5. Built-in Types

The following sections describe the standard types that are built into the interpreter.

> **Note:** Historically (until release 2.2), Python's built-in types have differed from user-defined types because it was not possible to use the built-in types as the basis for object-oriented inheritance. This limitation no longer exists.

The principal built-in types are numerics, sequences, mappings, files, classes, instances and exceptions.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the repr() function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

5.1. Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below. The following values are considered false:

- `None`
- `False`
- zero of any numeric type, for example, `0`, `0L`, `0.0`, `0j`.
- any empty sequence, for example, `''`, `()`, `[]`.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or `bool` value `False`. [1]

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)

5.2. Boolean Operations — `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
`x or y`	if *x* is false, then *y*, else *x*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is `False`.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is `True`.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

5.3. Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y and y <= z`, except that *y* is evaluated only once (but in both cases *z* is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

Operation	Meaning	Notes
`<`	strictly less than	
`<=`	less than or equal	
`>`	strictly greater than	
`>=`	greater than or equal	
`==`	equal	
`!=`	not equal	(1)
`is`	object identity	
`is not`	negated object identity	

Notes:

1. `!=` can also be written `<>`, but this is an obsolete usage kept for backwards compatibility only. New code should always use `!=`.

Objects of different types, except different numeric types and different string types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (for example, file objects) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently. The `<`, `<=`, `>` and `>=` operators will raise a `TypeError` exception when any operand is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method or the `__cmp__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines either enough of the rich comparison methods (`__lt__()`, `__le__()`, `__gt__()`, and `__ge__()`) or the `__cmp__()` method.

> **CPython implementation detail:** Objects of different types except numbers are ordered by their type names; objects of the same types that don't support proper comparison are ordered by their address.

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).

5.4. Numeric Types — `int`, `float`, `long`, `complex`

There are four distinct numeric types: *plain integers*, *long integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of plain integers. Plain integers (also just called *integers*) are implemented using `long` in C, which gives them at least 32 bits of precision (`sys.maxint` is always set to the maximum plain integer value for the current platform, the minimum value is `-sys.maxint - 1`). Long integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number *z*, use `z.real` and `z.imag`. (The standard library includes additional numeric types, `fractions` that hold rationals, and `decimal` that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including binary, hex, and octal numbers) yield plain integers unless the value they denote is too large to be represented as a plain integer, in which case they yield a long integer. Integer literals with an `'L'` or `'l'` suffix yield long integers (`'L'` is preferred because `1l` looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields a complex number with a zero real part. A complex numeric literal is the sum of a real and an imaginary part.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where plain integer is narrower than long integer is narrower than floating point is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `long()`, `float()`, and `complex()` can be used to produce numbers of a

specific type.

All built-in numeric types support the following operations. See The power operator and later sections for the operators' priorities.

Operation	Result	Notes
`x + y`	sum of *x* and *y*	
`x - y`	difference of *x* and *y*	
`x * y`	product of *x* and *y*	
`x / y`	quotient of *x* and *y*	(1)
`x // y`	(floored) quotient of *x* and *y*	(4)(5)
`x % y`	remainder of `x / y`	(4)
`-x`	*x* negated	
`+x`	*x* unchanged	
`abs(x)`	absolute value or magnitude of *x*	(3)
`int(x)`	*x* converted to integer	(2)
`long(x)`	*x* converted to long integer	(2)
`float(x)`	*x* converted to floating point	(6)
`complex(re,im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.	
`c.conjugate()`	conjugate of the complex number *c*. (Identity on real numbers)	
`divmod(x, y)`	the pair `(x // y, x % y)`	(3)(4)
`pow(x, y)`	*x* to the power *y*	(3)(7)
`x ** y`	*x* to the power *y*	(7)

Notes:

1. For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: 1/2 is 0, (-1)/2 is -1, 1/(-2) is -1, and (-1)/(-2) is 0. Note that the result is a long integer if either operand is a long integer, regardless of the numeric value.

2. Conversion from floats using **`int()`** or **`long()`** truncates toward zero like the related function, **`math.trunc()`**. Use the function **`math.floor()`** to round downward and **`math.ceil()`** to round upward.

3. See Built-in Functions for a full description.

4. *Deprecated since version 2.3:* The floor division operator, the modulo operator, and the **`divmod()`** function are no longer defined for complex numbers. Instead, convert to a floating point number using the **`abs()`** function if appropriate.

5. Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily int.

6. float also accepts the strings "nan" and "inf" with an optional prefix "+" or "-" for Not a Number (NaN) and positive or negative infinity.

   New in version 2.6.

7. Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages.

All `numbers.Real` types (`int`, `long`, and `float`) also include the following operations:

Operation	Result	Notes
`math.trunc(x)`	*x* truncated to Integral	
`round(x[, n])`	*x* rounded to n digits, rounding ties away from zero. If n is omitted, it defaults to 0.	
`math.floor(x)`	the greatest integral float <= *x*	
`math.ceil(x)`	the least integral float >= *x*	

5.4.1. Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. Negative numbers are treated as their 2's complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
`x | y`	bitwise *or* of *x* and *y*	
`x ^ y`	bitwise *exclusive or* of *x* and *y*	
`x & y`	bitwise *and* of *x* and *y*	
`x << n`	*x* shifted left by *n* bits	(1)(2)
`x >> n`	*x* shifted right by *n* bits	(1)(3)
`~x`	the bits of *x* inverted	

Notes:

1. Negative shift counts are illegal and cause a `ValueError` to be raised.
2. A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`. A long integer is returned if the result exceeds the range of plain integers.

3. A right shift by *n* bits is equivalent to division by `pow(2, n)`.

5.4.2. Additional Methods on Integer Types

The integer types implement the **`numbers.Integral`** abstract base class. In addition, they provide one more method:

`int.`**`bit_length`**()

`long.`**`bit_length`**()

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if `x` is nonzero, then `x.bit_length()` is the unique positive integer `k` such that `2**(k-1) <= abs(x) < 2**k`. Equivalently, when `abs(x)` is small enough to have a correctly rounded logarithm, then `k = 1 + int(log(abs(x), 2))`. If `x` is zero, then `x.bit_length()` returns `0`.

Equivalent to:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

New in version 2.7.

5.4.3. Additional Methods on Float

The float type implements the **`numbers.Real`** abstract base class. float also has the following additional methods.

`float.`**`as_integer_ratio`**()

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises **`OverflowError`** on infinities and a **`ValueError`** on NaNs.

New in version 2.6.

`float.`**`is_integer`**()

Return `True` if the float instance is finite with integral value, and `False` otherwise:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

New in version 2.6.

Two methods support conversion to and from hexadecimal strings. Since Python's floats are stored internally as binary numbers, converting a float to or from a *decimal* string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

`float.`**`hex`**()

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading `0x` and a trailing `p` and exponent.

New in version 2.6.

`float.`**`fromhex`**(*s*)

Class method to return the float represented by a hexadecimal string *s*. The string *s* may have leading and trailing whitespace.

New in version 2.6.

Note that **`float.hex()`** is an instance method, while **`float.fromhex()`** is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

where the optional `sign` may by either `+` or `-`, `integer` and `fraction` are strings of hexadecimal digits, and `exponent` is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of **`float.hex()`** is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's `%a` format character or Java's `Double.toHexString` are accepted by **`float.fromhex()`**.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number `(3 + 10./16 + 7./16**2) * 2.0**10`, or `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to `3740.0` gives a different hexadecimal string representing the same number:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

5.5. Iterator Types

New in version 2.2.

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

container.**__iter__**()
: Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the *iterator protocol*:

iterator.**__iter__**()
: Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

iterator.**next**()
: Return the next item from the container. If there are no further items, raise the `StopIteration` exception. This method corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

The intention of the protocol is that once an iterator's `next()` method raises `StopIteration`, it will continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken. (This constraint was added in Python 2.3; in Python 2.2, various iterators are broken according to this rule.)

5.5.1. Generator Types

Python's generators provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `next()` methods. More information about generators can be found in the documentation for the yield expression.

5.6. Sequence Types — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange`

There are seven sequence types: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

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

String literals are written in single or double quotes: `'xyzzy'`, `"frobozz"`. See String literals for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceding `'u'` character: `u'abc'`, `u"def"`. In addition to the functionality described here, there are also string-specific methods described in the String Methods section. Lists are constructed with square brackets, separating items with commas: `[a, b, c]`. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as `a, b, c` or `()`. A single item tuple must have a trailing comma, such as `(d,)`.

Bytearray objects are created with the built-in function `bytearray()`.

Buffer objects are not directly supported by Python syntax, but can be created by calling the built-in function `buffer()`. They don't support concatenation or repetition.

Objects of type xrange are similar to buffers in that there is no specific syntax to create them, but they are created using the `xrange()` function. They don't support slicing, concatenation or repetition, and using `in`, `not in`, `min()` or `max()` on them is inefficient.

Most sequence types support the following operations. The `in` and `not in` operations have the same priorities as the comparison operations. The `+` and `*` operations have the same priority as the corresponding numeric operations. [3] Additional methods are provided for Mutable Sequence Types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type; *n*, *i* and *j* are integers:

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`		

	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)
`s * n, n * s`	equivalent to adding *s* to itself *n* times	(2)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x)`	index of the first occurrence of *x* in *s*	
`s.count(x)`	total number of occurrences of *x* in *s*	

Sequence types also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see Comparisons in the language reference.)

Notes:

1. When *s* is a string or Unicode string object the `in` and `not in` operations act like a substring test. In Python versions before 2.3, *x* had to be a string of length 1. In Python 2.3 and beyond, *x* may be a string of any length.

2. Values of *n* less than `0` are treated as `0` (which yields an empty sequence of the same type as *s*). Note that items in the sequence *s* are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that `[[]]` is a one-element list containing an empty list, so all three elements of `[[]] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Further explanation is available in the FAQ entry How do I create a multidimensional

list?.

3. If *i* or *j* is negative, the index is relative to the end of the string: `len(s) + i` or `len(s) + j` is substituted. But note that `-0` is still `0`.

4. The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that `i <= k < j`. If *i* or *j* is greater than `len(s)`, use `len(s)`. If *i* is omitted or `None`, use `0`. If *j* is omitted or `None`, use `len(s)`. If *i* is greater than or equal to *j*, the slice is empty.

5. The slice of *s* from *i* to *j* with step *k* is defined as the sequence of items with index `x = i + n*k` such that `0 <= n < (j-i)/k`. In other words, the indices are `i`, `i+k`, `i+2*k`, `i+3*k` and so on, stopping when *j* is reached (but never including *j*). If *i* or *j* is greater than `len(s)`, use `len(s)`. If *i* or *j* are omitted or `None`, they become "end" values (which end depends on the sign of *k*). Note, *k* cannot be zero. If *k* is `None`, it is treated like `1`.

6. **CPython implementation detail:** If *s* and *t* are both strings, some Python implementations such as CPython can usually perform an in-place optimization for assignments of the form `s = s + t` or `s += t`. When applicable, this optimization makes quadratic run-time much less likely. This optimization is both version and implementation dependent. For performance sensitive code, it is preferable to use the **`str.join()`** method which assures consistent linear concatenation performance across versions and implementations.

   Changed in version 2.4: Formerly, string concatenation never occurred in-place.

5.6.1. String Methods

Below are listed the string methods which both 8-bit strings and Unicode objects support. Some of them are also available on **`bytearray`** objects.

In addition, Python's strings support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section. To output formatted strings use template strings or the `%` operator described in the String Formatting Operations section. Also, see the **`re`** module for string functions based on regular expressions.

`str.`**`capitalize`**()
Return a copy of the string with its first character capitalized and the rest lowercased.

For 8-bit strings, this method is locale-dependent.

`str.`**`center`**(*width*[, *fillchar*])
Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is a space).

Changed in version 2.4: Support for the *fillchar* argument.

`str.`**`count`**(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.`**`decode`**([*encoding*[, *errors*]])

Decodes the string using the codec registered for *encoding*. *encoding* defaults to the default string encoding. *errors* may be given to set a different error handling scheme. The default is `'strict'`, meaning that encoding errors raise **`UnicodeError`**. Other possible values are `'ignore'`, `'replace'` and any other name registered via **`codecs.register_error()`**, see section Codec Base Classes.

New in version 2.2.

Changed in version 2.3: Support for other error handling schemes added.

Changed in version 2.7: Support for keyword arguments added.

`str.`**`encode`**([*encoding*[, *errors*]])

Return an encoded version of the string. Default encoding is the current default string encoding. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a **`UnicodeError`**. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via **`codecs.register_error()`**, see section Codec Base Classes. For a list of possible encodings, see section Standard Encodings.

New in version 2.0.

Changed in version 2.3: Support for `'xmlcharrefreplace'` and `'backslashreplace'` and other error handling schemes added.

Changed in version 2.7: Support for keyword arguments added.

`str.`**`endswith`**(*suffix*[, *start*[, *end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

Changed in version 2.5: Accept tuples as *suffix*.

`str.`**`expandtabs`**([*tabsize*])

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined

character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

`str.`**`find`**(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

> **Note:** The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:
>
> ```
> >>> 'Py' in 'Python'
> True
> ```

`str.`**`format`**(**args*, ***kwargs*)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

This method of string formatting is the new standard in Python 3, and should be preferred to the `%` formatting described in String Formatting Operations in new code.

New in version 2.6.

`str.`**`index`**(*sub*[, *start*[, *end*]])

Like `find()`, but raise `ValueError` when the substring is not found.

`str.`**`isalnum`**()

Return true if all characters in the string are alphanumeric and there is at least one

character, false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`isalpha`**()
Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`isdigit`**()
Return true if all characters in the string are digits and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`islower`**()
Return true if all cased characters [4] in the string are lowercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`isspace`**()
Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`istitle`**()
Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`isupper`**()
Return true if all cased characters [4] in the string are uppercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

`str.`**`join`**(*iterable*)
Return a string which is the concatenation of the strings in the iterable *iterable*. The separator between elements is the string providing this method.

`str.`**`ljust`**(*width*[, *fillchar*])
Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than

or equal to `len(s)`.

Changed in version 2.4: Support for the *fillchar* argument.

`str.`**`lower`**()
Return a copy of the string with all the cased characters [4] converted to lowercase.

For 8-bit strings, this method is locale-dependent.

`str.`**`lstrip`**([*chars*])
Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.lstrip()                                    >>>
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

Changed in version 2.2.2: Support for the *chars* argument.

`str.`**`partition`**(*sep*)
Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

New in version 2.5.

`str.`**`replace`**(*old*, *new*[, *count*])
Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.`**`rfind`**(*sub*[, *start*[, *end*]])
Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.`**`rindex`**(*sub*[, *start*[, *end*]])
Like **`rfind()`** but raises **`ValueError`** when the substring *sub* is not found.

`str.`**`rjust`**(*width*[, *fillchar*])
Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than or equal to `len(s)`.

Changed in version 2.4: Support for the *fillchar* argument.

`str.`**`rpartition`**(*sep*)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

New in version 2.5.

`str.`**`rsplit`**([*sep*[, *maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, **`rsplit()`** behaves like **`split()`** which is described in detail below.

New in version 2.4.

`str.`**`rstrip`**([*chars*])

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()                                    >>>
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

Changed in version 2.2.2: Support for the *chars* argument.

`str.`**`split`**([*sep*[, *maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace.

Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example, `' 1  2   3  '.split()` returns `['1', '2', '3']`, and `'  1  2   3  '.split(None, 1)` returns `['1', '2  3  ']`.

`str.`**`splitlines`**([*keepends*])

Return a list of the lines in the string, breaking at line boundaries. This method uses the universal newlines approach to splitting lines. Line breaks are not included in the resulting list unless *keepends* is given and true.

For example, `'ab c\n\nde fg\rkl\r\n'.splitlines()` returns `['ab c', '', 'de fg', 'kl']`, while the same call with `splitlines(True)` returns `['ab c\n', '\n', 'de fg\r', 'kl\r\n']`.

Unlike **`split()`** when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line.

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

Return `True` if string starts with the *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

Changed in version 2.5: Accept tuples as *prefix*.

`str.`**`strip`**([*chars*])

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.strip()                                    >>>
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

Changed in version 2.2.2: Support for the *chars* argument.

`str.`**`swapcase`**()

Return a copy of the string with uppercase characters converted to lowercase and vice versa.

For 8-bit strings, this method is locale-dependent.

`str.`**`title`**()

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> "they're bill's friends from the UK".title()
"They're Bill's Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

For 8-bit strings, this method is locale-dependent.

str.**translate**(*table*[, *deletechars*])

Return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

You can use the **maketrans()** helper function in the **string** module to create a translation table. For string objects, set the *table* argument to None for translations that only delete characters:

```
>>> 'read this short text'.translate(None, 'aeiou')
'rd ths shrt txt'
```

New in version 2.6: Support for a None *table* argument.

For Unicode objects, the **translate()** method does not accept the optional *deletechars* argument. Instead, it returns a copy of the *s* where all characters have been mapped through the given translation table which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None. Unmapped characters are left untouched. Characters mapped to None are deleted. Note, a more flexible approach is to create a custom character mapping codec using the **codecs** module (see **encodings.cp1251** for an example).

str.**upper**()

Return a copy of the string with all the cased characters [4] converted to uppercase. Note that str.upper().isupper() might be False if s contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but

e.g. “Lt” (Letter, titlecase).

For 8-bit strings, this method is locale-dependent.

`str.`**`zfill`**(*width*)
Return the numeric string left filled with zeros in a string of length *width*. A sign prefix is handled correctly. The original string is returned if *width* is less than or equal to `len(s)`.

New in version 2.2.2.

The following methods are present only on unicode objects:

`unicode.`**`isnumeric`**()
Return `True` if there are only numeric characters in S, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH.

`unicode.`**`isdecimal`**()
Return `True` if there are only decimal characters in S, `False` otherwise. Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

5.6.2. String Formatting Operations

String and Unicode objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string or Unicode object), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to the using **`sprintf()`** in the C language. If *format* is a Unicode object, or if any of the objects being converted using the `%s` conversion are Unicode objects, the result will also be a Unicode object.

If *format* requires a single argument, *values* may be a single non-tuple object. [5] Otherwise, *values* must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'`

(an asterisk), the actual width is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print '%(language)s has %(number)03d quote types.' % \
...       {"language": "Python", "number": 2}
Python has 002 quote types.
```

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
`'#'`	The value conversion will use the “alternate form” (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a “space” flag).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python – so e.g. `%ld` is identical to `%d`.

The conversion types are:

Conversion	Meaning	Notes
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type – it is identical to `'d'`.	(7)
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating point exponential format (lowercase).	(3)
`'E'`	Floating point exponential format (uppercase).	(3)
`'f'`	Floating point decimal format.	(3)
`'F'`	Floating point decimal format.	(3)

`'g'`	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single character (accepts integer or single character string).	
`'r'`	String (converts any Python object using repr()).	(5)
`'s'`	String (converts any Python object using `str()`).	(6)
`'%'`	No argument is converted, results in a `'%'` character in the result.	

Notes:

1. The alternate form causes a leading zero (`'0'`) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

2. The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the `'x'` or `'X'` format was used) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.

4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

   The precision determines the number of significant digits before and after the decimal point and defaults to 6.

5. The `%r` conversion was added in Python 2.0.

   The precision determines the maximal number of characters used.

6. If the object or format provided is a `unicode` string, the resulting string will also be `unicode`.

   The precision determines the maximal number of characters used.

7. See **PEP 237**.

Since Python strings have an explicit length, `%s` conversions do not assume that `'\0'` is the end of the string.

Changed in version 2.7: `%f` conversions for numbers whose absolute value is over 1e50 are no longer replaced by `%g` conversions.

Additional string operations are defined in standard modules **`string`** and **`re`**.

5.6.3. XRange Type

The **`xrange`** type is an immutable sequence which is commonly used for looping. The advantage of the **`xrange`** type is that an **`xrange`** object will always take the same amount of memory, no matter the size of the range it represents. There are no consistent performance advantages.

XRange objects have very little behavior: they only support indexing, iteration, and the **`len()`** function.

5.6.4. Mutable Sequence Types

List and **`bytearray`** objects support additional operations that allow in-place modification of the object. Other mutable sequence types (when added to the language) should also support these operations. Strings and tuples are immutable sequence types: such objects cannot be modified once created. The following operations are defined on mutable sequence types (where *x* is an arbitrary object):

Operation	Result	Notes
`s[i] = x`	item *i* of *s* is replaced by *x*	
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*	
`del s[i:j]`	same as `s[i:j] = []`	
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*	(1)
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list	
`s.append(x)`	same as `s[len(s):len(s)] = [x]`	(2)
`s.extend(x)` or `s += t`	for the most part the same as `s[len(s):len(s)] = x`	(3)
`s *= n`	updates *s* with its contents repeated *n* times	(11)
`s.count(x)`	return number of *i*'s for which `s[i] == x`	
`s.index(x[, i[, j]])`	return smallest *k* such that `s[k] == x` and `i <= k < j`	(4)
`s.insert(i, x)`	same as `s[i:i] = [x]`	(5)
`s.pop([i])`	same as `x = s[i]; del s[i]; return x`	(6)

`s.remove(x)`	same as `del s[s.index(x)]`	(4)
`s.reverse()`	reverses the items of *s* in place	(7)
`s.sort([cmp[, key[, reverse]]])`	sort the items of *s* in place	(7)(8)(9)(10)

Notes:

1. *t* must have the same length as the slice it is replacing.

2. The C implementation of Python has historically accepted multiple parameters and implicitly joined them into a tuple; this no longer works in Python 2.0. Use of this misfeature has been deprecated since Python 1.4.

3. *x* can be any iterable object.

4. Raises **`ValueError`** when *x* is not found in *s*. When a negative index is passed as the second or third parameter to the **`index()`** method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

   Changed in version 2.3: Previously, **`index()`** didn't have arguments for specifying start and stop positions.

5. When a negative index is passed as the first parameter to the **`insert()`** method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

   Changed in version 2.3: Previously, all negative indices were truncated to zero.

6. The **`pop()`** method's optional argument *i* defaults to `-1`, so that by default the last item is removed and returned.

7. The **`sort()`** and **`reverse()`** methods modify the list in place for economy of space when sorting or reversing a large list. To remind you that they operate by side effect, they don't return the sorted or reversed list.

8. The **`sort()`** method takes optional arguments for controlling the comparisons.

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

   key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None`.

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

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

Changed in version 2.3: Support for `None` as an equivalent to omitting *cmp* was added.

Changed in version 2.4: Support for *key* and *reverse* was added.

9. Starting with Python 2.3, the `sort()` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

10. **CPython implementation detail:** While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python 2.3 and newer makes the list appear empty for the duration, and raises `ValueError` if it can detect that the list has been mutated during a sort.

11. The value *n* is an integer, or an object implementing `__index__()`. Zero and negative values of *n* clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange.

5.7. Set Types — `set`, `frozenset`

A *set* object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built in `dict`, `list`, and `tuple` classes, and the `collections` module.)

New in version 2.4.

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, `set` and `frozenset`. The `set` type is mutable — the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The `frozenset` type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

As of Python 2.7, non-empty sets (not frozensets) can be created by placing a comma-

separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the `set` constructor.

The constructors for both classes work the same:

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

Return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be hashable. To represent sets of sets, the inner sets must be `frozenset` objects. If *iterable* is not specified, a new empty set is returned.

Instances of `set` and `frozenset` provide the following operations:

len(s)

Return the number of elements in set *s* (cardinality of *s*).

x in s

Test *x* for membership in *s*.

x not in s

Test *x* for non-membership in *s*.

isdisjoint(*other*)

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

New in version 2.6.

issubset(*other*)
set <= other

Test whether every element in the set is in *other*.

set < other

Test whether the set is a proper subset of *other*, that is, `set <= other and set != other`.

issuperset(*other*)
set >= other

Test whether every element in *other* is in the set.

set > other

Test whether the set is a proper superset of *other*, that is, `set >= other and set != other`.

union(*other*, *...*)
set | other | ...

Return a new set with elements from the set and all others.

Changed in version 2.6: Accepts multiple input iterables.

intersection(*other*, *...*)
set & other & ...
Return a new set with elements common to the set and all others.

Changed in version 2.6: Accepts multiple input iterables.

difference(*other*, *...*)
set - other - ...
Return a new set with elements in the set that are not in the others.

Changed in version 2.6: Accepts multiple input iterables.

symmetric_difference(*other*)
set ^ other
Return a new set with elements in either the set or *other* but not both.

copy()
Return a new set with a shallow copy of *s*.

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()`, `issubset()`, and `issuperset()` methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both `set` and `frozenset` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of `set` are compared to instances of `frozenset` based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two non-empty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a<b`, `a==b`, or `a>b`. Accordingly, sets do not implement the `__cmp__()` method.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

Set elements, like dictionary keys, must be hashable.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of `frozenset`.

The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

update(*other*, *...*)
set |= other | ...
Update the set, adding elements from all others.

Changed in version 2.6: Accepts multiple input iterables.

intersection_update(*other*, *...*)
set &= other & ...
Update the set, keeping only elements found in it and all others.

Changed in version 2.6: Accepts multiple input iterables.

difference_update(*other*, *...*)
set -= other | ...
Update the set, removing elements found in others.

Changed in version 2.6: Accepts multiple input iterables.

symmetric_difference_update(*other*)
set ^= other
Update the set, keeping only elements found in either set, but not in both.

add(*elem*)
Add element *elem* to the set.

remove(*elem*)
Remove element *elem* from the set. Raises `KeyError` if *elem* is not contained in the set.

discard(*elem*)
Remove element *elem* from the set if it is present.

pop()
Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

clear()
Remove all elements from the set.

Note, the non-operator versions of the `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` methods will accept any iterable as an argument.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, the *elem* set is temporarily mutated during the search and then restored. During the search, the *elem* set should not be read or mutated since it does not have a meaningful value.

See also:

Comparison to the built-in set types
Differences between the `sets` module and the built-in set types.

5.8. Mapping Types — `dict`

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary's keys are *almost* arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of `key: value` pairs within braces, for example: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`, or by the `dict` constructor.

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

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

New in version 2.2.

Changed in version 2.3: Support for building a dictionary from keyword arguments added.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

`len(d)`

Return the number of items in the dictionary *d*.

`d[key]`

Return the item of *d* with key *key*. Raises a **`KeyError`** if *key* is not in the map.

If a subclass of dict defines a method **`__missing__()`** and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke **`__missing__()`**. If **`__missing__()`** is not defined, **`KeyError`** is raised. **`__missing__()`** must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of `collections.Counter`. A different `__missing__` method is used by `collections.defaultdict`.

New in version 2.5: Recognition of __missing__ methods of dict subclasses.

d[key] = value
Set `d[key]` to *value*.

del d[key]
Remove `d[key]` from *d*. Raises a `KeyError` if *key* is not in the map.

key in d
Return `True` if *d* has a key *key*, else `False`.

New in version 2.2.

key not in d
Equivalent to `not key in d`.

New in version 2.2.

iter(d)
Return an iterator over the keys of the dictionary. This is a shortcut for `iterkeys()`.

clear()
Remove all items from the dictionary.

copy()
Return a shallow copy of the dictionary.

fromkeys(*seq*[, *value*])
Create a new dictionary with keys from *seq* and values set to *value*.

`fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`.

New in version 2.3.

get(*key*[, *default*])
Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

has_key(*key*)
Test for the presence of *key* in the dictionary. `has_key()` is deprecated in favor of `key in d`.

items()

Return a copy of the dictionary's list of `(key, value)` pairs.

> **CPython implementation detail:** Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions.

If **`items()`**, **`keys()`**, **`values()`**, **`iteritems()`**, **`iterkeys()`**, and **`itervalues()`** are called with no intervening modifications to the dictionary, the lists will directly correspond. This allows the creation of `(value, key)` pairs using **`zip()`**: `pairs = zip(d.values(), d.keys())`. The same relationship holds for the **`iterkeys()`** and **`itervalues()`** methods: `pairs = zip(d.itervalues(), d.iterkeys())` provides the same value for `pairs`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.iteritems()]`.

`iteritems`()

Return an iterator over the dictionary's `(key, value)` pairs. See the note for **`dict.items()`**.

Using **`iteritems()`** while adding or deleting entries in the dictionary may raise a **`RuntimeError`** or fail to iterate over all entries.

New in version 2.2.

`iterkeys`()

Return an iterator over the dictionary's keys. See the note for **`dict.items()`**.

Using **`iterkeys()`** while adding or deleting entries in the dictionary may raise a **`RuntimeError`** or fail to iterate over all entries.

New in version 2.2.

`itervalues`()

Return an iterator over the dictionary's values. See the note for **`dict.items()`**.

Using **`itervalues()`** while adding or deleting entries in the dictionary may raise a **`RuntimeError`** or fail to iterate over all entries.

New in version 2.2.

`keys`()

Return a copy of the dictionary's list of keys. See the note for **`dict.items()`**.

`pop`(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a **`KeyError`** is raised.

New in version 2.3.

popitem()

Remove and return an arbitrary `(key, value)` pair from the dictionary.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

setdefault(*key*[, *default*])

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update([*other*])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

`update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

Changed in version 2.4: Allowed the argument to be an iterable of key/value pairs and allowed keyword arguments.

values()

Return a copy of the dictionary's list of values. See the note for `dict.items()`.

viewitems()

Return a new view of the dictionary's items (`(key, value)` pairs). See below for documentation of view objects.

New in version 2.7.

viewkeys()

Return a new view of the dictionary's keys. See below for documentation of view objects.

New in version 2.7.

viewvalues()

Return a new view of the dictionary's values. See below for documentation of view objects.

New in version 2.7.

Dictionaries compare equal if and only if they have the same `(key, value)` pairs.

5.8.1. Dictionary view objects

The objects returned by `dict.viewkeys()`, `dict.viewvalues()` and `dict.viewitems()` are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

`len(dictview)`
Return the number of entries in the dictionary.

`iter(dictview)`
Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of `(value, key)` pairs using `zip()`: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

`x in dictview`
Return `True` if *x* is in the underlying dictionary's keys, values or items (in the latter case, *x* should be a `(key, value)` tuple).

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) Then these set operations are available ("other" refers either to another view or a set):

`dictview & other`
Return the intersection of the dictview and the other object as a new set.

`dictview | other`
Return the union of the dictview and the other object as a new set.

`dictview - other`
Return the difference between the dictview and the other object (all elements in *dictview* that aren't in *other*) as a new set.

`dictview ^ other`

Return the symmetric difference (all elements either in *dictview* or *other*, but not in both) of the dictview and the other object as a new set.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.viewkeys()
>>> values = dishes.viewvalues()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
```

5.9. File Objects

File objects are implemented using C's `stdio` package and can be created with the built-in **`open()`** function. File objects are also returned by some other built-in functions and methods, such as **`os.popen()`** and **`os.fdopen()`** and the **`makefile()`** method of socket objects. Temporary files can be created using the **`tempfile`** module, and high-level file operations such as copying, moving, and deleting files and directories can be achieved with the **`shutil`** module.

When a file operation fails for an I/O-related reason, the exception **`IOError`** is raised. This includes situations where the operation is not defined for some reason, like **`seek()`** on a tty device or writing a file opened for reading.

Files have the following methods:

`file.`**`close`**()

Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a **`ValueError`** after the file has been closed. Calling **`close()`** more than once is allowed.

As of Python 2.5, you can avoid having to call this method explicitly if you use the `with` statement. For example, the following code will automatically close *f* when the `with` block is exited:

```
from __future__ import with_statement # This isn't required in Python 2.6

with open("hello.txt") as f:
    for line in f:
        print line,
```

In older versions of Python, you would have needed to do this to get the same effect:

```
f = open("hello.txt")
try:
    for line in f:
        print line,
finally:
    f.close()
```

> **Note:** Not all "file-like" types in Python support use as a context manager for the `with` statement. If your code is intended to work with any file-like object, you can use the function `contextlib.closing()` instead of using the object directly.

`file.`**`flush`**()

Flush the internal buffer, like `stdio`'s **`fflush()`**. This may be a no-op on some file-like objects.

> **Note:** `flush()` does not necessarily write the file's data to disk. Use `flush()` followed by `os.fsync()` to ensure this behavior.

`file.`**`fileno`**()

Return the integer "file descriptor" that is used by the underlying implementation to request I/O operations from the operating system. This can be useful for other, lower level interfaces that use file descriptors, such as the `fcntl` module or `os.read()` and friends.

> **Note:** File-like objects which do not have a real file descriptor should *not* provide this method!

`file.`**`isatty`**()

Return `True` if the file is connected to a tty(-like) device, else `False`.

> **Note:** If a file-like object is not associated with a real file, this method should *not* be implemented.

`file.`**`next`**()

A file object is its own iterator, for example `iter(f)` returns *f* (unless *f* is closed). When a file is used as an iterator, typically in a **`for`** loop (for example, `for line in f: print line.strip()`), the **`next()`** method is called repeatedly. This method returns the next input line, or raises **`StopIteration`** when EOF is hit when the file is open for reading (behavior is undefined when the file is open for writing). In order to make a **`for`** loop the most efficient way of looping over the lines of a file (a very common operation), the **`next()`** method uses a hidden read-ahead buffer. As a consequence of using a read-ahead buffer, combining **`next()`** with other file methods (like **`readline()`**) does not work right. However, using **`seek()`** to reposition the file to an absolute position will flush the read-ahead buffer.

New in version 2.3.

