2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

abs(*x*)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(*iterable*)

Return True if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

New in version 2.5.

any(*iterable*)

Return True if any element of the *iterable* is true. If the iterable is empty, return False. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

New in version 2.5.

basestring()

This abstract type is the superclass for `str` and `unicode`. It cannot be called or instantiated, but it can be used to test whether an object is an instance of `str` or `unicode`. `isinstance(obj, basestring)` is equivalent to `isinstance(obj, (str, unicode))`.

New in version 2.3.

bin(*x*)

Convert an integer number to a binary string. The result is a valid Python expression. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer.

New in version 2.6.

bool([*x*])

Convert a value to a Boolean, using the standard truth testing procedure. If *x* is false or omitted, this returns `False`; otherwise it returns `True`. `bool` is also a class, which is a subclass of `int`. Class `bool` cannot be subclassed further. Its only instances are `False` and `True`.

New in version 2.2.1.

Changed in version 2.3: If no argument is given, this function returns `False`.

bytearray([*source*[, *encoding*[, *errors*]]])

Return a new array of bytes. The `bytearray` type is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in *Mutable Sequence Types*, as well as most methods that the `str` type has, see *String Methods*.

The optional *source* parameter can be used to initialize the array in a few different ways:

- If it is a *string*, you must also give the *encoding* (and optionally, *errors*) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- If it is an *integer*, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the *buffer* interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an *iterable*, it must be an iterable of integers in the range `0 <= x < 256`, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

New in version 2.6.

callable(*object*)

Return `True` if the *object* argument appears callable, `False` if not. If this returns true, it is still possible that a call fails, but if it is false, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a `__call__()` method.

chr(*i*)

Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of `ord()`. The argument must be in the range [0..255], inclusive; `ValueError` will be raised if *i* is outside that range. See also `unichr()`.

classmethod(*function*)

Return a class method for *function*.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C(object):
    @classmethod
    def f(cls, arg1, arg2, ...):
        ...
```

The `@classmethod` form is a function *decorator* – see the description of function definitions in *Function definitions* for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()` in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in *The standard type hierarchy*.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

cmp(*x*, *y*)

Compare the two objects *x* and *y* and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y` and strictly positive if `x > y`.

compile(*source*, *filename*, *mode*[, *flags*[, *dont_inherit*]])

Compile the *source* into a code or AST object. Code objects can be executed by an `exec` statement or evaluated by a call to `eval()`. *source* can either be a string or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file (`'<string>'` is commonly used).

The *mode* argument specifies what kind of code must be compiled; it can be `'exec'` if *source* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).

The optional arguments *flags* and *dont_inherit* control which future statements (see **PEP 236**) affect the compilation of *source*. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module.

This function raises `SyntaxError` if the compiled source is invalid, and `TypeError` if the source contains null bytes.

> **Note:** When compiling a string with multi-line code in `'single'` or `'eval'` mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

Changed in version 2.3: The *flags* and *dont_inherit* arguments were added.

Changed in version 2.6: Support for compiling AST objects.

Changed in version 2.7: Allowed use of Windows and Mac newlines. Also input in `'exec'` mode does not have to end in a newline anymore.

complex([*real*[, *imag*]])

Create a complex number with the value *real* + *imag**j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the function serves as a numeric conversion function like `int()`, `long()` and `float()`. If both arguments are omitted, returns `0j`.

> **Note:** When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

The complex type is described in *Numeric Types — int, float, long, complex*.

delattr(*object*, *name*)

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`.

dict(***kwarg*)
dict(*mapping*, ***kwarg*)
dict(*iterable*, ***kwarg*)

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and *Mapping Types — dict* for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

dir([*object*])

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape(object):
```

```
        def __dir__(self):
            return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'perimeter', 'location']
```

> **Note:** Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(*a*, *b*)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 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)`.

Changed in version 2.3: Using `divmod()` with complex numbers is deprecated.

enumerate(*sequence*, *start=0*)

Return an enumerate object. *sequence* must be a sequence, an *iterator*, or some other object which supports iteration. The `next()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *sequence*:

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

New in version 2.3.

Changed in version 2.6: The *start* parameter was added.

