object or obj from list |
| 7 | [list.remove(obj)](http://www.tutorialspoint.com/python/list_remove.htm)  Removes object obj from list |
| 8 | [list.reverse()](http://www.tutorialspoint.com/python/list_reverse.htm)  Reverses objects of list in place |
| 9 | [list.sort([func])](http://www.tutorialspoint.com/python/list_sort.htm)  Sorts objects of list, use compare func if given |

Pasted from <<http://www.tutorialspoint.com/python/python_lists.htm>>

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The only difference is that tuples can't be changed i.e., tuples are immutable and tuples use parentheses and lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values and optionally you can put these comma-separated values between parentheses also. For example:

tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5 );  
tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing:

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value:

tup1 = (50,);

Like string indices, tuple indices start at 0, and tuples can be sliced, concatenated and so on.

Accessing Values in Tuples:

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. Following is a simple example:

#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]  
print "tup2[1:5]: ", tup2[1:5]

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

tup1[0]: physics  
tup2[1:5]: [2, 3, 4, 5]

Updating Tuples:

Tuples are immutable which means you cannot update them or change values of tuple elements. But we able to take portions of an existing tuples to create a new tuples as follows. Following is a simple example:

#!/usr/bin/python

tup1 = (12, 34.56);  
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples  
# tup1[0] = 100;

# So let's create a new tuple as follows  
tup3 = tup1 + tup2;  
print tup3;

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

(12, 34.56, 'abc', 'xyz')

Delete Tuple Elements:

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. Following is a simple example:

#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;  
del tup;  
print "After deleting tup : "  
print tup;

This will produce following result. Note an exception raised, this is because after **del tup** tuple does not exist any more:

('physics', 'chemistry', 1997, 2000)  
After deleting tup :  
Traceback (most recent call last):  
 File "test.py", line 9, in <module>  
 print tup;  
NameError: name 'tup' is not defined

Basic Tuples Operations:

Tuples respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter :

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len((1, 2, 3)) | 3 | Length |
| (1, 2, 3) + (4, 5, 6) | (1, 2, 3, 4, 5, 6) | Concatenation |
| ['Hi!'] \* 4 | ('Hi!', 'Hi!', 'Hi!', 'Hi!') | Repetition |
| 3 in (1, 2, 3) | True | Membership |
| for x in (1, 2, 3): print x, | 1 2 3 | Iteration |

Indexing, Slicing, and Matrixes:

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input:

L = ('spam', 'Spam', 'SPAM!')

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

No Enclosing Delimiters:

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples:

#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';  
x, y = 1, 2;  
print "Value of x , y : ", x,y;

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

abc -4.24e+93 (18+6.6j) xyz  
Value of x , y : 1 2

Built-in Tuple Functions:

Python includes the following tuple functions:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [cmp(tuple1, tuple2)](http://www.tutorialspoint.com/python/tuple_cmp.htm)  Compares elements of both tuples. |
| 2 | [len(tuple)](http://www.tutorialspoint.com/python/tuple_len.htm)  Gives the total length of the tuple. |
| 3 | [max(tuple)](http://www.tutorialspoint.com/python/tuple_max.htm)  Returns item from the tuple with max value. |
| 4 | [min(tuple)](http://www.tutorialspoint.com/python/tuple_min.htm)  Returns item from the tuple with min value. |
| 5 | [tuple(seq)](http://www.tutorialspoint.com/python/tuple_tuple.htm)  Converts a list into tuple. |

Pasted from <<http://www.tutorialspoint.com/python/python_tuples.htm>>

A dictionary is mutable and is another container type that can store any number of Python objects, including other container types. Dictionaries consist of pairs (called items) of keys and their corresponding values.

Python dictionaries are also known as associative arrays or hash tables. The general syntax of a dictionary is as follows:

dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}

You can create dictionary in the following way as well:

dict1 = { 'abc': 456 };  
dict2 = { 'abc': 123, 98.6: 37 };

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary:

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example:

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];  
print "dict['Age']: ", dict['Age'];

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

dict['Name']: Zara  
dict['Age']: 7

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows:

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice'];

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

dict['Zara']:  
Traceback (most recent call last):  
 File "test.py", line 4, in <module>  
 print "dict['Alice']: ", dict['Alice'];  
KeyError: 'Alice'

Updating Dictionary:

You can update a dictionary by adding a new entry or item (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry as shown below in the simple example:

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry  
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age'];  
print "dict['School']: ", dict['School'];

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

dict['Age']: 8  
dict['School']: DPS School

Delete Dictionary Elements:

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example:

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # remove entry with key 'Name'  
dict.clear(); # remove all entries in dict  
del dict ; # delete entire dictionary

print "dict['Age']: ", dict['Age'];  
print "dict['School']: ", dict['School'];

This will produce the following result. Note an exception raised, this is because after **del dict** dictionary does not exist any more:

dict['Age']:  
Traceback (most recent call last):  
 File "test.py", line 8, in <module>  
 print "dict['Age']: ", dict['Age'];  
TypeError: 'type' object is unsubscriptable

**Note:** del() method is discussed in subsequent section.

Properties of Dictionary Keys:

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys:

**(a)** More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. Following is a simple example:

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];

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

dict['Name']: Manni

**(b)** Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example:

#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];

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

Traceback (most recent call last):  
 File "test.py", line 3, in <module>  
 dict = {['Name']: 'Zara', 'Age': 7};  
TypeError: list objects are unhashable

Built-in Dictionary Functions & Methods:

Python includes the following dictionary functions:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [cmp(dict1, dict2)](http://www.tutorialspoint.com/python/dictionary_cmp.htm)  Compares elements of both dict. |
| 2 | [len(dict)](http://www.tutorialspoint.com/python/dictionary_len.htm)  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | [str(dict)](http://www.tutorialspoint.com/python/dictionary_str.htm)  Produces a printable string representation of a dictionary |
| 4 | [type(variable)](http://www.tutorialspoint.com/python/dictionary_type.htm)  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes following dictionary methods

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [dict.clear()](http://www.tutorialspoint.com/python/dictionary_clear.htm)  Removes all elements of dictionary *dict* |
| 2 | [dict.copy()](http://www.tutorialspoint.com/python/dictionary_copy.htm)  Returns a shallow copy of dictionary *dict* |
| 3 | [dict.fromkeys()](http://www.tutorialspoint.com/python/dictionary_fromkeys.htm)  Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | [dict.get(key, default=None)](http://www.tutorialspoint.com/python/dictionary_get.htm)  For *key* key, returns value or default if key not in dictionary |
| 5 | [dict.has\_key(key)](http://www.tutorialspoint.com/python/dictionary_has_key.htm)  Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | [dict.items()](http://www.tutorialspoint.com/python/dictionary_items.htm)  Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | [dict.keys()](http://www.tutorialspoint.com/python/dictionary_keys.htm)  Returns list of dictionary dict's keys |
| 8 | [dict.setdefault(key, default=None)](http://www.tutorialspoint.com/python/dictionary_setdefault.htm)  Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | [dict.update(dict2)](http://www.tutorialspoint.com/python/dictionary_update.htm)  Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | [dict.values()](http://www.tutorialspoint.com/python/dictionary_values.htm)  Returns list of dictionary *dict*'s values |

Pasted from <<http://www.tutorialspoint.com/python/python_dictionary.htm>>

A Python program can handle date & time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

What is Tick?