`file.`**`read`**([*size*])

Read at most *size* bytes from the file (less if the read hits EOF before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.) Note that this method may call the underlying C function **`fread()`** more than once in an effort to acquire as close to *size* bytes as possible. Also note that when in non-blocking mode, less data than was requested may be returned, even if no *size* parameter was given.

> **Note:** This function is simply a wrapper for the underlying **`fread()`** C function, and will behave the same in corner cases, such as whether the EOF value is cached.

`file.`**`readline`**([*size*])

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). [6] If the *size* argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. When *size* is not 0, an empty string is returned *only* when EOF is encountered immediately.

> **Note:** Unlike `stdio`'s **`fgets()`**, the returned string contains null characters (`'\0'`) if they occurred in the input.

`file.`**`readlines`**([*sizehint*])

Read until EOF using **`readline()`** and return a list containing the lines thus read. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totalling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read. Objects implementing a file-like interface may choose to ignore *sizehint* if it cannot be implemented, or cannot be implemented efficiently.

`file.`**`xreadlines`**()

This method returns the same thing as `iter(f)`.

New in version 2.1.

> *Deprecated since version 2.3:* Use `for line in file` instead.

`file.`**`seek`**(*offset*[, *whence*])

Set the file's current position, like `stdio`'s **`fseek()`**. The *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end). There is no return value.

For example, `f.seek(2, os.SEEK_CUR)` advances the position by two and `f.seek(-3, os.SEEK_END)` sets the position to the third to last.

Note that if the file is opened for appending (mode `'a'` or `'a+'`), any **`seek()`** operations will be undone at the next write. If the file is only opened for writing in append mode (mode `'a'`), this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode `'a+'`). If the file is opened in text mode (without `'b'`), only offsets returned by **`tell()`** are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

Changed in version 2.6: Passing float values as offset has been deprecated.

`file.`**`tell`**()

Return the file's current position, like `stdio`'s **`ftell()`**.

> **Note:** On Windows, **`tell()`** can return illegal values (after an **`fgets()`**) when reading files with Unix-style line-endings. Use binary mode (`'rb'`) to circumvent this problem.

`file.`**`truncate`**([*size*])

Truncate the file's size. If the optional *size* argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content. Availability: Windows, many Unix variants.

`file.`**`write`**(*str*)

Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the **`flush()`** or **`close()`** method is called.

`file.`**`writelines`**(*sequence*)

Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match `readlines()`; `writelines()` does not add line separators.)

Files support the iterator protocol. Each iteration returns the same result as `readline()`, and iteration ends when the `readline()` method returns an empty string.

File objects also offer a number of other interesting attributes. These are not required for file-like objects, but should be implemented if they make sense for the particular object.

`file.`**`closed`**

bool indicating the current state of the file object. This is a read-only attribute; the `close()` method changes the value. It may not be available on all file-like objects.

`file.`**`encoding`**

The encoding that this file uses. When Unicode strings are written to a file, they will be converted to byte strings using this encoding. In addition, when the file is connected to a terminal, the attribute gives the encoding that the terminal is likely to use (that information might be incorrect if the user has misconfigured the terminal). The attribute is read-only and may not be present on all file-like objects. It may also be `None`, in which case the file uses the system default encoding for converting Unicode strings.

New in version 2.3.

`file.`**`errors`**

The Unicode error handler used along with the encoding.

New in version 2.6.

`file.`**`mode`**

The I/O mode for the file. If the file was created using the `open()` built-in function, this will be the value of the *mode* parameter. This is a read-only attribute and may not be present on all file-like objects.

`file.`**`name`**

If the file object was created using `open()`, the name of the file. Otherwise, some string that indicates the source of the file object, of the form `<...>`. This is a read-only attribute and may not be present on all file-like objects.

`file.`**`newlines`**

If Python was built with universal newlines enabled (the default) this read-only attribute exists, and for files opened in universal newline read mode it keeps track of the types of newlines encountered while reading the file. The values it can take are `'\r'`, `'\n'`, `'\r\n'`, `None` (unknown, no newlines read yet) or a tuple containing all the newline types seen, to indicate that multiple newline conventions were encountered. For files not opened in universal newlines read mode the value of this attribute will be `None`.

file.**softspace**

Boolean that indicates whether a space character needs to be printed before another value when using the `print` statement. Classes that are trying to simulate a file object should also have a writable `softspace` attribute, which should be initialized to zero. This will be automatic for most classes implemented in Python (care may be needed for objects that override attribute access); types implemented in C will have to provide a writable `softspace` attribute.

> **Note:** This attribute is not used to control the `print` statement, but to allow the implementation of `print` to keep track of its internal state.

5.10. memoryview type

New in version 2.7.

`memoryview` objects allow Python code to access the internal data of an object that supports the buffer protocol without copying. Memory is generally interpreted as simple bytes.

class **memoryview**(*obj*)

Create a `memoryview` that references *obj*. *obj* must support the buffer protocol. Built-in objects that support the buffer protocol include `str` and `bytearray` (but not `unicode`).

A `memoryview` has the notion of an *element*, which is the atomic memory unit handled by the originating object *obj*. For many simple types such as `str` and `bytearray`, an element is a single byte, but other third-party types may expose larger elements.

`len(view)` returns the total number of elements in the memoryview, *view*. The `itemsize` attribute will give you the number of bytes in a single element.

A `memoryview` supports slicing to expose its data. Taking a single index will return a single element as a `str` object. Full slicing will result in a subview:

```
>>> v = memoryview('abcefg')
>>> v[1]
'b'
>>> v[-1]
'g'
>>> v[1:4]
<memory at 0x77ab28>
>>> v[1:4].tobytes()
'bce'
```

If the object the memoryview is over supports changing its data, the memoryview supports slice assignment:

```
>>> data = bytearray('abcefg')
>>> v = memoryview(data)
```

```
>>> v.readonly
False
>>> v[0] = 'z'
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = '123'
>>> data
bytearray(b'z123fg')
>>> v[2] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot modify size of memoryview object
```

Notice how the size of the memoryview object cannot be changed.

