# Python Date & Time

A Python program can handle date and 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.

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: 1507888743.720282

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=2017, tm\_mon=10, tm\_mday=13, tm\_hour=16, tm\_min=2, tm\_sec=8, tm\_wday=4, tm\_yday=286, 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 : Fri Oct 13 16:10:30 2017

## **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 2017 )

#!/usr/bin/python

import calendar

cal = calendar.month(2017, 1)

print(cal)

This would produce the following result −

January 2017

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**  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.  print("time.altzone %d " % time.altzone)  time.altzone -23400 |
| 2 | **time.asctime([tupletime])**  Accepts a time-tuple and returns as character readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.  print(time.gmtime())  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=10, tm\_min=54, tm\_sec=4, tm\_wday=4, tm\_yday=286, tm\_isdst=0)  # Output in UTC (Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT).  time.asctime(time.gmtime())  'Fri Oct 13 10:56:04 2017'  time.asctime(time.localtime())  'Fri Oct 13 16:26:18 2017' |
| 3 | **time.clock( )**  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(). On Unix and in Windows it returns wall-clock seconds elapsed since the first call to this function, as a floating point number.  time.clock( )  2.1935462828870445  time.clock( )  4.097568009977395  time.clock( )  5.349900923483391 |
| 4 | **time.ctime([secs])**  Like asctime(localtime(secs)) and without arguments is like asctime( )  time.asctime(time.gmtime())  'Fri Oct 13 11:25:41 2017'  time.asctime(time.localtime())  'Fri Oct 13 16:57:43 2017'  time.ctime()  'Fri Oct 13 16:57:52 2017' |
| 5 | **time.gmtime([secs])**  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  time.gmtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=11, tm\_min=31, tm\_sec=33, tm\_wday=4, tm\_yday=286, tm\_isdst=0) |
| 6 | **time.localtime([secs])**  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).  time.localtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=17, tm\_min=3, tm\_sec=35, tm\_wday=4, tm\_yday=286, tm\_isdst=0) |
| 7 | **time.mktime(tupletime)**  The method **mktime()** is the inverse function of localtime(). Its argument is the struct\_time or full 9-tuple and it returns a floating point number, for compatibility with time().  If the input value cannot be represented as a valid time, either *OverflowError*or *ValueError* will be raised.  time.localtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=17, tm\_min=18, tm\_sec=50, tm\_wday=4, tm\_yday=286, tm\_isdst=0)  t =(2017,10,13,17,18,50,4,286,0);  time.mktime(t);  1507895330.0  time.mktime(time.localtime())  1507895862.0 |
| 8 | **time.sleep(secs)**  Suspends the calling thread for secs seconds.  print("Start : %s" % time.ctime())  Start : Fri Oct 13 18:46:57 2017  time.sleep( 5 )  print("End : %s" % time.ctime())  End : Fri Oct 13 18:47:02 2017 |
| 9 | **time.strftime(fmt[,tupletime])**  Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.  Syntax  Following is the syntax for strftime() method:  time.strftime(format[, t])  Format elements   * %a - abbreviated weekday name * %A - full weekday name * %b - abbreviated month name * %B - full month name * %c - preferred date and time representation * %C - century number (the year divided by 100, range 00 to 99) * %d - day of the month (01 to 31) * %D - same as %m/%d/%y * %e - day of the month (1 to 31) * %g - like %G, but without the century * %G - 4-digit year corresponding to the ISO week number (see %V). * %h - same as %b * %H - hour, using a 24-hour clock (00 to 23) * %I - hour, using a 12-hour clock (01 to 12) * %j - day of the year (001 to 366) * %m - month (01 to 12) * %M - minute * %n - newline character * %p - either am or pm according to the given time value * %r - time in a.m. and p.m. notation * %R - time in 24 hour notation * %S - second * %t - tab character * %T - current time, equal to %H:%M:%S * %u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1 * %U - week number of the current year, starting with the first Sunday as the first day of the first week * %V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week * %W - week number of the current year, starting with the first Monday as the first day of the first week * %w - day of the week as a decimal, Sunday=0 * %x - preferred date representation without the time * %X - preferred time representation without the date * %y - year without a century (range 00 to 99) * %Y - year including the century * %Z or %z - time zone or name or abbreviation   t =(2017,10,13,17,18,50,4,286,0);  t = time.mktime(t)  print(time.strftime("%b %d %Y %H:%M:%S", time.gmtime(t)))  Oct 13 2017 11:48:50 |
| 10 | [**time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')**](https://www.tutorialspoint.com/python/time_strptime.htm)  Parses str according to format string fmt and returns the instant in time-tuple format. The method strptime() parses a string representing a time according to a format. The return value is a struct\_time as returned by gmtime() or localtime().  The format parameter uses the same directives as those used by strftime(); it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime().  If string cannot be parsed according to format, or if it has excess data after parsing, ValueError is raised.  **Syntax**  Following is the syntax for strptime() method:  **time.strptime(string[, format]**  t = time.strptime("30 Nov 2017 16 30 45", "%d %b %Y %H %M %S");  print(t)  time.struct\_time(tm\_year=2017, tm\_mon=11, tm\_mday=30, tm\_hour=16, tm\_min=30, tm\_sec=45, tm\_wday=3, tm\_yday=334, tm\_isdst=-1) |
| 11 | **time.time( )**  Returns the current time instant, a floating-point number of seconds since the epoch.  time.time( )  1507904917.6235135 |
| 12 | **time.tzset()**  Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.  Availability: Unix. |

Let us go through the functions briefly −

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).  time.timezone  -19800 |
| 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.  time.tzname  ('India Standard Time', 'India Daylight Time') |

## **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.  print(calendar.calendar(2017,w=1,l=1,c=1));  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  July August September  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 1 2  2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9  9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16  16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23  23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30  30 31  October November December  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2  8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9  15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16  22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23  29 30 31 26 27 28 29 30 24 25 26 27 28 29 30  31 |
| 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.  calendar.firstweekday( )  0 |
| 3 | **calendar.isleap(year)**  Returns True if year is a leap year; otherwise, False.  calendar.isleap(2016)  True  calendar.isleap(2017)  False |
| 4 | **calendar.leapdays(y1,y2)**  Returns the total number of leap days in the years within range(y1,y2).  calendar.leapdays(2000,2017)  5 |
| 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.  print(calendar.month(2017,10,w=2,l=1))  October 2017  Su Mo Tu We Th Fr Sa  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 |
| 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.  print(calendar.monthcalendar(2017,10))  [[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, 0, 0, 0, 0]] |
| 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; 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.  >>> print(calendar.calendar(2017,w=1,l=1,c=1));  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  calendar.monthrange(2017,1)  (6, 31)  calendar.monthrange(2017,2)  (2, 28)  calendar.monthrange(2017,3)  (2, 31)  calendar.monthrange(2017,4)  (5, 30)  calendar.monthrange(2017,5)  (0, 31)  calendar.monthrange(2017,6)  (3, 30) |
| 8 | **calendar.prcal(year,w=2,l=1,c=6)**  Like print(calendar.calendar(year,w,l,c));  calendar.prcal(2017,w=2,l=1,c=2)  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  July August September  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 1 2  2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9  9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16  16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23  23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30  30 31  October November December  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2  8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9  15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16  22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23  29 30 31 26 27 28 29 30 24 25 26 27 28 29 30  31 |
| 9 | **calendar.prmonth(year,month,w=2,l=1)**  Like print(calendar.month(year,month,w=1,l=1));  print(calendar.month(2017,1,w=1,l=1));  January 2017  Su Mo Tu We Th Fr Sa  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  calendar.prmonth(2017,1,w=2,l=1)  January 2017  Su Mo Tu We Th Fr Sa  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 |
| 10 | **calendar.setfirstweekday(weekday)**  Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).  import calendar  calendar.firstweekday( )  0  calendar.setfirstweekday(calendar.SUNDAY);  calendar.firstweekday( )  6 |
| 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.  import calendar  import time  import datetime  d = datetime.datetime(2017, 10, 31)  calendar.timegm(d.timetuple())  1509408000  time.mktime(d.timetuple())  1509388200.0  d = datetime.datetime(2017, 10, 31, 16, 30, 30)  calendar.timegm(d.timetuple())  1509467430  time.mktime(d.timetuple())  1509447630.0 |
| 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).  print(calendar.month(2017,10,w=2,l=1));  October 2017  Su Mo Tu We Th Fr Sa  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  calendar.weekday(2017,10,3)  1 |

## **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)

# Python Functions

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 function\_name( parameter1, parameter2, parameter3 ):

"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**

The following function takes a string as input parameter and prints it on standard screen.

def print\_myname( str ):

"This function print your name."

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 **print\_myname()** function −

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This function print your name."

print(str)

return;

# Now you can call printme function

print\_myname("Hari Yadav")

Hari Yadav

## **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 change\_mylist( 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 change\_mylist function by using a list variable mylist as reference

mylist = [10,20,30];

print("Values before passing in the function: ", mylist)

change\_mylist( mylist );

print("Values after passing in 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 before passing in the function: [10, 20, 30]

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values after passing in 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 change\_mylist1( 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)

print("Object ID of mylist inside function: ", id(mylist))

return

# Now you can call change\_mylist1 function

mylist = [10,20,30];

print("Object ID of mylist outside function: ", id(mylist))

change\_mylist1( mylist );

print("Values outside the function: ", mylist)

The parameter *mylist* is local to the function change\_mylist1. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result:

Object ID of mylist outside function: 2301171593544

Values inside the function: [1, 2, 3, 4]

Object ID of mylist inside function: 2301171594056

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 *print\_myname()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str);

return;

# Now you can call print\_myname function

print\_myname()

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

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

TypeError: print\_myname() missing 1 required positional argument: 'str'

## **Keyword arguments (parameter named notation)**

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 *print\_myname()* function in the following ways −

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str)

return;

# Now you can call print\_myname function

print\_myname( str = "Michael Gerard Tyson")

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

Michael Gerard Tyson

The following example gives more clear picture. Note that the order of parameters does not matter.

#!/usr/bin/python

# Function definition is here

def personal\_info( name, age ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age ", age)

return;

# Now you can call personal\_info function

personal\_info( age=50, name="Michael Gerard Tyson")

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

Name: Michael Gerard Tyson

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. The following example gives an idea on default arguments, it prints default age if it is not passed −

#!/usr/bin/python

# Function definition is here

def personal\_info( name, age, design = "Trainer" ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age ", age)

print("Designation ", design)

return;

# Now you can call personal\_info function

personal\_info( age=50, name="Michael Gerard Tyson" )

personal\_info( age=50, name="Michael Gerard Tyson" , design='Chairman')

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

Name: Michael Gerard Tyson

Age 50

Designation Trainer

Name: Michael Gerard Tyson

Age 50

Designation Chairman

## **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.

Syntax for a function with non-keyword variable arguments is this −

def function\_name([formal\_args,] **\***var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds 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 student\_marks( name, **\***score\_tuple ):

"This prints a variable length passed arguments"

print("Student subject details is: ")

print(name)

for var in score\_tuple:

print(var)

return;

# Now you can call student\_marks function

student\_marks("Dev Singh")

student\_marks ("Sami", 70, 60, 50 )

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

Student subject details is:

Dev Singh

Student subject details is:

Sami

70

60

50

def student\_marks( name, \*score\_tuple, \*class ):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name, class)

for var in score\_tuple:

print(var)

return;

**NOTE: Only one variable-length arguments can be specified.**

def student\_marks(\*score\_tuple, name):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name)

for var in score\_tuple:

print(var)

return;

print("Good Bye. . . . . ")

student\_marks("Dev Singh")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: student\_marks() missing 1 required keyword-only argument: 'name'

student\_marks(name="Dev Singh")

Student subject details is:

Dev Singh

**NOTE: Any statement written after return keyword is not executed.**

student\_marks (70, 60, 50, "Sami" )

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: student\_marks() missing 1 required keyword-only argument: 'name'

student\_marks (70, 60, 50, name="Sami" )

Student subject details is:

Sami

70

60

50

## **The *Anonymous* Functions**

These functions are called anonymous because they are not declared in the standard manner by using the ***def* keyword**. You can use the ***lambda*** keyword to create small anonymous functions.

* 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. Any statement written after return keyword is not executed.

All the above examples are not returning any value. 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("Total Inside the function : ", total)

return total;

# Now you can call sum function

v\_total = sum( 10, 20 );

print("Total Outside the function : ", v\_total);

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

Total Inside the function : 30

Total 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

30

Outside the function global total : 0

# Python Modules

Modules refer to a file containing Python statements and definitions like function and class definitions.. A module allows you to logically organize your Python code.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

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

## **Example**

Let’s create a module. Type the following and save it as **example.py**.

# Python Module example

def add(a, b):

"""This program adds two

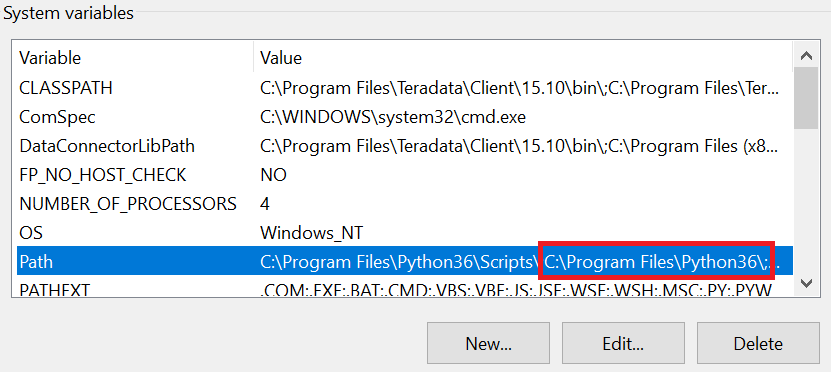
numbers and return the result"""

result = a + b

return result

Here, we have defined a [function](https://www.programiz.com/python-programming/function) **add()** inside a module named example. The function takes in two numbers and returns their sum.

I have copied this python module file **example.py** to **C:\Program Files\Python36** directory because this is the environment variable path.



## How to import modules in Python?

We can import the definitions of a module on the interactive prompt of Python or inside another module.

We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Welcome to Python!

Hari Yadav

>>> import example

>>> example.add(50,45.5);

95.5

## Import with renaming

We can import a module by renaming it as follows.

>>> import example as ex

>>> ex.add(50,45.5);

95.5

>>> example.add(50,45.5);

95.5

Using the module name we can access the function using dot (.) operation. A module is loaded only once, regardless of the number of times it is imported.

