2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are alwa 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 floatir 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). Equivale 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 Fals 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 esca
the non-ASCII characters in the string returned by `repr()` using \x, \u or \U escape
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. I
is not a Python `int` object, it has to define an `__index__()` method that returns a
integer.

class **bool**([*x*])

Return a Boolean value, i.e. one of `True` or `False`. *x* is converted using the standa
truth testing procedure. If *x* is false or omitted, this returns `False`; otherwise it retur
`True`. The `bool` class is a subclass of `int` (see Numeric Types — int, float, complex).
cannot be subclassed further. Its only instances are `False` and `True` (see Boolea
Values).

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

Return a new array of bytes. The `bytearray` class is a mutable sequence of integers
the range 0 <= x < 256. It has most of the usual methods of mutable sequence
described in Mutable Sequence Types, as well as most methods that the `bytes` type ha
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*) parameter
  `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 obje
  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 < 25
  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 Bytearr
Objects.

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

Return a new "bytes" object, which is an immutable sequence of integers in the range
<= x < 256. `bytes` is an immutable version of `bytearray` – it has the same no

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 String and Bytes literals.

See also Binary Sequence Types — bytes, bytearray, memoryview, Bytes, and Bytes an Bytearray Operations.

`callable`(*object*)

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

New in version 3.2: This function was first removed in Python 3.0 and then brought ba in Python 3.2.

`chr`(*i*)

Return the string representing a character whose Unicode code point is the integer *i*. F example, `chr(97)` returns the string `'a'`, while `chr(957)` returns the string `'ν'`. This the inverse of `ord()`.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 1(`ValueError` will be raised if *i* is outside that range.

`classmethod`(*function*)

Return a class method for *function*.

A class method receives the class as implicit first argument, just like an instance meth 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 functi 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 clas 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, s `staticmethod()` in this section.

For more information on class methods, consult the documentation on the standard ty

hierarchy in The standard type hierarchy.

`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 normal string, a byte string, or an AST object. Refer 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 son 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'` *source* consists of a sequence of statements, `'eval'` if it consists of a single expressic or `'single'` if it consists of a single interactive statement (in the latter case, expressic statements that evaluate to something other than `None` will be printed).

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

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

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

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

If you want to parse Python code into its AST representation, see `ast.parse()`.

> **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 `'exec'` mode does not have to end in a newline anymore. Added the *optimi.* parameter.

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

Return a complex number with the value *real* + *imag**1j or convert a string or number to complex number. If the first parameter is a string, it will be interpreted as a comple number and the function must be called without a second parameter. The secor parameter can never be a string. Each argument may be any numeric type (includir complex). If *imag* is omitted, it defaults to zero and the constructor serves as a numer conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

> **Note** When converting from a string, the string must not contain whitespace aroun the central + or – operator. For example, `complex('1+2j')` is fine, but `complex('` `+ 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 strir must be the name of one of the object's attributes. The function deletes the name attribute, provided the object allows it. For example, `delattr(x, 'foobar')` equivalent to `del x.foobar`.

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

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

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

dir([*object*])

Without arguments, return the list of names in the current local scope. With an argumer 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 retu the list of attributes. This allows objects that implement a custom `__getattr__()` `__getattribute__()` function to customize the way `dir()` reports their attributes.

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

The default `dir()` mechanism behaves differently with different types of objects, as 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 attributes.

- If the object is a type or class object, the list contains the names of its attribute and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class 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__', '__name__', 'struct']
>>> dir(struct)    # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__f
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
```

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 chang 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 consisti of their quotient and remainder when using integer division. With mixed operand type the rules for binary arithmetic operators apply. For integers, the result is the same (a // b, a % b). For floating point numbers the result is (q, a % b), where *q* usually `math.floor(a / b)` but may be 1 less than that. In any case q * b + a % 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 oth object which supports iteration. The `__next__()` method of the iterator returned `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and t 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:

```python
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 dictionary. If provided, *locals* can be any mapping object.

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

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

This function can also be used to execute arbitrary code objects (such as those creat by `compile()`). In this case pass a code object instead of a string. If the code object h 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. Tl `globals()` and `locals()` functions returns the current global and local dictiona 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 wi expressions containing only literals.