`memoryview` has two methods:

`tobytes`()

Return the data in the buffer as a bytestring (an object of class `str`).

```
>>> m = memoryview("abc")
>>> m.tobytes()
'abc'
```

`tolist`()

Return the data in the buffer as a list of integers.

```
>>> memoryview("abc").tolist()
[97, 98, 99]
```

There are also several readonly attributes available:

`format`

A string containing the format (in `struct` module style) for each element in the view. This defaults to `'B'`, a simple bytestring.

`itemsize`

The size in bytes of each element of the memoryview.

`shape`

A tuple of integers the length of `ndim` giving the shape of the memory as a N-dimensional array.

`ndim`

An integer indicating how many dimensions of a multi-dimensional array the memory represents.

`strides`

A tuple of integers the length of `ndim` giving the size in bytes to access each

element for each dimension of the array.

readonly
A bool indicating whether the memory is read only.

5.11. Context Manager Types

New in version 2.5.

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using two separate methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

The *context management protocol* consists of a pair of methods that need to be provided for a context manager object to define a runtime context:

`contextmanager.`**`__enter__`**()
Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from __enter__() to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

`contextmanager.`**`__exit__`**(*exc_type*, *exc_val*, *exc_tb*)
Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`.

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does

not want to suppress the raised exception. This allows context management code (such as `contextlib.nested`) to easily detect whether or not an **`__exit__()`** method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the **`contextlib`** module for some examples.

Python's generators and the `contextlib.contextmanager` decorator provide a convenient way to implement these protocols. If a generator function is decorated with the `contextlib.contextmanager` decorator, it will return a context manager implementing the necessary **`__enter__()`** and **`__exit__()`** methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

5.12. Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

5.12.1. Modules

The only special operation on a module is attribute access: `m.name`, where *m* is a module and *name* accesses a name defined in *m*'s symbol table. Module attributes can be assigned to. (Note that the **`import`** statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named *foo* to exist, rather it requires an (external) *definition* for a module named *foo* somewhere.)

A special attribute of every module is **`__dict__`**. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the **`__dict__`** attribute is not possible (you can write `m.__dict__['a'] = 1`, which defines `m.a` to be `1`, but you can't write `m.__dict__ = {}`). Modifying **`__dict__`** directly is not recommended.

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

5.12.2. Classes and Class Instances

See Objects, values and types and Class definitions for these.

5.12.3. Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See Function definitions for more information.

5.12.4. Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as **`append()`** on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: `m.im_self` is the object on which the method operates, and `m.im_func` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)`.

Class instance methods are either *bound* or *unbound*, referring to whether the method was accessed through an instance or a class, respectively. When a method is unbound, its `im_self` attribute will be `None` and if called, an explicit `self` object must be passed as the first argument. In this case, `self` must be an instance of the unbound method's class (or a subclass of that class), otherwise a **`TypeError`** is raised.

Like function objects, methods objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`meth.im_func`), setting method attributes on either bound or unbound methods is disallowed. Attempting to set an attribute on a method results in an **`AttributeError`** being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'instancemethod' object has no attribute 'whoami'
>>> c.method.im_func.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See The standard type hierarchy for more information.

5.12.5. Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `func_code` attribute. See also the `code` module.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec` statement or the built-in `eval()` function.

See The standard type hierarchy for more information.

5.12.6. Type Objects

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<type 'int'>`.

5.12.7. The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name).

It is written as `None`.

5.12.8. The Ellipsis Object

This object is used by extended slice notation (see Slicings). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name).

It is written as `Ellipsis`. When in a subscript, it can also be written as `...`, for example `seq[...]`.

5.12.9. The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don’t support. See Comparisons for more information.

It is written as `NotImplemented`.

5.12.10. Boolean Values

Boolean values are the two constant objects `False` and `True`. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function **`bool()`** can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section Truth Value Testing above).

They are written as `False` and `True`, respectively.

5.12.11. Internal Objects

See The standard type hierarchy for this information. It describes stack frame objects, traceback objects, and slice objects.

5.13. Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the **`dir()`** built-in function.

`object.`**`__dict__`**
A dictionary or other mapping object used to store an object's (writable) attributes.

`object.`**`__methods__`**

Deprecated since version 2.2: Use the built-in function **`dir()`** to get a list of an object's attributes. This attribute is no longer available.

`object.`**`__members__`**

Deprecated since version 2.2: Use the built-in function **`dir()`** to get a list of an object's attributes. This attribute is no longer available.

`instance.`**`__class__`**
The class to which a class instance belongs.

`class.`**`__bases__`**
The tuple of base classes of a class object.

`class.`**`__name__`**
The name of the class or type.

The following attributes are only supported by new-style classes.

`class.`**`__mro__`**

This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

`class.`**`mro`**()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in `__mro__`.

`class.`**`__subclasses__`**()

Each new-style class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```
>>> int.__subclasses__()
[<type 'bool'>]
```

Footnotes

[1] Additional information on these special methods may be found in the Python Reference Manual (Basic customization).

[2] As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

[3] They must have since the parser can't tell the type of the operands.

[4] *(1, 2, 3, 4)* Cased characters are those with general category property being one of "Lu" (Letter, uppercase), "Ll" (Letter, lowercase), or "Lt" (Letter, titlecase).

[5] To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

[6] The advantage of leaving the newline on is that returning an empty string is then an unambiguous EOF indication. It is also possible (in cases where it might matter, for example, if you want to make an exact copy of a file while scanning its lines) to tell whether the last line of a file ended in a newline or not (yes this happens!).