BUILT-IN FUNCTIONS

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

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

abs(*x*)

Return the absolute value of a number. The argument may be an 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:

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

any(*iterable*)

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

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

ascii(*object*)
>   As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `repr()` in Python 2.

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.

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` (see *Numeric Types — int, float, complex*). Class `bool` cannot be subclassed further. Its only instances are `False` and `True` (see *Boolean Values*).

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 `bytes` type has, see *Bytes and Byte Array 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.

bytes([*source*[, *encoding*[, *errors*]]])
>   Return a new "bytes" object, which is an immutable sequence of integers in the range `0 <= x < 256`. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.
>
>   Accordingly, constructor arguments are interpreted as for `bytearray()`.
>
>   Bytes objects can also be created with literals, see *strings*.

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); instances are callable if their class has a `__call__()` method. New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

chr(*i*)
>   Return the string representing a character whose Unicode codepoint is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of `ord()`. The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if *i* is outside that range.
>
>   Note that on narrow Unicode builds, the result is a string of length two for *i* greater than 65,535 (0xFFFF in hexadecimal).

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

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

compile (*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

Compile the *source* into a code or AST object. Code objects can be executed by `exec()` or `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.

The argument *optimize* specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

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 3.2: Allowed use of Windows and Mac newlines. Also input in `'exec'` mode does not have to end in a newline anymore. Added the *optimize* parameter.

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()` 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, 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([*arg*])

Create a new data dictionary, optionally with items taken from *arg*. The dictionary type is described in *Mapping Types — dict*.

For other containers see the built in `list`, `set`, and `tuple` classes, and 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:

```python
>>> 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 integer division. With mixed operand types, the rules for binary arithmetic operators apply. For 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)`.

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

Return an enumerate object. *iterable* 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 *iterable*.

```
>>> 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
```

eval(*expression*, *globals=None*, *locals=None*)

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.

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 `builtins` 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
>>> 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()` 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 `exec()`.

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

exec (*object*[, *globals*[, *locals*]])

This function supports dynamic execution of Python code. *object* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section "File input" in the Reference Manual). Be aware that the `return` and `yield` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary, which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object.

If the *globals* dictionary does not contain a value for the key __builtins__, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own __builtins__ dictionary into *globals* before passing it to `exec()`.

Note: The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

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 `exec()` returns.

---

filter (*function*, *iterable*)

Construct an iterator 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 *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 the generator expression `(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.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* returns false.

float ([*x*])

Convert a string or a number to floating point.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be '+' or '−'; a '+' sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
sign           ::=   "+" | "−"
infinity       ::=   "Infinity" | "inf"
nan            ::=   "nan"
numeric_value  ::=   floatnumber | infinity | nan
numeric_string ::=   [sign] numeric_value
```

---

[1] Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.

Here `floatnumber` is the form of a Python floating-point literal, described in *floating*. Case is not significant, so, for example, "inf", "Inf", "INFINITY" and "iNfINity" are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python's floating point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object x, `float(x)` delegates to `x.__float__()`.

If no argument is given, `0.0` is returned.

Examples:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

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

The default *format_spec* is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(format_spec)` which bypasses the instance dictionary when searching for the value's `__format__()` method. A `TypeError` exception is raised if the method is not found or if either the *format_spec* or the return value are not strings.

frozenset ([*iterable*])

Return a frozenset object, optionally with elements taken from *iterable*. The frozenset type is described in *Set Types — set, frozenset*.

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

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 `AttributeError` 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.

hex(*x*)
> Convert an integer number to a hexadecimal 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.

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

id(*object*)
> Return the "identity" of an object. This is an 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*])
> 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 = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

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

int([*number* | *string*[, *base*]])
> Convert a number or string to an integer. If no arguments are given, return `0`. If a number is given, return `number.__int__()`. Conversion of floating point numbers to integers truncates towards zero. A string must be a base-radix integer literal optionally preceded by '+' or '-' (with no space in between) and optionally 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`, or `0x/0X`, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.
>
> The integer type is described in *Numeric Types — int, float, 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. If *object* is not an object of the given type, the function always returns false. If *classinfo* is not a class (type object), it may be a tuple of type objects, or may recursively contain other such tuples (other sequence types are not accepted). If *classinfo* is not a type or tuple of types and such tuples, a `TypeError` exception is raised.

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.

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

> Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* 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 *object* must be a callable object. The iterator created in this case will call *object* 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:

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

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, bytes, bytearray, list, tuple, range*.

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.

---

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

> Return an iterator that applies *function* to every item of *iterable*, yielding 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. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

max(*iterable*[, *args...*], *[, *key*])

> With a single argument *iterable*, return the largest item of a non-empty iterable (such as a string, tuple or list). With more than one argument, return the largest of the arguments.

> The optional keyword-only *key* argument specifies a one-argument ordering function like that used for `list.sort()`.

> If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

memoryview(*obj*)

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

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

    With a single argument *iterable*, return the smallest item of a non-empty iterable (such as a string, tuple or list). With more than one argument, return the smallest of the arguments.

    The optional keyword-only *key* argument specifies a one-argument ordering function like that used for `list.sort()`.

    If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.

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.

object()

    Return a new featureless object. `object` is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.