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

This function supports dynamic execution of Python code. *object* must be either a stri or a code object. If it is a string, the string is parsed as a suite of Python statements whi is then executed (unless a syntax error occurs). [1] If it is a code object, it is simp executed. In all cases, the code that's executed is expected to be valid as file input (s) the section "File input" in the Reference Manual). Be aware that the `return` and `yie` statements may not be used outside of function definitions even within the context 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.

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. 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()`.

> **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.

class **float**([*x*])

Return a floating point number constructed from a number or string *x*.

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
```

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

Otherwise, if the argument is an integer or a floating point number, a floating poi
number with the same value (within Python's floating point precision) is returned. If th
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*. Th
interpretation of *format_spec* will depend on the type of the *value* argument, howev
there is a standard formatting syntax that is used by most built-in types: Form
Specification Mini-Language.

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

A call to `format(value, format_spec)` is translated to `type(value).__format_
(value, format_spec)` which bypasses the instance dictionary when searching for th
value's `__format__()` method. A `TypeError` exception is raised if the method sear
reaches `object` and the *format_spec* is non-empty, or if either the *format_spec* or th
return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises `TypeError`
format_spec is not an empty string.

class **frozenset**([*iterable*])

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

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

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

Return the value of the named attribute of *object*. *name* must be a string. If the string
the name of one of the object's attributes, the result is the value of that attribute. F
example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribu
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 tl
dictionary of the current module (inside a function or method, this is the module where
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
one of the object's attributes, `False` if not. (This is implemented by calling `getatt`
`(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).

Note For object's with custom `__hash__()` methods, note that `hash()` truncates th
return value based on the bit width of the host machine. See `__hash__()` for details.

help([*object*])

Invoke the built-in help system. (This function is intended for interactive use.) If r
argument is given, the interactive help system starts on the interpreter console. If tl
argument is a string, then the string is looked up as the name of a module, functic
class, method, keyword, or documentation topic, and a help page is printed on tl
console. If the argument is any other kind of object, a help page on the object
generated.

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

Changed in version 3.4: Changes to `pydoc` and `inspect` mean that the reporte signatures for callables are now more comprehensive and consistent.

hex(*x*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x", f example:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If x is not a Python `int` object, it has to define an __index__() method that returns a integer.

See also `int()` for converting a hexadecimal string to an integer using a base of 16.

> **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 ar constant for this object during its lifetime. Two objects with non-overlapping lifetimes ma 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 traili newline. The function then reads a line from input, converts it to a string (stripping 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 li editing and history features.

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

Return an integer object constructed from a number or string *x*, or return 0 if r arguments are given. If *x* is a number, return `x.__int__()`. For floating point numbe

this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray`
instance representing an integer literal in radix *base*. Optionally, the literal can
preceded by + or – (with no space in between) and surrounded by whitespace. A base
literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. Th
default *base* is 10. The allowed values are 0 and 2-36. Base-2, -8, and -16 literals can
optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. Base
means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, a
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.

Changed in version 3.4: If *base* is not an instance of `int` and the *base* object has
`base.__index__` method, that method is called to obtain an integer for the bas
Previous versions used `base.__int__` instead of `base.__index__`.

isinstance(*object*, *classinfo*)

Return true if the *object* argument is an instance of the *classinfo* argument, or of a (dire
indirect or virtual) subclass thereof. If *object* is not an object of the given type, the functi
always returns false. If *classinfo* is not a class (type object), it may be a tuple of ty
objects, or may recursively contain other such tuples (other sequence types are n
accepted). If *classinfo* is not a type or tuple of types and such tuples, a `TypeErr`
exception is raised.

issubclass(*class*, *classinfo*)

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

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

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

See also Iterator Types.

One useful application of the second form of `iter()` is to read lines of a file until
certain line is reached. The following example reads a file until the `readline()` meth

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 (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, frozen set).

class **list**([*iterable*])

Rather than being a function, `list` is actually a mutable sequence type, as documented in Lists and Sequence Types — 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. additional *iterable* arguments are passed, *function* must take that many arguments and applied to the items from all iterables in parallel. With multiple iterables, the iterator stop when the shortest iterable is exhausted. For cases where the function inputs are alrea arranged into argument tuples, see `itertools.starmap()`.

max(*iterable*, *[, *key*, *default*])
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, it should be an iterable. The largest item in th iterable is returned. If two or more positional arguments are provided, the largest of th positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a on argument ordering function like that used for `list.sort()`. The *default* argume specifies an object to return if the provided iterable is empty. If the iterable is empty an *default* is not provided, a `ValueError` is raised.

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

```
key=keyfunc).
```

New in version 3.4: The *default* keyword-only argument.

memoryview(*obj*)

Return a "memory view" object created from the given argument. See Memory Views f
more information.

min(*iterable*, *[, key, default]*)
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, it should be an iterable. The smallest item in tl
iterable is returned. If two or more positional arguments are provided, the smallest of tl
positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a on
argument ordering function like that used for `list.sort()`. The *default* argume
specifies an object to return if the provided iterable is empty. If the iterable is empty al
default is not provided, a `ValueError` is raised.

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

New in version 3.4: The *default* keyword-only argument.

next(*iterator[, default]*)

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

class **object**

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

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

oct(*x*)

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

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=Nor*
closefd=True, *opener=None*)

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

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

mode is an optional string that specifies the mode in which the file is opened. It defau
to `'r'` which means open for reading in text mode. Other common values are `'w'` f
writing (truncating the file if it already exists), `'x'` for exclusive creation and `'a'` f
appending (which on *some* Unix systems, means that *all* writes append to the end of tl
file regardless of the current seek position). In text mode, if *encoding* is not specified tl
encoding used is platform dependent: `locale.getpreferredencoding(False)`
called to get the current locale encoding. (For reading and writing raw bytes use bina
mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'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 newlines mode (deprecated)

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

As mentioned in the Overview, Python distinguishes between binary and text I/O. Fil
opened in binary mode (including `'b'` in the *mode* argument) return contents as `byte`
objects without any decoding. In text mode (the default, or when `'t'` is included in tl
mode argument), the contents of the file are returned as `str`, the bytes having been fi
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 bufferi
off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), al

an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no *bufferi* 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 usi a heuristic trying to determine the underlying device's "block size" and falling ba on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 409 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns `True`) use line bufferin 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 or be used in text mode. The default encoding is platform dependent (whatev `locale.getpreferredencoding()` returns), but any text encoding supported 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 handled–this cannot be used in binary mode. A variety of standard error handlers a available (listed under Error Handlers), though any error handling name that has be registered with `codecs.register_error()` is also valid. The standard names include

- `'strict'` to raise a `ValueError` exception if there is an encoding error. T default value of `None` has the same effect.
- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss
- `'replace'` causes a replacement marker (such as `'?'`) to be inserted where the is malformed data.
- `'surrogateescape'` will represent any incorrect bytes as code points in t Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private co points will then be turned back into the same bytes when the `surrogateesca` error handler is used when writing data. This is useful for processing files in unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters n supported by the encoding are replaced with the appropriate XML charact reference `&#nnn;`.
- `'backslashreplace'` replaces malformed data by Python's backslashed esca sequences.
- `'namereplace'` (also only supported when writing) replaces unsupport characters with `\N{...}` escape sequences.

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

- When reading input from the stream, if *newline* is `None`, universal newlines mode enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these a translated into `'\n'` before being returned to the caller. If it is `''`, univers newlines mode is enabled, but line endings are returned to the caller untranslate If it has any of the other legal values, input lines are only terminated by the giv string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if *newline* is `None`, any `'\n'` characters writt are translated to the system default line separator, `os.linesep`. If *newline* is `''`

'\n', no translation takes place. If *newline* is any of the other legal values, a
'\n' characters written are translated to the given string.

If *closefd* is `False` and a file descriptor rather than a filename was given, the underlyir
file descriptor will be kept open when the file is closed. If a filename is given *closefd* mu
be `True` (the default) otherwise an error will be raised.

A custom opener can be used by passing a callable as *opener*. The underlying f
descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *open*
must return an open file descriptor (passing `os.open` as *opener* results in functional
similar to passing `None`).

The newly created file is non-inheritable.

The following example uses the dir_fd parameter of the `os.open()` function to open
file relative to a given directory:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=
...
```

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

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

Changed in version 3.3: The *opener* parameter was added. The 'x' mode was adde
`IOError` used to be raised, it is now an alias of `OSError`. `FileExistsError` is nc
raised if the file opened in exclusive creation mode ('x') already exists.

Changed in version 3.4: The file is now non-inheritable.

Deprecated since version 3.4, will be removed in version 4.0: The 'U' mode.

Changed in version 3.5: If the system call is interrupted and the signal handler does n
raise an exception, the function now retries the system call instead of raising a
`InterruptedError` exception (see **PEP 475** for the rationale).

ord(*c*)

Given a string representing one Unicode character, return an integer representing th
Unicode code point of that character. For example, `ord('a')` returns the integer `97` ar
`ord('ν')` returns `957`. This is the inverse of `chr()`.

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

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

The arguments must have numeric types. With mixed operand types, the coercion rul
for binary arithmetic operators apply. For `int` operands, the result has the same type a
the operands (after coercion) unless the second argument is negative; in that case,
arguments are converted to float and a float result is delivered. For example, $10*$
returns `100`, but `10**-2` returns `0.01`. If the second argument is negative, the thi
argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* mu
be non-negative.

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

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

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

The *file* argument must be an object with a `write(string)` method; if it is not present
`None`, `sys.stdout` will be used. Since printed arguments are converted to text string
`print()` cannot be used with binary mode file objects. For these, use `file.wri
(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argume
is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

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

Return a property attribute.

fget is a function for getting an attribute value. *fset* is a function for setting an attribu
value. *fdel* is a function for deleting an attribute value. And *doc* creates a docstring for th

attribute.

