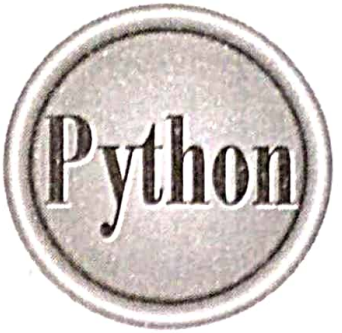
The Complete Reference



Chapter 8

Python's Built-In Functions

The bulk of functionality that Python provides in its standard form is handled by a series of external modules. These modules cover everything from the basic mechanics of getting command-line arguments and system configuration and information right up to complex modules for talking to SMTP and HTTP servers and to other systems and APIs.

However, despite the large library that comes standard with Python, you still need some built-in functionality to create, manipulate, and otherwise determine information about the objects, types, and classes that make up the Python language. This chapter concentrates on these built-in functions, while the rest of this part is given over to looking at specific modules within the Python standard library.

Note *The information contained in this chapter is based on the documentation available for Python 2.0. Updated versions of the full documentation supplied with Python can be found on the Python and MCwords web sites. See Appendix A for details.*

The functions in this part are part of the **__builtin__** module.

__import__ (name [, globals [, locals [, fromlist]]])

The **__import__()** function is invoked automatically by the **import** statement. For example, the statement

```
import module
```

results in the following call to **__import__()**:

```
__import__('module',globals(),locals(),[])
```

while the call

```
from module import class
```

results in the following call to **__import__()**:

```
__import('module',globals(),locals(),['class'])
```

The **__import__()** function exists mainly so that you can optionally replace it with your own import function. See the **ihooks** and **rexec** modules for more examples.

abs(x)

The **abs()** function returns the absolute value of a number (plain, long integer, or floating-point). If you supply a complex number, only the magnitude is returned. Here's an example:

```
>>> print abs(-2.4)
2.4
>>> print abs(4+2j)
4.472135955
```


apply(function, args [, keywords])

The **apply()** function applies the arguments **args** to **function**, which must be a callable object (function, method, or other callable object). The **args** must be supplied as a sequence; lists are converted to tuples before being applied. The **function** is called using **args** as individual arguments. For example,

```
apply(add,(1,3,4))
```

is equivalent to

```
add(1,3,4)
```

You need to use the **apply()** function in situations where you are building up a list of arguments in a list or tuple and you want to supply the list as individual arguments. This is especially useful in situations where you want to supply a varying list of arguments to a function.

The optional **keywords** argument should be a dictionary whose keys are strings; these strings will be used as keyword arguments to be supplied to the end of the argument list.

buffer(object [, offset [, size]])

The **buffer()** function creates a new buffer on **object**, providing that **object** supports the buffer call interface; such objects include strings, arrays, and buffers. The new buffer references **object** using a slice starting from **offset** and extending to the end of the object or to the length **size**. If no arguments are given, the buffer covers the entire sequence. The resulting buffer object is a read-only copy of the object's data.

Buffer objects are used to create a more friendly interface to certain object types. For example, the string object type is made available through a buffer object that allows you to access the information in the string on a byte-by-byte basis.

callable(object)

The **callable()** function returns true if **object** is callable, false if **object** is not callable. Callable objects include functions, methods, and code objects, and also classes (which return a new instance when called) and class instances that have the **call** method defined.

chr(i)

The **chr()** function returns a single character string matching the ASCII code **i**, as in the following example:

```
>>> print chr(72)+chr(101)+chr(108)+chr(108)+chr(111)
Hello
```

The **chr()** function is the opposite of the **ord()** function, which converts characters back to ASCII integer codes. The argument **i** should be in the range 0–255; a **ValueError** exception is raised if the argument is outside that limit.

cmp(x, y)

The **cmp()** function compares the two objects **x** and **y** and returns an integer depending on the outcome. The return value is negative if **x < y**, 0 if **x == y**, and strictly positive if **x > y**. Note that this specifically compares the values rather than any reference relationship, such that

```
>>> a=99
>>> b=int('99')
>>> cmp(a,b)
0
```

coerce(x, y)