## Python **from...import** statement

We can import specific function from a module without importing the module as a whole. Here we will add another function in **example.py** file. In such case we don't use the dot operator.

def sub(a, b):

"""This program subtract two

numbers and return the result"""

result = a - b

return result

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Welcome to Python!

Hari Yadav

>>> from example import sub

>>> sub(50,45.5);

4.5

>>> add(50,45.5);

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'add' is not defined

>>> from example import add

>>> add(50,45.5);

95.5

## 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.

**Add this private function in the module example**

def \_income\_tax(income, rate): # private method

"""This program calculate tax and return the result"""

result = (income\*rate)/100

print("Your tax is Rs : ")

return result

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Welcome to Python!

Hari Yadav

>>> import example

>>> example.add(50,45.5);

95.5

>>> example.sub(50,45.5);

4.5

>>> example.\_income\_tax(400000,10)

Your tax is Rs :

40000.0

>>> exit()

This import all object definition except those beginnig with an underscore (private objects), visible in our scope. Objects are access without dot operator (.)

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Welcome to Python!

Hari Yadav

**>>> from example import \***

>>> example.add(50,45.5);

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'example' is not defined

>>> add(50,45.5);

95.5

>>> sub(50,45.5);

4.5

>>> \_income\_tax(400000,10)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**NameError: name '\_income\_tax' is not defined**

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier.

## **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.

>>> import sys

>>> sys.path

['', 'C:\\Program Files\\Python36\\python36.zip', 'C:\\Program Files\\Python36\\DLLs', 'C:\\Program Files\\Python36\\lib', 'C:\\Program Files\\Python36', 'C:\\Program Files\\Python36\\lib\\site-packages']

C:\Users\haryadav>echo %PATH%

C:\Program Files\Python36\Scripts\;C:\Program Files\Python36\;C:\Program Files\Teradata\Client\15.10\bin;C:\Program Files (x86)\Teradata\Client\15.10\bin;C:\app\haryadav\product\11.2.0\dbhome\_1\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\QuickTime\QTSystem\;C:\Program Files\CapgeminiScripts\Support Tools\;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\DTS\Binn\;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\VSShell\Common7\IDE\;C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\Microsoft SQL Server\100\DTS\Binn\;C:\Program Files\Microsoft SQL Server\110\DTS\Binn\;C:\Program Files (x86)\Microsoft SQL Server\110\Tools\Binn\;C:\Program Files\Microsoft SQL Server\110\Tools\Binn\;C:\Program Files (x86)\Microsoft SQL Server\110\Tools\Binn\ManagementStudio\;C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\Microsoft SQL Server\110\DTS\Binn\;C:\Users\haryadav\AppData\Local\Microsoft\WindowsApps;

import sys

sys.builtin\_module\_names

('\_ast', '\_bisect', '\_blake2', '\_codecs', '\_codecs\_cn', '\_codecs\_hk', '\_codecs\_iso2022', '\_codecs\_jp', '\_codecs\_kr', '\_codecs\_tw', '\_collections', '\_csv', '\_datetime', '\_functools', '\_heapq', '\_imp', '\_io', '\_json', '\_locale', '\_lsprof', '\_md5', '\_multibytecodec', '\_opcode', '\_operator', '\_pickle', '\_random', '\_sha1', '\_sha256', '\_sha3', '\_sha512', '\_signal', '\_sre', '\_stat', '\_string', '\_struct', '\_symtable', '\_thread', '\_tracemalloc', '\_warnings', '\_weakref', '\_winapi', 'array', 'atexit', 'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parser', 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')

## **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:

C:\Users\haryadav> set PYTHONPATH=c:\python20\lib

C:\Users\haryadav> set PYTHONPATH

PYTHONPATH=c:\python20\lib

C:\Users\haryadav> echo %PYTHONPATH%

c:\python20\lib

# To unset the environment variable

C:\Users\haryadav> set PYTHONPATH=

C:\Users\haryadav> echo %PYTHONPATH%

%PYTHONPATH%

C:\Users\haryadav> set PYTHONPATH

Environment variable PYTHONPATH not defined

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.

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 use the global statement to reference the variable defined outside the function.

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 Add*Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable.

#!/usr/bin/python

Money = 2000

def AddMoney():

# local variable Money is referenced before assignment. It will throw error.

Money = Money + 1

return Money

print(Money)

2000

AddMoney()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 4, in AddMoney

UnboundLocalError: local variable 'Money' referenced before assignment

def AddMoney():

# local variable Money is assigned:

Money = 5000

Money = Money + 1

return Money

AddMoney()

5001

print(Money)

**2000**

def AddMoney():

# Here global variable Money is referenced using global keyword

# local variable Money is commented

global Money

# Money = 5000

Money = Money + 1

return Money

AddMoney()

2001

print(Money)

2001

## **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.

#!/usr/bin/python

Add the following lines in the example module which you defined in previous exercise.

# Global variables defined

g\_result = 0

g\_score = 0

**import example**

**content = dir(example)**

**print(content)**

**['\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '\_income\_tax', 'add', 'g\_result', 'g\_score', 'sub']**

**type(content)**

**<class 'list'>**

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.

g\_result = 0

g\_score = 0

def currency():

Rupees = 5000

Usd = 5000

Euro= 5000

Ruble=5000

v\_result = Rupees + Usd + Euro + Ruble

print('Local variables are : ', '\n', locals() )

print('Global variables are : ', '\n', globals())

return v\_result

currency()

Local variables are :

{'v\_result': 20000, 'Ruble': 5000, 'Euro': 5000, 'Usd': 5000, 'Rupees': 5000}

Global variables are :

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0000020EDB74F0B8>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_cached\_\_': None, 'os': <module 'os' from 'C:\\Program Files\\Python36\\lib\\os.py'>, 'cls': <function <lambda> at 0x0000020EDB5C3E18>, 'AddMoney': <function AddMoney at 0x0000020EDB8AD8C8>, 'g\_result': 0, 'g\_score': 0, 'currency': <function currency at 0x0000020EDB8AD950>}

20000

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

def currency():

Rupees = 5000

Usd = 5000

Euro= 5000

Ruble=5000

v\_result = Rupees + Usd + Euro + Ruble

print('Local variables are : ', '\n', locals() )

dict = locals()

print(dict.keys())

print(dict.values())

return v\_result

currency()

Local variables are :

{'v\_result': 20000, 'Ruble': 5000, 'Euro': 5000, 'Usd': 5000, 'Rupees': 5000}

dict\_keys(['v\_result', 'Ruble', 'Euro', 'Usd', 'Rupees'])

dict\_values([20000, 5000, 5000, 5000, 5000])

20000

## **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)

reload(example)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'reload' is not defined

**For >= Python3.4:**

import example

import importlib

importlib.reload(example)

<module 'example' from 'C:\\Program Files\\Python36\\example.py'>

**For <= Python3.3:**

import imp

imp.reload(example)

<module 'example' from 'C:\\Program Files\\Python36\\example.py'>

**For Python2.x:**

reload(example)

## **Packages in Python**

A bundle of multiple modules together form a package.

A package is basically a hierarchical directory structure with Python files and a file with the name \_\_init\_\_.py. This means that every directory inside of the Python path, which contains a file named \_\_init\_\_.py, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python’s module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name. The submodule A of the package P1 and the submodule A of the package P2 can be totally different.