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular **time** module available in Python which provides functions for working with times, and for converting between representations. The function *time.time()* returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

Example:

#!/usr/bin/python  
import time; # This is required to include time module.

ticks = time.time()  
print "Number of ticks since 12:00am, January 1, 1970:", ticks

This would produce a result something as follows:

Number of ticks since 12:00am, January 1, 1970: 7186862.73399

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

What is TimeTuple?

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below:

|  |  |  |
| --- | --- | --- |
| **Index** | **Field** | **Values** |
| 0 | 4-digit year | 2008 |
| 1 | Month | 1 to 12 |
| 2 | Day | 1 to 31 |
| 3 | Hour | 0 to 23 |
| 4 | Minute | 0 to 59 |
| 5 | Second | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | Day of Week | 0 to 6 (0 is Monday) |
| 7 | Day of year | 1 to 366 (Julian day) |
| 8 | Daylight savings | -1, 0, 1, -1 means library determines DST |

The above tuple is equivalent to **struct\_time** structure. This structure has following attributes:

|  |  |  |
| --- | --- | --- |
| **Index** | **Attributes** | **Values** |
| 0 | tm\_year | 2008 |
| 1 | tm\_mon | 1 to 12 |
| 2 | tm\_mday | 1 to 31 |
| 3 | tm\_hour | 0 to 23 |
| 4 | tm\_min | 0 to 59 |
| 5 | tm\_sec | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | tm\_wday | 0 to 6 (0 is Monday) |
| 7 | tm\_yday | 1 to 366 (Julian day) |
| 8 | tm\_isdst | -1, 0, 1, -1 means library determines DST |

Getting current time -:

To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all nine items valid.

#!/usr/bin/python  
import time;

localtime = time.localtime(time.time())  
print "Local current time :", localtime

This would produce the following result, which could be formatted in any other presentable form:

Local current time : time.struct\_time(tm\_year=2013, tm\_mon=7,   
tm\_mday=17, tm\_hour=21, tm\_min=26, tm\_sec=3, tm\_wday=2, tm\_yday=198, tm\_isdst=0)

Getting formatted time -:

You can format any time as per your requirement, but simple method to get time in readable format is asctime():

#!/usr/bin/python  
import time;

localtime = time.asctime( time.localtime(time.time()) )  
print "Local current time :", localtime

This would produce the following result:

Local current time : Tue Jan 13 10:17:09 2009

Getting calendar for a month -:

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month ( Jan 2008 ):

#!/usr/bin/python  
import calendar

cal = calendar.month(2008, 1)  
print "Here is the calendar:"  
print cal;

This would produce the following result:

Here is the calendar:  
 January 2008  
Mo Tu We Th Fr Sa Su  
 1 2 3 4 5 6  
 7 8 9 10 11 12 13  
14 15 16 17 18 19 20  
21 22 23 24 25 26 27  
28 29 30 31

The *time* Module:

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [time.altzone](http://www.tutorialspoint.com/python/time_altzone.htm)  The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero. |
| 2 | [time.asctime([tupletime])](http://www.tutorialspoint.com/python/time_asctime.htm)  Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'. |
| 3 | [time.clock( )](http://www.tutorialspoint.com/python/time_clock.htm)  Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time(). |
| 4 | [time.ctime([secs])](http://www.tutorialspoint.com/python/time_ctime.htm)  Like asctime(localtime(secs)) and without arguments is like asctime( ) |
| 5 | [time.gmtime([secs])](http://www.tutorialspoint.com/python/time_gmtime.htm)  Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm\_isdst is always 0 |
| 6 | [time.localtime([secs])](http://www.tutorialspoint.com/python/time_localtime.htm)  Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm\_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules). |
| 7 | [time.mktime(tupletime)](http://www.tutorialspoint.com/python/time_mktime.htm)  Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch. |
| 8 | [time.sleep(secs)](http://www.tutorialspoint.com/python/time_sleep.htm)  Suspends the calling thread for secs seconds. |
| 9 | [time.strftime(fmt[,tupletime])](http://www.tutorialspoint.com/python/time_strftime.htm)  Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt. |
| 10 | [time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')](http://www.tutorialspoint.com/python/time_strptime.htm)  Parses str according to format string fmt and returns the instant in time-tuple format. |
| 11 | [time.time( )](http://www.tutorialspoint.com/python/time_time.htm)  Returns the current time instant, a floating-point number of seconds since the epoch. |
| 12 | [time.tzset()](http://www.tutorialspoint.com/python/time_tzset.htm)  Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. |

There are following two important attributes available with time module:

|  |  |
| --- | --- |
| **SN** | **Attribute with Description** |
| 1 | **time.timezone**  Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa). |
| 2 | **time.tzname**  Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively. |

The *calendar* Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function.

Here is a list of functions available with the *calendar* module:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | **calendar.calendar(year,w=2,l=1,c=6)**  Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21\*w+18+2\*c. l is the number of lines for each week. |
| 2 | **calendar.firstweekday( )**  Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday. |
| 3 | **calendar.isleap(year)**  Returns True if year is a leap year; otherwise, False. |
| 4 | **calendar.leapdays(y1,y2)**  Returns the total number of leap days in the years within range(y1,y2). |
| 5 | **calendar.month(year,month,w=2,l=1)**  Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7\*w+6. l is the number of lines for each week. |
| 6 | **calendar.monthcalendar(year,month)**  Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up. |
| 7 | **calendar.monthrange(year,month)**  Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12. |
| 8 | **calendar.prcal(year,w=2,l=1,c=6)**  Like print calendar.calendar(year,w,l,c). |
| 9 | **calendar.prmonth(year,month,w=2,l=1)**  Like print calendar.month(year,month,w,l). |
| 10 | **calendar.setfirstweekday(weekday)**  Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday). |
| 11 | **calendar.timegm(tupletime)**  The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch. |
| 12 | **calendar.weekday(year,month,day)**  Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December). |

Other Modules & Functions:

If you are interested, then here you would find a list of other important modules and functions to play with date & time in Python:

* + [The *datetime* Module](http://docs.python.org/library/datetime.html#module-datetime)
  + [The *pytz*Module](http://www.twinsun.com/tz/tz-link.htm)
  + [The *dateutil* Module](http://labix.org/python-dateutil)

Pasted from <<http://www.tutorialspoint.com/python/python_date_time.htm>>

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* + Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
  + Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
  + The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
  + The code block within every function starts with a colon (:) and is indented.
  + The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

def functionname( parameters ):  
 "function\_docstring"  
 function\_suite  
 return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example:

Here is the simplest form of a Python function. This function takes a string as input parameter and prints it on standard screen.

def printme( str ):  
 "This prints a passed string into this function"  
 print str  
 return

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function:

#!/usr/bin/python

# Function definition is here  
def printme( str ):  
 "This prints a passed string into this function"  
 print str;  
 return;

# Now you can call printme function  
printme("I'm first call to user defined function!");  
printme("Again second call to the same function");

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

I'm first call to user defined function!  
Again second call to the same function

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

#!/usr/bin/python

# Function definition is here  
def changeme( mylist ):  
 "This changes a passed list into this function"  
 mylist.append([1,2,3,4]);  
 print "Values inside the function: ", mylist  
 return

# Now you can call changeme function  
mylist = [10,20,30];  
changeme( mylist );  
print "Values outside the function: ", mylist

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result:

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]  
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