    Note: `object` does *not* have a __dict__, so you can't assign arbitrary attributes to an instance of the `object` class.

---

oct(*x*)

    Convert an integer number to an octal 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.

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*)

    Open *file* and return a corresponding stream. If the file cannot be opened, an `IOError` is raised.

    file is either a string or bytes object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless *closefd* is set to `False`.)

    mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'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). In text mode, if *encoding* is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newline mode (for backwards compatibility; should not be used in new code)

    The default mode is `'r'` (open for reading text, synonym of `'rt'`). For binary read-write access, the mode `'w+b'` opens and truncates the file to 0 bytes. `'r+b'` opens the file without truncation.

    As mentioned in the *Overview*, Python distinguishes between binary and text I/O. Files opened in binary mode (including `'b'` in the *mode* argument) return contents as `bytes` objects without any decoding. In text mode (the default, or when `'t'` is included in the *mode* argument), the contents of the file are returned as `str`, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

Note: Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

---

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.

- "Interactive" text files (files for which `isatty()` returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any encoding supported by Python can be used. See the `codecs` module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled–this cannot be used in binary mode. Pass `'strict'` to raise a `ValueError` exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'backslashreplace'` (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

newline controls how universal newlines works (it only applies to text mode). It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- On input, if *newline* is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `''`, universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

- On output, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If *newline* is `''`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *closefd* is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* has no effect and must be `True` (the default).

The type of file object returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), it returns a subclass of `io.TextIOBase` (specifically `io.TextIOWrapper`). When used to open a file in a binary mode with buffering, the returned class is a subclass of `io.BufferedIOBase`. The exact class varies: in read binary mode, it returns a `io.BufferedReader`; in write binary and append binary modes, it returns a `io.BufferedWriter`, and in read/write mode, it returns a `io.BufferedRandom`. When buffering is disabled, the raw stream, a subclass of `io.RawIOBase`, `io.FileIO`, is returned.

See also the file handling modules, such as, `fileinput`, `io` (where `open()` is declared), `os`, `os.path`, `tempfile`, and `shutil`.

ord(*c*)

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('\u2020')` returns `8224`. This is the inverse of `chr()`.

On wide Unicode builds, if the argument length is not one, a `TypeError` will be raised. On narrow Unicode builds, strings of length two are accepted when they form a UTF-16 surrogate pair.

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` 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`. 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.

print ([*object*, ...], *, *sep=' '*, *end='\n'*, *file=sys.stdout*)

Print *object*(s) 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 *object* is 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.

property (*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Return a property attribute.

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`:

```python
class C:
    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*:

```python
class Parrot:
    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:

```python
class C:
    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.

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

This is a versatile function to create iterables yielding arithmetic progressions. It is most often used in `for` loops. The arguments must be 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 an iterable of 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:

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

Range objects implement the `collections.Sequence` ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see *Sequence Types — str, bytes, bytearray, list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Ranges containing absolute values larger than `sys.maxsize` are permitted but some features (such as `len()`) will raise `OverflowError`. Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test integers for membership in constant time instead of iterating through all items.

repr(*object*)

Return a string containing a printable representation of an object. 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`).

round(*x*[, *n*])

Return the floating point value *x* rounded to *n* digits after the decimal point. If *n* is omitted, it defaults to zero. Delegates to `x.__round__(n)`.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *n*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). The return value is an integer if called with one argument, otherwise of the same type as *x*.

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 *tut-fp-issues* for more information.

---

set([*iterable*])

Return a new set, optionally with elements taken from *iterable*. The set type is described in *Set Types — set, frozenset*.

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([*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[, key][, reverse]*)

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

Has two optional arguments which must be specified as keyword arguments.

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.

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.

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:

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

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

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

Return a string version of an object, using one of the following modes:

If *encoding* and/or *errors* are given, `str()` will decode the *object* which can either be a byte 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.

When only *object* is given, this returns its nicely printable representation. For strings, this is 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. With no arguments, this returns the empty string.

Objects can specify what `str(object)` returns by defining a `__str__()` special method.

For more information on strings see *Sequence Types — str, bytes, bytearray, list, tuple, range* which describes sequence functionality (strings are sequences), and also the string-specific methods described in the *String Methods* section. To output formatted strings, see the *String Formatting* section. In addition see the *String Services* section.

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().

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).

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:

```python
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # 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. The zero argument form automatically searches the stack frame for the class (__class__) and the first argument.

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

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, bytes, bytearray, list, tuple, range*.

type(*object*)
     Return the type of an *object*. The return value is a type object and generally the same object as returned by `object.__class__`.

     The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

     With three arguments, `type()` functions as a constructor as detailed below.

type(*name*, *bases*, *dict*)
     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:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

vars([*object*])
     Without an argument, act like `locals()`.

     With a module, class or class instance object as argument (or anything else that has a `__dict__` attribute), return that attribute.

     Note: The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined. [2]

---

zip(**iterables*)
     Make an iterator that aggregates elements from each of the iterables.

     Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

     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()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

---

[2] 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.

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

__import__(*name*, *globals={}*, *locals={}*, *fromlist=[ ]*, *level=0*)

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 `builtins` module and assigning to `builtins.__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. `0` (the default) 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(), [], 0)
```

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

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

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'], 0)
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()`.