A package is imported like a "normal" module.

Create a  *Phone* directory where your package will be stored as shown below :-

**NOTE: Run this terminial as administrator if you have privilege issue.**

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>cd ..

C:\Users>cd ..

C:\>chdir C:\Program Files\Python36\

C:\Program Files\Python36>mkdir Phone

C:\Program Files\Python36>cd Phone

C:\Program Files\Python36\Phone>

C:\Program Files\Python36\Phone>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Welcome to Python!

Hari Yadav

Create a file Basicphone*.py* available in *Phone* directory. This file has following line of source code −

def Basicphone():

print("This is Basic Phone")

Similar way, we have another two files having different functions as shown below :−

* *Phone/Isdnphone.py* file having function Isdnphone() code as below:

**def Isdnphone():**

**print("This is ISDN Phone")**

* *Phone/IPphone.py* file having function *IPphone*() code as below:

**def IPphone():**

**print("This is IP Phone")**

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 Basicphone import Basicphone

from Isdnphone import Isdnphone

from IPphone import IPphone

After you add these lines to \_\_init\_\_.py, you have all of these functions available when you import the Phone package.

#!/usr/bin/python

# Now import your Phone Package.

import Phone

Phone.Basicphone()

Phone.Isdnphone()

Phone.IPphone()

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

This is Basic Phone

This is ISDN Phone

This is IP 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.

# Python Files I/O

This topic covers 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,", "it is very interesting?")

This produces the following result on your standard screen −

Python is really a great language,", "it is very interesting?

## **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 ( prior to Python 3.0)
* 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 prompts 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, it assumes the input as character string.

#!/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: Capgemini India

Recieved input is : Capgemini India

**NOTE: Input is treated as character string and arithmetic operation is not possible.**

str = input("Enter your input: ");

Enter your input: 100

v\_result = str + 200

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: must be str, not int

v\_result = int(str) + 200

print('Result is : ', v\_result)

Result is : 300

## **Opening and Closing Files**

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do 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 are parameter details:

* **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 takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is 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 placed at the beginning of the file. |
| rb+ | Opens a file for both reading and writing in binary format. The file pointer placed 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. |

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "w")

print("Name of the file: ", v\_file.name)

print("Closed or not : ", v\_file.closed)

print("Opening mode : ", v\_file.mode)

v\_file.close()

**NOTE: Run this terminial as administrator if you have privilege issue.**

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>cd ..

C:\Users>cd ..

C:\>chdir C:\Program Files\Python36\

C:\Program Files\Python36>mkdir fileIO

C:\Program Files\Python36>cd fileIO

C:\Program Files\Python36\fileIO>

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Welcome to Python!

Hari Yadav

v\_file = open("resume.txt", "wb")

print("Name of the file: ", v\_file.name)

print("Closed or not : ", v\_file.closed)

print("Open mode of the file : ", v\_file.mode)

v\_file.close()

print("File closed : ", v\_file.closed)

This produces the following result −

Name of the file: resume.txt

Closed or not : False

Open mode of the file : wb

File closed : True

We create a file object using open() function and get a list of all possible methods that can be used with a file object, using Python built-in dir() function. The dir() function accepts a Python object as an argument and returns a list of attributes and methods related to them.

dir(v\_file)

['\_\_class\_\_', '\_\_del\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_enter\_\_', '\_\_eq\_\_', '\_\_exit\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_checkClosed', '\_checkReadable', '\_checkSeekable', '\_checkWritable', '\_dealloc\_warn', '\_finalizing', 'close', 'closed', 'detach', 'fileno', 'flush', 'isatty', 'mode', 'name', 'raw', 'read', 'read1', 'readable', 'readinto', 'readinto1', 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']

## **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

v\_file = open("resume.txt", "wb")

print("Name of the file: ", v\_file.name)

# Close opend file

v\_file.close()

print("File is closed : ", v\_file.closed)

This produces the following result −

Name of the file: resume.txt

File is closed : True

## **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. You have to do it explicitly.

**Syntax**

fileObject.write(string);

Here, passed parameter is the content to be written into the opened file.

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "w")

print("Open mode of the file : ", v\_file.mode)

v\_file.write( "Python is a great language.\nYeah its great!!\n");

# Close opend file

v\_file.close()

The above method would create *resume.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.

Open mode of the file : w

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. apart from text data.

### **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 read the data from our file *resume.txt*, which we created above.

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "r+")

str = v\_file.read(10);

print("Read String is : ", str)

str = v\_file.read();

print("Read String is : ", str)

v\_file.write( "Python can be installed on both window and linux.\nLinux provide better security!!\n");

v\_file.close()

v\_file = open("resume.txt", "r+")

str = v\_file.read();

print("Read String is : ", str)

# Close opend file

v\_file.close()

# Readlines from the file.

import os

os.getcwd()

'C:\\Program Files\\Python36\\fileIO'

myFileObject = open('C:\\Program Files\\Python36\\fileIO\\resume.txt')

filedata = myFileObject.readlines()

filedata

['Python is a great language.\n', 'Yeah its great!!\n', 'Python can be installed on both window and linux.\n', 'Linux provide better security!!\n']

myFileObject.close()

This produces the following result −

Read String is : Python is

Read String is : a great language.

Yeah its great!!

Read String is : Python is a great language.

Yeah its great!!

Python can be installed on both window and linux.

Linux provide better security!!

## **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 means use the beginning of the file as the reference position.

1 means use the current position as the reference position.

2 means at the end of the file would be taken as the reference position.

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "r+")

str = v\_file.read(10);

print("Read String is : ", str)

# Check current position

v\_position = v\_file.tell();

print("Current file position is : ", v\_position)

# Reposition pointer at the beginning once again

v\_position = v\_file.seek(0, 0);

print("Current file position is : ", v\_position)

str = v\_file.read(10);

print("Again Read String is : ", str)

v\_position = v\_file.seek(0, 1);

print("Current file position is now : ", v\_position)

str = v\_file.read(10);

print("Again Read String is : ", str)

v\_position = v\_file.seek(0, 1);

print("Current file position is now : ", v\_position)

# Reposition pointer to the end

v\_position = v\_file.seek(0, 2);

print("Current file position is : ", v\_position)

str = v\_file.read(10);

print("Again Read String is : ", str)

# Reposition pointer back to the begining

v\_position = v\_file.seek(0, 0);

str = v\_file.read(20);

print("Again Read String is : ", str)

# Close opend file

v\_file.close()

## **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 *resume.txt*:

#!/usr/bin/python

import os

# Rename a file from resume.txt to Hari\_Yadav\_Resume.txt

os.rename( "resume.txt", "Hari\_Yadav\_Resume.txt" )

os.rename( "Hari\_Yadav\_Resume.txt", "C:\\Program Files\\Python36\\fileIO\\Hari\_Yadav\_Resume\_2.txt"))

os.rename("Hari\_Yadav\_Resume.txt","resume.txt")

## **The *copyfile()* Method**

from shutil import copyfile

copyfile("resume.txt","Hari\_Yadav\_Resume\_1.txt")

'Hari\_Yadav\_Resume\_1.txt'

copyfile("Hari\_Yadav\_Resume\_1.txt","Hari\_Yadav\_Resume\_2.txt")

'Hari\_Yadav\_Resume\_2.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**

#!/usr/bin/python

import os

os.remove("Hari\_Yadav\_Resume\_2.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

os.mkdir("fileIO\_1")

os.mkdir("fileIO\_2")

## **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('C:\\Program Files\\Python36\\fileIO\\fileIO\_1')

os.getcwd()

'C:\\Program Files\\Python36\\fileIO\\fileIO\_1'

os.chdir('C:\\Program Files\\Python36\\fileIO')

os.getcwd()

'C:\\Program Files\\Python36\\fileIO'

# Switch to sub directory

os.chdir("fileIO\_1")

os.getcwd()

'C:\\Program Files\\Python36\\fileIO\\fileIO\_1'

## **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

os.listdir()

['fileIO\_1', 'fileIO\_2', 'Hari\_Yadav\_Resume\_1.txt', 'resume.txt']

os.rmdir("fileIO\_1")

os.rmdir('C:\\Program Files\\Python36\\fileIO\\fileIO\_2')

os.listdir()

['Hari\_Yadav\_Resume\_1.txt', 'resume.txt']

## **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: The *file* object provides functions to manipulate files.

A **file** object is created using *open* function and here is a list of functions which can be called on this object:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [**file.close()**](https://www.tutorialspoint.com/python/file_close.htm)  Close the file. A closed file cannot be read or written any more. |
| 2 | [**file.flush()**](https://www.tutorialspoint.com/python/file_flush.htm)  Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects. |
| 3 | [**file.fileno()**](https://www.tutorialspoint.com/python/file_fileno.htm)  Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system. |
| 4 | [**file.isatty()**](https://www.tutorialspoint.com/python/file_isatty.htm)  Returns True if the file is connected to a tty(-like) device, else False. |
| 5 | [**file.next()**](https://www.tutorialspoint.com/python/file_next.htm)  Returns the next line from the file each time it is being called. |
| 6 | [**file.read([size])**](https://www.tutorialspoint.com/python/file_read.htm)  Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). |
| 7 | [**file.readline([size])**](https://www.tutorialspoint.com/python/file_readline.htm)  Reads one entire line from the file. A trailing newline character is kept in the string. |
| 8 | [**file.readlines([sizehint])**](https://www.tutorialspoint.com/python/file_readlines.htm)  Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. |
| 9 | [**file.seek(offset[, whence])**](https://www.tutorialspoint.com/python/file_seek.htm)  Sets the file's current position |
| 10 | [**file.tell()**](https://www.tutorialspoint.com/python/file_tell.htm)  Returns the file's current position |
| 11 | [**file.truncate([size])**](https://www.tutorialspoint.com/python/file_truncate.htm)  Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size. |
| 12 | [**file.write(str)**](https://www.tutorialspoint.com/python/file_write.htm)  Writes a string to the file. There is no return value. |
| 13 | [**file.writelines(sequence)**](https://www.tutorialspoint.com/python/file_writelines.htm)  Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. |

[OS Object Methods](https://www.tutorialspoint.com/python/os_file_methods.htm): This provides methods to process files as well as directories.

The **os** module provides a big range of useful methods to manipulate files and directories.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [**os.access(path, mode)**](https://www.tutorialspoint.com/python/os_access.htm)  Use the real uid/gid to test for access to path. |
| 2 | [**os.chdir(path)**](https://www.tutorialspoint.com/python/os_chdir.htm)  Change the current working directory to path |
| 3 | [**os.chflags(path, flags)**](https://www.tutorialspoint.com/python/os_chflags.htm)  Set the flags of path to the numeric flags. |
| 4 | [**os.chmod(path, mode)**](https://www.tutorialspoint.com/python/os_chmod.htm)  Change the mode of path to the numeric mode. |
| 5 | [**os.chown(path, uid, gid)**](https://www.tutorialspoint.com/python/os_chown.htm)  Change the owner and group id of path to the numeric uid and gid. |
| 6 | [**os.chroot(path)**](https://www.tutorialspoint.com/python/os_chroot.htm)  Change the root directory of the current process to path. |
| 7 | [**os.close(fd)**](https://www.tutorialspoint.com/python/os_close.htm)  Close file descriptor fd. |
| 8 | [**os.closerange(fd\_low, fd\_high)**](https://www.tutorialspoint.com/python/os_closerange.htm)  Close all file descriptors from fd\_low (inclusive) to fd\_high (exclusive), ignoring errors. |
| 9 | [**os.dup(fd)**](https://www.tutorialspoint.com/python/os_dup.htm)  Return a duplicate of file descriptor fd. |
| 10 | [**os.dup2(fd, fd2)**](https://www.tutorialspoint.com/python/os_dup2.htm)  Duplicate file descriptor fd to fd2, closing the latter first if necessary. |
| 11 | [**os.fchdir(fd)**](https://www.tutorialspoint.com/python/os_fchdir.htm)  Change the current working directory to the directory represented by the file descriptor fd. |
| 12 | [**os.fchmod(fd, mode)**](https://www.tutorialspoint.com/python/os_fchmod.htm)  Change the mode of the file given by fd to the numeric mode. |
| 13 | [**os.fchown(fd, uid, gid)**](https://www.tutorialspoint.com/python/os_fchown.htm)  Change the owner and group id of the file given by fd to the numeric uid and gid. |
| 14 | [**os.fdatasync(fd)**](https://www.tutorialspoint.com/python/os_fdatasync.htm)  Force write of file with filedescriptor fd to disk. |
| 15 | [**os.fdopen(fd[, mode[, bufsize]])**](https://www.tutorialspoint.com/python/os_fdopen.htm)  Return an open file object connected to the file descriptor fd. |
| 16 | [**os.fpathconf(fd, name)**](https://www.tutorialspoint.com/python/os_fpathconf.htm)  Return system configuration information relevant to an open file. name specifies the configuration value to retrieve. |
| 17 | [**os.fstat(fd)**](https://www.tutorialspoint.com/python/os_fstat.htm)  Return status for file descriptor fd, like stat(). |
| 18 | [**os.fstatvfs(fd)**](https://www.tutorialspoint.com/python/os_fstatvfs.htm)  Return information about the filesystem containing the file associated with file descriptor fd, like statvfs(). |
| 19 | [**os.fsync(fd)**](https://www.tutorialspoint.com/python/os_fsync.htm)  Force write of file with filedescriptor fd to disk. |
| 20 | [**os.ftruncate(fd, length)**](https://www.tutorialspoint.com/python/os_ftruncate.htm)  Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size. |
| 21 | [**os.getcwd()**](https://www.tutorialspoint.com/python/os_getcwd.htm)  Return a string representing the current working directory. |
| 22 | [**os.getcwdu()**](https://www.tutorialspoint.com/python/os_getcwdu.htm)  Return a Unicode object representing the current working directory. |
| 23 | [**os.isatty(fd)**](https://www.tutorialspoint.com/python/os_isatty.htm)  Return True if the file descriptor fd is open and connected to a tty(-like) device, else False. |
| 24 | [**os.lchflags(path, flags)**](https://www.tutorialspoint.com/python/os_lchflags.htm)  Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links. |
| 25 | [**os.lchmod(path, mode)**](https://www.tutorialspoint.com/python/os_lchmod.htm)  Change the mode of path to the numeric mode. |
| 26 | [**os.lchown(path, uid, gid)**](https://www.tutorialspoint.com/python/os_lchown.htm)  Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links. |
| 27 | [**os.link(src, dst)**](https://www.tutorialspoint.com/python/os_link.htm)  Create a hard link pointing to src named dst. |
| 28 | [**os.listdir(path)**](https://www.tutorialspoint.com/python/os_listdir.htm)  Return a list containing the names of the entries in the directory given by path. |
| 29 | [**os.lseek(fd, pos, how)**](https://www.tutorialspoint.com/python/os_lseek.htm)  Set the current position of file descriptor fd to position pos, modified by how. |
| 30 | [**os.lstat(path)**](https://www.tutorialspoint.com/python/os_lstat.htm)  Like stat(), but do not follow symbolic links. |
| 31 | [**os.major(device)**](https://www.tutorialspoint.com/python/os_major.htm)  Extract the device major number from a raw device number. |
| 32 | [**os.makedev(major, minor)**](https://www.tutorialspoint.com/python/os_makedev.htm)  Compose a raw device number from the major and minor device numbers. |
| 33 | [**os.makedirs(path[, mode])**](https://www.tutorialspoint.com/python/os_makedirs.htm)  Recursive directory creation function. |
| 34 | [**os.minor(device)**](https://www.tutorialspoint.com/python/os_minor.htm)  Extract the device minor number from a raw device number. |
| 35 | [**os.mkdir(path[, mode])**](https://www.tutorialspoint.com/python/os_mkdir.htm)  Create a directory named path with numeric mode mode. |
| 36 | [**os.mkfifo(path[, mode])**](https://www.tutorialspoint.com/python/os_mkfifo.htm)  Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal). |
| 37 | [**os.mknod(filename[, mode=0600, device])**](https://www.tutorialspoint.com/python/os_mknod.htm)  Create a filesystem node (file, device special file or named pipe) named filename. |
| 38 | [**os.open(file, flags[, mode])**](https://www.tutorialspoint.com/python/os_open.htm)  Open the file file and set various flags according to flags and possibly its mode according to mode. |
| 39 | [**os.openpty()**](https://www.tutorialspoint.com/python/os_openpty.htm)  Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively. |
| 40 | [**os.pathconf(path, name)**](https://www.tutorialspoint.com/python/os_pathconf.htm)  Return system configuration information relevant to a named file. |
| 41 | [**os.pipe()**](https://www.tutorialspoint.com/python/os_pipe.htm)  Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively. |
| 42 | [**os.popen(command[, mode[, bufsize]])**](https://www.tutorialspoint.com/python/os_popen.htm)  Open a pipe to or from command. |
| 43 | [**os.read(fd, n)**](https://www.tutorialspoint.com/python/os_read.htm)  Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned. |
| 44 | [**os.readlink(path)**](https://www.tutorialspoint.com/python/os_readlink.htm)  Return a string representing the path to which the symbolic link points. |
| 45 | [**os.remove(path)**](https://www.tutorialspoint.com/python/os_remove.htm)  Remove the file path. |
| 46 | [**os.removedirs(path)**](https://www.tutorialspoint.com/python/os_removedirs.htm)  Remove directories recursively. |
| 47 | [**os.rename(src, dst)**](https://www.tutorialspoint.com/python/os_rename.htm)  Rename the file or directory src to dst. |
| 48 | [**os.renames(old, new)**](https://www.tutorialspoint.com/python/os_renames.htm)  Recursive directory or file renaming function. |
| 49 | [**os.rmdir(path)**](https://www.tutorialspoint.com/python/os_rmdir.htm)  Remove the directory path |
| 50 | [**os.stat(path)**](https://www.tutorialspoint.com/python/os_stat.htm)  Perform a stat system call on the given path. |
| 51 | [**os.stat\_float\_times([newvalue])**](https://www.tutorialspoint.com/python/os_stat_float_times.htm)  Determine whether stat\_result represents time stamps as float objects. |
| 52 | [**os.statvfs(path)**](https://www.tutorialspoint.com/python/os_statvfs.htm)  Perform a statvfs system call on the given path. |
| 53 | [**os.symlink(src, dst)**](https://www.tutorialspoint.com/python/os_symlink.htm)  Create a symbolic link pointing to src named dst. |
| 54 | [**os.tcgetpgrp(fd)**](https://www.tutorialspoint.com/python/os_tcgetpgrp.htm)  Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()). |
| 55 | [**os.tcsetpgrp(fd, pg)**](https://www.tutorialspoint.com/python/os_tcsetpgrp.htm)  Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg. |
| 56 | [**os.tempnam([dir[, prefix]])**](https://www.tutorialspoint.com/python/os_tempnam.htm)  Return a unique path name that is reasonable for creating a temporary file. |
| 57 | [**os.tmpfile()**](https://www.tutorialspoint.com/python/os_tmpfile.htm)  Return a new file object opened in update mode (w+b). |
| 58 | [**os.tmpnam()**](https://www.tutorialspoint.com/python/os_tmpnam.htm)  Return a unique path name that is reasonable for creating a temporary file. |
| 59 | [**os.ttyname(fd)**](https://www.tutorialspoint.com/python/os_ttyname.htm)  Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised. |
| 60 | [**os.unlink(path)**](https://www.tutorialspoint.com/python/os_unlink.htm)  Remove the file path. |
| 61 | [**os.utime(path, times)**](https://www.tutorialspoint.com/python/os_utime.htm)  Set the access and modified times of the file specified by path. |
| 62 | [**os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])**](https://www.tutorialspoint.com/python/os_walk.htm)  Generate the file names in a directory tree by walking the tree either top-down or bottom-up. |
| 63 | [**os.write(fd, str)**](https://www.tutorialspoint.com/python/os_write.htm)  Write the string str to file descriptor fd. Return the number of bytes actually written. |

# Python Exceptions Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them −

* **Exception Handling:**
* **Assertions:**

List of Standard Exceptions:

|  |  |
| --- | --- |
| **EXCEPTION NAME** | **DESCRIPTION** |
| Exception | Base class for all exceptions |
| StopIteration | Raised when the next() method of an iterator does not point to any object. |
| SystemExit | Raised by the sys.exit() function. |
| StandardError | Base class for all built-in exceptions ***except StopIteration and SystemExit.*** |
| ArithmeticError | Base class for all errors that occur for numeric calculation. |
| OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| FloatingPointError | Raised when a floating point calculation fails. |
| ZeroDivisionError | Raised when division or modulo by zero takes place for all numeric types. |
| AssertionError | Raised in case of failure of the Assert statement. |
| AttributeError | Raised in case of failure of attribute reference or assignment. |
| EOFError | Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| ImportError | Raised when an import statement fails. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| LookupError | Base class for all lookup errors. |
| IndexError  KeyError | Raised when an index is not found in a sequence.  Raised when the specified key is not found in the dictionary. |
| NameError | Raised when an identifier is not found in the local or global namespace. |
| UnboundLocalError  EnvironmentError | Raised when trying to access a local variable in a function or method but no value has been assigned to it.  Base class for all exceptions that occur outside the Python environment. |
| IOError  IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.  Raised for operating system-related errors. |
| SyntaxError  IndentationError | Raised when there is an error in Python syntax.  Raised when indentation is not specified properly. |
| SystemError | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| SystemExit | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| ValueError | Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| RuntimeError | Raised when a generated error does not fall into any category. |
| NotImplementedError | Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |

### **Assertions in Python**

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is like to a **raise-if** statement (or to be more accurate, a **raise-if-not** statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for ***valid input***, and after a function call to check for ***valid output.***

### **The *assert* Statement**

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The syntax for assert is −

assert Expression[, Arguments]

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the ***try-except*** statement, but if not handled, they will terminate the program and produce a traceback.