The **coerce()** function returns a tuple consisting of the two numeric arguments converted to a common type, using the same rules used by arithmetic operations. Here's are two examples:

```
>>> a = 1
>>> b = 1.2
```

```
>>> coerce(a,b)
(1.0, 1.2)
>>> a=1+2j
>>> b=4.3e10
>>> coerce(a,b)
((1+2j), (43000000000+0j))
```

compile(string, filename, kind)

The **compile()** function compiles **string** into a code object, which can later be executed by the **exec** statement to be evaluated using the **eval** function. The **filename** should be the name of the file from which the code was read, or a suitable identifier if generated internally. The **kind** argument specifies what kind of code is contained in **string**. See Table 8-1 for more information of the possible values of **kind**.

For example:

```
>>> a=compile('print "Hello World"','<string>','single')
>>> exec(a)
Hello World
>>> eval(a)
Hello World
```

complex(real [, imag])

The **complex()** function returns a complex number with the real component **real** and the imaginary component **imag**, if supplied. If **imag** isn't supplied, the imaginary component is 0j.

kind Value	Code Compiled
exec	Sequence of statements
eval	Single expression
single	Single interactive statement

Table 8-1. *The Kinds of Code Compiled by the **compile()** Function*

delattr(object, name)

The **delattr()** function deletes the attribute **name** from **object**, providing that **object** allows you to. This function is identical to the statement

```
del object.attr
```

but **delattr()** allows you to define **object** and **name** programmatically, rather than explicitly in the code.

dir([object])

When supplied without an argument, the **dir()** function lists the names in the current local symbol table, as in the following example:

```
>>> import smtplib, sys, os
>>> dir()
['__builtins__', '__doc__', '__name__', 'os', 'smtplib', 'sys']
```

When supplied with an argument, the **dir()** function returns a list of attributes for that object. This can be useful for determining the objects and methods defined within a module:

```
>>> import sys
>>> dir(sys)
['__doc__', '__name__', '__stderr__', '__stdin__', '__stdout__',
 'argv', 'builtin_module_names', 'byteorder', 'copyright',
 'exc_info', 'exc_type', 'exec_prefix', 'executable', 'exit',
 'getdefaultencoding', 'getrecursionlimit', 'getrefcount',
 'hexversion', 'maxint', 'modules', 'path', 'platform', 'prefix',
 'ps1', 'ps2', 'setcheckinterval', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info']
```

The information is built from the **__dict__**, **__methods__**, and **__members__** attributes of the given object and may not be complete. For example, methods and attributes inherited from other classes are not normally included.

divmod(a, b)

The **divmod()** function returns a tuple that contains the quotient and remainder of **a** divided by **b**, as in the following example:

```
>>> divmod(7,4)
(1, 3)
```

For integers, the value returned is the same as **a / b** and **a % b**. If the values supplied are floating-point numbers, the result is **(q, a % b)**, where **q** is usually **math.floor(a / b)** but may be one less than that. In any case, **q * b + a % b** is very close to **a**; if **a % b** is non-zero, it has the same sign as **b**, and **0 <= abs(a % b) < abs(b)**. The following examples show how **divmod()** works with floating-point numbers:

```
>>> divmod(3.75, 1.125)
(3.0, 0.375)
>>> divmod(4.99, 1.001)
(4.0, 0.98600000000000065)
>>> divmod(-3.5, 1.1)
(-4.0, 0.90000000000000036)
```


eval(expression [, globals [, locals]])

The **eval()** function evaluates the string **expression**, parsing and evaluating it as a standard Python expression. When called without any additional arguments, **expression** has access to the same global and local objects in which it is called. Alternatively, you can supply the global and local symbol tables as dictionaries (see the descriptions of the **globals()** and **locals()** functions later in this chapter).

The return value of the **eval()** function is the value of the evaluated expression, as in the following example:

```
>>> a=99
>>> eval('divmod(a,7)')
(14,1)
```

Any syntax errors are raised as exceptions.

You can also use **eval()** to compile code objects such as those created by the **compile()** function, but only when the code object has been compiled using the "eval" mode.

To execute arbitrary Python code incorporating statements and expressions, use the **exec** statement or the **execfile()** function to dynamically execute a file. See the section "Executing Arbitrary Statements" at the end of this chapter for more information.

execfile(file [, globals [, locals]])

The **execfile()** function is identical to the **exec** statement (see the section "Executing Arbitrary Statements" at the end of this chapter for more information), except that it executes statements from a file instead of from a string. The **globals** and **locals** arguments

should be dictionaries containing the symbol tables that will be available to the file during execution. If **locals** is omitted, then all references use the **globals** namespace. If both arguments are omitted, the file has access to the current symbol tables at the time of execution.

filter(function, list)

The **filter()** function filters the items in **list** according to whether **function** returns true, returning the new list, as in the following example:

```
a=[1,2,3,4,5,6,7,8,9]
b=filter(lambda x: x > 6, a)
print b
```

If **function** is **None**, the identity function is used and all the elements in **list** that are false are removed instead.

float(x)

The **float()** function converts **x**, which can be a string or number, to a floating-point number.

getattr(object, name [, default])

The **getattr()** function returns the value of the attribute **name** of **object**. Syntactically the statement

```
getattr(x,'myvalue')
```

is identical to

```
x.myvalue
```

If **name** does not exist, the function returns **default** if supplied, or raises an **AttributeError** otherwise.

globals()

The **globals()** function returns a dictionary that represents the current global symbol table. This is always the dictionary of the current module. If **globals()** is called within a function or method, it returns the symbol table for the module where the function or method is defined, not the function from where it is called.

hasattr(object, name)

The **hasattr()** function returns true if **object** has an attribute matching the string **name**. Returns 0 otherwise.

hash(object)

The **hash()** function returns the integer hash value for an object. The hash value is the same for any two objects that compare equally. This function does not apply to mutable objects.

hex(x)

The **hex()** function converts an integer to a hexadecimal string that is a valid Python expression.

id(object)

The **id()** function returns an integer (or long integer)—the object's *identity*—which is guaranteed to be unique and constant during the lifetime of the object.

input([prompt])

The **input()** function is equivalent to **eval(raw_input(prompt))**. See the description of the **raw_input()** function later in this chapter for more information.

int(x [, radix])

The **int()** function converts the number or string **x** to a plain integer. The **radix** argument, if supplied, is used as the base for the conversion and should be an integer in the range 2–36.

intern(string)

The **intern()** function adds **string** to the table of interned strings, returning the interned version. *Interned strings* are available through a pointer, rather than a raw string, allowing lookups of dictionary keys to be made using pointer comparisons instead of string comparisons. This provides a small performance gain over the normal string-comparison methods.

Names used within the Python namespace tables and the dictionaries used to hold module, class, or instance attributes are normally interned to speed up the execution of the script.

Interned strings are not garbage collected, so be aware that using interned strings on large dictionary key sets increases the memory requirements significantly, even after the dictionary keys have gone out of scope.

isinstance(object, class)

The **isinstance()** function returns true if **object** is an instance of **class**. The determination follows the normal inheritance rules and subclasses. You can also use the **isinstance()** function to identify if **object** is of a particular type by using the type class definitions in the **types** module. If **class** is not a class or type object, a **TypeError** exception is raised.

issubclass(class1, class2)

The **issubclass()** function returns true if **class1** is a subclass of **class2**. A class is always considered a subclass of itself. A **TypeError** exception is raised if either argument is not a class object.

len(s)

The **len()** function returns the length of a sequence (string, tuple, or list) or dictionary object.

list(sequence)

The **list()** function returns a list whose items and order are the same as those in **sequence**, as in the following example:

```
>>> list('abc')
['a', 'b', 'c']
>>> list([1,2,3])
[1, 2, 3]
```

locals()

The **locals()** function returns a dictionary that represents the current local symbol table.

long(x)

The **long()** function converts a string or number to a long integer. The conversion of a floating-point number follows the same rules as **int()**.

map(function, list, ...)

The **map()** function applies **function** to each item of **list** and returns the new list, as in the following example:

```
>>> a=[1,2,3,4]
>>> map(lambda x: pow(x,2), a)
[1,4,9,16]
```

If additional lists are supplied, they are supplied to **function** in parallel. Lists are padded with **None** until all lists are of the same length.

If **function** is **None**, the identity function is assumed, causing **map()** to return list with all false arguments removed. If the **function** is **None** and multiple list arguments are supplied, a list of tuples of each argument of the list is returned, as in the following example:

```
>>> map(None, [1,2,3,4], [4,5,6,7])
[(1, 4), (2, 5), (3, 6), (4, 7)]
```

The result of the preceding example is identical to the result produced by the **zip()** function.

max(s [, args...])

When supplied with a single argument, the **max()** function returns the maximum value in the sequence **s**. When supplied a list of arguments, the **max()** function returns the largest argument from those supplied. See the description of the **min()** function for more details.

min(s [, args...])

When supplied with a single argument, the **min()** function returns the minimum value in the sequence **s**. When supplied a list of arguments, the **min()** function returns the smallest value of all the arguments. Note that sequences in a multi-argument call are not traversed—each argument is compared as a whole, such that

```
min([1,2,3],[4,5,6])
```

returns

```
[1, 2, 3]
```

and not the often expected 1. To get the minimum value of one or more lists, use concatenation:

```
min([1,2,3]+[4,5,6])
```

oct(x)

The **oct()** function converts an integer to an octal string. The result is a valid Python expression, as in the following example:

```
>>> oct(2001)
'03721'
```

Note that the returned value is always unsigned, such that **oct(-1)** yields *'037777777777'* on a 32-bit machine.

open(filename [, mode [, bufsize]])

The **open()** function opens the file identified by **filename**, using **mode** and the buffering type **bufsize**. This function returns a file object. (See Chapter 3 and Chapter 6 for more information on file objects.)

The **mode** is the same as that used by the system **fopen()** function; see Table 8-2 for a list of valid modes. If **mode** is omitted, it defaults to **r**.

Mode	Meaning
r	Open for reading.
w	Open for writing.
a	Open for appending (file position automatically seeks to the end during the open).
r+	Open for updating (reading and writing).
w+	Truncates (empties) the file and then opens it for reading and writing.
a+	Opens the file for reading and writing and automatically changes current file position to the end of the file.
b	When appended to any option, opens the file in binary rather than text mode. (This mode is available for Windows, DOS and some other operating systems only. Unix/MacOS/BeOS treat all files as binary, regardless of this option.)

Table 8-2. *File Modes for the* ***open()*** *Function*

The optional **bufsize** argument of the **open()** function determines the size of the buffer to use when reading from the file. Table 8-3 lists the supported **bufsize** values. If the **bufsize** argument is omitted, the system default is used.

ord(c)

The **ord()** function returns the ASCII or Unicode numeric code of the string of one character **c**. The **ord()** function is the inverse of the **chr()** and **unichr()** functions.

pow(x, y [, z])

The **pow()** function returns the value of **x** raised to the power of **y**. If **z** is supplied, this function calculates **x** raised to the power **y** modulo **z**. This calculation is more efficient than using

```
pow(x,y) % z
```

The arguments supplied to **pow()** should be numeric types, and the types supplied will determine the type of the return value. If the calculated value cannot be represented by the supplied argument types, an exception is raised. For example, the following call to **pow()** will fail:

```
pow(2,-1)
```

But

```
pow(2.0,-1)
```

is valid.

Bufsize Value	Description
0	Disable buffering.
1	Line buffered.
>1	Use a buffer that is approximately **bufsize** characters in length.
<0	Use the system default (line buffered for **tty** devices and fully buffered for any other file.

Table 8-3. *Buffer Sizes Supported by the **open()** Function*

range([start,] stop [, step])

The **range()** function returns a list of numbers starting from **start** and ending before **stop** using **step** as the interval. All numbers should be supplied and are returned as plain integers. If **step** is omitted, the step value defaults to 1. If **start** is omitted, the sequence starts at 0. Note that the two-argument form of the call assumes that **start** and **stop** are supplied; if you want to specify a **step**, you must supply all three arguments.

The following calls to **range()** use positive values of **step**:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
[5, 6, 7, 8, 9]
>>> range(5,25,5)
[5, 10, 15, 20]
```

Note that the final number is **stop** minus **step**; the range goes up to but not including the **stop** value.

If you supply a negative value to **step**, the range counts down rather than up. **stop** must be lower than **start**; otherwise the returned list will be empty. The following examples illustrate the use of **step** as a negative value:

```
>>> range(10,0,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> range (25,0,-5)
[25, 20, 15, 10, 5]
>>> range(0,10,-1)
[]
```

raw_input([prompt])

The **raw_input()** function accepts raw input from **sys.stdin** and returns a string. Input is terminated by a newline character, which is stripped before the string is returned to the caller. If **prompt** is supplied, it is written to **sys.stdout** without a trailing newline character and used as the prompt for input, as in the following example:

```
>>> name=raw_input('Name? ')
Name? Martin
```

If the **readline** module has been loaded, features such as line editing and history are supported during input.

reduce(function, sequence [, initializer])

The **reduce()** function applies **function** (supporting two arguments) cumulatively to each element of **sequence**, reducing the entire statement to a single value. For example, the following statement emulates the ! mathematical operator:

```
reduce(lambda x,y: x*y, [1,2,3,4,5])
```

The effect is to perform the calculation

```
((((1*2)*3)*4)*5)
```

which equals 120.

If **initializer** is supplied, it's used as the first element in the sequence:

```
>>> reduce(lambda x,y: x*y, [1,2,3,4,5],10)
1200
```

reload(module)

The **reload()** function reloads an already imported module. The reload includes the normal parsing and initializing processes employed when the module was imported originally. This allows you to reload a changed Python module without needing to exit the interpreter.

There are a number of caveats for using **reload()**:

- If the module is syntactically correct but fails during initialization, the import process does not bind its name correctly in the symbol table. You will need to use the **import()** function to load the module before it can be reloaded.
- The reloaded module does not delete entries in the symbol table for the old version of the module first. For identically named objects and functions this is not a problem, but if you rename an entity, its value remains in the symbol table after a reload.
- The reloading of extension modules (which rely on built-in or dynamically loaded libraries for support) is supported, but is probably pointless and may actually fail, depending entirely on how well behaved the dynamically loaded library is.

- If a module imports objects from another module using the **from...import...** form, the **reload()** function does not redefine the objects imported. You can get around this by using the **import...** form.
- Reloading modules that provide classes does not affect any existing instances of that class—the existing instances will continue to use the old method definitions. Only new instances of the class will use the new forms. This also holds true for derived classes.

repr(object)

The **repr()** function returns a string representation of **object**. This is identical to using back quotes (`) on an object or attribute. The string returned yields an object with the same value as that when passed to **eval()**, as in the following example:

```
>>> dict = {'One':1, 'Two':2, 'Many': {'Many':4, 'ManyMany':8}}
>>> repr(dict)
"{'One': 1, 'Many': {'Many': 4, 'ManyMany': 8}, 'Two': 2}"
```

round(x[, n])

The **round()** function returns the floating-point value **x** rounded to **n** digits after the decimal point, as in the following examples:

```
>>> round(0.4)
0.0
>>> round(0.5)
1.0
>>> round(-0.5)
-1.0
>>> round(1985,-2)
2000.0
```

setattr(object, name, value)

The **setattr()** function sets the attribute **name** of **object** to **value**. The **setattr()** function is the opposite of the **getattr()** function, which merely gets the information. The statement

```
setattr(myobj, 'myattr', 'new value')
```

is equivalent to

```
myobj.myattr = 'new value'
```

The **setattr()** function can be used in situations where the attribute is known pragmatically by name, rather than explicitly as an attribute.

slice([start,] stop [, step])

The **slice()** function returns a slice object that represents the set of indexes specified by **range(start, stop, step)**. If one argument is supplied, it's used as **stop**; if two arguments are supplied, they're used as **start** and **stop**. The default value for any unsupplied argument is **None**. Slice objects have three attributes (**start**, **stop**, and **step**) that merely return the argument supplied to the **slice()** function.

str(object)

The **str()** function returns a string representation of **object**. This is similar to the **repr()** function except that the return value is designed to be a printable string rather than a string that is compatible with the **eval()** function.

tuple(sequence)

The **tuple()** function returns a tuple whose items are the same and in the same order as the items in **sequence**. Here are two examples of the **tuple()** function:

```
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple([1,2,3])
(1, 2, 3)
```

type(object)

The **type()** function returns the type of **object**. The return value is a type object, as described by the **types** module. Here's an example:

```
>>> import types
>>> if type(string) == types.StringType:
    print "This is a string"
```

unichr(i)

The **unichr()** function returns a Unicode string of one character whose code is the integer **i**. This function is the Unicode equivalent of the **chr()** function described earlier in this chapter. Note that to convert a Unicode character back to its integer form, you can use **ord()**; there is no **uniord()** function. A **ValueError** exception is raised if the integer supplied is outside the range 0–65535.

unicode(string [, encoding [, errors]])

The **unicode()** function decodes a given **string** from one form to another, using the encoding format codec. Any error in encoding is tagged with the string in errors. Typically used to convert between string and Unicode encoding formats. The default behavior (when **encoding** is not supplied) is to decode UTF-8 in strict mode, with **errors** raising the **ValueError** exception. See the **codecs** module for a list of suitable codecs. See Chapter 10 for more information on how Unicode in Python works.

vars([object])

The **vars()** function returns a dictionary corresponding to the current local symbol table. When supplied with a module, class, or class instance, the **vars()** function returns a dictionary corresponding to that object's symbol table. Do not modify the returned dictionary because the effects are undefined.

xrange([start,] stop [, step])

The **xrange()** function works the same as the **range()** function, except that **xrange()** returns an **xrange** object. An **xrange** object is an opaque object type that returns the same information as the list that was requested, without having to store each individual element in the list. This is particularly useful in situations where you are creating very large lists; the memory saved by using **xrange()** over **range()** can be considerable.

zip(seq1, ...)

The **zip()** function takes a series of sequences and returns them as a list of tuples, where each tuple contains the nth element of each of the supplied sequences. Here's an example:

```
>>> a=[1,2,3,4]
>>> b=[5,6,7,8]
>>> zip(a,b)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Executing Arbitrary Statements

Python supports three constructs that allow you to execute some arbitrary file or a string of Python code. It can be useful to execute a statement that has either been dynamically built by the script, or perhaps provided by the user. The three constructs are the **exec** statement and the **execfile()** and **eval()** functions. However, make sure you are using the correct construct to execute the code or fragment because each statement is designed to perform a specific function.

exec Statement

The **exec** statement is designed to execute any piece of Python code that can use any combination of functions and statements. The code you execute has access to the same globally and locally defined objects, classes, and methods or functions. Here's a simple example using the **exec** statement:

```
exec "print 'Hello World'"
```

You can also restrict the available resources for the **exec** statement by supplying a dictionary containing the list of objects and their values, as in the example:

```
exec "print message" in mynamespace
```

where **mynamespace** is the dictionary you want to use.

You can also explicitly supply global and local dictionary namespaces using the statement

```
exec "print message" in myglobals, mylocals
```

You can use the **globals()** and **locals()** functions to get dictionaries of the current tables.

Note that the **exec** statement executes/evaluates expressions and statements, but it does not return a value. A syntax error is raised if you try to get a return value because **exec** is a statement, not a function:

```
>>> a=exec '3+4'
  File "<stdin>", line 1
    a=exec '3+4'
         ^
SyntaxError: invalid syntax
```

execfile() Function

The **execfile()** function, explained earlier in this chapter, performs the same operation as the **exec** statement except that it reads the statements to be executed from a file, rather than through a string or code object. In all other respects **execfile()** is identical to the **exec** statement.

eval() Function

The **eval()** function does not allow you to execute arbitrary Python statements; the **eval()** function is designed to execute a Python expression and return a value, as in the following example:

```
result=eval(userexpression)
```

or more explicitly in this statement

```
result=eval("99+45")
```

You *cannot* use the **eval()** function to execute a statement:

```
>>> a=eval('print "Hello world"')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<string>", line 1
    print "Hello world"
          ^
SyntaxError: invalid syntax
```

As a rule of thumb, use the **eval()** function to evaluate an expression to a return value, but use the **exec** statement in all other situations.