

4. Built-in Types

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

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

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

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.

4.1. 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`, `0.0`, `0j`.
- any empty sequence, for example, `'`, `()`, `[]`.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__bool__()` 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.)

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

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
<code>x or y</code>	if <code>x</code> is false, then <code>y</code> , else <code>x</code>	(1)
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>	(2)
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>	(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.

4.3. Comparisons

There are eight comparison operations in Python. 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
<code><</code>	strictly less than
<code><=</code>	less than or equal
<code>></code>	strictly greater than
<code>>=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	object identity
<code>is not</code>	negated object identity

Objects of different types, except different numeric types, never compare equal. Furthermore, some types (for example, function objects) support only a degenerate notion of comparison where any two objects of that type are unequal. The `<`, `<=`, `>` and `>=` operators will raise a `TypeError` exception when comparing a complex number with another built-in numeric type, when the objects are of different types that cannot be

compared, or in other cases where there is no defined ordering.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` 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 enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

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

4.4. Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. 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 hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

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 integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

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

Notes:

1. Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily `int`. 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`.
2. Not for complex numbers. Instead convert to floats using [abs\(\)](#) if appropriate.
3. Conversion from floating point to integer may round or truncate as in C; see functions [math.floor\(\)](#) and [math.ceil\(\)](#) for well-defined conversions.
4. `float` also accepts the strings "nan" and "inf" with an optional prefix "+" or "-" for Not a Number (NaN) and positive or negative infinity.
5. Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages.
6. The numeric literals accepted include the digits `0` to `9` or any Unicode equivalent

(code points with the `Nd` property).

See <http://www.unicode.org/Public/6.3.0/ucd/extracted/DerivedNumericType.txt> for a complete list of code points with the `Nd` property.

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

Operation	Result	Notes
<code>math.trunc(x)</code>	<code>x</code> truncated to Integral	
<code>round(x[, n])</code>	<code>x</code> rounded to <code>n</code> digits, rounding half to even. If <code>n</code> is omitted, it defaults to 0.	
<code>math.floor(x)</code>	the greatest integral float $\leq x$	
<code>math.ceil(x)</code>	the least integral float $\geq x$	

For additional numeric operations see the `math` and `cmath` modules.

4.4.1. 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
<code>x y</code>	bitwise <i>or</i> of <code>x</code> and <code>y</code>	
<code>x ^ y</code>	bitwise <i>exclusive or</i> of <code>x</code> and <code>y</code>	
<code>x & y</code>	bitwise <i>and</i> of <code>x</code> and <code>y</code>	
<code>x << n</code>	<code>x</code> shifted left by <code>n</code> bits	(1)(2)
<code>x >> n</code>	<code>x</code> shifted right by <code>n</code> bits	(1)(3)
<code>~x</code>	the bits of <code>x</code> 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)` without overflow check.

3. A right shift by n bits is equivalent to division by `pow(2, n)` without overflow check.

4.4.2. Additional Methods on Integer Types

The `int` type implements the `numbers.Integral abstract base class`. In addition, it provides a few more methods:

`int.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 $2^{k-1} \leq \text{abs}(x) < 2^k$. Equivalently, when $\text{abs}(x)$ is small enough to have a correctly rounded logarithm, then $k = 1 + \text{int}(\log(\text{abs}(x), 2))$. 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 3.1.

`int.to_bytes(length, byteorder, *, signed=False)`

Return an array of bytes representing an integer.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')
b'\xe8\x03'
```

The integer is represented using *length* bytes. An `OverflowError` is raised if the integer is not representable with the given number of bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument determines whether two's complement is used to represent the integer. If *signed* is `False` and a negative integer is given, an `OverflowError` is raised. The default value for *signed* is `False`.

New in version 3.2.

classmethod `int.from_bytes(bytes, byteorder, *, signed=False)`

Return the integer represented by the given array of bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

The argument *bytes* must either be a *bytes-like object* or an iterable producing bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument indicates whether two's complement is used to represent the integer.

New in version 3.2.

4.4.3. 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.

`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
```

```
>>>
```

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.

classmethod `float.fromhex(s)`

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

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 be 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'
```

>>>

4.4.4. Hashing of numeric types

For numbers `x` and `y`, possibly of different types, it's a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo `P` for a fixed prime `P`. The value of `P` is made available to Python as the `modulus` attribute of `sys.hash_info`.

CPython implementation detail: Currently, the prime used is $P = 2^{31} - 1$ on machines with 32-bit C longs and $P = 2^{61} - 1$ on machines with 64-bit C longs.

Here are the rules in detail:

- If $x = m / n$ is a nonnegative rational number and `n` is not divisible by `P`, define `hash(x)` as `m * invmod(n, P) % P`, where `invmod(n, P)` gives the inverse of `n` modulo `P`.
- If $x = m / n$ is a nonnegative rational number and `n` is divisible by `P` (but `m` is not) then `n` has no inverse modulo `P` and the rule above doesn't apply; in this case define `hash(x)` to be the constant value `sys.hash_info.inf`.
- If $x = m / n$ is a negative rational number define `hash(x)` as `-hash(-x)`. If the resulting hash is `-1`, replace it with `-2`.
- The particular values `sys.hash_info.inf`, `-sys.hash_info.inf` and `sys.hash_info.nan` are used as hash values for positive infinity, negative infinity, or nans (respectively). (All hashable nans have the same hash value.)

- For a `complex` number `z`, the hash values of the real and imaginary parts are combined by computing `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, reduced modulo `2**sys.hash_info.width` so that it lies in `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. Again, if the result is `-1`, it's replaced with `-2`.

To clarify the above rules, here's some example Python code, equivalent to the built-in `hash`, for computing the hash of a rational number, `float`, or `complex`:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P. (Unnecessary if m and n already coprime.
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_ = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_ = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_ = -hash_
    if hash_ == -1:
        hash_ = -2
    return hash_

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_ = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
```

```
# do a signed reduction modulo 2**sys.hash_info.width
M = 2**(sys.hash_info.width - 1)
hash_ = (hash_ & (M - 1)) - (hash & M)
if hash_ == -1:
    hash_ == -2
return hash_
```

4.5. Iterator Types

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.

Once an iterator's `__next__()` method raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

4.5.1. 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](#).

4.6. Sequence Types — list, tuple, range

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of [binary data](#) and [text strings](#) are described in dedicated sections.

4.6.1. Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom 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*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.

Operation	Result	Notes
<code>x in s</code>	True if an item of <i>s</i> is equal to <i>x</i> , else False	(1)
<code>x not in s</code>	False if an item of <i>s</i> is equal to <i>x</i> , else True	(1)
<code>s + t</code>	the concatenation of <i>s</i> and <i>t</i>	(6)(7)
<code>s * n</code> or <code>n * s</code>	<i>n</i> shallow copies of <i>s</i> concatenated	(2)(7)
<code>s[i]</code>	<i>i</i> th item of <i>s</i> , origin 0	(3)
<code>s[i:j]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i>	(3)(4)
<code>s[i:j:k]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> with step <i>k</i>	(3)(5)
<code>len(s)</code>	length of <i>s</i>	

<code>min(s)</code>	smallest item of <code>s</code>	
<code>max(s)</code>	largest item of <code>s</code>	
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)	(8)
<code>s.count(x)</code>	total number of occurrences of <code>x</code> in <code>s</code>	

Sequences of the same type 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. While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as [str](#), [bytes](#) and [bytearray](#)) also use them for subsequence testing:

```
>>> "gg" in "eggs"
True
```

>>>

2. Values of `n` less than 0 are treated as 0 (which yields an empty sequence of the same type as `s`). Note also that the copies are shallow; nested structures are not copied. 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 (pointers 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]]
```

>>>

3. If i or j is negative, the index is relative to the end of the string: $\text{len}(s) + i$ or $\text{len}(s) + j$ is substituted. But note that $-\emptyset$ is still \emptyset .
4. The slice of s from i to j is defined as the sequence of items with index k such that $i \leq k < j$. If i or j is greater than $\text{len}(s)$, use $\text{len}(s)$. If i is omitted or `None`, use \emptyset . If j is omitted or `None`, use $\text{len}(s)$. 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 $\emptyset \leq n < (j-i)/k$. 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 $\text{len}(s)$, use $\text{len}(s)$. 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. Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:
 - if concatenating `str` objects, you can build a list and use `str.join()` at the end or else write to a `io.StringIO` instance and retrieve its value when complete
 - if concatenating `bytes` objects, you can similarly use `bytes.join()` or `io.BytesIO`, or you can do in-place concatenation with a `bytearray` object. `bytearray` objects are mutable and have an efficient overallocation mechanism
 - if concatenating `tuple` objects, extend a `list` instead
 - for other types, investigate the relevant class documentation
7. Some sequence types (such as `range`) only support item sequences that follow specific patterns, and hence don't support sequence concatenation or repetition.
8. `index` raises `ValueError` when x is not found in s . When supported, the additional arguments to the `index` method allow efficient searching of subsections of the sequence. Passing the extra arguments is roughly equivalent to using `s[i:j].index(x)`, only without copying any data and with the returned index being relative to the start of the sequence rather than the start of the slice.

4.6.2. Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the `hash()` built-in.

This support allows immutable sequences, such as `tuple` instances, to be used as `dict`

keys and stored in `set` and `frozenset` instances.

Attempting to hash an immutable sequence that contains unhashable values will result in `TypeError`.

4.6.3. Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The `collections.abc.MutableSequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table `s` is an instance of a mutable sequence type, `t` is any iterable object and `x` is an arbitrary object that meets any type and value restrictions imposed by `s` (for example, `bytearray` only accepts integers that meet the value restriction $0 \leq x \leq 255$).

Operation	Result	Notes
<code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>	
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>	
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>	(1)
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list	
<code>s.append(x)</code>	appends <i>x</i> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)	
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code>)	(5)
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code>)	(5)
<code>s.extend(t)</code>	extends <i>s</i> with the contents of <i>t</i> (same as <code>s[len(s):len(s)] = t</code>)	
<code>s.insert(i, x)</code>	inserts <i>x</i> into <i>s</i> at the index given by <i>i</i> (same as <code>s[i:i] = [x]</code>)	
<code>s.pop([i])</code>	retrieves the item at <i>i</i> and also removes it from <i>s</i>	(2)
<code>s.remove(x)</code>	remove the first item from <i>s</i> where <code>s[i] == x</code>	(3)

`s.reverse()`reverses the items of `s` in place

(4)

Notes:

1. `t` must have the same length as the slice it is replacing.
2. The optional argument `i` defaults to `-1`, so that by default the last item is removed and returned.
3. `remove` raises `ValueError` when `x` is not found in `s`.
4. The `reverse()` method modifies the sequence in place for economy of space when reversing a large sequence. To remind users that it operates by side effect, it does not return the reversed sequence.
5. `clear()` and `copy()` are included for consistency with the interfaces of mutable containers that don't support slicing operations (such as `dict` and `set`)

New in version 3.3: `clear()` and `copy()` methods.

4.6.4. Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

class `list([iterable])`

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a]`, `[a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

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

Many other operations also produce lists, including the `sorted()` built-in.

Lists implement all of the *common* and *mutable* sequence operations. Lists also provide the following additional method:

`sort(*, key=None, reverse=None)`

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

`sort()` accepts two arguments that can only be passed by keyword (*keyword-only arguments*):

`key` specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The `functools.cmp_to_key()` utility is available to convert a 2.x style *cmp* function to a *key* function.

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

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use `sorted()` to explicitly request a new sorted list instance).

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).

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 makes the list appear empty for the duration, and raises `ValueError` if it can detect that the list has been mutated during a sort.

4.6.5. Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the `enumerate()` built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a `set` or `dict` instance).

`class tuple([iterable])`

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the `tuple()` built-in: `tuple()` or `tuple(iterable)`

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

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, `f(a, b, c)` is a function call with three arguments, while `f((a, b, c))` is a function call with a 3-tuple as the sole argument.

Tuples implement all of the *common* sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, `collections.namedtuple()` may be a more appropriate choice than a simple tuple object.

4.6.6. Ranges

The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