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

The arguments are a string and optional globals and locals. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

Changed in version 2.4: formerly *locals* was required to be a dictionary.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and lacks '__builtins__', the current globals are copied into *globals* before *expression* is parsed. This means that *expression* normally has full access to the standard `__builtin__` module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval()` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with `'exec'` as the *mode* argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec` statement. Execution of statements from a file is supported by the `execfile()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `execfile()`.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

execfile(*filename*[, *globals*[, *locals*]])

This function is similar to the `exec` statement, but parses a file instead of a string. It is different from the `import` statement in that it does not use the module administration — it reads the file unconditionally and does not create a new module. [1]

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the *globals* and *locals* dictionaries as global and local namespace. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are passed as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

Changed in version 2.4: formerly *locals* was required to be a dictionary.

If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `execfile()` is called. The return value is `None`.

> **Note:** The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `execfile()` returns. `execfile()` cannot be used reliably to modify a function's locals.

file(*name*[, *mode*[, *buffering*]])

Constructor function for the `file` type, described further in section *File Objects*. The constructor's arguments are the same as those of the `open()` built-in function described below.

When opening a file, it's preferable to use `open()` instead of invoking this constructor directly. `file` is more suited to type testing (for example, writing `isinstance(f, file)`).

New in version 2.2.

filter(*function*, *iterable*)

Construct a list from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *iterable* is a string or a tuple, the result also has that type; otherwise it is always a list. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to `[item for item in iterable if function(item)]` if function is not `None` and `[item for item in iterable if item]` if function is `None`.

See `itertools.ifilter()` and `itertools.ifilterfalse()` for iterator versions of this function, including a variation that filters for elements where the *function* returns false.

float([*x*])

Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. The argument may also be [+|-]nan or [+|-]inf. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns `0.0`.

> **Note:** When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. Float accepts the strings nan, inf and -inf for NaN and positive or negative infinity. The case and a leading + are ignored as well as a leading - is ignored for NaN. Float always represents NaN and infinity as nan, inf or -inf.

The float type is described in *Numeric Types — int, float, long, complex*.

format(*value*[, *format_spec*])

Convert a *value* to a "formatted" representation, as controlled by *format_spec*. The

interpretation of *format_spec* will depend on the type of the *value* argument, however there is a standard formatting syntax that is used by most built-in types: *Format Specification Mini-Language*.

> **Note:** `format(value, format_spec)` merely calls `value.__format__(format_spec)`.

New in version 2.6.

frozenset([*iterable*])

Return a new **frozenset** object, optionally with elements taken from *iterable*. `frozenset` is a built-in class. See **frozenset** and *Set Types — set, frozenset* for documentation about this class.

For other containers see the built-in **set**, **list**, **tuple**, and **dict** classes, as well as the **collections** module.

New in version 2.4.

getattr(*object*, *name*[, *default*])

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise **AttributeError** is raised.

globals()

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

hasattr(*object*, *name*)

The arguments are an object and a string. The result is **True** if the string is the name of one of the object's attributes, **False** if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

hash(*object*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

help([*object*])

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the

argument is any other kind of object, a help page on the object is generated.

This function is added to the built-in namespace by the `site` module.

New in version 2.2.

hex(*x*)

Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression.

> **Note:** To obtain a hexadecimal string representation for a float, use the `float.hex()` method.

Changed in version 2.4: Formerly only returned an unsigned literal.

id(*object*)

Return the "identity" of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

> **CPython implementation detail:** This is the address of the object in memory.

input([*prompt*])

Equivalent to `eval(raw_input(prompt))`.

This function does not catch user errors. If the input is not syntactically valid, a `SyntaxError` will be raised. Other exceptions may be raised if there is an error during evaluation.

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Consider using the `raw_input()` function for general input from users.

int(*x=0*)
int(*x*, *base=10*)

Convert a number or string *x* to an integer, or return `0` if no arguments are given. If *x* is a number, it can be a plain integer, a long integer, or a floating point number. If *x* is floating point, the conversion truncates towards zero. If the argument is outside the integer range, the function returns a long object instead.

If *x* is not a number or if *base* is given, then *x* must be a string or Unicode object representing an *integer literal* in radix *base*. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2-36. Base-2, -8, and -16 literals can be optionally prefixed with

`0b`/`0B`, `0o`/`0O`/`0`, or `0x`/`0X`, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16.

