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()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	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()	

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. If *x* defines `__abs__()`, `abs(x)` returns `x.__abs__()`.

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
```

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
```

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 prefixed with “0b”. 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. Some examples:

```
>>>
```

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If prefix “0b” is desired or not, you can use either of the following ways.

```
>>>
```

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also `format()` for more information.

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

Return a new array of bytes. The `bytearray` class 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 Bytearray Operations.

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.

See also Binary Sequence Types — bytes, bytearray, memoryview and Bytearray Objects.

class **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()`.

chr(*i*)

Return the string representing a character whose Unicode code point is the integer *i*. For example, `chr(97)` returns the string `'a'`, while `chr(8364)` returns the string `'€'`. 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.

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*[, *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.

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 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 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 with the *globals* and *locals* in the environment where `eval()` is called. Note, *eval()* does not have access to the nested scopes (non-locals) in the enclosing environment.

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.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

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 (and not a subclass of 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. Remember that at module level, globals and locals are the same dictionary. If exec gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

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()`.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

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.

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).

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 that at the module level, `locals()` and `globals()` are the same dictionary.

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.

class **str**(*object=''*)
class **str**(*object=b''*, *encoding='utf-8'*, *errors='strict'*)
Return a `str` version of *object*.

`str` is the built-in string class.

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, classes use a `types.MappingProxyType` 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.

A `TypeError` exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute).