Python Memoire

[builtins: 2](#_Toc67181964)

[os: 4](#_Toc67181965)

[os.path: 6](#_Toc67181966)

[sys 7](#_Toc67181967)

[re 8](#_Toc67181968)

[datetime 9](#_Toc67181969)

[math 10](#_Toc67181970)

[random 11](#_Toc67181971)

[statistics 11](#_Toc67181972)

[itertools 12](#_Toc67181973)

[time 13](#_Toc67181974)

[glob 13](#_Toc67181975)

[site 13](#_Toc67181976)

[subprocess 13](#_Toc67181977)

[functools 13](#_Toc67181978)

[collections 13](#_Toc67181979)

[inspect 14](#_Toc67181980)

[operator 14](#_Toc67181981)

[requests 14](#_Toc67181982)

[textwrap 14](#_Toc67181983)

[multiprocessing 15](#_Toc67181984)

[cProfile 15](#_Toc67181985)

[socket 15](#_Toc67181986)

[hashlib 15](#_Toc67181987)

[traceback 15](#_Toc67181988)

[http.client 15](#_Toc67181989)

[numpy 16](#_Toc67181990)

[numpy.linalg 16](#_Toc67181991)

[sympy 16](#_Toc67181992)

[sympy.solvers.solveset 16](#_Toc67181993)

[set 17](#_Toc67181994)

[dictionary 18](#_Toc67181995)

[string 19](#_Toc67181996)

[list 22](#_Toc67181997)

[bytes 23](#_Toc67181998)

[bytearray 23](#_Toc67181999)

[memoryview 23](#_Toc67182000)

# packing and unpacking

<https://stackabuse.com/variable-length-arguments-in-python-with-args-and-kwargs>

<https://stackabuse.com/unpacking-in-python-beyond-parallel-assignment>

Get arbitrary number of values from beginning/end, only restriction is that there can be only on star-ed *‘\*’* variable on the LHS. The expressions are equivelant, regardless of brackets around LHS variables.

first, second, \*body, semifinal, last = list(range(20))

[first, second, \*body, semifinal, last] = list(range(20))

Replace group variable with *\*\_* or singular variable with *\_* to disregard that value.

Merge iterables using *star ‘\*’* into single list.

[\*list(range(5)), \*list(range(5, 10))]

Merge dictionaries using *‘\*\*’* - note that for any duplicate keys, the value on the RHS is kept

{\*\*dict1, \*\*dict2}

## Defining functions with ‘\*’ and ‘\*\*’

Positional arguments come before named arguments, and fixed length arguments must come before variable length arguments, giving us the order

def super\_function(num1, num2, \*args, callback=None, messages=[], \*\*kwargs):

1. Known positional arguments
2. *\*args*
3. Known named arguments (arguments after *\*args* are keyword only, whether they have default values or not)
4. *\*\*kwargs*

Both *args* and *kwargs* are optional when calling function, and default to *()* and *{}* respectively. *args* collects multiple items, as per their position in the function call (either multiple variables, an unpacked list, or both). The named arguments with default values *callback*, *messages*,are optional. *kwargs* is a dictionary, containing all pairs from unpacked dictionaries passed as arguments, as well as pairs corresponding to any named arguments passed to function which do not match named arguments in function definition.

Note that other names can be used in place of *args* and *kwargs* (although it is generally better to use them), but only one of each is allowed. (Also) note that if an unpacked dictionary pair has a key that matches a named argument, the corresponding value is supplied as that named argument, and if that argument has been supplied separately from said unpacked dictionary, there will be an error.

## Defining functions with ‘/’ and ‘\*’

def func(num1, num2, /)

def func(num1, num2, /, \*, n=4)

Any arguments before *‘/’* can only be used positionally. Any arguments after *‘\*’* can only be keyword arguments. Between them, either positional or keyword arguments can be used. Keyword arguments are only optional if they have a default value.

# Comprehensions

## nested comprehension

# [builtins](https://docs.python.org/3/library/functions.html)

range(*stop*) or range(*start, stop[, step=1]*)

The arguments to the range constructor must be integers

zip(*\*iterables*)

Make an iterator that aggregates elements from each of the iterables.

len(*s*)

Return length (number of items) of an object.

abs(*x*)

Return the absolute value of a number.

sum(*iterable, /, start=0*)

Sums *start* and the items of an iterable from left to right, and returns total.

max(*iterable \*[, key, default]*) *or* max(*arg1, arg2, \*args[, key]*)

min(*iterable \*[, key, default])* *or* min(*arg1, arg2, \*args[, key]*)

Return the smallest/largest item in an iterable or the smallest of two or more arguments

isinstance(*object, classinfo*)

Return True if the *object* argument is an instance of the *classinfo* argument.

issubclass(*class, classinfo*)

Return True if *class* is a subclass of *classinfo*. A class is considered a subclass of itself.

dir(*[object]*)

Without an argument, return the list of names in the current local scope. With an argument, return a list of valid attributes for that object.

all(*iterable*)

Return True if all elements of the iterable are True (or the iterable is empty).

any(*iterable*)

Return True if any of the elements in iterable is True. If the iterable is empty, return False.

ord(*c*)

Return integer representing Unicode of chararacter *c*.

chr(*i*)

Return the string representing character whose Unicode is integer *i*.

id(*object*)

Return identity of *object*; integer which is unique and constant for the lifetime of the object.

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

With one argument, return the type of *object*. With three arguments, return a new type object.

bin(*x*)

Convert an integer to a binary string prefixed with “*0b”*.

hex(*x*)

Convert an integer to a hex string prefixed with *“0x”*.

enumerate(*iterable, start=0*)

Return a enumerate object. Yields a tuple containing a count, and the next value of the iterable.

eval(*expression[, globals[, locals]]*)

The expression argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the globals and locals dictionaries as global and local namespace.

sorted(*iterable, \*, key=None, reverse=False*)

Return a new sorted list from the items in *iterable*. *key* specifies a function of one argument that extracts the comparison key for each element.

del

Deletion of a name removes the binding of that name from the local or global namespace.

next(*iterator[, default]*)

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

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

Return an iterator that applies function to every item of iterable, yielding the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

iter(*object[, sentinel]*)

Returns an iterator object. Without the second argument, *object* must be a collection object which supports iteration. If the *sentinel* argument is given, the object must be callable; the created iterator will call object with no arguments until value returned is equal to *sentinel*.

setattr(*object, name, value*)

Assign *value* to the attribute *name*, provided the object allows it. setattr(x, 'foobar', 123) is equivalent to x.foobar = 123

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

Return the value of the named attribute of the object. getattr(x, 'foobar') is equivalent to x.foobar

hasattr(*object, name*)

True if *name* is the string of the name of one of the objects attributes, False if not.

delattr(*object, name*)

*del attr(x, 'foobar')* is equivalent to *del x.foobar*

reversed(*seq*)

Return a reverse iterator.

round(*number[, ndigits]*)

Return number rounded to ndigits precision after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input.

callable(*object*)

Return True if the *object* appears callable, False if not.

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

Returns a complex number with the value *real* + *imag* \* j, or convert a string or number to a complex number.

breakpoint(*\*args, \*\*kws*)

This function drops you into the debugger at the call site.

filter(*function, iterable*)

Construct an iterator from those elements of *iterable* for which *function* returns True.

slice(*stop*) or slice(*start, stop[, step]*)

Return a slice object representing the set of indices specified by *range(start, stop, step)*. *values[slice(a,b,c)]* is equivalent to *values[a:b:c]*

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

Return a property attribute. *fget* is a function for getting an attribute value, *fset* is a function for setting an attribute value. *fdel* is a function for deleting an attribute value.

vars(*[object]*)

Return the *\_\_dict\_\_* attribute for a module, class, instance, or other object. Without argument, like *locals()*

classmethod()

staticmethod()

super()

# [os](https://docs.python.org/3/library/os.html)

name

environ

system(*command*)

Execute string *command* in a subshell. Implemented by calling C function system(). On Unix, returns the exit status of the process in the format for wait(). On Windows, return the value returned by system shell.

replace(*src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None*)

Rename the file or directory *src* to *dst*. Raises error if *dst* is a directory, if it is a file, said file is replaced.

scandir(*path=’.’*)

Returns an iterator corresponding to the entries in the directory given by *path*. In arbitrary order, and does not include *‘.’* or *‘..’*.

listdir(*path=’.’*)

Returns a list containing the names of the entries in the directory given by *path*. List is in arbitrary order, and does not include *‘.’ or ‘..’*.

walk(*top, topdown=True, onerror=None, followlinks=False*)

Generate the file names in a directory tree by wakling the tree, either top-down or bottom-up. For each directory in the tree rooted at directory *top*, yields a 3-tuple, (dirpath, dirnames, filenames).

stat(*path, \*, dir\_fd=None, follow\_symlinks=True*)

Get the status of a file or file descriptor – performs equivalent of a stat() system call on given *path*.

remove(*path, \*, dir\_fd=None*)

Remove (delete) the file *path*. Raises *IsADirectoryError* if *path* is a directory.

rmdir(*path, \*, dir\_fd=None*)

Remove (delete) the directory *path*.

readlink(*path, \*, dir\_fd=None*)

Return a string representing the path to which the symbolic link points.

link(*src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None, follow\_symlinks=True*)

Create a hard link pointing to *src* named *dst*.

symlink(*src, dst, target\_is\_directory=False, \*, dir\_fd=None*)

Create a symbolic link pointing to *src* named *dst*.

chown(*path, uid, gid, \*, dir\_fd=None, follow\_symlinks=True*)

Change the owner and group id of path to the numeric *uid* and *gid.* Leave unchanged for -1.

chmod(*path, mode, \*, dir\_fd=None, follow\_symlinks=True*)

Change the mode of *path* to the numeric *mode*.

chroot(*path*)

Change the root directory of the current process to *path*.

sync()

Force write of everything to disk.

getlogin()

getpid()

getpgrp()

getppid()

getuid()

# [os.path](https://docs.python.org/3/library/os.path.html)

realpath(*path*)

Return the canonical path of the specified filename, eliminating any symbolic links in the path.

abspath(*path*)

Return a normalized absolutized version of the pathname path.

isfile(*path*)

Return True if path is an existing regular file. This follows symbolic links, so both islink() and isfile() can be true for the same path.

isdir(*path*)

Return True if path is an existing directory. This follows symbolic links, so both islink() and isdir() can be true for the same path.

spitext(*path*)

Split the pathname path into a pair *(root, ext)* such that *root + ext == path*, and ext is empty or begins with a period and contains at most one period.

getmtime(*path*)

Return the time of last modification of path. The return value is a floating point number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.

getctime(*path*)

Return the system’s ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for path. The return value is a number giving the number of seconds since the epoch

dirname(*path*)

Return the directory name of pathname path.

basename(*path*)

Return the base name of pathname path.

getsize(*path*)

Return the size, in bytes, of path.

expanduser(*path*)

On Unix and Windows, return the argument with an initial component of ~ or ~user replaced by that user’s home directory.

# os.system

ttyname(*fd*)

# [sys](https://docs.python.org/3/library/sys.html)

argv

The list of arguments passed to a python script. *argv[0]* is the script name.

getsizeof(*object[, default]*)

Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

getrecursionlimit()

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack.

setrecursionlimit(*limit*)

Set the maximum depth of the Python interpreter stack to *limit*.

float\_info

A named tuple holding information about the float type.

path

A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.

exit(*[arg]*)

Exit Python. Implemented by raising *SystemExit* exception, meaning it will only exit the process when called from the main thread, and when the exception is not intercepted. Successful termination is considered to be *arg=0*, with other values being considered error codes.

flags

Status of command line flags

getdefaultencoding()

getfilesystemencoding()

getrefcount()

getsizeof()

platform

stdin

stdout

stderr

File objects for standard input, output and errors, function like text files returned by *open()*.

setswitchinterval()

settrace()

thread\_info

version

<…>

# [re](https://docs.python.org/3/library/re.html)

findall(*pattern, string, flags=0*)

Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

finditer(*pattern, string, flags=0*)

Return an iterator yielding match objects over all non-overlapping matches for the RE pattern in string. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

match(*pattern, string, flags=0*)

If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

search(*pattern, string, flags=0*)

Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding match object. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

sub(*pattern, repl, string, count=0, flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement *repl*. If the pattern isn’t found, string is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. Backreferences, such as *\6*, are replaced with the substring matched by group 6 in the pattern.

split(*pattern, string, maxsplit=0, flags=0*)

Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.

## Match Object

group(*[group1, ...]*)

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned). If a groupN argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group.

groups(*default=None*)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None.

groupdict(*default=None*)

Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None.

start(*[group]*)

end(*[group]*)

Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring). Return -1 if group exists but did not contribute to the match.

re

The regex objects whose match() or search() method produced this instance

string

The string passed to match() or search()

# [datetime](https://docs.python.org/3/library/datetime.html)

datetime(*year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, \*, fold=0*)

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass.

timedelta(*days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0*)

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

strftime(*format*)

Return a string representing the date, controlled by an explicit format string.

strptime(*date\_string, format*)

Return a datetime corresponding to date\_string, parsed according to format.

fromtimestamp()

fromisoformat()

combine()

# dateutil

dateutil.relativedelta.relativedelta()

dateutil.parser.parse(*timestr[, parserinfo=None, \*\*kwargs]*)

timestr String containing datetime

parserinfo object containing parameters for parser

kwargs:

# [dateparser](https://dateparser.readthedocs.io/en/latest/)

# [math](https://docs.python.org/3/library/math.html)

pi

exp()

expm1()

log()

log1p()

log2()

log10()

pow()

sqrt()

acos()

asin()

atan()

atan2()

sin()

cos()

tan()

degrees()

radians()

acosh()

asinh()

atanh()

cosh()

sinh()

tanh()

dist()

ceil()

floor()

comb()

copysign()

fabs()

factorial()

fmod()

fsum()

frexp()

gcd()

lcm()

isqrt()

modf()

nextafter()

perm()

prod()

remainder()

trunc()

ulp()

isclose()

isfinite()

isinf()

isnan()

erf()

# [random](https://docs.python.org/3/library/random.html)

seed(*a=None, version=2*)

Initialise rng. If *a* is *None*, seed is provided by OS, or if unavailable, the current time is used.

random()

Return a random float in the range *[0, 1)*.

uniform(*a, b*)

Return a randomly selected float in the range *[a, b)*.

randrange(*stop*) *or* randrange(*start, stop[, step]*)

Randomly selected element from *range(start, stop, step)*.

randint(*a, b*)

Random integer between *[a, b]*.

randbytes(*n*)

Get *n* random bytes.

sample(*population, k, \*, counts=None*)

Return a *k* sized list of elements chosen from *population* without replacement.

choice(*seq*)

Return a random element from the non-empty sequence *seq*.

choices(*population, weights=None, \*, cum\_weights=None, k=1*)

Return a *k* sized list of elements chosen from *population* with replacement.

shuffle(*x[, random]*)

Shuffle sequence *x* in-place.

getstate()

Return object capturing internal rng state.

setstate(*state*)

Restore rng state to that obtained from *getstate()*.

# [statistics](https://docs.python.org/3/library/statistics.html)

mean()

fmean()

geometric\_mean()

harmonic\_mean()

median()

mode()

multimode()

quantiles()

pstdev()

pvariance()

stdev()

variance()

# [itertools](https://docs.python.org/3/library/itertools.html)

repeat(*object[, times]*)

Make an iterator that returns object over and over again. Runs indefinitely unless the times argument is specified.

cycle(*iterable*)

Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely.

count(*start=0, step=1*)

Make an iterator that returns evenly spaced values starting with number start.

zip\_longest(*iterables, fillvalue=None*)

Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

permutations(*iterable, r=None*)

Return successive r-length permutations of elements in iterable. If r is not specified or is None, then r defaults to the length of the iterable. The permutation tuples are emitted in lexicographic ordering according to the order of the input iterable.

product(*\*iterables, repeat=1*)

Cartesian product of input iterables. Roughly equivalent to nested for-loops in a generator expression. For example, product(A, B) returns the same as ((x,y) for x in A for y in B). The nested loops cycle like an odometer with the rightmost element advancing on every iteration.

combinations(*iterable, r*) *or* combinations\_with\_replacement(iterable, r)

Return r length subsequences of elements from the input iterable. The combination tuples are emitted in lexicographic ordering according to the order of the input iterable. Elements are treated as unique based on their position, not on their value. With replacement: return r length subsequences of elements from the input iterable allowing individual elements to be repeated more than once.

tee(*iterable, n=*2)

Return n independent iterators from a single iterable.

islice(*iterable, stop*) *or*islice(*iterable, start, stop*[*, step*])

Make an iterator that returns selected elements from the iterable.

starmap(*function, iterable*)

Make an iterator that computes the function using arguments obtained from the iterable. The difference between map() and starmap() parallels the distinction between function(a,b) and function(\*c)

filterfalse(*predicate, iterable*)

Make an iterator that filters elements from iterable returning only those for which the predicate is False. If predicate is None, return the items that are false.

groupby(*iterable, key=None*)

Make an iterator that returns consecutive keys and groups from the iterable (which must be sorted by what will be the key). *key* is a function computing a key value for each element, if it is not given, the element itself is used as key. The group returned on each iteration is a grouper object, which can be converted into a list of the values of that group by passing it to *list()*. Example usage:

data = sorted(data, key=keyfunc)

for key, group in itertools.groupby(data, key=keyfunc):

print("key=(%s), group\_list=(%s)" % (key, list(group))

chain(*\*iterables*)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence.

accumulate(*iterable*[*, func, \*, initial=None*])

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional func argument). If func is supplied, it should be a function of two arguments.

# [time](https://docs.python.org/3/library/time.html)

time()

asctime()

ctime()

gmtime()

localtime()

mktime()

get\_clock\_info()

sleep()

strftime()

strptime()

thread\_time()

# [subprocess](https://docs.python.org/3/library/subprocess.html)

run()

check\_output()

poll()

wait()

communicate()

send\_signal()

terminate()

kill()

# [timeit](https://docs.python.org/3/library/timeit.html)

# [functools](https://docs.python.org/3/library/functools.html)

reduce(*function, iterable[, initializer]*)

Apply *function* of two arguments cumulatively to the items of *iterable*, LR, so as to reduce the iterable to a single value.

partial(*func, /, \*args, \*\*kw*)

Return a new partial object, which when called will behave like *func* called with the positional arguments *args* and keyword arguments *kw*.

partialmethod(*func, /, \*args, \*\*kw*)

Return a partialmethod descriptior, which behaves like *partial*(), except it is to be used as a method definition rather than directly called.

@cache(*user\_function*)

Lightweight unbounded function cache.

@cached\_property(*func*)

Transform a method of a class into a property whose value is computed once and then cached as a normal attribute for the life of the insance. Like *property()*, wit the addition of caching.

@lru\_cache(*user\_function*) or @lru\_cache(*maxsize=128, typed=False*)

Decorator to wrap a function with a memorizing callable that saves up to *maxsize* most recent calls.

@total\_ordering()

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. The class must define one of: *\_\_lt\_\_(), \_\_le\_\_(), \_\_gt\_\_(), or \_\_ge\_\_()*, in addition to *\_\_eq\_\_()*.

# [collections](https://docs.python.org/3/library/collections.html)

## Counter

Counter(*[iterable-or-mapping]*)

A counter is a dict subclass for counting hashable objects. It is a collection where elements are stored as dict keys, and counts are stored as dict values.

Counter.elements()

Return an iterator over elements, repeating each as many times as its count. Elements are returned in the order first encountered.

Counter.most\_common(*[n]*)

Return a list of the *n* most common elements and their counts from the most common to least. If *n* is omitted or None, all elements are returned.

Counter.subtract(*[iterable-or-mapping]*)

Elements are subtracted from an iterable or another mapping, but subtracts counts instead of replacing them.

Counter.update(*[iterable-or-mapping]*)

Elements are counted from an iterable or added-in from another mapping or counter. Like *dict.update()*, but adds counts instead of replacing them. *Iterable* is expected to be a sequence of elements (not *k-v* pairs)

## defaultdict

defaultdict(*default\_factor[, ...]*)

Return a dict like object (subclass of dict). If *default\_factor* is not None, it is called without arguments to provide a default value for the given *key*.

## deque

deque(*[iterable[, maxlen]]*)

Returns deque object, initialized LR with data from *iterable*. Deques are a generalization of stacks and queues.

deque.append(*x*)

Add *x* to RHS of deque

deque.appendleft(*x*)

Add *x* to LHS of deque

deque.clear()

Remove all elements from deque

deque.copy()

Create shallow copy of deque

deque.count(*x*)

Count number of deque elements equal to *x*

deque.extend(*iterable*)

Append elements from *iterable* to RHS of deque

deque.extendleft(*iterable*)

Append elements from *iterable* to LHS of deque in reversing order

deque.index(*x*[*, start*[*, stop*]])

Return the position of first *x* in the deque (between indexes *start* and *stop*)

deque.insert(*i, x*)

Insert *x* into the deque at position *i*

deque.pop()

Remove and return element from RHS, raise *IndexError* if empty

deque.popleft()

Remove and return element from LHS, raise *IndexError* if empty

deque.remove(*value*)

Remove first occurrence of *value*. If not found, raise *ValueError*

deque.reverse()

Reverse elements in-place

deque.rotate(*n=1*)

Rotate the deque *n* steps right. Rotate left for negative value.

## namedtuple

namedtuple()

## ordereddict

OrderedDict(*[items]*)

Return instance of dict subclass with methods for rearranging order.

popitem(*last=True*)

Removes and returns a *k-v* pair. Order is LIFO if *last* is True, or FIFO if it is False.

move\_to\_end(*key, last=True*)

Move an existing *key* to either end of ordered dict, right (end) if *last* is True, left (start) if it is False. Raises *KeyError* if *key* does not exist.

## chainmap

ChainMap(*\*maps*)

Group multiple dicts or other mappings together to create a single, updateable view. Order is that of a series of *dict.update()* calls, starting at the last mapping.

ChainMap.new\_child(*m=None*)

Return a new ChainMap containing a new map, followed by all of the maps in the current instance. If *m* is specified, it becomes the new map at the front of the list of mappings (an empty dict is used if None).

# [heapq](https://docs.python.org/3/library/heapq.html)

Note: heapsort is not a stable sort.

heappush(*heap, item*)

Push the value item onto the *heap*, maintaining the heap invariant

heappop(*heap*)

Pop and return the smallest item from the *heap*, maintaining the heap invariant. Raise *IndexError* if empty. Use *heap[0]* to access smallest element without removing it.

heappushpop(*heap, item*)

Push *item* on the heap, then pop and return the smallest item from *heap*.

heapify(*x*)

Transform list *x* into a heap.

heapreplace(*heap, item*)

Pop and return the smallest item from *heap*, and also push the new *item*. Raise *IndexError* if empty.

merge(*\*iterables, key=None, reverse=False*)

Merge multiple sorted inputs into a single sorted output, returning iterator over sorted values.

nlargest(*n, iterable, key=None*)

Return a list with the *n* largest elements from *iterable*.

nsmallest(*n, iterable, key=None*)

Return a list with the *n* smallest elements from *iterable*.

# [bisect](https://docs.python.org/3/library/bisect.html)

bisect\_left(*a, x, lo=0, hi=len(a)*)

Locate the insertion point for *x* in *a* to maintain sorted order. Parameters *lo* and *hi* may specify a subset of the list to be considered. The returned insertion point *i* partitions the array *a* into two halves so that   
*all(val < x for val in a[lo:i])* and *all(val >= x for val in a[i:hi])*

bisect\_right(*a, x, lo=0, hi=len(a)*)

bisect(*a, x, lo=0, hi=len(a)*)

As per *bisect\_left()*, but returns an insertion point which comes after any existing entries of *x* in *a*. The returned insertion *i* point partitions the array *a* so that:  
*all(val <= x for val in a[lo:i])* and *all(val > x for val in a[i:hi])*

insort\_left(*a, x, lo=0, hi=len(a)*)

Insert *x* in *a* in sorted order. Assuming *a* is sorted, equivalent to: *a.insert(bisect.bisect\_left(a, x, lo, hi), x)*

insort\_right(*a, x, lo=0, hi=len(a)*)

insort(*a, x, lo=0, hi=len(a)*)

As per *insort\_left()*, but insert *x* in *a* after any existing entries of *x*.

# [queue](https://docs.python.org/3/library/queue.html)

Queue(*maxsize=0*)

FIFO queue. Infinite for *maxsize=0*

LifoQueue(*maxsize=0*)

LIFO queue. Infinite for *maxsize=0*

PriorityQueue(*maxsize=0*)

Priority queue. Infinite for *maxsize=0*

*Queue elements are accessible through attribute*

queue.queue

*Queue Methods:*

qsize()

Approximate size of queue

empty()

True if queue is empty, False otherwise

full()

True if queue is full, False otherwise

put(*item, block=True, timeout=None*)

Put *item* into queue. If *block* is True and *timeout* is None, block until free spot is available. If *timeout* is given, block the given value of seconds before raising *Full* exception.

put\_nowait(*item*)

Equivalent to *put(item, False)*

get(*block=True, timeout=None*)

Remove and return an item from queue. If *block* is True and *timeout* is None, block until an item is available. If *timeout* is given, block the given number of seconds before raising *Empty* exception.

get\_nowait()

Equivalent to *get(False)*

task\_done()

Indicate that a formerly enqueued task is complete. Used by consumer threads. For each *get()*, a subsequent *task\_done()* tells queue processing is complete. If *join()* is blocking, resume when all items have been processed.

join()

Block until all items in the queue have been gotten and processed.

# [inspect](https://docs.python.org/3/library/inspect.html)

getmembers()

getmodule()

signature()

getclasstree()

ismodule()

isclass()

ismethod()

isfunction()

isgenerator()

isawaitable()

isbuiltin()

getmro(*cls*)

Return a tuples of the base classes of class *cls*, including *cls*, in method resolution order. No class appears more than once in this tuple. In most cases, *cls* will be the first element of the tuple.

getmembers(*sample, inspect.isclass*)

Get all classes present in a module

getmembers(*sample.X, inspect.isfunction*)

Get all methods in function

# [textwrap](https://docs.python.org/3/library/textwrap.html)

wrap(*text, width=70, \*\*kwargs*)

Wraps the single paragraph in *text* so every line is at most *width* characters long. Returns a list of lines, without final newlines.

fill(*text, width=70, \*\*kwargs*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph. Equivalent to *“\n”.join(wrap(text, ...))*

shorten(*text, width, \*\*kwargs*)

Collapse and truncate the given *text* to fit the given *width*. kwargs are those of TextWrapper (*placeholder, …*).

dedent(*text*)

Remove any common leading whitespace from every line in *text*.

indent(*text, prefix, predicate=None*)

Add *prefix* to the beginning of the selected lines in *text*.

# [BeautifulSoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/)

BeautifulSoup is a high level parser, built atop (and therefore slower than) lxml (which it requires). Any modified attributes are reflected in any subsequently generated HTML. Two *NavigableString* or *Tag* objects are equal (as per ==) if the represent exactly the same HTML or XML markup (Check whether variables refer to same object with *is*). The following Attributes/methods are applicable to most BeautifulSoup objects.

contents

The children of a tag as a list

children

The children of a tag as an iterator

descendants

All descendants (children, children of children, ect) as an iterator

string

If a tag has only a single child, a *NavigableString*, it is made available as *string* (which is otherwise None).

strings

Iterator of each string contained inside a tag

stripped\_strings

As per *strings*, but stripped of whitespace at beginning/end

parent

The tag containing the current element (the parent of a top-level tag is a *BeautifulSoup* object, the parent of which is None).

parents

Iterator over all of an elements parents

next\_sibling

previous\_sibling

Next/previous sibling tag to current with same parents

next\_siblings

previous\_siblings

Iterator for next/previous sibling tag to current with same parents

next\_element

previous\_element

Element parsed immediately before/after current tag

next\_elements

previous\_elements

Iterator for elements parsed immediately before/after current tag

find\_all(*name, attrs, recursive, string, limit, \*\*kwargs*)

Looks through a tags descendants and retrieves all descendants that match given filters. Calling a Tag object as if it were a function is equivalent to calling *find\_all()* for that object. Returns empty list for no results.   
Keyword arguments include: *id*, *href*, *class\_*  
To search for an element whose name is taken (ie: name), use: *attrs={‘name’: ‘value’}*.

find(*name, attrs, recursive, string, \*\*kwargs*)

Nearly equivalent to *find\_all()* with *limit=1* (returns result instance, instead of list of one result instance). Returns None for no results.

find\_parents()

find\_parent()

find\_next\_siblings()

find\_next\_sibling()

find\_previous\_siblings()

find\_previous\_sibling()

find\_all\_next()

find\_next()

find\_all\_previous()

find\_previous()

The *find\_all\_\** methods are akin to *find\_all()*, and the *find\_\** methods are akin to *find()*, only for searching different parts of the tree.

select()

Run a CSS selector against parsed document and return all matching elements. (Generally, this will be done with lxml)

append()

Add to a tags contents. Akin to *list.append()*

extend()

Add every element of a given list to a Tag (in order). Akin to *list.extend()*

new\_tag()

Create a new tag object (to be appended into document)

insert(*position, value*)

Add to a tags contents, at a specified numeric position. Akin to *list.insert()*

insert\_before()

insert\_after()

Insert tag or string immediately before/after something else in the parse tree

clear()

Remove the contents of a tag

extract()

Remove (and return) a tag or string from the tree

decompose()

Remove a tag from the tree (destroying it and its contents) (check the *disposed* attribute of the object)

replace\_with()

Replace a tag or string with value specified, returning replaced value

wrap()

Wrap an element in the specified tag, returning the wrapper

unwrap()

Replaces a tag with whatever is inside that tag. Returns tag that was replaced.

smooth()

Consolidate adjacent strings in the parse tree (can be necessary after modifying tree)

prettify(*formatter*)

Returns HTML representation of tree as formatted Unicode string, with separate line for each tag and string. Intended for visual representation of document – not for generating output.

encode(*formatter*)

get\_text()

Returns all text in a document or beneath a tag, as a single unicode string

sourceline

Line number in parsed document where tag was located

sourcepos

Position where start of tag was located within line in parsed document

copy()

diagnose()

## BeautifulSoup

Represents the parsed document as a whole. (Can be treated as a Tag in most respects).

BeautifulSoup(*markup='', features=None, builder=None, parse\_only=None, from\_encoding=None, exclude\_encodings=None, element\_classes=None, \*\*kwargs*)

## Tag

A Tag object corresponds to an XML or HTML tag in the document.

name

Name of the tag

attrs

Dictionary of tag attributes (can be accessed from object directly)

## NavigableString

Corresponds to a bit of text within a tag.

## Comment

A special type of *NavigableString*,

## SoupStrainer

## Encoding

# lxml

# [urllib](https://docs.python.org/3/library/urllib.html)

Used for low-level server communication – use *request* for high level tasks.

## [urllib.requests](https://docs.python.org/3/library/urllib.request.html#module-urllib.request)

urlopen(*url, data=None, [timeout,] \*, cafile=None, capath=None, cadefault=False, context=None*)

Open the URL *url*, (either a string or a Requests object). Returns an *HTTPResponse* instance upon successful connection.

## [HTTPResponse attributes](https://docs.python.org/3/library/http.client.html#http.client.HTTPResponse)

read()

readinto()

getheader()

getheaders()

fileno()

msg

version

url

headers

status

reason

debuglevel

closed

## [HTTPConnection attributes](https://docs.python.org/3/library/http.client.html#http.client.HTTPConnection)

request()

getresponse()

set\_debuglevel()

set\_tunnel()

connect()

close()

blocksize

putrequest()

putheader()

endheaders()

send()

## [urllib.error](https://docs.python.org/3/library/urllib.error.html#module-urllib.error)

URLError

HTTPError

ContentToShortError

## [urllib.parse](https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse)

## [urllib.robotparser](https://docs.python.org/3/library/urllib.robotparser.html#module-urllib.robotparser)

# [requests](https://requests.readthedocs.io/en/master/api/)

request(*method, url, \*\*kwargs*)

Constructs and sends a Request. Returns a Response object.

get(*url, params=None, \*\*kwargs*)

Send a Get request. Returns a Response object.

post(*url, data=None, json=None, \*\*kwargs*)

Send a Post request. Returns a Response object.

put(*url, data=None, \*\*kwargs*)

Send a Put request. Returns a Response object.

delete(*url, \*\*kwargs*)

Send a Delete request. Returns a Response object.

head(*url, \*\*kwargs*)

Send a Head request. Returns a Response object.

options(*url, \*\*kwargs*)

Send an Options request. Returns a Response object.

patch(*url, data=None, \*\*kwargs*)

Send a Patch request. Returns a Response object.

## [requests.Response attributes](https://docs.python-requests.org/en/latest/api/#requests.Response)

apparent\_encoding

close()

content

cookies

elapsed

encoding

history

is\_permanent\_redirect

is\_redirect

iter\_content()

iter\_lines()

json()

Returns the json-encoded content of response (if any) as a (possibly nested) dictionary

links

next

ok

raise\_for\_status()

raw

reason

request

status\_code

text

url

## urllib vs requests, getting an html page

from urllib.request import urlopen

html = urlopen(url)

html\_text = html.read()

import requests

r = requests.get(url)

html\_text = r.contents # Use r.text instead for html-as-unicode

# [operator](https://docs.python.org/3/library/operator.html)

add(*a, b*)

concat(*seq1, seq2*)

contains(*seq, obj*)

truediv(*a, b*)

floordiv(*a, b*)

and\_(*a, b*)

xor(*a, b*)

invert(*a*)

or\_(*a, b*)

pow(*a, b*)

is\_(*a, b*)

is\_not(*a, b*)

setitem(*obj, k, v*)

delitem(*obj, k*)

getitem(*obj, k*)

lshift(*a, b*)

mod(*a, b*)

mul(*a, b*)

matmul(*a, b*)

pos(*a*)

neg(*a*)

not\_(*a*)

rshift(*a, b*)

setitem(*seq, slice(i,j), values*)

delitem(*seq, slice(i,j)*)

getitem(*seq, slice(i,j))*

mod(*s, obj*)

sub(*a, b*)

truth(*obj*)

lt(*a, b*)

le(*a, b*)

eq(*a, b*)

ne(*a, b*)

ge(*a, b*)

gt(*a, b*)

attrgetter(*attr*) *or* attrgetter(*\*attrs*)