### **Example 1.**

Here is a function that converts a temperature from degrees Celsius to Fahrenheit. Since zero degrees celsius is as cold as it gets, the function bails out if it sees a negative temperature

#!/usr/bin/python

def CelsiusToFahrenheit(Temperature):

assert (Temperature >= 0),"Colder than absolute zero!"

return (Temperature \* 1.8)+32

print(CelsiusToFahrenheit(37))

print(int(CelsiusToFahrenheit(37)))

print(CelsiusToFahrenheit (-5))

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

print(CelsiusToFahrenheit(37))

98.60000000000001

print(int(CelsiusToFahrenheit(37)))

98

print(CelsiusToFahrenheit (-5))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in CelsiusToFahrenheit

AssertionError: Colder than absolute zero!

### **Example 2.**

def value\_comparison\_f():

return 3

def test\_value\_comparison\_f():

# argument not specified

assert value\_comparison\_f() == 4

test\_value\_comparison\_f()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in test\_value\_comparison\_f

**AssertionError**

def test\_set\_comparison():

set1 = set("12345")

set2 = set("678910")

assert (set1 == set2), "Set elements should match."

test\_set\_comparison()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 4, in test\_set\_comparison

AssertionError: Set elements should match.

## **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 cannot 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 terminates and quits.

## **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**

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all:

#!/usr/bin/python

NOTE: In my case without admin priviliges, I got this error.

C:\Users\haryadav>chdir C:\Program Files\Python36\fileIO\

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Hari Yadav

Welcome to python

try:

fh = open("testfile.txt", "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 produces the following result −

Error: can't find file or read data

NOTE: With admin priviliges.

C:\Users\haryadav>chdir C:\Program Files\Python36\fileIO\

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Hari Yadav

Welcome to python

try:

fh = open("testfile.txt", "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 produces the following result −

Written content in the file successfully

## **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.

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>chdir C:\Program Files\Python36\fileIO\

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

Hari Yadav

Welcome to python

try:

fh = open("testfile.txt", "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()

Error: can't find file or read data

try:

#fh = open("testfile.txt", "w")

v\_num =100/0

except:

print("Unknown error. Please check your programm carefully.")

Unknown error. Please check your programm carefully.

try:

fh = open("testfile.txt", "w")

# v\_num =100/0

except:

print("Unknown error. Please check your programm carefully.")

Unknown error. Please check your programm carefully.

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.

NOTE: dev\_resume.txt file is not available.

try:

f = open("dev\_resume.txt")

# name = input("Enter your name : ")

except (KeyboardInterrupt, FileNotFoundError, PermissionError):

print("\nPlease try again.")

NOTE: After run, Press control + c to interupt the input from keyboard.

try:

# f = open("dev\_resume.txt")

name = input("Enter your name : ")

except (KeyboardInterrupt, FileNotFoundError, PermissionError):

print("\nPlease try again.")

## **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.

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

You cannot 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  
Traceback (most recent call last):

File "<stdin>", line 2, in <module>

PermissionError: [Errno 13] Permission denied: 'testfile'

Same example can be written more cleanly as follows −

#!/usr/bin/python

try:

fh = open("testfile.txt", "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 write 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 receives 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 produces 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 is as follows.

### **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 the 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 the exception as follows −

try:

raise Networkerror("Bad hostname")

except Networkerror,e:

print e.args

# Python 3 - Object Oriented

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy.

## **Overview of OOP Terminology**

* **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
* **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
* **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
* **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
* **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
* **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
* **Instance** − An individual object of a certain class. An object **obj** that belongs to a class **Circle**, for example, is an instance of the class **Circle**.
* **Instantiation** − The creation of an instance of a class.
* **Method**− A special kind of function that is defined in a class definition.
* **Object** − A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.
* **Operator overloading** − The assignment of more than one function to a particular operator.

## **Creating Classes**

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

class ClassName:

'Optional class documentation string'

class\_suite

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

### **Example**

Following is an example of a simple Python class –

del(Employee)

class Employee:

'Common base class for all employees'

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

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

def displayEmployee(self):

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

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

## **Creating Instance Objects**

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

# This would create first object of Employee class

emp1 = Employee("Zara", 2000)

# This would create second object of Employee class

emp2 = Employee("Manni", 5000)

## **Accessing Attributes**

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

emp1.displayEmployee()

emp2.displayEmployee()

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

Now, putting all the concepts together −

#!/usr/bin/python3

class Employee:

'Common base class for all employees'

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

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

def displayEmployee(self):

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

#This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

#This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

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

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

Name : Zara ,Salary: 2000

Name : Manni ,Salary: 5000

Total Employee 2

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

emp1.displayEmployee()

Name : Zara , Salary: 2000

emp1.salary = 7000 # Update 'salary' attribute.

emp1.name = 'Mark Tully' # Update 'name' attribute.

emp1.displayEmployee()

Name : Mark Tully , Salary: 7000

emp1.age = 50 # Add 'age' attribute.

emp1.displayEmployee()

Name : Mark Tully , Salary: 7000

del emp1.salary # Delete 'salary' attribute.

emp1.displayEmployee()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 11, in displayEmployee

AttributeError: 'Employee' object has no attribute 'salary'

emp2.displayEmployee()

Name : Manni , Salary: 5000

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

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

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

False

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

True

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

True

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

'Mark Tully'

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

50

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

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**AttributeError: 'Employee' object has no attribute 'salary'**

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

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

7000

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

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

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Employee' object has no attribute 'salary'

## **Built-In Class Attributes**

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

* **\_\_dict\_\_** − Dictionary containing the class's namespace.
* **\_\_doc\_\_** − Class documentation string or none, if undefined.
* **\_\_name\_\_** − Class name.
* **\_\_module\_\_** − Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode.
* **\_\_bases\_\_** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

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

#!/usr/bin/python3

class Employee:

'Common base class for all employees'

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

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

def displayEmployee(self):

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

emp1 = Employee("Zara", 2000)

emp2 = Employee("Manni", 5000)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

<function Employee.displayEmployee at 0x0160D300>,

'\_\_weakref\_\_':

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

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

}

## **Destroying Objects (Garbage Collection)**

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

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

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

a = 40 # Create object <40>

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

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

del a # Decrease ref. count of <40>

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

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

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

### **Example**

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

#!/usr/bin/python3

class Point:

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

self.x = x

self.y = y

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

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

print (class\_name, "destroyed")

pt1 = Point()

pt2 = pt1

pt3 = pt1

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

del pt1

del pt2

del pt3

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

3083401324 3083401324 3083401324

Point destroyed

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

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

#!/usr/bin/python3

import point

p1 = point.Point()

## **Class Inheritance**

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

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

### **Syntax**

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

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

'Optional class documentation string'

class\_suite

### **Example**

#!/usr/bin/python3

class Parent: # define parent class

parentAttr = 100

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

print ("Calling parent constructor")

def parentMethod(self):

print ('Calling parent method')

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

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

class Child(Parent): # define child class

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

print ("Calling child constructor")

def childMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.childMethod() # child calls its method

c.parentMethod() # calls parent's method

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

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

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

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

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

class A: # define your class A

.....

class B: # define your calss B

.....