The integer type is described in *Numeric Types — int, float, long, complex*.

isinstance(*object*, *classinfo*)

Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or *virtual*) subclass thereof. Also return true if *classinfo* is a type object (new-style class) and *object* is an object of that type or of a (direct, indirect or *virtual*) subclass thereof. If *object* is not a class instance or an object of the given type, the function always returns false. If *classinfo* is neither a class object nor a type object, it may be a tuple of class or type objects, or may recursively contain other such tuples (other sequence types are not accepted). If *classinfo* is not a class, type, or tuple of classes, types, and such tuples, a `TypeError` exception is raised.

Changed in version 2.2: Support for a tuple of type information was added.

issubclass(*class*, *classinfo*)

Return true if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a `TypeError` exception is raised.

Changed in version 2.3: Support for a tuple of type information was added.

iter(*o*[, *sentinel*])

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *o* must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, *sentinel*, is given, then *o* must be a callable object. The iterator created in this case will call *o* with no arguments for each call to its `next()` method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned.

One useful application of the second form of `iter()` is to read lines of a file until a certain line is reached. The following example reads a file until the `readline()` method returns an empty string:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

New in version 2.2.

len(*s*)

Return the length (the number of items) of an object. The argument may be a sequence

(string, tuple or list) or a mapping (dictionary).

list([*iterable*])

Return a list whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a list, a copy is made and returned, similar to `iterable[:]`. For instance, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`. If no argument is given, returns a new empty list, `[]`.

list is a mutable sequence type, as documented in *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange*. For other containers see the built in **dict**, **set**, and **tuple** classes, and the **collections** module.

locals()

Update and return a dictionary representing the current local symbol table. Free variables are returned by **locals()** when it is called in function blocks, but not in class blocks.

> **Note:** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

long(*x=0*)
long(*x*, *base=10*)

Convert a string or number to a long integer. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace. The *base* argument is interpreted in the same way as for **int()**, and may only be given when *x* is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns `0L`.

The long type is described in *Numeric Types — int, float, long, complex*.

map(*function*, *iterable*, *...*)

Apply *function* to every item of *iterable* and return a list of the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with **None** items. If *function* is **None**, the identity function is assumed; if there are multiple arguments, **map()** returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The *iterable* arguments may be a sequence or any iterable object; the result is always a list.

max(*iterable*[, *key*])
max(*arg1*, *arg2*, **args*[, *key*])

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, *iterable* must be a non-empty iterable (such as a non-empty string, tuple or list). The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

The optional *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *key* argument, if supplied, must be in keyword form (for example, `max(a,b,c,key=func)`).

Changed in version 2.5: Added support for the optional *key* argument.

memoryview(*obj*)

Return a "memory view" object created from the given argument. See *memoryview type* for more information.

min(*iterable*[, *key*])

min(*arg1*, *arg2*, **args*[, *key*])

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, *iterable* must be a non-empty iterable (such as a non-empty string, tuple or list). The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

The optional *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *key* argument, if supplied, must be in keyword form (for example, `min(a,b,c,key=func)`).

Changed in version 2.5: Added support for the optional *key* argument.

next(*iterator*[, *default*])

Retrieve the next item from the *iterator* by calling its `next()` method. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

New in version 2.6.

object()

Return a new featureless object. `object` is a base for all new style classes. It has the methods that are common to all instances of new style classes.

New in version 2.2.

Changed in version 2.3: This function does not accept any arguments. Formerly, it accepted arguments but ignored them.

oct(*x*)

Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

Changed in version 2.4: Formerly only returned an unsigned literal.

open(*name*[, *mode*[, *buffering*]])

Open a file, returning an object of the `file` type described in section *File Objects*. If the file cannot be opened, `IOError` is raised. When opening a file, it's preferable to use `open()` instead of invoking the `file` constructor directly.

The first two arguments are the same as for `stdio`'s `fopen()`: *name* is the file name to be opened, and *mode* is a string indicating how the file is to be opened.

The most commonly-used values of *mode* are `'r'` for reading, `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on *some* Unix systems means that *all* writes append to the end of the file regardless of the current seek position). If *mode* is omitted, it defaults to `'r'`. The default is to use text mode, which may convert `'\n'` characters to a platform-specific representation on writing and back on reading. Thus, when opening a binary file, you should append `'b'` to the *mode* value to open the file in binary mode, which will improve portability. (Appending `'b'` is useful even on systems that don't treat binary and text files differently, where it serves as documentation.) See below for more possible values of *mode*.

The optional *buffering* argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size (in bytes). A negative *buffering* means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used. [2]

Modes `'r+'`, `'w+'` and `'a+'` open the file for updating (note that `'w+'` truncates the file). Append `'b'` to the mode to open the file in binary mode, on systems that differentiate between binary and text files; on systems that don't have this distinction, adding the `'b'` has no effect.

In addition to the standard `fopen()` values *mode* may be `'U'` or `'rU'`. Python is usually built with *universal newlines* support; supplying `'U'` opens the file as a text file, but lines may be terminated by any of the following: the Unix end-of-line convention `'\n'`, the Macintosh convention `'\r'`, or the Windows convention `'\r\n'`. All of these external representations are seen as `'\n'` by the Python program. If Python is built without universal newlines support a *mode* with `'U'` is the same as normal text mode. Note that file objects so opened also have an attribute called `newlines` which has a value of `None` (if no newlines have yet been seen), `'\n'`, `'\r'`, `'\r\n'`, or a tuple containing all the newline types seen.

Python enforces that the mode, after stripping `'U'`, begins with `'r'`, `'w'` or `'a'`.

Python provides many file handling modules including `fileinput`, `os`, `os.path`, `tempfile`, and `shutil`.

Changed in version 2.5: Restriction on first letter of mode string introduced.

ord(*c*)

Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, `ord('a')` returns the integer `97`, `ord(u'\u2020')` returns `8224`. This is the inverse of **chr()** for 8-bit strings and of **unichr()** for unicode objects. If a unicode argument is given and Python was built with UCS2 Unicode, then the character's code point must be in the range [0..65535] inclusive; otherwise the string length is two, and a **TypeError** will be raised.

pow(*x*, *y*[, *z*])

Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than `pow(x, y) % z`). The two-argument form `pow(x, y)` is equivalent to using the power operator: `x**y`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised.) If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument `pow()` returned platform-dependent results depending on floating-point rounding accidents.)

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*)

Print *objects* to the stream *file*, separated by *sep* and followed by *end*. *sep*, *end* and *file*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like **str()** does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be **None**, which means to use the default values. If no *objects* are given, **print()** will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or **None**, **sys.stdout** will be used. Output buffering is determined by *file*. Use `file.flush()` to ensure, for instance, immediate appearance on a screen.

> **Note:** This function is not normally available as a built-in since the name `print` is recognized as the **print** statement. To disable the statement and use the **print()** function, use this future statement at the top of your module:

```
from __future__ import print_function
```

New in version 2.6.

property([*fget*[, *fset*[, *fdel*[, *doc*]]]])

Return a property attribute for *new-style classes* (classes that derive from **object**).

fget is a function for getting an attribute value, likewise *fset* is a function for setting, and *fdel* a function for del'ing, an attribute. Typical use is to define a managed attribute `x`:

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If then *c* is an instance of *C*, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter.