Return a callable object that fetches *attr* from its operand. If more than one attribute requested, returns tuple.

itemgetter(*item*) *or* itemgetter(*\*items*)

Return a callable object that fetches *item* from its operand, using *\_\_getitem\_\_()* method. Returns a tuple if multiple items given.

methodcaller(*name, /, \*args, \*\*kwargs*)

Return a callable object that calls the method *name* on its operand. If additional arguments / keywords are given, they are passed to method.

# [json](https://docs.python.org/3/library/json.html)

dumps(*obj, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw*)

Serialize *obj* to a JSON formatted str. Arguments as per *dump()*.

loads(*s, \*, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw*)

Deserialize *s* (str, bytes, or bytearray) containing JSON document to a python object. Arguments as per *load()*.

load(*fp, \*, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw*)

Deserialize file-like *fp* containing JSON document to python object.

dump(*obj, fp, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw*)

Serialize *obj* as JSON formatted stream to file-like object *fp*. Produces str (not bytes), *fp* must support str input.

Translation: JSON Python

object dict

array list

string str

number (int) int

number (real) float

true True

false False

null None

# [pickle](https://docs.python.org/3/library/pickle.html)

# [tempfile](https://docs.python.org/3/library/tempfile.html)

TemporaryFile()

TemporaryDirectory()

gettempdir()

mkstemp()

mkdtemp()

# [difflib](https://docs.python.org/3/library/difflib.html)

SequenceMatcher(*isjunk=None, a=’’, b=’’, autojunk=True*)

SequenceMatcher.find\_longest\_match(*alo=0, ahi=None, blo=0, bhi=None*)

SequenceMatcher.get\_opcodes()

SequenceMatcher.get\_grouped\_opcodes(*n=3*)

SequenceMatcher.ratio()

# [glob](https://docs.python.org/3/library/glob.html)

glob()

iglob()

escape()

# [site](https://docs.python.org/3/library/site.html)

getsitepackages()

getuserbase()

# [multiprocessing](https://docs.python.org/3/library/multiprocessing.html)

cpu\_count()

# [cProfile](https://docs.python.org/3/library/profile.html)

run()

# socket

# hashlib

sha256()

# [traceback](https://docs.python.org/3/library/traceback.html)

print\_stack()

print\_exc()

# http.client

HTTPConnection()

# [array](https://docs.python.org/3/tutorial/datastructures.html)

array(*typecode[, initalizer]*)

A new array whose items are restricted by typecode, and initalised from *initalizer*, which must be a list, bytes-like object, or iterable.

append(*x*)

Append a new item with value *x* to the end of the array

buffer\_info()

Return a tuple (*address, base*) giving memory address and length of the array buffer

byteswap()

Byteswap all items of the array

count(*x*)

Return the number of occurrences of *x* in the array

extend(*iterable*)

Append items from *iterable* to end of the array. If *iterable* is an array, it must have the same typecode.

frombytes(*s*)

Append items from the string, interpreting the string as an array of machine values.

fromfile(*f, n*)

Read n items from file object *f* and append them to the end of the array

fromlist(*list*)

Append items from *list*.

fromunicode(*s*)

Extend this array with data from the given Unicode string. Array must be type *‘u’*

index(*x*)

Return the smallest *i* such that *i* is the index of the first occurrence of *x* in array.

insert(*i, x*)

Insert a new item with value *x* in the array before position *i*

pop([*i*])

Remove the item with the index *i* from the array and returns it. *i* defaults to *-1*.

remove(*x*)

Remove the first occurrence of *x* from the array

reverse()

tobytes()

tofile(*f*)

tolist()

tounicode()

*typecodes:*

b signed char l signed long

B unsigned char L unsigned long

u wchar\_t q signed long long

h signed short Q unsigned long long

H unsigned short f float

i signed int d double

I unsigned int

# [set](https://docs.python.org/3/library/stdtypes.html#set)

set(*[iterable]*)

frozenset(*[iterable]*)

add(*elem*)

Add element *elem* to set.

update(*\*others*) *or* set |= other ...

Add elements from all *others* to set.

remove(*elem*)

Remove element *elem* from set. Raises *KeyError* if not found.

discard(*elem*)

Remove element *elem* from 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.

copy()

Return a shallow copy of the set.

isdisjoint(*other*)

Return True if the set has no elements in common with other.

issubset(other) *or* set <= other

Test whether every element in the set is in other.

issuperset(*other*) *or* set >= other

Test whether every element in other is in the set.

union(*\*others*) *or* set | other ...

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

intersection(*\*others*) *or* set & other ...

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

difference(*\*others*) *or* set – other ...

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

symmetric\_difference(*other*) *or* set ^ other

Return a new set with elements in either the set or other but not both.

intersection\_update(*\*others*) *or* set &= other ...

difference\_update(*\*others*) *or* set -= other ...

symmetric\_difference\_update(*\*others*) *or* set -= other ...

# [dictionary](https://docs.python.org/3/library/stdtypes.html#dict)

dict(*\*\*kwarg*) *or* dict(*mapping, \*\*kwarg*) *or* dict(*iterable, \*\*kwarg*)

clear()

Remove all items from dict.

copy()

Return shallow copy of the dict.

get(*key[, default=None]*)

Return value for *key* if key is in the dict, else *default*.

pop(*key[, default]*)

Return and remove value for key from dict. Raises exception if key not in dict and default not given.

popitem()

Remove and return a (*key, value*) pair. Pairs are returned in LIFO order.

items()

Return view of the dictionaries items (*key, value*) pairs.

keys()

Return view of the dictionaries keys.

values()

Return view of the dictionaries values.

setdefault(*key[, default=None]*)

If *key* is in the dict, return its value. If not, insert *key* with value of *default.*

update([*other*])

Update the dictionary with the key/value pairs from *other*, (overwrites existing keys).

d | other

Create new dict with merged keys and value of *d* and *other*. The values of *other* take priority if there are conflicts.

d |= other

Update the dict *d* with keys and values from *other*. Values from *other* take priority if there are conflicts.

# [string](https://docs.python.org/3/library/stdtypes.html#string-methods)

str(*object=’’*) or str(*object=b’’, encoding=’utf-8’, errors=’strict’*)

Return a string version of object. If neither *encoding*, nor *errors* is given, str(*object*) returns *object.\_\_str\_\_()*. If *encoding* or *errors* is given, *object* should be a bytes-like object.

join(*iterable*)

Return a string which is the concatenation of the strings in iterable, using string providing join as separator.

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

Return the number of non-overlapping occurrences of substring *sub*, in the range [*start, end*] (if given).

replace(*old, new[, count]*)

Return a copy of the string with all occurrence of substring *old* replaced by *new*. If *count* is given, only the first *count* occurrences are replaced.

Simulating rreplace()

new.join(*\_str.rsplit(old, count)*)

strip(*[chars]*)

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

find(*sub[, start[, end]]*)

Return the lowest index in the string where substring *sub* is found within range [*start, end*] (if given). If not found, return *-1*.

rfind(*sub[, start[, end]]*)

Return the highest index in the string where substring *sub* is found within range [*start, end*] (if given). If not found, return *-1*.

index(*sub[, start[, end]]*)

Like *find()*, but raise *ValueError* if substring not found.

rindex()

Like *rfind()*, but raise *ValueError* if substring not found.

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

Return a list of words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (resulting in at most maxsplit+1 elements). If *sep* is given, consecutive delimiters are deemed to delimit empty strings. If *sep* is None, runs of consecutive whitespace are regarded as seperator.

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.

capitalize()

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

center(*width[, fillchar]*)

Return centred in a string of length *width*. Padding is done using *fillchar* (default is space). Origional string is returned if *width* *<= len(s)*.

ljust(*width[, fillchar]*)

Return the string left-justified in a string of length *width*. Padding is done using *fillchar* (default is space). Original string is returned if *width <= len(s)*.

rjust(*width[, fillchar]*)

As per *ljust()*, except right-justified.

encode(*encoding=’utf8’, errors=’strict’*)

Return an encoded version of the string as bytes object.

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.

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.

rstrip(*[chars]*)

*strip()* on RHS of string.

lstrip(*[chars]*)

*strip()* on LHS of string.

lower()

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

upper()

Return a copy of the string with all cased characters converted to uppercase.

islower()

Return True if all cased characters are lower case (and there is at least one cased character).

isupper()

Return True if all cased characters are upper case (and there is at least one cased character).

swapcase()

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

title()

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

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.

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.

zfill(*width*)

Return a copy of the string left-filled with *‘0’* digits to make a string of length *width*. A leading sign is handled by inserting the padding after the sign character. The origional string is returned if *width <= len(s)*.

partition(*sep*)

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

rpartition(*sep*)

As per *partition()*, but split at the last occurrence of *sep*.

removeprefix(*prefix, /*)

If the string starts with the *prefix* string, return string with prefix removed, otherwise remove original string.

removesuffix(*suffix, /*)

If the string ends with the *suffix* string, return string with suffix removed, otherwise remove original string.

isprintable()

Return True if all characters in the string are printable, or the string is empty, otherwise False.

isnumeric()

Return True if all characters in the string are numeric characters, and there is at least one character, False otherwise.

isalpha()

Return True if all characters in the string are alphabetic and there is at least one character, False otherwise.

isalnum()

Return True if all characters in the string are alphanumeric and there is at least one character, False otherwise.

isascii()

Return True if the string is empty or all characters in the string are ASCII, False otherwise.

translate(*table*)

Return a copy of the string in which each character has been mapped through the given translation table.

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

(static) returns a translation table usable for *translate()*.

## string formatting

# [list](https://docs.python.org/3/library/stdtypes.html#list)

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.

append(*x*)

Appends *x* to the end of the sequence. Equivalent to *s[len(s):len(s)] = [x]*.

pop(*[i]*)

Retrieves the item at *i*, and also removes it (Defaults to *-1*).

sort(*\*, key=None, reverse=False*)

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). This method sorts the sequence in place. It is a stable sort.

clear()

Remove all items from list. Equivalent to *del s[:]*

copy()

Create a shallow copy of the list. Equivalent to *s[:]*

extend(*t*)

Extends the list with the contents of *t*. Equivalent to *s[len(s):len(s)] = t*

insert(*i, x*)

Inserts *x* into list at index given by *i*. Equivalent to *s[i:i] = [x]*

remove(*x*)

Remove the first item from list where *s[i]* is equal to *x*.

reverse()

Reverses list in-place.

index(*x[, i[, j]]*)

Index of the first occurrence of *x* in list (at or after index *I* and before index *j*).

count(*x*)

Total number of occurrences of *x* in list.

[i for i, c in enumerate(\_vals) if c == n]

List of indexes of all occurrences of *n* in list *\_vals*.

# bytes

# bytearray

count()

removeprefix()

removesuffix()

decode()

find()

index()

join()

replace()

startswith()

endswith()

strip()

split()

# memoryview