#!/usr/bin/python

# Function definition is here  
def changeme( mylist ):  
 "This changes a passed list into this function"  
 mylist = [1,2,3,4]; # This would assig new reference in mylist  
 print "Values inside the function: ", mylist  
 return

# Now you can call changeme function  
mylist = [10,20,30];  
changeme( mylist );  
print "Values outside the function: ", mylist

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result:

Values inside the function: [1, 2, 3, 4]  
Values outside the function: [10, 20, 30]

Function Arguments:

You can call a function by using the following types of formal arguments:

* + Required arguments
  + Keyword arguments
  + Default arguments
  + Variable-length arguments

Required arguments:

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it would give a syntax error as follows:

#!/usr/bin/python

# Function definition is here  
def printme( str ):  
 "This prints a passed string into this function"  
 print str;  
 return;

# Now you can call printme function  
printme();

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

Traceback (most recent call last):  
 File "test.py", line 11, in <module>  
 printme();  
TypeError: printme() takes exactly 1 argument (0 given)

Keyword arguments:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways:

#!/usr/bin/python

# Function definition is here  
def printme( str ):  
 "This prints a passed string into this function"  
 print str;  
 return;

# Now you can call printme function  
printme( str = "My string");

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

My string

Following example gives more clear picture. Note, here order of the parameter does not matter.

#!/usr/bin/python

# Function definition is here  
def printinfo( name, age ):  
 "This prints a passed info into this function"  
 print "Name: ", name;  
 print "Age ", age;  
 return;

# Now you can call printinfo function  
printinfo( age=50, name="miki" );

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

Name: miki  
Age 50

Default arguments:

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. Following example gives an idea on default arguments, it would print default age if it is not passed:

#!/usr/bin/python

# Function definition is here  
def printinfo( name, age = 35 ):  
 "This prints a passed info into this function"  
 print "Name: ", name;  
 print "Age ", age;  
 return;

# Now you can call printinfo function  
printinfo( age=50, name="miki" );  
printinfo( name="miki" );

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

Name: miki  
Age 50  
Name: miki  
Age 35

Variable-length arguments:

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

The general syntax for a function with non-keyword variable arguments is this:

def functionname([formal\_args,] \*var\_args\_tuple ):  
 "function\_docstring"  
 function\_suite  
 return [expression]

An asterisk (\*) is placed before the variable name that will hold the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example:

#!/usr/bin/python

# Function definition is here  
def printinfo( arg1, \*vartuple ):  
 "This prints a variable passed arguments"  
 print "Output is: "  
 print arg1  
 for var in vartuple:  
 print var  
 return;

# Now you can call printinfo function  
printinfo( 10 );  
printinfo( 70, 60, 50 );

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

Output is:  
10  
Output is:  
70  
60  
50

The *Anonymous* Functions:

You can use the *lambda* keyword to create small anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.

* + Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
  + An anonymous function cannot be a direct call to print because lambda requires an expression.
  + Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
  + Although it appears that lambda's are a one-line version of a function, they are not equivalent to*inline* statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax:

The syntax of *lambda* functions contains only a single statement, which is as follows:

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works:

#!/usr/bin/python

# Function definition is here  
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function  
print "Value of total : ", sum( 10, 20 )  
print "Value of total : ", sum( 20, 20 )

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

Value of total : 30  
Value of total : 40

The *return* Statement:

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value, but if you like you can return a value from a function as follows:

#!/usr/bin/python

# Function definition is here  
def sum( arg1, arg2 ):  
 # Add both the parameters and return them."  
 total = arg1 + arg2  
 print "Inside the function : ", total  
 return total;

# Now you can call sum function  
total = sum( 10, 20 );  
print "Outside the function : ", total

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

Inside the function : 30  
Outside the function : 30

Scope of Variables:

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

* + Global variables
  + Local variables

Global vs. Local variables:

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example:

#!/usr/bin/python

total = 0; # This is global variable.  
# Function definition is here  
def sum( arg1, arg2 ):  
 # Add both the parameters and return them."  
 total = arg1 + arg2; # Here total is local variable.  
 print "Inside the function local total : ", total  
 return total;

# Now you can call sum function  
sum( 10, 20 );  
print "Outside the function global total : ", total

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

Inside the function local total : 30  
Outside the function global total : 0

Pasted from <<http://www.tutorialspoint.com/python/python_functions.htm>>

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example:

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

def print\_func( par ):  
 print "Hello : ", par  
 return

The *import* Statement:

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax:

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script:

#!/usr/bin/python

# Import module support  
import support

# Now you can call defined function that module as follows  
support.print\_func("Zara")

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

Hello : Zara

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax:

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement:

from fib import fibonacci

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The *from...import \** Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement:

from modname import \*

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules:

When you import a module, the Python interpreter searches for the module in the following sequences:

* + The current directory.
  + If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
  + If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The *PYTHONPATH* Variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

set PYTHONPATH=c:\python20\lib;

And here is a typical PYTHONPATH from a UNIX system:

set PYTHONPATH=/usr/local/lib/python

Namespaces and Scoping:

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

#!/usr/bin/python

Money = 2000  
def AddMoney():  
 # Uncomment the following line to fix the code:  
 # global Money  
 Money = Money + 1

print Money  
AddMoney()  
print Money

The dir( ) Function:

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example:

#!/usr/bin/python

# Import built-in module math  
import math

content = dir(math)

print content;

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

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',   
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',   
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',  
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',   
'sqrt', 'tan', 'tanh']

Here, the special string variable *\_\_name\_\_* is the module's name, and *\_\_file\_\_* is the filename from which the module was loaded.

The *globals()* and *locals()* Functions:

The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

The *reload()* Function:

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this:

reload(module\_name)

Here, *module\_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following:

reload(hello)

Packages in Python:

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code:

#!/usr/bin/python

def Pots():  
 print "I'm Pots Phone"

Similar way, we have another two files having different functions with the same name as above:

* + *Phone/Isdn.py* file having function Isdn()
  + *Phone/G3.py* file having function G3()

Now, create one more file \_\_init\_\_.py in *Phone* directory:

* + Phone/\_\_init\_\_.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in \_\_init\_\_.py as follows:

from Pots import Pots  
from Isdn import Isdn  
from G3 import G3

After you've added these lines to \_\_init\_\_.py, you have all of these classes available when you've imported the Phone package.

#!/usr/bin/python

# Now import your Phone Package.  
import Phone

Phone.Pots()  
Phone.Isdn()  
Phone.G3()

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

I'm Pots Phone  
I'm 3G Phone  
I'm ISDN Phone

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

Pasted from <<http://www.tutorialspoint.com/python/python_modules.htm>>

This chapter will cover all the basic I/O functions available in Python. For more functions, please refer to standard Python documentation.

Printing to the Screen:

The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows:

#!/usr/bin/python

print "Python is really a great language,", "isn't it?";

This would produce the following result on your standard screen:

Python is really a great language, isn't it?

Reading Keyboard Input:

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are:

* + raw\_input
  + input

The *raw\_input* Function:

The *raw\_input([prompt])* function reads one line from standard input and returns it as a string (removing the trailing newline).

#!/usr/bin/python

str = raw\_input("Enter your input: ");  
print "Received input is : ", str

This would prompt you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this:

Enter your input: Hello Python  
Received input is : Hello Python

The *input* Function:

The *input([prompt])* function is equivalent to raw\_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

#!/usr/bin/python

str = input("Enter your input: ");  
print "Received input is : ", str

This would produce the following result against the entered input:

Enter your input: [x\*5 for x in range(2,10,2)]  
Recieved input is : [10, 20, 30, 40]

Opening and Closing Files:

Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do your most of the file manipulation using a **file** object.

The *open* Function:

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

SYNTAX:

file object = open(file\_name [, access\_mode][, buffering])

Here is paramters' detail:

* + **file\_name:** The file\_name argument is a string value that contains the name of the file that you want to access.
  + **access\_mode:** The access\_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
  + **buffering:** If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Here is a list of the different modes of opening a file:

|  |  |
| --- | --- |
| **Modes** | **Description** |
| r | Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| rb | Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| r+ | Opens a file for both reading and writing. The file pointer will be at the beginning of the file. |
| rb+ | Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file. |
| w | Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| wb | Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| w+ | Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| wb+ | Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| a | Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| ab | Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| a+ | Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| ab+ | Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

The *file* object attributes:

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| file.closed | Returns true if file is closed, false otherwise. |
| file.mode | Returns access mode with which file was opened. |
| file.name | Returns name of the file. |
| file.softspace | Returns false if space explicitly required with print, true otherwise. |

EXAMPLE:

#!/usr/bin/python

# Open a file  
fo = open("foo.txt", "wb")  
print "Name of the file: ", fo.name  
print "Closed or not : ", fo.closed  
print "Opening mode : ", fo.mode  
print "Softspace flag : ", fo.softspace

This would produce the following result:

Name of the file: foo.txt  
Closed or not : False  
Opening mode : wb  
Softspace flag : 0

The *close()* Method:

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

SYNTAX:

fileObject.close();

**EXAMPLE:**

#!/usr/bin/python

# Open a file  
fo = open("foo.txt", "wb")  
print "Name of the file: ", fo.name

# Close opend file  
fo.close()

This would produce the following result:

Name of the file: foo.txt

Reading and Writing Files:

The *file* object provides a set of access methods to make our lives easier. We would see how to use*read()* and *write()* methods to read and write files.

The *write()* Method:

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string:

SYNTAX:

fileObject.write(string);

Here, passed parameter is the content to be written into the opened file.

EXAMPLE:

#!/usr/bin/python

# Open a file  
fo = open("foo.txt", "wb")  
fo.write( "Python is a great language.\nYeah its great!!\n");

# Close opend file  
fo.close()

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

Python is a great language.  
Yeah its great!!

The *read()* Method:

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.

SYNTAX:

fileObject.read([count]);

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

EXAMPLE:

Let's take a file *foo.txt*, which we have created above.

#!/usr/bin/python

# Open a file  
fo = open("foo.txt", "r+")  
str = fo.read(10);  
print "Read String is : ", str  
# Close opend file  
fo.close()

This would produce the following result:

Read String is : Python is

File Positions:

The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

EXAMPLE:

Let's take a file *foo.txt*, which we have created above.

#!/usr/bin/python

# Open a file  
fo = open("foo.txt", "r+")  
str = fo.read(10);  
print "Read String is : ", str

# Check current position  
position = fo.tell();  
print "Current file position : ", position

# Reposition pointer at the beginning once again  
position = fo.seek(0, 0);  
str = fo.read(10);  
print "Again read String is : ", str  
# Close opend file  
fo.close()

This would produce the following result:

Read String is : Python is  
Current file position : 10  
Again read String is : Python is

Renaming and Deleting Files:

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

The rename() Method:

The *rename()* method takes two arguments, the current filename and the new filename.

SYNTAX:

os.rename(current\_file\_name, new\_file\_name)

**EXAMPLE:**

Following is the example to rename an existing file *test1.txt*:

#!/usr/bin/python  
import os

# Rename a file from test1.txt to test2.txt  
os.rename( "test1.txt", "test2.txt" )

The *remove()* Method:

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

SYNTAX:

os.remove(file\_name)

**EXAMPLE:**

Following is the example to delete an existing file *test2.txt*:

#!/usr/bin/python  
import os

# Delete file test2.txt  
os.remove("text2.txt")

Directories in Python:

All files are contained within various directories, and Python has no problem handling these too. The**os** module has several methods that help you create, remove and change directories.

The *mkdir()* Method:

You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

SYNTAX:

os.mkdir("newdir")

**EXAMPLE:**

Following is the example to create a directory *test* in the current directory:

#!/usr/bin/python  
import os

# Create a directory "test"  
os.mkdir("test")

The *chdir()* Method:

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

SYNTAX:

os.chdir("newdir")

**EXAMPLE:**

Following is the example to go into "/home/newdir" directory:

#!/usr/bin/python  
import os

# Changing a directory to "/home/newdir"  
os.chdir("/home/newdir")

The *getcwd()* Method:

The *getcwd()* method displays the current working directory.

SYNTAX:

os.getcwd()

**EXAMPLE:**

Following is the example to give current directory:

#!/usr/bin/python  
import os

# This would give location of the current directory  
os.getcwd()

The *rmdir()* Method:

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

SYNTAX:

os.rmdir('dirname')

**EXAMPLE:**

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

#!/usr/bin/python  
import os

# This would remove "/tmp/test" directory.  
os.rmdir( "/tmp/test" )

File & Directory Related Methods:

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows:

* + [File Object Methods](http://www.tutorialspoint.com/python/file_methods.htm): The *file* object provides functions to manipulate files.
  + [OS Object Methods](http://www.tutorialspoint.com/python/os_file_methods.htm): This provides methods to process files as well as directories.

Pasted from <<http://www.tutorialspoint.com/python/python_files_io.htm>>

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:

* + **Exception Handling:** This would be covered in this tutorial. Here is a list standard Exceptions available in Python: [Standard Exceptions](http://www.tutorialspoint.com/python/standard_exceptions.htm).
  + **Assertions:** This would be covered in [Assertions in Python](http://www.tutorialspoint.com/python/assertions_in_python.htm) tutorial.

What is Exception?

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Handling an exception:

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

SYNTAX:

Here is simple syntax of *try....except...else* blocks:

try:  
 You do your operations here;  
 ......................  
except *ExceptionI*:  
 If there is ExceptionI, then execute this block.  
except *ExceptionII*:  
 If there is ExceptionII, then execute this block.  
 ......................  
else:  
 If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax:

* + A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
  + You can also provide a generic except clause, which handles any exception.
  + After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
  + The else-block is a good place for code that does not need the try: block's protection.

EXAMPLE:

Here is simple example, which opens a file and writes the content in the file and comes out gracefully because there is no problem at all:

#!/usr/bin/python

try:  
 fh = open("testfile", "w")  
 fh.write("This is my test file for exception handling!!")  
except IOError:  
 print "Error: can\'t find file or read data"  
else:  
 print "Written content in the file successfully"  
 fh.close()

This will produce the following result:

Written content in the file successfully

**EXAMPLE:**

Here is one more simple example, which tries to open a file where you do not have permission to write in the file, so it raises an exception:

#!/usr/bin/python

try:  
 fh = open("testfile", "r")  
 fh.write("This is my test file for exception handling!!")  
except IOError:  
 print "Error: can\'t find file or read data"  
else:  
 print "Written content in the file successfully"

This will produce the following result:

Error: can't find file or read data

The *except* clause with no exceptions:

You can also use the except statement with no exceptions defined as follows:

try:  
 You do your operations here;  
 ......................  
except:  
 If there is any exception, then execute this block.  
 ......................  
else:  
 If there is no exception then execute this block.