If given, *doc* will be the docstring of the property attribute. Otherwise, the property will copy *fget*'s docstring (if it exists). This makes it possible to create read-only properties easily using **property()** as a *decorator*:

```
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

turns the **voltage()** method into a "getter" for a read-only attribute with the same name.

A property object has **getter**, **setter**, and **deleter** methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
```

```
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

New in version 2.2.

Changed in version 2.5: Use *fget*'s docstring if no *doc* given.

Changed in version 2.6: The `getter`, `setter`, and `deleter` attributes were added.

range(*stop*)

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

This is a versatile function to create lists containing arithmetic progressions. It is most often used in `for` loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. The full form returns a list of plain integers `[start, start + step, start + 2 * step, ...]`. If *step* is positive, the last element is the largest `start + i * step` less than *stop*; if *step* is negative, the last element is the smallest `start + i * step` greater than *stop*. *step* must not be zero (or else `ValueError` is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([*prompt*])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `raw_input()` will use it to provide elaborate line editing and history features.

reduce(*function*, *iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initializer* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initializer* is not given and *iterable* contains only one item, the first item is returned. Roughly equivalent to:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

reload(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

If a module is syntactically correct but its initialization fails, the first `import` statement for it does not bind its name locally, but does store a (partially initialized) module object in `sys.modules`. To reload the module you must first `import` it again (this will bind the name to the partially initialized module object) before you can `reload()` it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__` and `__builtin__`. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` ... `import` ..., calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module*.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

repr(*object*)

Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

reversed(*seq*)

Return a reverse *iterator*. *seq* must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with integer arguments starting at `0`).

New in version 2.4.

Changed in version 2.6: Added the possibility to write a custom `__reversed__()` method.

round(*number*[, *ndigits*])

Return the floating point value *number* rounded to *ndigits* digits after the decimal point. If *ndigits* is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done away from 0 (so. for example, `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

> **Note:** The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See *Floating Point Arithmetic: Issues and Limitations* for more information.

set([*iterable*])

Return a new `set` object, optionally with elements taken from *iterable*. `set` is a built-in class. See `set` and *Set Types — set, frozenset* for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

New in version 2.4.

setattr(*object*, *name*, *value*)

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

slice(*stop*)

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

Return a *slice* object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an iterator.

sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]])

Return a new sorted list from the items in *iterable*.

The optional arguments *cmp*, *key*, and *reverse* have the same meaning as those for the **`list.sort()`** method (described in section *Mutable Sequence Types*).

cmp specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: `cmp=lambda x,y: cmp(x.lower(), y.lower())`. The default value is **`None`**.

key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is **`None`** (compare the elements directly).

reverse is a boolean value. If set to **`True`**, then the list elements are sorted as if each comparison were reversed.

In general, the *key* and *reverse* conversion processes are much faster than specifying an equivalent *cmp* function. This is because *cmp* is called multiple times for each list element while *key* and *reverse* touch each element only once. Use **`functools.cmp_to_key()`** to convert an old-style *cmp* function to a *key* function.

For sorting examples and a brief sorting tutorial, see Sorting HowTo.

New in version 2.4.

`staticmethod`(*function*)

Return a static method for *function*.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C(object):
    @staticmethod
    def f(arg1, arg2, ...):
        ...
```

The `@staticmethod` form is a function *decorator* – see the description of function definitions in *Function definitions* for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. Also see **`classmethod()`** for a variant that is useful for creating alternate class constructors.

For more information on static methods, consult the documentation on the standard type hierarchy in *The standard type hierarchy*.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

str(*object=''*)

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to **eval()**; its goal is to return a printable string. If no argument is given, returns the empty string, `''`.

For more information on strings see *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange* which describes sequence functionality (strings are sequences), and also the string-specific methods described in the *String Methods* section. To output formatted strings use template strings or the `%` operator described in the *String Formatting Operations* section. In addition see the *String Services* section. See also **unicode()**.

sum(*iterable*[, *start*])

Sums *start* and the items of an *iterable* from left to right and returns the total. *start* defaults to `0`. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to **sum()**. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see **math.fsum()**. To concatenate a series of iterables, consider using **itertools.chain()**.

New in version 2.3.

super(*type*[, *object-or-type*])

Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by **getattr()** except that the *type* itself is skipped.

The **__mro__** attribute of the *type* lists the method resolution search order used by both **getattr()** and **super()**. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

> **Note:** **super()** only works for *new-style class*es.

There are two typical use cases for *super*. In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution

environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references.

For practical suggestions on how to design cooperative classes using `super()`, see guide to using super().

New in version 2.2.

tuple([*iterable*])

Return a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, returns a new empty tuple, `()`.

`tuple` is an immutable sequence type, as documented in *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange*. For other containers see the built in `dict`, `list`, and `set` classes, and the `collections` module.

type(*object*)
type(*name*, *bases*, *dict*)

With one argument, return the type of an *object*. The return value is a type object. The `isinstance()` built-in function is recommended for testing the type of an object.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute;