```
class range(stop)
```

```
class range(start, stop[, step])
```

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__` special method). If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0. If *step* is zero, `ValueError` is raised.

For a positive *step*, the contents of a range *r* are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than `sys.maxsize` are permitted but some features (such as `len()`) may raise `OverflowError`.

Range examples:

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

Ranges implement all of the *common* sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

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

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

Testing range objects for equality with `==` and `!=` compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different start, stop and step attributes, for example `range(0) == range(2, 1, 3)` or `range(0, 3, 2) == range(0, 4, 2)`.)

Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test `int` objects for membership in constant time instead of iterating through all items.

Changed in version 3.3: Define `'=='` and `'!='` to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

New in version 3.3: The start, stop and step attributes.

4.7. Text Sequence Type — `str`

Textual data in Python is handled with `str` objects, or *strings*. Strings are immutable *sequences* of Unicode code points. String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: `"allows embedded 'single' quotes"`.
- Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, `("spam " "eggs") == "spam eggs"`.

See *[String and Bytes literals](#)* for more about the various forms of string literal, including supported escape sequences, and the `r` (“raw”) prefix that disables most escape sequence processing.

Strings may also be created from other objects using the `str` constructor.

Since there is no separate “character” type, indexing a string produces strings of length 1. That is, for a non-empty string `s`, `s[0] == s[0:1]`.

There is also no mutable string type, but `str.join()` or `io.StringIO` can be used to efficiently construct strings from multiple fragments.

Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

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

Return a [string](#) version of *object*. If *object* is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether *encoding* or *errors* is given, as follows.

If neither *encoding* nor *errors* is given, `str(object)` returns `object.__str__()`, which is the “informal” or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.

If at least one of *encoding* or *errors* is given, *object* should be a [bytes-like object](#) (e.g. [bytes](#) or [bytearray](#)). In this case, if *object* is a [bytes](#) (or [bytearray](#)) object, then `str(bytes, encoding, errors)` is equivalent to `bytes.decode(encoding, errors)`. Otherwise, the bytes object underlying the buffer object is obtained before calling `bytes.decode()`. See [Binary Sequence Types — bytes, bytearray, memoryview](#) and [Buffer Protocol](#) for information on buffer objects.

Passing a [bytes](#) object to `str()` without the *encoding* or *errors* arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

>>>

For more information on the `str` class and its methods, see [Text Sequence Type — str](#) and the [String Methods](#) section below. To output formatted strings, see the [String Formatting](#) section. In addition, see the [Text Processing Services](#) section.

4.7.1. String Methods

Strings implement all of the [common](#) sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, [Format String Syntax](#) and [String Formatting](#)) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can

handle ([printf-style String Formatting](#)).

The [Text Processing Services](#) section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the [re](#) module).

str.capitalize()

Return a copy of the string with its first character capitalized and the rest lowercased.

str.casefold()

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, [lower\(\)](#) would do nothing to 'ß'; [casefold\(\)](#) converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

str.center(*width*[, *fillchar*])

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

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.encode(*encoding*="utf-8", *errors*="strict")

Return an encoded version of the string as a bytes object. Default encoding is 'utf-8'. *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 [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

Changed in version 3.1: 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.

`str.expandtabs(tabsize=8)`

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, such that *sub* is contained in 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 {}".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.

`str.format_map(mapping)`

Similar to `str.format(**mapping)`, except that `mapping` is used directly and not copied to a `dict`. This is useful if for example `mapping` is a `dict` subclass:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

New in version 3.2.

`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. A character `c` is alphanumeric if one of the following returns True: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

`str.isalpha()`

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the “Alphabetic” property defined in the Unicode Standard.

`str.isdecimal()`

Return true if all characters in the string are decimal characters and there is at least one character, false otherwise. Decimal characters are those from general category “Nd”. This category includes digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

`str.isdigit()`

Return true if all characters in the string are digits and there is at least one character, false otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. Formally, a digit is a character that has the property value `Numeric_Type=Digit` or `Numeric_Type=Decimal`.

`str.isidentifier()`

Return true if the string is a valid identifier according to the language definition, section [Identifiers and keywords](#).

Use `keyword.iskeyword()` to test for reserved identifiers such as `def` and `class`.

`str.islower()`

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

`str.isnumeric()`

Return true if all characters in the string are numeric characters, and there is at least one character, 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. Formally, numeric characters are those with the property value `Numeric_Type=Digit`, `Numeric_Type=Decimal` or `Numeric_Type=Numeric`.

`str.isprintable()`

Return true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

`str.isspace()`

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise. Whitespace characters are those characters defined in the Unicode character database as “Other” or “Separator” and those with bidirectional property being one of “WS”, “B”, or “S”.

`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.

`str.isupper()`

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

`str.join(iterable)`

Return a string which is the concatenation of the strings in the [iterable](#) *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes`

objects. 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 an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.lower()

Return a copy of the string with all the cased characters [4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

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'
```

>>>

static str.maketrans(*x*[, *y*[, *z*]])

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or `None`. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *x* will be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

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.

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 an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

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.

str.rsplit(sep=None, maxsplit=-1)

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.

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'
```

```
>>>
```

str.split(sep=None, maxsplit=-1)

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 `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3'.split(',')
['1', '2', '', '3', '']
```

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()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

`str.splitlines([keepends])`

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of [universal newlines](#).

Representation	Description
<code>\n</code>	Line Feed
<code>\r</code>	Carriage Return
<code>\r\n</code>	Carriage Return + Line Feed

<code>\v</code> or <code>\x0b</code>	Line Tabulation
<code>\f</code> or <code>\x0c</code>	Form Feed
<code>\x1c</code>	File Separator
<code>\x1d</code>	Group Separator
<code>\x1e</code>	Record Separator
<code>\x85</code>	Next Line (C1 Control Code)
<code>\u2028</code>	Line Separator
<code>\u2029</code>	Paragraph Separator

Changed in version 3.2: `\v` and `\f` added to list of line boundaries.

For example:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['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:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, `split('\n')` gives:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

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.

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'
```

>>>

str.**swapcase()**

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

str.**title()**

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

For example:

```
>>> 'Hello world'.title()
'Hello World'
```

>>>

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.'
```

>>>

str.**translate(map)**

Return a copy of the *s* where all characters have been mapped through the *map* which must be a dictionary of Unicode ordinals (integers) to Unicode ordinals, strings

or None. Unmapped characters are left untouched. Characters mapped to None are deleted.

You can use `str.maketrans()` to create a translation map from character-to-character mappings in different formats.

Note: An even 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).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.zfill(width)`

Return a copy of the string left filled with ASCII ‘0’ digits to make a string of length *width*. A leading sign prefix (‘+’/‘-’) is handled by inserting the padding *after* the sign character rather than before. The original string is returned if *width* is less than or equal to `len(s)`.

For example:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

```
>>>
```

4.7.2. printf-style String Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer `str.format()` interface helps avoid these errors, and also provides a generally more powerful, flexible and extensible approach to formatting text.

String 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), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` in the C language.

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 precision 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()).	(5)
'a'	String (converts any Python object using ascii()).	(5)
'%'	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. If precision is N, the output is truncated to N characters.

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 3.1: %f conversions for numbers whose absolute value is over 1e50 are no longer replaced by %g conversions.

4.8. Binary Sequence Types — [bytes](#), [bytearray](#), [memoryview](#)

The core built-in types for manipulating binary data are [bytes](#) and [bytearray](#). They are supported by [memoryview](#) which uses the [buffer protocol](#) to access the memory of other binary objects without needing to make a copy.

The [array](#) module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

4.8.1. Bytes

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a b prefix is added:

- Single quotes: b'still allows embedded "double" quotes'
- Double quotes: b"still allows embedded 'single' quotes".

- Triple quoted: `b'''3 single quotes'''`, `b"""3 double quotes"""`

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a `r` prefix to disable processing of escape sequences. See [String and Bytes literals](#) for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that $0 \leq x < 256$ (attempts to violate this restriction will trigger `ValueError`). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

Also see the [bytes](#) built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:

classmethod `bytes.fromhex(string)`

This [bytes](#) class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII spaces being ignored.

```
>>> bytes.fromhex('2Ef0 F1f2 ')  
b'.\xf0\xf1\xf2'
```

```
>>>
```

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format `(b'...')` since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

Note: For Python 2.x users: In the Python 2.x series, a variety of implicit conversions between 8-bit strings (the closest thing 2.x offers to a built-in binary data type) and Unicode strings were permitted. This was a backwards compatibility workaround to account for the fact that Python originally only supported 8-bit text, and Unicode text was a later addition. In Python 3.x, those implicit conversions are gone - conversions between 8-bit binary data and Unicode text must be explicit, and bytes and string objects will always compare unequal.

4.8.2. Bytearray Objects

`bytearray` objects are a mutable counterpart to `bytes` objects. There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi!')`

As bytearray objects are mutable, they support the *mutable* sequence operations in addition to the common bytes and bytearray operations described in *Bytes and Bytearray Operations*.

Also see the `bytearray` built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

classmethod `bytearray.fromhex(string)`

This `bytearray` class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII spaces being ignored.

```
>>> bytearray.fromhex('2Ef0 F1f2 ')  
bytearray(b'.\xf0\xf1\xf2')
```

```
>>>
```

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format (`bytearray(b'...')`) since it is often more useful than e.g. `bytearray([46, 46, 46])`. You can always convert

a bytearray object into a list of integers using `list(b)`.

4.8.3. Bytes and Bytearray Operations

Both bytes and bytearray objects support the *common* sequence operations. They interoperate not just with operands of the same type, but with any *bytes-like object*. Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

Note: The methods on bytes and bytearray objects don't accept strings as their arguments, just as the methods on strings don't accept bytes as their arguments. For example, you have to write:

```
a = "abc"
b = a.replace("a", "f")
```

and:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

Note: Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The following methods on bytes and bytearray objects can be used with arbitrary binary data.

bytes. **count**(*sub*[, *start*[, *end*]])

bytearray. **count**(*sub*[, *start*[, *end*]])

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

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

```
bytes.decode(encoding="utf-8", errors="strict")
```

```
bytearray.decode(encoding="utf-8", errors="strict")
```

Return a string decoded from the given bytes. Default encoding is 'utf-8'. *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' and any other name registered via [codecs.register_error\(\)](#), see section [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

Note: Passing the *encoding* argument to [str](#) allows decoding any *bytes-like object* directly, without needing to make a temporary bytes or bytearray object.

Changed in version 3.1: Added support for keyword arguments.

```
bytes.endswith(suffix[, start[, end]])
```

```
bytearray.endswith(suffix[, start[, end]])
```

Return True if the binary data 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.

The suffix(es) to search for may be any *bytes-like object*.

```
bytes.find(sub[, start[, end]])
```

```
bytearray.find(sub[, start[, end]])
```

Return the lowest index in the data where the subsequence *sub* is found, such that *sub* is contained in the slice *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

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:

```
>>> b'Py' in b'Python'
True
```

```
>>>
```

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

```
bytes.index(sub[, start[, end]])
```

```
bytearray.index(sub[, start[, end]])
```

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

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.join(iterable)`

`bytearray.join(iterable)`

Return a bytes or bytearray object which is the concatenation of the binary data sequences in the *iterable* `iterable`. A `TypeError` will be raised if there are any values in *iterable* that are not *bytes-like objects*, including `str` objects. The separator between elements is the contents of the bytes or bytearray object providing this method.

static `bytes.maketrans(from, to)`

static `bytearray.maketrans(from, to)`

This static method returns a translation table usable for `bytes.translate()` that will map each character in *from* into the character at the same position in *to*; *from* and *to* must both be *bytes-like objects* and have the same length.

New in version 3.1.

`bytes.partition(sep)`

`bytearray.partition(sep)`

Split the sequence at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray objects.

The separator to search for may be any *bytes-like object*.

`bytes.replace(old, new[, count])`

`bytearray.replace(old, new[, count])`

Return a copy of the sequence with all occurrences of subsequence *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The subsequence to search for and its replacement may be any *bytes-like object*.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.rfind(sub[, start[, end]])`

`bytearray.rfind(sub[, start[, end]])`

Return the highest index in the sequence where the subsequence *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.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rindex(sub[, start[, end]])`

`bytearray.rindex(sub[, start[, end]])`

Like `rfind()` but raises `ValueError` when the subsequence *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rpartition(sep)`

`bytearray.rpartition(sep)`

Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray objects.

The separator to search for may be any *bytes-like object*.

`bytes.startswith(prefix[, start[, end]])`

`bytearray.startswith(prefix[, start[, end]])`

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

The prefix(es) to search for may be any *bytes-like object*.

`bytes.translate(table[, delete])`

`bytearray.translate(table[, delete])`

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument *delete* are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the `bytes.maketrans()` method to create a translation table.

Set the *table* argument to `None` for translations that only delete characters:

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

>>>

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

`bytes.center(width[, fillbyte])`

`bytearray.center(width[, fillbyte])`

Return a copy of the object centered in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.ljust(width[, fillbyte])`

`bytearray.ljust(width[, fillbyte])`

Return a copy of the object left justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.lstrip([chars])`

`bytearray.lstrip([chars])`

Return a copy of the sequence with specified leading bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

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

>>>

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.rjust(width[, fillbyte])`

`bytearray.rjust(width[, fillbyte])`

Return a copy of the object right justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For *bytes* objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.rsplit(sep=None, maxsplit=-1)`

`bytearray.rsplit(sep=None, maxsplit=-1)`

Split the binary sequence into subsequences of the same type, 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 subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

`bytes.rstrip([chars])`

`bytearray.rstrip([chars])`

Return a copy of the sequence with specified trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or None, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

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

>>>

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**split**(sep=None, maxsplit=-1)

bytearray.**split**(sep=None, maxsplit=-1)

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given and non-negative, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified or is -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 subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence (for example, `b'1<>2<>3'.split(b'<>')` returns `[b'1', b'2', b'3']`). Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

For example:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

If *sep* is not specified or is None, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

For example:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b' 1 2 3 '.split()
[b'1', b'2', b'3']
```

bytes.**strip**([chars])

bytearray.**strip**([chars])

Return a copy of the sequence with specified leading and trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or None, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

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

>>>

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytearray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

bytes.**capitalize()**

bytearray.**capitalize()**

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**expandtabs(*tabsize*=8)**

bytearray.**expandtabs(*tabsize*=8)**

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (b'\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 current byte is an ASCII newline (b'\n') or carriage return (b'\r'), it is copied and the current column

is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

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

>>>

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**isalnum()**

bytearray.**isalnum()**

Return true if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'. ASCII decimal digits are those byte values in the sequence b'0123456789'.

For example:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

>>>

bytes.**isalpha()**

bytearray.**isalpha()**

Return true if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.

For example:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

>>>

bytes.**isdigit()**

bytearray.**isdigit()**

Return true if all bytes in the sequence are ASCII decimal digits and the sequence is

not empty, false otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

>>>

`bytes.islower()`

`bytearray.islower()`

Return true if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, false otherwise.

For example:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

>>>

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.isspace()`

`bytearray.isspace()`

Return true if all bytes in the sequence are ASCII whitespace and the sequence is not empty, false otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\r\n\b'` (space, tab, newline, carriage return, vertical tab, form feed).

`bytes.istitle()`

`bytearray.istitle()`

Return true if the sequence is ASCII titlecase and the sequence is not empty, false otherwise. See [bytes.title\(\)](#) for more details on the definition of “titlecase”.

For example:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

>>>

bytes.**isupper()**

bytearray.**isupper()**

Return true if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, false otherwise.

For example:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

>>>

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**lower()**

bytearray.**lower()**

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

For example:

```
>>> b'Hello World'.lower()
b'hello world'
```

>>>

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**splitlines(keepends=False)**

bytearray.**splitlines(keepends=False)**

Return a list of the lines in the binary sequence, breaking at ASCII 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:

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
```

>>>

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'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:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

>>>

bytes. **swapcase()**

bytearray. **swapcase()**

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

For example:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

>>>

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike `str.swapcase()`, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes. **title()**

bytearray. **title()**

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining characters are lowercase. Uncased byte values are left unmodified.

For example:

```
>>> b'Hello world'.title()
b'Hello World'
```

>>>

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

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:

```
>>> b"they're bill's friends from the UK".title()
b'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(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                               mo.group(0)[1:].lower(),
...                   s)
>>> titlecase(b"they're bill's friends.")
b'They're Bill's Friends.'
```

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes. **upper()**

bytearray. **upper()**

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

For example:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: The bytearray version of this method does *not* operate in place - it always

produces a new object, even if no changes were made.

`bytes.zfill(width)`

`bytearray.zfill(width)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length *width*. A leading sign prefix (`b'+' / b'-'`) is handled by inserting the padding *after* the sign character rather than before. For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(seq)`.

For example:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

>>>

Note: The `bytearray` version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

4.8.4. Memory Views

`memoryview` objects allow Python code to access the internal data of an object that supports the *buffer protocol* without copying.

`class memoryview(obj)`

Create a `memoryview` that references *obj*. *obj* must support the buffer protocol. Built-in objects that support the buffer protocol include `bytes` and `bytearray`.

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 `bytes` and `bytearray`, an element is a single byte, but other types such as `array.array` may have bigger elements.

`len(view)` is equal to the length of `tolist`. If `view.ndim = 0`, the length is 1. If `view.ndim = 1`, the length is equal to the number of elements in the view. For higher dimensions, the length is equal to the length of the nested list representation of the view. The `itemsize` attribute will give you the number of bytes in a single element.

A `memoryview` supports slicing to expose its data. If `format` is one of the native format specifiers from the `struct` module, indexing will return a single element with the correct type. Full slicing will result in a subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

Other native formats:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> a[0]
-11111111
>>> a[-1]
44444444
>>> a[2:3].tolist()
[-33333333]
>>> a[::2].tolist()
[-11111111, -33333333]
>>> a[::-1].tolist()
[44444444, -33333333, 22222222, -11111111]
```

New in version 3.3.

If the underlying object is writable, the memoryview supports slice assignment. Resizing is not allowed:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different str
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

One-dimensional memoryviews of hashable (read-only) types with formats 'B', 'b' or 'c' are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Changed in version 3.3: One-dimensional memoryviews with formats 'B', 'b' or 'c' are now hashable.

Changed in version 3.4: `memoryview` is now registered automatically with `collections.abc.Sequence`

`memoryview` has several methods:

`__eq__`(*exporter*)

A memoryview and a [PEP 3118](#) exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands' respective format codes are interpreted using `struct` syntax.

For the subset of `struct` format strings currently supported by `tolist()`, `v` and `w` are equal if `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the `struct` module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating point numbers, `v is w` does *not* imply `v == w` for `memoryview` objects.

Changed in version 3.3: Previous versions compared the raw memory disregarding the item format and the logical array structure.

`tobytes()`

Return the data in the buffer as a bytestring. This is equivalent to calling the `bytes` constructor on the `memoryview`.

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

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. `tobytes()` supports all format strings, including those that are not in `struct` module syntax.

`tolist()`

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

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Changed in version 3.3: `tolist()` now supports all single character native formats in `struct` module syntax as well as multi-dimensional representations.

release()

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a `bytearray` would temporarily forbid resizing); therefore, calling `release()` is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a `ValueError` (except `release()` itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

New in version 3.2.

cast(format[, shape])

Cast a memoryview to a new format or shape. `shape` defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-contiguous and C-contiguous -> 1D.

Both formats are restricted to single element native formats in `struct` syntax. One of the formats must be a byte format ('B', 'b' or 'c'). The byte length of the result must be the same as the original length.

Cast 1D/long to 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
```

```

>>> x.format
'1'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24