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* clause with multiple exceptions:

You can also use the same *except* statement to handle multiple exceptions as follows:

try:  
 You do your operations here;  
 ......................  
except(Exception1[, Exception2[,...ExceptionN]]]):  
 If there is any exception from the given exception list,   
 then execute this block.  
 ......................  
else:  
 If there is no exception then execute this block.

The try-finally clause:

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

try:  
 You do your operations here;  
 ......................  
 Due to any exception, this may be skipped.  
finally:  
 This would always be executed.  
 ......................

Note that you can provide except clause(s), or a finally clause, but not both. You can not use *else* clause as well along with a finally clause.

EXAMPLE:

#!/usr/bin/python

try:  
 fh = open("testfile", "w")  
 fh.write("This is my test file for exception handling!!")  
finally:  
 print "Error: can\'t find file or read data"

If you do not have permission to open the file in writing mode, then this will produce the following result:

Error: can't find file or read data

Same example can be written more cleanly as follows:

#!/usr/bin/python

try:  
 fh = open("testfile", "w")  
 try:  
 fh.write("This is my test file for exception handling!!")  
 finally:  
 print "Going to close the file"  
 fh.close()  
except IOError:  
 print "Error: can\'t find file or read data"

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the finally block are executed, the exception is raised again and is handled in the*except* statements if present in the next higher layer of the *try-except* statement.

Argument of an Exception:

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows:

try:  
 You do your operations here;  
 ......................  
except *ExceptionType, Argument*:  
 You can print value of Argument here...

If you are writing the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable will receive the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

EXAMPLE:

Following is an example for a single exception:

#!/usr/bin/python

# Define a function here.  
def temp\_convert(var):  
 try:  
 return int(var)  
 except ValueError, Argument:  
 print "The argument does not contain numbers\n", Argument

# Call above function here.  
temp\_convert("xyz");

This would produce the following result:

The argument does not contain numbers  
invalid literal for int() with base 10: 'xyz'

Raising an exceptions:

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise**statement.

SYNTAX:

raise [Exception [, args [, traceback]]]

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

EXAMPLE:

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:

def functionName( level ):  
 if level < 1:  
 raise "Invalid level!", level  
 # The code below to this would not be executed  
 # if we raise the exception

**Note:** In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write our except clause as follows:

try:  
 Business Logic here...  
except "Invalid level!":  
 Exception handling here...  
else:  
 Rest of the code here...

User-Defined Exceptions:

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from*RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

class Networkerror(RuntimeError):  
 def \_\_init\_\_(self, arg):  
 self.args = arg

So once you defined above class, you can raise your exception as follows:

try:  
 raise Networkerror("Bad hostname")  
except Networkerror,e:  
 print e.args

Pasted from <<http://www.tutorialspoint.com/python/python_exceptions.htm>>

Python has been an object-oriented language from day one. Because of 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 don't 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 small introduction of Object-Oriented Programming (OOP) to bring you at speed:

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 aren't 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 (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's 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 the 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 would be shared among all instances of a 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 don't 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/python

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.age = 7 # Add an 'age' attribute.  
emp1.age = 8 # Modify 'age' attribute.  
del emp1.age # Delete 'age' attribute.

Instead of using the normal statements to access attributes, you can use 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, 'age') # Returns true if 'age' attribute exists  
getattr(emp1, 'age') # Returns value of 'age' attribute  
setattr(emp1, 'age', 8) # Set attribute 'age' at 8  
delattr(empl, 'age') # Delete attribute 'age'

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's try to access all these attributes:

#!/usr/bin/python

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

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\_\_: ()  
Employee.\_\_dict\_\_: {'\_\_module\_\_': '\_\_main\_\_', 'displayCount':  
<function displayCount at 0xb7c84994>, 'empCount': 2,   
'displayEmployee': <function displayEmployee at 0xb7c8441c>,   
'\_\_doc\_\_': 'Common base class for all employees',   
'\_\_init\_\_': <function \_\_init\_\_ at 0xb7c846bc>}

Destroying Objects (Garbage Collection):

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

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

An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary). The object's reference count decreases when it's 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 won't notice when the garbage collector destroys an orphaned instance and reclaims its space. But 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 nonmemory resources used by an instance.

EXAMPLE:

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

#!/usr/bin/python

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 following result:

3083401324 3083401324 3083401324  
Point destroyed

**Note:** Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement. Kindly check [Python - Modules](http://www.tutorialspoint.com/python/python_modules.htm) chapter for more details on importing modules and classes.

Class Inheritance:

Instead of starting from scratch, you can create a class by deriving it from a preexisting 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 are given after the class name:

class SubClassName (ParentClass1[, ParentClass2, ...]):  
 'Optional class documentation string'  
 class\_suite

**EXAMPLE:**

#!/usr/bin/python

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

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 because you may want special or different functionality in your subclass.

EXAMPLE:

#!/usr/bin/python

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:

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

|  |  |
| --- | --- |
| **SN** | **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 : *dell 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've 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/python

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. For these cases, you can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders.

EXAMPLE:

#!/usr/bin/python

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 would work for you:

.........................  
print counter.\_JustCounter\_\_secretCount

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

1  
2  
2

Pasted from <<http://www.tutorialspoint.com/python/python_classes_objects.htm>>

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. But 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'**.

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:

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| pattern | This is the regular expression to be matched. |
| string | This is the string, which would be searched to match the pattern at the beginning of string. |
| 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 would use *group(num)*or *groups()*function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **Match Object Methods** | **Description** |
| group(num=0) | This method returns entire match (or specific subgroup num) |
| groups() | This method returns all matching subgroups in a tuple (empty if there weren't any) |

EXAMPLE:

#!/usr/bin/python  
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 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:

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| pattern | This is the regular expression to be matched. |
| string | This is the string, which would be searched to match the pattern anywhere in the string. |
| 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 would use *group(num)*or *groups()*function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **Match Object Methods** | **Description** |
| group(num=0) | This method returns entire match (or specific subgroup num) |
| groups() | This method returns all matching subgroups in a tuple (empty if there weren't any) |

EXAMPLE:

#!/usr/bin/python  
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 following result:

matchObj.group() : Cats are smarter than dogs  
matchObj.group(1) : Cats  
matchObj.group(2) : smarter

Matching vs 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/python  
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:

Some 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* provided. This method would return modified string.