the *bases* tuple itemizes the base classes and becomes the **__bases__** attribute; and the *dict* dictionary is the namespace containing definitions for class body and becomes the **__dict__** attribute. For example, the following two statements create identical **type** objects:

```
>>> class X(object):
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

New in version 2.2.

unichr(*i*)

Return the Unicode string of one character whose Unicode code is the integer *i*. For example, `unichr(97)` returns the string `u'a'`. This is the inverse of **ord()** for Unicode strings. The valid range for the argument depends how Python was configured – it may be either UCS2 [0..0xFFFF] or UCS4 [0..0x10FFFF]. **ValueError** is raised otherwise. For ASCII and 8-bit strings see **chr()**.

New in version 2.0.

unicode(*object=''*)

unicode(*object*[, *encoding*[, *errors*]])

Return the Unicode string version of *object* using one of the following modes:

If *encoding* and/or *errors* are given, `unicode()` will decode the object which can either be an 8-bit string or a character buffer using the codec for *encoding*. The *encoding* parameter is a string giving the name of an encoding; if the encoding is not known, **LookupError** is raised. Error handling is done according to *errors*; this specifies the treatment of characters which are invalid in the input encoding. If *errors* is `'strict'` (the default), a **ValueError** is raised on errors, while a value of `'ignore'` causes errors to be silently ignored, and a value of `'replace'` causes the official Unicode replacement character, `U+FFFD`, to be used to replace input characters which cannot be decoded. See also the **codecs** module.

If no optional parameters are given, `unicode()` will mimic the behaviour of `str()` except that it returns Unicode strings instead of 8-bit strings. More precisely, if *object* is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.

For objects which provide a **__unicode__()** method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in `'strict'` mode.

For more information on Unicode strings see *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange* which describes sequence functionality (Unicode strings are sequences), and also the string-specific methods described in the *String Methods* section. To output formatted strings use template strings or the `%` operator described in the *String*

Formatting Operations section. In addition see the *String Services* section. See also `str()`.

New in version 2.0.

Changed in version 2.2: Support for `__unicode__()` added.

vars([*object*])

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, new-style classes use a dictproxy to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

xrange(*stop*)

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

This function is very similar to `range()`, but returns an *xrange object* instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of `xrange()` over `range()` is minimal (since `xrange()` still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with `break`). For more information on xrange objects, see *XRange Type* and *Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange*.

> **CPython implementation detail:** `xrange()` is intended to be simple and fast. Implementations may impose restrictions to achieve this. The C implementation of Python restricts all arguments to native C longs ("short" Python integers), and also requires that the number of elements fit in a native C long. If a larger range is needed, an alternate version can be crafted using the `itertools` module: `islice(count(start, step), (stop-start+step-1+2*(step<0))//step)`.

zip([*iterable*, ...])

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length, `zip()` is similar to `map()` with an initial argument of `None`. With a single sequence argument, it returns a list of 1-tuples. With no arguments, it returns an empty list.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`.

`zip()` in conjunction with the `*` operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

New in version 2.0.

Changed in version 2.4: Formerly, `zip()` required at least one argument and `zip()` raised a `TypeError` instead of returning an empty list.

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

> **Note:** This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.

This function is invoked by the `import` statement. It can be replaced (by importing the `__builtin__` module and assigning to `__builtin__.__import__`) in order to change semantics of the `import` statement, but nowadays it is usually simpler to use import hooks (see **PEP 302**). Direct use of `__import__()` is rare, except in cases where you want to import a module whose name is only known at runtime.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. The default is `-1` which indicates both absolute and relative imports will be attempted. `0` means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling `__import__()`.

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], -1)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], -1)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

Changed in version 2.5: The level parameter was added.

Changed in version 2.5: Keyword support for parameters was added.

3. Non-essential Built-in Functions

There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and book writers should feel free to bypass these functions without concerns about missing something important.

apply(*function*, *args*[, *keywords*])

The *function* argument must be a callable object (a user-defined or built-in function or method, or a class object) and the *args* argument must be a sequence. The *function* is called with *args* as the argument list; the number of arguments is the length of the tuple. If the optional *keywords* argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the argument list. Calling `apply()` is different from just calling `function(args)`, since in that case there is always exactly one argument. The use of `apply()` is equivalent to `function(*args, **keywords)`.

> *Deprecated since version 2.3:* Use `function(*args, **keywords)` instead of `apply(function, args, keywords)` (see *Unpacking Argument Lists*).

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

The *object* argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the *object*

argument. The buffer object will be a slice from the beginning of *object* (or from the specified *offset*). The slice will extend to the end of *object* (or will have a length given by the *size* argument).

coerce(*x*, *y*)

Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations. If coercion is not possible, raise `TypeError`.

intern(*string*)

Enter *string* in the table of "interned" strings and return the interned string – which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Changed in version 2.3: Interned strings are not immortal (like they used to be in Python 2.2 and before); you must keep a reference to the return value of `intern()` around to benefit from it.

Footnotes

[1] It is used relatively rarely so does not warrant being made into a statement.

[2] Specifying a buffer size currently has no effect on systems that don't have `setvbuf()`. The interface to specify the buffer size is not done using a method that calls `setvbuf()`, because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.

[3] In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (such as modules) can be. This may change.