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

If given, *doc* will be the docstring of the property attribute. Otherwise, the property w
copy *fget*'s docstring (if it exists). This makes it possible to create read-only properti
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
```

The `@property` decorator turns the `voltage()` method into a "getter" for a read-on
attribute with the same name, and it sets the docstring for *voltage* to "Get the curre
voltage."

A property object has `getter`, `setter`, and `deleter` methods usable as decorators th
create a copy of the property with the corresponding accessor function set to th
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 addition functions the same name as the original property (x in this case.)

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

Changed in version 3.5: The docstrings of property objects are now writeable.

range(*stop*)
range(*start*, *stop*[, *step*])
    Rather than being a function, `range` is actually an immutable sequence type, documented in Ranges and Sequence Types — list, tuple, range.

repr(*object*)
    Return a string containing a printable representation of an object. For many types, th function makes an attempt to return a string that would yield an object with the san value when passed to `eval()`, otherwise the representation is a string enclosed in anc brackets that contains the name of the type of the object together with addition information often including the name and address of the object. A class can control wh 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__()` meth or supports the sequence protocol (the `__len__()` method and the `__getitem__` method with integer arguments starting at 0).

round(*number*[, *ndigits*])
    Return the floating point value *number* rounded to *ndigits* digits after the decimal point. *ndigits* is omitted, it returns the nearest integer to its input. Delegates `number.__round__(ndigits)`.

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

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

class **set**([*iterable*])

Return a new `set` object, optionally with elements taken from *iterable*. `set` is a built 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 w as the `collections` module.

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

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

class **slice**(*stop*)
class **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 da attributes `start`, `stop` and `step` which merely return the argument values (or the default). They have no other explicit functionality; however they are used by Numeric Python and other third party extensions. Slice objects are also generated when extende 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 fro each list element: `key=str.lower`. The default value is `None` (compare the elemer directly).

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

Use `functools.cmp_to_key()` to convert an old-style *cmp* function to a *key* function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantee not to change the relative order of elements that compare equal — this is helpful f sorting in multiple passes (for example, sort by department, then by salary grade).

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 metho use this idiom:

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

The `@staticmethod` form is a function decorator – see the description of functi 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 s `classmethod()` for a variant that is useful for creating alternate class constructors.

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

class **str**(*object=''*)
class **str**(*object=b'', encoding='utf-8', errors='strict'*)

Return a `str` version of *object*. See `str()` for details.

`str` is the built-in string class. For general information about strings, see Text Sequen Type — str.

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

Sums *start* and the items of an *iterable* from left to right and returns the total. *st* defaults to `0`. The *iterable*'s items are normally numbers, and the start value is n allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way concatenate a sequence of strings is by calling `''.join(sequence)`. To add floati point values with extended precision, see `math.fsum()`. To concatenate a series 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*. Th
> is useful for accessing inherited methods that have been overridden in a class. Tl
> 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 bc
> `getattr()` and `super()`. The attribute is dynamic and can change whenever tl
> inheritance hierarchy is updated.
>
> If the second argument is omitted, the super object returned is unbound. If the seco
> argument is an object, `isinstance(obj, type)` must be true. If the second argume
> 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 inheritanc
> *super* can be used to refer to parent classes without naming them explicitly, thus makii
> the code more maintainable. This use closely parallels the use of *super* in oth
> programming languages.
>
> The second use case is to support cooperative multiple inheritance in a dynam
> execution environment. This use case is unique to Python and is not found in statica
> compiled languages or languages that only support single inheritance. This makes
> possible to implement "diamond diagrams" where multiple base classes implement tl
> same method. Good design dictates that this method have the same calling signature
> every case (because the order of calls is determined at runtime, because that ord
> adapts to changes in the class hierarchy, and because that order can include siblii
> 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 dott
> attribute lookups such as `super().__getitem__(name)`. It does so by implementing
> own `__getattribute__()` method for searching classes in a predictable order th
> supports cooperative multiple inheritance. Accordingly, `super()` is undefined for impli
> lookups using statements or operators such as `super()[name]`.
>
> Also note that, aside from the zero argument form, `super()` is not limited to use insi
> methods. The two argument form specifies the arguments exactly and makes tl
> appropriate references. The zero argument form only works inside a class definition, a
> the compiler fills in the necessary details to correctly retrieve the class being defined, a
> well as accessing the current instance for ordinary methods.

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

tuple([*iterable*])

Rather than being a function, `tuple` is actually an immutable sequence type, a
documented in Tuples and Sequence Types — list, tuple, range.

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

With one argument, return the type of an *object*. The return value is a type object a
generally the same object as returned by `object.__class__`.

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

With three arguments, return a new type object. This is essentially a dynamic form of tl
`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 bo
and becomes the `__dict__` attribute. For example, the following two statements crea
identical `type` objects:

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

See also Type Objects.

vars([*object*])

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

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

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

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
the argument sequences or iterables. The iterator stops when the shortest input iterat
is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With r

arguments, it returns an empty iterator. Equivalent to:

```python
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)`. Th repeats the *same* iterator n times so that each output tuple has the result of n calls to t iterator. This has the effect of dividing the input into n-length chunks.

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

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

```python
>>> 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=None*, *locals=None*, *fromlist=()*, *level=0*)

> **Note** This is an advanced function that is not needed in everyday Pytho
> 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 chang semantics of the `import` statement, but doing so is **strongly** discouraged as it is usua simpler to use import hooks (see **PEP 302**) to attain the same goals and does not cau issues with code which assumes the default import implementation is in use. Direct use `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module *name*, potentially using the given *globals* and *locals* determine how to interpret the name in a package context. The *fromlist* gives the nam

of objects or submodules that should be imported from the module given by *name*. Th
standard implementation does not use its *locals* argument at all, and uses its *globals* or
to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. `0` (the default) means or
perform absolute imports. Positive values for *level* indicate the number of pare
directories to search relative to the directory of the module calling `__import__()` (se
PEP 328 for the details).

When the *name* variable is of the form `package.module`, normally, the top-level packag
(the name up till the first dot) is returned, *not* the module named by *name*. Howeve
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 followin
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 th
is bound to a name by the `import` statement.

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

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausa
eggs = _temp.eggs
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, th
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, us
`importlib.import_module()`.

Changed in version 3.3: Negative values for *level* are no longer supported (which al
changes the default value to 0).

Footnotes

[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.