EXAMPLE:

Following is the example:

#!/usr/bin/python  
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:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| re.I | Performs case-insensitive matching. |
| 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). |
| 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). |
| re.S | Makes a period (dot) match any character, including a newline. |
| re.U | Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B. |
| 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 control characters, **(+ ? . \* ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python:

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| ^ | Matches beginning of line. |
| $ | Matches end of line. |
| . | Matches any single character except newline. Using m option allows it to match newline as well. |
| [...] | Matches any single character in brackets. |
| [^...] | Matches any single character not in brackets |
| re\* | Matches 0 or more occurrences of preceding expression. |
| re+ | Matches 1 or more occurrence of preceding expression. |
| re? | Matches 0 or 1 occurrence of preceding expression. |
| re{ n} | Matches exactly n number of occurrences of preceding expression. |
| re{ n,} | Matches n or more occurrences of preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of preceding expression. |
| a| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers matched text. |
| (?imx) | Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| (?-imx) | Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| (?: re) | Groups regular expressions without remembering matched text. |
| (?imx: re) | Temporarily toggles on i, m, or x options within parentheses. |
| (?-imx: re) | Temporarily toggles off i, m, or x options within parentheses. |
| (?#...) | Comment. |
| (?= re) | Specifies position using a pattern. Doesn't have a range. |
| (?! re) | Specifies position using pattern negation. Doesn't have a range. |
| (?> re) | Matches independent pattern without backtracking. |
| \w | Matches word characters. |
| \W | Matches nonword characters. |
| \s | Matches whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches nonwhitespace. |
| \d | Matches digits. Equivalent to [0-9]. |
| \D | Matches nondigits. |
| \A | Matches beginning of string. |
| \Z | Matches end of string. If a newline exists, it matches just before newline. |
| \z | Matches end of string. |
| \G | Matches point where last match finished. |
| \b | Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets. |
| \B | Matches nonword boundaries. |
| \n, \t, etc. | Matches newlines, carriage returns, tabs, etc. |
| \1...\9 | Matches nth grouped subexpression. |
| \10 | Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code. |

Regular-expression Examples

Literal characters:

|  |  |
| --- | --- |
| **Example** | **Description** |
| python | Match "python". |

Character classes:

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

Special Character Classes:

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

Repetition Cases:

|  |  |
| --- | --- |
| **Example** | **Description** |
| ruby? | Match "rub" or "ruby": the y is optional |
| ruby\* | Match "rub" plus 0 or more ys |
| ruby+ | Match "rub" plus 1 or more ys |
| \d{3} | Match exactly 3 digits |
| \d{3,} | Match 3 or more digits |
| \d{3,5} | Match 3, 4, or 5 digits |

Nongreedy repetition:

This matches the smallest number of repetitions:

|  |  |
| --- | --- |
| **Example** | **Description** |
| <.\*> | Greedy repetition: matches "<python>perl>" |
| <.\*?> | Nongreedy: matches "<python>" in "<python>perl>" |

Grouping with parentheses:

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

Backreferences:

This matches a previously matched group again:

|  |  |
| --- | --- |
| **Example** | **Description** |
| ([Pp])ython&\1ails | Match python&pails or Python&Pails |
| (['"])[^\1]\*\1 | Single or double-quoted string. \1 matches whatever the 1st group matched . \2 matches whatever the 2nd group matched, etc. |

Alternatives:

|  |  |
| --- | --- |
| **Example** | **Description** |
| python|perl | Match "python" or "perl" |
| rub(y|le)) | Match "ruby" or "ruble" |
| Python(!+|\?) | "Python" followed by one or more ! or one ? |

Anchors:

This needs to specify match position.

|  |  |
| --- | --- |
| **Example** | **Description** |
| ^Python | Match "Python" at the start of a string or internal line |
| Python$ | Match "Python" at the end of a string or line |
| \APython | Match "Python" at the start of a string |
| Python\Z | Match "Python" at the end of a string |
| \bPython\b | Match "Python" at a word boundary |
| \brub\B | \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone |
| Python(?=!) | Match "Python", if followed by an exclamation point |
| Python(?!!) | Match "Python", if not followed by an exclamation point |

Special syntax with parentheses:

|  |  |
| --- | --- |
| **Example** | **Description** |
| R(?#comment) | Matches "R". All the rest is a comment |
| R(?i)uby | Case-insensitive while matching "uby" |
| R(?i:uby) | Same as above |
| rub(?:y|le)) | Group only without creating \1 backreference |

Pasted from <<http://www.tutorialspoint.com/python/python_reg_expressions.htm>>

What is 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 and NCSA defines CGI is as follows:
  + 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, lets 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 will parse the URL and will look for the filename in if it finds that file then sends it back to the browser, otherwise sends an error message indicating that you have 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 & 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 will 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, I assumed 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](http://www.tutorialspoint.com/cgi-bin/hello.py). This file is being 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>'

If you click hello.py, then this produces the following output:

Hello Word! This is my first CGI program

This hello.py script is a simple Python script, which is writing 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 specifiy the content type to be displayed on the browser screen.

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.

|  |  |
| --- | --- |
| **Header** | **Description** |
| Content-type: | A MIME string defining the format of the file being returned. Example is Content-type:text/html |
| Expires: Date | The date the information becomes invalid. This should be used by the browser to decide when a page needs to be refreshed. A valid date string should be in the format 01 Jan 1998 12:00:00 GMT. |
| Location: URL | The URL that should be returned instead of the URL requested. You can use this field to redirect a request to any file. |
| Last-modified: Date | The date of last modification of the resource. |
| 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. |
| Set-Cookie: String | Set the cookie passed through the *string* |

CGI Environment Variables

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

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| CONTENT\_TYPE | The data type of the content. Used when the client is sending attached content to the server. For example, file upload, etc. |
| CONTENT\_LENGTH | The length of the query information. It's available only for POST requests. |
| HTTP\_COOKIE | Returns the set cookies in the form of key & value pair. |
| HTTP\_USER\_AGENT | The User-Agent request-header field contains information about the user agent originating the request. Its name of the web browser. |
| PATH\_INFO | The path for the CGI script. |
| QUERY\_STRING | The URL-encoded information that is sent with GET method request. |
| REMOTE\_ADDR | The IP address of the remote host making the request. This can be useful for logging or for authentication purpose. |
| 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. |
| REQUEST\_METHOD | The method used to make the request. The most common methods are GET and POST. |
| SCRIPT\_FILENAME | The full path to the CGI script. |
| SCRIPT\_NAME | The name of the CGI script. |
| SERVER\_NAME | The server's hostname or IP Address |
| 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 limtation: 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 will pass two values to hello\_get.py program using GET method.

[/cgi-bin/hello\_get.py?first\_name=ZARA&last\_name=ALI](http://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

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same CGI script hello\_get.py to handle this imput.

<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 are going to use same CGI script hello\_get.py to handle this imput.

<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:

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:

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. But for a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. But 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 will be sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. So cookies setting will be 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.

Here, 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 will produce 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 will create 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 are running above script on Unix/Linux, then you would have 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 will 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 will be achieved through HTTP header. This HTTP header will 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 will be 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 hear.  
fo = open("foo.txt", "rb")

str = fo.read();  
print str

# Close opend file  
fo.close()

Hope you enjoyed this tutorial. If yes, please send me your feedback at: [Contact Us](http://www.tutorialspoint.com/about/contact_us.htm)

Pasted from <<http://www.tutorialspoint.com/python/python_cgi_programming.htm>>

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:

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

Here is the list of available Python database interfaces: [Python Database Interfaces and APIs](http://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

We would learn all the concepts using MySQL, so let's talk about MySQLdb module only.

What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

How do I install the MySQLdb?

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

#!/usr/bin/python

import MySQLdb

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 MySQLdb  
ImportError: No module named MySQLdb

To install MySQLdb module, download it from [MySQLdb Download](http://sourceforge.net/projects/mysql-python) page and proceed as follows:

$ gunzip MySQL-python-1.2.2.tar.gz  
$ tar -xvf MySQL-python-1.2.2.tar  
$ cd MySQL-python-1.2.2  
$ python setup.py build  
$ python setup.py install

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

Database Connection:

Before connecting to a MySQL database, make sure of the followings:

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

EXAMPLE:

Following is the example of connecting with MySQL database "TESTDB"

#!/usr/bin/python

import MySQLdb

# Open database connection  
db = MySQLdb.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 is producing the following result in my Linux machine.

Database version : 5.0.45

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 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:

First, let's create Database table EMPLOYEE:

#!/usr/bin/python

import MySQLdb

# Open database connection  
db = MySQLdb.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:

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

EXAMPLE:

Following is the example, which executes SQL *INSERT* statement to create a record into EMPLOYEE table:

#!/usr/bin/python

import MySQLdb

# Open database connection  
db = MySQLdb.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()

Above example can be written as follows to create SQL queries dynamically:

#!/usr/bin/python

import MySQLdb

# Open database connection  
db = MySQLdb.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:**

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 databasse means to fetch some useful information from the database.

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

* + **fetchone():** This method 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():** This method fetches all the rows in a result set. If some rows have already been extracted from the result set, the fetchall() method 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:

Following is the procedure to query all the records from EMPLOYEE table having salary more than 1000:

#!/usr/bin/python

import MySQLdb

# Open database connection  
db = MySQLdb.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 fecth data"

# disconnect from server  
db.close()

This will produce the following result:

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

Update Operation:

UPDATE Operation on any databasse means to update one or more records, which are already available in the database. Following is the procedure to update all the records having SEX as 'M'. Here, we will increase AGE of all the males by one year.

EXAMPLE:

#!/usr/bin/python

import MySQLdb

# Open database connection  
db = MySQLdb.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/python

import MySQLdb

# Open database connection  
db = MySQLdb.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 should 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 have seen how we have implemented transations. Here is again 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 the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

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

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

db.rollback()

Disconnecting Database:

To disconnect Database connection, use 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 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.

|  |  |
| --- | --- |
| **Exception** | **Description** |
| Warning | Used for non-fatal issues. Must subclass StandardError. |
| Error | Base class for errors. Must subclass StandardError. |
| InterfaceError | Used for errors in the database module, not the database itself. Must subclass Error. |
| DatabaseError | Used for errors in the database. Must subclass Error. |
| DataError | Subclass of DatabaseError that refers to errors in the data. |
| 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. |
| IntegrityError | Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys. |
| InternalError | Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active. |
| ProgrammingError | Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you. |
| 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.

Pasted from <<http://www.tutorialspoint.com/python/python_database_access.htm>>

Python provides two levels of access to 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 tutorial gives you understanding on 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:

|  |  |
| --- | --- |
| **Term** | **Description** |
| domain | The family of protocols that will be used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on. |
| type | The type of communications between the two endpoints, typically SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols. |
| protocol | Typically zero, this may be used to identify a variant of a protocol within a domain and type. |
| 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. |
| 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 *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 required functions to create your client or server program. Following is the list of functions required:

Server Socket Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.bind() | This method binds address (hostname, port number pair) to socket. |
| s.listen() | This method sets up and start TCP listener. |
| s.accept() | This passively accept TCP client connection, waiting until connection arrives (blocking). |

Client Socket Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.connect() | This method actively initiates TCP server connection. |

General Socket Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.recv() | This method receives TCP message |
| s.send() | This method transmits TCP message |
| s.recvfrom() | This method receives UDP message |
| s.sendto() | This method transmits UDP message |
| s.close() | This method closes socket |
| 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 **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/python # This is server.py file

import socket # Import socket module

s = socket.socket() # Create a socket object  
host = socket.gethostname() # Get local machine name  
port = 12345 # Reserve a port for your service.  
s.bind((host, port)) # Bind to the port

s.listen(5) # Now wait for client connection.  
while True:  
 c, addr = s.accept() # Establish connection with client.  
 print 'Got connection from', addr  
 c.send('Thank you for connecting')  
 c.close() # Close the connection

A Simple Client:

Now we will write a very simple client program which will open a connection to a given port 12345 and given host. This is very simple to create a socket client using 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.

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/python # This is client.py file

import socket # Import socket module

s = socket.socket() # Create a socket object  
host = socket.gethostname() # Get local machine name  
port = 12345 # Reserve a port for your service.

s.connect((host, port))  
print s.recv(1024)  
s.close # Close the socket when done

Now run this server.py in background and then run 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

This would produce following result:

Got connection from ('127.0.0.1', 48437)  
Thank you for connecting

Python Internet modules

A list of some important modules which could be used in Python Network/Internet programming.

|  |  |  |  |
| --- | --- | --- | --- |
| **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.

Further Readings:

I have given you a quick start with Socket Programming. It's a big subject so its recommended to go through the following link to find more detail on:

* + [Unix Socket Programming](http://www.tutorialspoint.com/unix_sockets/index.htm).
  + [Python Socket Library and Modules](http://docs.python.org/3.0/library/socket.html).

Pasted from <<http://www.tutorialspoint.com/python/python_networking.htm>>

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending 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 mail 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 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* as of this option.

An SMTP object has an instance method called **sendmail**, which will typically be 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/python

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 **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 and 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 aren't always used to route mail).

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

smtplib.SMTP('mail.your-domain.com', 25)

Sending an HTML e-mail using Python:

When you send a text message using Python, then all the content will be treated as simple text. Even if you will include HTML tags in a text message, it will be displayed as simple text and HTML tags will not be formatted according to HTML syntax. But Python provides 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 character set to send an HTML e-mail.

EXAMPLE:

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

#!/usr/bin/python

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 to set **Content-type** header to **multipart/mixed**. Then, text and attachment sections can be specified within **boundaries**.

A boundary is started with two hyphens followed by a unique number, which can not 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.

Attached files should be encoded with the **pack("m")** function to have base64 encoding before transmission.

EXAMPLE:

Following is the example, which will send a file **/tmp/test.txt** as an attachment. Try it once:

#!/usr/bin/python

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"

Pasted from <<http://www.tutorialspoint.com/python/python_sending_email.htm>>

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 sometimes called light-weight processes and they do not require much memory overhead; they care 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 it is 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.

Starting a New Thread:

To spawn another thread, you need to call following method available in *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 *agrs*. When 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/python

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

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

Thread-1: Thu Jan 22 15:42:17 2009  
Thread-1: Thu Jan 22 15:42:19 2009  
Thread-2: Thu Jan 22 15:42:19 2009  
Thread-1: Thu Jan 22 15:42:21 2009  
Thread-2: Thu Jan 22 15:42:23 2009  
Thread-1: Thu Jan 22 15:42:23 2009  
Thread-1: Thu Jan 22 15:42:25 2009  
Thread-2: Thu Jan 22 15:42:27 2009  
Thread-2: Thu Jan 22 15:42:31 2009  
Thread-2: Thu Jan 22 15:42:35 2009

Although it is very effective for low-level threading, but 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 will in turn call *run()* method.

EXAMPLE:

#!/usr/bin/python

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:  
 thread.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()

print "Exiting Main Thread"

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

Starting Thread-1  
Starting Thread-2  
Exiting Main Thread  
Thread-1: Thu Mar 21 09:10:03 2013  
Thread-1: Thu Mar 21 09:10:04 2013  
Thread-2: Thu Mar 21 09:10:04 2013  
Thread-1: Thu Mar 21 09:10:05 2013  
Thread-1: Thu Mar 21 09:10:06 2013  
Thread-2: Thu Mar 21 09:10:06 2013  
Thread-1: Thu Mar 21 09:10:07 2013  
Exiting Thread-1  
Thread-2: Thu Mar 21 09:10:08 2013  
Thread-2: Thu Mar 21 09:10:10 2013  
Thread-2: Thu Mar 21 09:10:12 2013  
Exiting Thread-2

Synchronizing Threads:

The threading module provided with Python includes a simple-to-implement locking mechanism that will allow 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 would be used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread will wait to acquire the lock.

