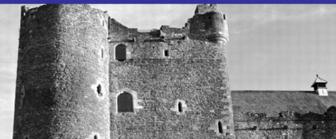
Python Essential Reference

Fourth Edition

Developer's Library



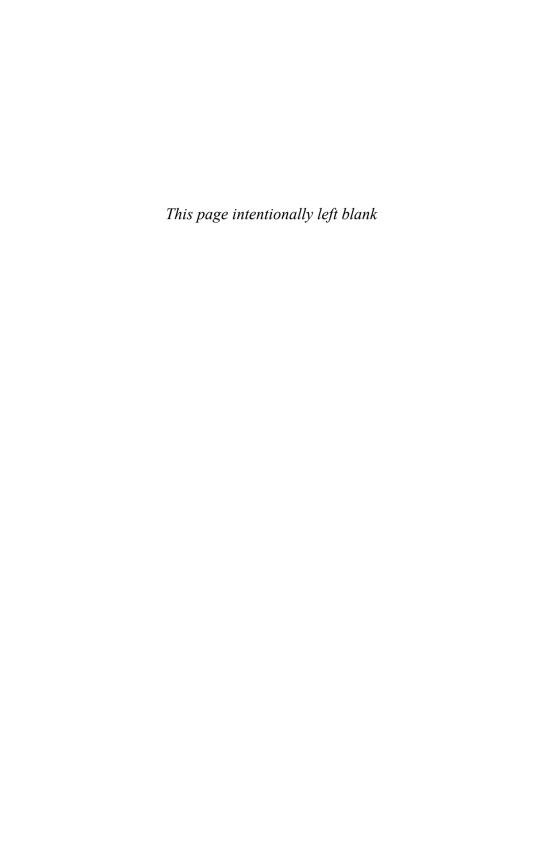
Python

ESSENTIAL REFERENCE

Fourth Edition

The Python Library

- 12 Built-In Functions
- 13 Python Runtime Services
- **14** Mathematics
- 15 Data Structures, Algorithms, and Utilities
- **16** String and Text Handling
- 17 Python Database Access
- 18 File and Directory Handling
- 19 Operating System Services
- 20 Threads and Concurrency
- 21 Network Programming and Sockets
- 22 Internet Application Programming
- 23 Web Programming
- 24 Internet Data Handling and Encoding
- 25 Miscellaneous Library Modules



Built-In Functions and Exceptions

This chapter describes Python's built-in functions and exceptions. Much of this material is covered less formally in earlier chapters of this book. This chapter merely consolidates all this information into one section and expands upon some of the more subtle features of certain functions. Also, Python 2 includes a number of built-in functions that are considered to be obsolete and which have been removed from Python 3. Those functions are not documented here—instead the focus is on modern functionality.

Built-in Functions and Types

Certain types, functions, and variables are always available to the interpreter and can be used in any source module. Although you don't need to perform any extra imports to access these functions, they are contained in a module __builtin__ in Python 2 and in a module builtins in Python 3. Within other modules that you import, the variable builtins is also bound to this module.

abs(x)

Returns the absolute value of x

all(s)

Returns True if all of the values in the iterable s evaluate as True.

any(s)

Returns True if any of the values in the iterable s evaluate as True.

ascii(x)

Creates a printable representation of the object x just like the repr(), but only uses ASCII characters in the result. Non-ASCII characters are turned into appropriate escape sequences. This can be used to view Unicode strings in a terminal or shell that doesn't support Unicode. Python 3 only.

basestring

This is an abstract data type that is the superclass of all strings in Python 2 (str and unicode). It is only used for type testing of strings. For example, isinstance (s, basestring) returns True if s is either kind of string. Python 2 only.

Returns a string containing the binary representation of the integer x.

bool([x])

bin(x)

Type representing Boolean values True and False. If used to convert x, it returns True if x evaluates to true using the usual truth-testing semantics (that is, nonzero number, non-empty list, and so on). Otherwise, False is returned. False is also the default value returned if bool () is called without any arguments. The bool class inherits from int so the Boolean values True and False can be used as integers with values 1 and 0 in mathematical calculations.

bytearray([x])

A type representing a mutable array of bytes. When creating an instance, x may be an iterable sequence of integers in the range 0 to 255, an 8-bit string or bytes literal, or an integer that specifies the size of the byte array (in which case every entry will be initialized to 0). A bytearray object a looks like an array of integers. If you perform a lookup such as a[i], you will get an integer value representing the byte value at index i. Assignments such as a[i] = v also require v to be an integer byte value. However, a bytearray also provides all of the operations normally associated with strings (that is, slicing, find(), split(), replace(), and so on). When using these string operations, you should be careful to preface all string literals with b in order to indicate that you're working with bytes. For example, if you wanted to split a byte array a into fields using a comma character separator, you would use a.split(b',') not a.split(','). The result of these operations is always new bytearray objects, not strings. To turn a bytearray a into a string, use the a.decode(encoding) method. An encoding of 'latin-1' will directly turn a bytearray of 8-bit characters into a string without any modification of the underlying character values.

bytearray(s, encoding)

An alternative calling convention for creating a bytearray instance from characters in a string s where *encoding* specifies the character encoding to use in the conversion.

bytes([x])

A type representing an immutable array of bytes. In Python 2, this is an alias for str() which creates a standard 8-bit string of characters. In Python 3, bytes is a completely separate type that is an immutable version of the bytearray type described earlier. In that case, the argument x has the same interpretation and can be used in the same manner. One portability caution is that even though bytes is defined in Python 2, the resulting object does not behave consistently with Python 3. For example, if a is an instance created by bytes(), then a[i] returns a character string in Python 2, but returns an integer in Python 3.