```

Cast 1D/unsigned bytes to 1D/char:

```

>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')

```

Cast 1D/bytes to 3D/ints to 1D/signed char:

```

>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format

```

```
'b'  
>>> z.itemsize  
1  
>>> len(z)  
48  
>>> z.nbytes  
48
```

Cast 1D/unsigned char to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))  
>>> x = memoryview(buf)  
>>> y = x.cast('L', shape=[2,3])  
>>> len(y)  
2  
>>> y.nbytes  
48  
>>> y.tolist()  
[[0, 1, 2], [3, 4, 5]]
```

New in version 3.3.

There are also several readonly attributes available:

obj

The underlying object of the memoryview:

```
>>> b = bytearray(b'xyz')  
>>> m = memoryview(b)  
>>> m.obj is b  
True
```

New in version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)`:

```
>>> import array  
>>> a = array.array('i', [1,2,3,4,5])  
>>> m = memoryview(a)  
>>> len(m)  
5  
>>> m.nbytes  
20  
>>> y = m[::2]
```



```
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.0]]
>>> len(y)
3
>>> y.nbytes
96
```

New in version 3.3.

readonly

A bool indicating whether the memory is read only.

format

A string containing the format (in `struct` module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. `tolist()`) are restricted to native single element formats.

Changed in version 3.3: format 'B' is now handled according to the struct module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

The size in bytes of each element of the memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

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

shape

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

Changed in version 3.3: An empty tuple instead of None when `ndim = 0`.

strides

A tuple of integers the length of `ndim` giving the size in bytes to access each element for each dimension of the array.

Changed in version 3.3: An empty tuple instead of None when `ndim = 0`.

suboffsets

Used internally for PIL-style arrays. The value is informational only.

c_contiguous

A bool indicating whether the memory is C-contiguous.

New in version 3.3.

f_contiguous

A bool indicating whether the memory is Fortran contiguous.

New in version 3.3.

contiguous

A bool indicating whether the memory is contiguous.

New in version 3.3.

4.9. 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.)

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.

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 cardinality of set *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.

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.

intersection(*other*, ...)

set & other & ...

Return a new set with elements common to the set and all others.

difference(*other*, ...)

set - other - ...

Return a new set with elements in the set that are not in the others.

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 nonempty 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`.

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.

intersection_update(*other*, ...)

set &= **other** & ...

Update the set, keeping only elements found in it and all others.

difference_update(*other*, ...)

set -= **other** | ...

Update the set, removing elements found in others.

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.

4.10. 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.

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](#).

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.

key not in d

Equivalent to `not key in d`.

iter(d)

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

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod **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.

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](#).

items()

Return a new view of the dictionary's items ((*key*, *value*) pairs). See the [documentation of view objects](#).

keys()

Return a new view of the dictionary's keys. See the [documentation of view objects](#).

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.

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)`.

values()

Return a new view of the dictionary's values. See the [documentation of view objects](#).

See also: [types.MappingProxyType](#) can be used to create a read-only view of a dict.

4.10.1. Dictionary view objects

The objects returned by [dict.keys\(\)](#), [dict.values\(\)](#) and [dict.items\(\)](#) 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.) For set-like views, all of the operations defined for the abstract base class [collections.abc.Set](#) are available (for example, `==`, `<`, or `^`).

An example of dictionary view usage:

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

>>> # 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'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11. Context Manager Types

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of 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:

`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 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.

4.12. Other Built-in Types

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

4.12.1. 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'>.
```

4.12.2. Classes and Class Instances ¶

See [Objects, values and types](#) and [Class definitions](#) for these.

4.12.3. 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.

4.12.4. 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.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called *instance method*) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`meth.__func__`), setting method attributes on bound 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: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See [The standard type hierarchy](#) for more information.

4.12.5. 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 `__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()` or `eval()` built-in functions.

See [The standard type hierarchy](#) for more information.

4.12.6. 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: `<class 'int'>`.

4.12.7. 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). `type(None)()` produces the same singleton.

It is written as `None`.

4.12.8. The Ellipsis Object

This object is commonly used by slicing (see [Slicings](#)). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name). `type(Ellipsis)()`

produces the [Ellipsis](#) singleton.

It is written as `Ellipsis` or `...`.

4.12.9. 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. There is exactly one `NotImplemented` object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

4.12.10. 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.

4.12.11. Internal Objects

See [The standard type hierarchy](#) for this information. It describes stack frame objects, traceback objects, and slice objects.

4.13. 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.

`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.

`class.__qualname__`

The *qualified name* of the class or type.

New in version 3.3.

`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 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__()
[<class '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.