If *blocking* is set to 0, the thread will return 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 will block and wait for the lock to be released.

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

EXAMPLE:

#!/usr/bin/python

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"

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

Starting Thread-1  
Starting Thread-2  
Thread-1: Thu Mar 21 09:11:28 2013  
Thread-1: Thu Mar 21 09:11:29 2013  
Thread-1: Thu Mar 21 09:11:30 2013  
Thread-2: Thu Mar 21 09:11:32 2013  
Thread-2: Thu Mar 21 09:11:34 2013  
Thread-2: Thu Mar 21 09:11:36 2013  
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/python

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"

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

Pasted from <<http://www.tutorialspoint.com/python/python_multithreading.htm>>

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 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.

XML is extremely useful for keeping track of small to medium amounts of data without requiring a 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 disk and the entire file is never stored in memory.
  + **Document Object Model (DOM) API :**This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously can't process information as fast as DOM can when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of 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 can't use them both for large projects.

For all our XML code examples, let's 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 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:

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 is the detail 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:

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

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

Here is the detail 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 is the detail 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/python

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")

This would produce 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 standard [Python SAX APIs](http://docs.python.org/library/xml.sax.html).

Parsing XML with DOM APIs:

The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying 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 quickly load an XML document and to create a minidom object using the xml.dom module. The minidom object provides a simple parser method that will quickly create 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.

#!/usr/bin/python

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

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

For a complete detail on DOM API documentation, please refer to standard [Python DOM APIs](http://docs.python.org/library/xml.dom.html).

Pasted from <<http://www.tutorialspoint.com/python/python_xml_processing.htm>>

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below:

* + **Tkinter:** Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this tutorial.
  + **wxPython:** This is an open-source Python interface for wxWindows [http://wxpython.org](http://wxpython.org/).
  + **JPython:** JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine [http://www.jython.org](http://www.jython.org/).

There are many other interfaces available which I'm not listing here. You can find them over the net.

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:

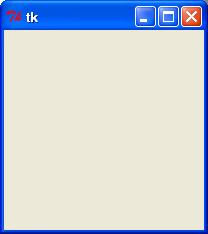
* + Import the *Tkinter* module.
  + Create the GUI application main window.
  + Add one or more of the above-mentioned widgets to the GUI application.
  + Enter the main event loop to take action against each event triggered by the user.

Example:

#!/usr/bin/python

import Tkinter  
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:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| [Button](http://www.tutorialspoint.com/python/tk_button.htm) | The Button widget is used to display buttons in your application. |
| [Canvas](http://www.tutorialspoint.com/python/tk_canvas.htm) | The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application. |
| [Checkbutton](http://www.tutorialspoint.com/python/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. |
| [Entry](http://www.tutorialspoint.com/python/tk_entry.htm) | The Entry widget is used to display a single-line text field for accepting values from a user. |
| [Frame](http://www.tutorialspoint.com/python/tk_frame.htm) | The Frame widget is used as a container widget to organize other widgets. |
| [Label](http://www.tutorialspoint.com/python/tk_label.htm) | The Label widget is used to provide a single-line caption for other widgets. It can also contain images. |
| [Listbox](http://www.tutorialspoint.com/python/tk_listbox.htm) | The Listbox widget is used to provide a list of options to a user. |
| [Menubutton](http://www.tutorialspoint.com/python/tk_menubutton.htm) | The Menubutton widget is used to display menus in your application. |
| [Menu](http://www.tutorialspoint.com/python/tk_menu.htm) | The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton. |
| [Message](http://www.tutorialspoint.com/python/tk_message.htm) | The Message widget is used to display multiline text fields for accepting values from a user. |
| [Radiobutton](http://www.tutorialspoint.com/python/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. |
| [Scale](http://www.tutorialspoint.com/python/tk_scale.htm) | The Scale widget is used to provide a slider widget. |
| [Scrollbar](http://www.tutorialspoint.com/python/tk_scrollbar.htm) | The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes. |
| [Text](http://www.tutorialspoint.com/python/tk_text.htm) | The Text widget is used to display text in multiple lines. |
| [Toplevel](http://www.tutorialspoint.com/python/tk_toplevel.htm) | The Toplevel widget is used to provide a separate window container. |
| [Spinbox](http://www.tutorialspoint.com/python/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. |
| [PanedWindow](http://www.tutorialspoint.com/python/tk_panedwindow.htm) | A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically. |
| [LabelFrame](http://www.tutorialspoint.com/python/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. |
| [tkMessageBox](http://www.tutorialspoint.com/python/tk_messagebox.htm) | This module is used to display message boxes in your applications. |

Standard attributes:

Let's take a look at how some of their common attributes.such as sizes, colors and fonts are specified.

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

Geometry Management:

All Tkinter widgets have access to 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](http://www.tutorialspoint.com/python/tk_pack.htm) - This geometry manager organizes widgets in blocks before placing them in the parent widget.
  + [The *grid()* Method](http://www.tutorialspoint.com/python/tk_grid.htm) - This geometry manager organizes widgets in a table-like structure in the parent widget.
  + [The *place()* Method](http://www.tutorialspoint.com/python/tk_place.htm) -This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Pasted from <<http://www.tutorialspoint.com/python/python_gui_programming.htm>>

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-Requisite:

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](http://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 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'll be grouping your code into four parts:

* + 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 will see them to C functions inside the extension module.
  + An initialization function.

The header file *Python.h*

Start including *Python.h* header file in your C source file, which will give you access to the internal Python API used to hook your module into the interpreter.

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

The C functions:

The signatures of the C implementations of your functions will always take 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's no such thing as a *void*function in Python as there is in C. If you don't 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 will never be seen outside of the extension module. So they would be 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 would be a Python function called *func* inside of the module *module*. You'll 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 will present it when it is used in Python programs.
  + **ml\_meth:** This must be the address to a function that has any one of the signatures described in previous seection.
  + **ml\_flags:** This tells the interpreter which of the three signatures ml\_meth is using.
  + This flag will usually have 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 don't want to accept any arguments.
  + **ml\_doc:** This is the docstring for the function, which could be NULL if you don't 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 would have 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's 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'll be building. The Python headers define PyMODINIT\_FUNC to include the appropriate incantations for that to happen for the particular environment in which we're 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 init*Module*() {  
 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 this all together 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 init*Module*() {  
 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 source form and built and installed via a setup script usually called *setup.py* as follows.

For the above module, you would have to prepare 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'll most likely need to run this command as root in order to have permissions to write to the site-packages directory. This usually isn't a problem on Windows

Import Extensions:

Once you installed your extension, you would be able to import and call that extension in your Python script as follows:

#!/usr/bin/python  
import helloworld

print helloworld.helloworld()

This would produce the following result:

Hello, Python extensions!!

Passing Function Parameters:

Because you'll most likely want to define functions that do accept arguments, you can use one of the other signatures for your C functions. For example, 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 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'll 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;  
}

Compiling the new version of your module and importing it will enable 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 **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 **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're building, you pass in the actual values. Here's 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 cauptured 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.

Following table lists the commonly used code strings, of which zero or more are joined into 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}.

Pasted from <<http://www.tutorialspoint.com/python/python_further_extensions.htm>>