bytes(s, encoding)

An alternative calling convention for creating a bytes instance from characters in a string s where *encoding* specifies the character encoding to use. Python 3 only.

chr(x)

Converts an integer value, x, into a one-character string. In Python 2, x must be in the range 0 <= x <= 255, and in Python 3, x must represent a valid Unicode code point. If x is out of range, a ValueError exception is raised.

classmethod(func)

This function creates a class method for the function <code>func</code>. It is typically only used inside class definitions where it is implicitly invoked by the <code>@classmethod</code> decorator. Unlike a normal method, a class method receives the class as the first argument, not an instance. For example, if you had an object, <code>f</code>, that is an instance of class <code>Foo</code>, invoking a class method on <code>f</code> will pass the class <code>Foo</code> as the first argument to the method, not the instance <code>f</code>.

cmp(x, y)

Compares x and y and returns a negative number if x < y, a positive number if x > y, or 0 if x == y. Any two objects can be compared, although the result may be meaningless if the two objects have no meaningful comparison method defined (for example, comparing a number with a file object). In certain circumstances, such comparisons may also raise an exception.

compile(string, filename, kind [, flags [, dont_inherit]])

Compiles <code>string</code> into a code object for use with <code>exec()</code> or <code>eval()</code>. <code>string</code> is a string containing valid Python code. If this code spans multiple lines, the lines must be terminated by a single newline ('\n') and not platform-specific variants (for example, '\r\n' on Windows). <code>filename</code> is a string containing the name of the file in which the string was defined. <code>kind</code> is 'exec' for a sequence of statements, 'eval' for a single expression, or 'single' for a single executable statement. The <code>flags</code> parameter determines which optional features (associated with the <code>__future__</code> module) are enabled. Features are specified using the bitwise OR of flags defined in the <code>__future__</code> module. For example, if you wanted to enable new division semantics, you would set <code>flags</code> to <code>__future__</code>.division.compiler_flag. If <code>flags</code> is omitted or set to 0, the code is compiled with whatever features are currently in effect. If <code>flags</code> is supplied, the features specified are added to those features already in effect. If <code>dont_inherit</code> is set, only those features specified in <code>flags</code> are enabled—features currently enabled are ignored.

complex([real [, imag]])

Type representing a complex number with real and imaginary components, real and imag, which can be supplied as any numeric type. If imag is omitted, the imaginary component is set to zero. If real is passed as a string, the string is parsed and converted to a complex number. In this case, imag should be omitted. If no arguments are given, 0j is returned.

delattr(object, attr)

Deletes an attribute of an object. attr is a string. Same as del object.attr.

dict([m]) or dict(key1 = value1, key2 = value2, ...)

Type representing a dictionary. If no argument is given, an empty dictionary is returned. If m is a mapping object (such as a dictionary), a new dictionary having the same keys and same values as m is returned. For example, if m is a dictionary, dict (m) simply makes a shallow copy of it. If m is not a mapping, it must support iteration in which a sequence of (key, value) pairs is produced. These pairs are used to populate the dictionary. dict() can also be called with keyword arguments. For example, dict(foo=3, bar=7) creates the dictionary { 'foo': 3, 'bar': 7 }.

dir([object])

Returns a sorted list of attribute names. If object is a module, it contains the list of symbols defined in that module. If object is a type or class object, it returns a list of attribute names. The names are typically obtained from the object's __dict__ attribute if defined, but other sources may be used. If no argument is given, the names in the current local symbol table are returned. It should be noted that this function is primarily used for informational purposes (for example, used interactively at the command line). It should not be used for formal program analysis because the information obtained may be incomplete. Also, user-defined classes can define a special method __dir__() that alters the result of this function.

divmod(a, b)

Returns the quotient and remainder of long division as a tuple. For integers, the value (a // b, a % b) is returned. For floats, (math.floor(a / b), a % b) is returned. This function may not be called with complex numbers.

enumerate(iter[, initial value)

Given an iterable object, *iter*, returns a new iterator (of type enumerate) that produces tuples containing a count and the value produced from *iter*. For example, if *iter* produces a, b, c, then enumerate (*iter*) produces (0, a), (1, b), (2, c).

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

Evaluates an expression. expr is a string or a code object created by compile(). globals and locals are mapping objects that define the global and local namespaces, respectively, for the operation. If omitted, the expression is evaluated in the namespace of the caller. It is most common for globals and locals to be specified as dictionaries, but advanced applications can supply custom mapping objects.

exec(code [, global [, locals]])

Executes Python statements. code is a string, a file, or a code object created by compile(). globals and locals define the global and local namespaces, respectively, for the operation. If omitted, the code is executed in the namespace of the caller. If no global or local dictionaries are given, the behavior of this function is a little muddled between Python versions. In Python 2, exec is actually implemented as a special language statement, whereas Python 3 implements it as a standard library function. A subtle side effect of this implementation difference is that in Python 2, code evaluated by exec can freely mutate local variables in the caller's namespace. In Python 3, you can execute code that makes such changes, but they don't seem to have any lasting effect beyond the exec() call itself. This is because Python 3 uses locals() to obtain the local namespace if one isn't supplied. As you will note in the documentation for locals(), the returned dictionary is only safe to inspect, not modify.