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

.....

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

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

## **Overriding Methods**

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

### **Example**

#!/usr/bin/python3

class Parent: # define parent class

def myMethod(self):

print ('Calling parent method')

class Child(Parent): # define child class

def myMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.myMethod() # child calls overridden method

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

Calling child method

## **Base Overloading Methods**

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

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

## **Overloading Operators**

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

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

### **Example**

#!/usr/bin/python3

class Vector:

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

self.a = a

self.b = b

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

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

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

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

v1 = Vector(2,10)

v2 = Vector(5,-2)

print (v1 + v2)

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

Vector(7,8)

## **Data Hiding**

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

### **Example**

#!/usr/bin/python3

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print (self.\_\_secretCount)

counter = JustCounter()

counter.count()

counter.count()

print (counter.\_\_secretCount)

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

1

2

Traceback (most recent call last):

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

print counter.\_\_secretCount

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

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

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

print (counter.\_JustCounter\_\_secretCount)

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

1

2

2

# Python 3 - Regular Expressions

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

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

We would cover two important functions, which would be used to handle regular expressions. Nevertheless, a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

### **Basic patterns that match single chars**

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

### **Compilation flags**

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

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

## **The match Function**

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

Here is the syntax for this function −

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

Here is the description of the parameters −

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

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

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

### **Example 1.**

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs"

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

if matchObj:

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

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

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

else:

print ("No match!!")

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

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

### **Example 2.**

#!/usr/bin/python3

import re

# Sample strings.

list = ["dog dot", "do don't", "dumb-dumb", "no match"]

# Loop.

for element in list:

# Match if two words starting with letter d.

m = re.match("(d\w+)\W(d\w+)", element)

# See if success.

if m:

print(m.groups())

NOTE: Pattern details

Pattern: (d\w+)\W(d\w+)

d Lowercase letter d.

\w+ One or more word characters.

\W A non-word character.

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

('dog', 'dot')

('do', 'don')

('dumb', 'dumb')

### **Example 3.**

# Sample strings.

list = ["dog dot", "do don't", "dumb-dumb", "no match"]

# Loop.

for element in list:

# Match if two words starting with letter d.

m = re.match("(n\w+)\W(m\w+)", element)

# See if success.

if m:

print(m.groups())

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

('no', 'match')

## **The search Function**

**Search**. This method is different from match. Both apply a pattern. But search attempts this at all possible starting points in the string. Match just tries the first starting point.

So:

**Search** scans through the input string and tries to match at any location.

Here is the syntax for this function −

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

Here is the description of the parameters −

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

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

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

### **Example 1.**

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs";

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

if searchObj:

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

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

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

else:

print ("Nothing found!!")

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

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

### **Example 2.**

#!/usr/bin/python3

import re

# Input.

value = "voorheesville"

m = re.search("(vi.\*)", value)

if m:

# This is reached.

print("search:", m.group(1))

m = re.match("(vi.\*)", value)

if m:

# This is not reached.

print("match:", m.group(1))

NOTE: Pattern details

Pattern: (vi.\*)

vi The lowercase letters v and i together.

.\* Zero or more characters of any type.

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

Output

search: ville

## **Matching Versus Searching**

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string.

### **Example**

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs";

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

if matchObj:

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

else:

print ("No match!!")

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

if searchObj:

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

else:

print ("Nothing found!!")

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

No match!!

search --> matchObj.group() : dogs

## **The Search and Replace function (sub)**

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

### **Syntax**

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

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

### **Example**

#!/usr/bin/python3

import re

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

# Delete Python-style comments

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

print ("Phone Num : ", num)

# Remove anything other than digits

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

print ("Phone Num : ", num)

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

Phone Num : 2004-959-559

Phone Num : 2004959559

## **The split Function**

**Split**. The **re.split()** method accepts a pattern argument. This pattern specifies the delimiter. With it, we can use any text that matches a pattern as the delimiter to separate text data.

A split() method is also available directly on a string. This method handles no regular expressions. It is simpler.

### **Syntax**

re.split(pattern, string)

We split the string on one or more non-digit characters.

**Example 1.**

#!/usr/bin/python3

import re

# Input string.

value = "one 1 two 2 three 3"

# Separate on one or more non-digit characters.

result = re.split("\D+", value)

# Print results.

for element in result:

print(element)

NOTE: Pattern details

Pattern: \D+

\D+ One or more non-digit characters.

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

1

2

3

**Example 2.**

**Split the string where ever there is capital letter.**

import re

# Input string.

value = "DogCatElephant"

# Separate on one or more non-digit characters.

result = re.split("([A-Z][^A-Z]\*)", value)

# Print results.

for element in result:

print(element)

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

Dog

Cat

Elephant

[s for s in re.split("([A-Z][^A-Z]\*)", "DogCatElephant") if s]

['Dog', 'Cat', 'Elephant']

## **The Findall Function**

**Findall**. This is similar to **split()**. **Findall** accepts a pattern that indicates which strings to return in a list. It is like **split()** but we specify matching parts, not delimiters.

### **Syntax**

re.findall(pattern, **string**[, pos[, endpos]])

We scan a string for all words starting with the letter d or p, and with one or more following word characters.

**Example 1.**

import re

# Input.

value = "abc 123 def 456 dot map pat"

# Find all words starting with d or p.

list = re.findall("[dp]\w+", value)

# Print result.

print(list)

**NOTE: Pattern details**

Pattern: [dp]\w+

[dp] lowercase d, or a lowercase p.

\w+ One or more word characters.

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

['def', 'dot', 'pat']

## **The Finditer Function**

Unlike **re.findall**, which returns strings, finditer returns matches. For each match, we call methods like start() or end(). And we can access the value of the match with group().

**start(), end()**. We can use special characters in an expression to match the start and end of a string. For the start, we use the character **"^"** and for the end, we use the **"$"** sign.

### **Syntax**

re.findall(pattern, string[, pos[, endpos]])

We loop over a list of strings and call re.match. We detect all the strings that start or end with a digit character "\d."

**Tip:**

The match method tests from the leftmost part of the string. So to test the end, we use ".\*" to handle these initial characters.

**Example 1.**

import re

list = ["123", "4cat", "dog5", "6mouse"]

for element in list:

# See if string starts in digit.

m = re.match("^\d", element)

if m:

print("START:", element)

# See if string ends in digit.

m = re.match(".\*\d$", element)

if m:

print(" END:", element)

Pattern details

^\d Match at the start, check for single digit.

.\*\d$ Check for zero or more of any char.

Check for single digit.

Match at the end.

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

START: 123

END: 123

START: 4cat

END: dog5

START: 6mouse

## **Regular Expression Modifiers: Option Flags**

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

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

## **Regular Expression Patterns**

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

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

Here is the list of regular expression syntax in Python.

## **Regular Expression Examples**

### **Literal characters**

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

## **Character classes**

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

## **Special Character Classes**

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

## **Repetition Cases**

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

## **Nongreedy repetition**

This matches the smallest number of repetitions −

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

## **Grouping with Parentheses**

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

## **Backreferences**

This matches a previously matched group again −

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

## **Alternatives**

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

## **Anchors**

This needs to specify match position.

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

## **Special Syntax with Parentheses**

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