Python Memoire

[Builtins 2](#__RefHeading___Toc2427_3851815450)

[os 4](#__RefHeading___Toc2429_3851815450)

[os.path 5](#__RefHeading___Toc2431_3851815450)

[sys 6](#__RefHeading___Toc2433_3851815450)

[re 7](#__RefHeading___Toc2435_3851815450)

[datetime 8](#__RefHeading___Toc2437_3851815450)

[math 8](#__RefHeading___Toc2439_3851815450)

[itertools 9](#__RefHeading___Toc2441_3851815450)

[time 10](#__RefHeading___Toc2443_3851815450)

[glob 10](#__RefHeading___Toc2445_3851815450)

[site 10](#__RefHeading___Toc2447_3851815450)

[subprocess 10](#__RefHeading___Toc2449_3851815450)

[functools 10](#__RefHeading___Toc2451_3851815450)

[collections 10](#__RefHeading___Toc2453_3851815450)

[inspect 10](#__RefHeading___Toc2455_3851815450)

[multiprocessing 11](#__RefHeading___Toc2457_3851815450)

[cProfile 11](#__RefHeading___Toc2459_3851815450)

[socket 11](#__RefHeading___Toc2461_3851815450)

[hashlib 11](#__RefHeading___Toc2463_3851815450)

[traceback 11](#__RefHeading___Toc2465_3851815450)

[http.client 11](#__RefHeading___Toc2467_3851815450)

[numpy 12](#__RefHeading___Toc2469_3851815450)

[numpy.linalg 12](#__RefHeading___Toc2471_3851815450)

[sympy 12](#__RefHeading___Toc2473_3851815450)

[sympy.solvers.solveset 12](#__RefHeading___Toc2475_3851815450)

[set 13](#__RefHeading___Toc2477_3851815450)

[dictionary 14](#__RefHeading___Toc2479_3851815450)

[string 15](#__RefHeading___Toc2481_3851815450)

[list 18](#__RefHeading___Toc2483_3851815450)

[bytes 19](#__RefHeading___Toc2485_3851815450)

[bytearray 19](#__RefHeading___Toc2487_3851815450)

[memoryview 19](#__RefHeading___Toc2489_3851815450)

# builtins:

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*])

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

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

Return the smallest 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.

# os:

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.

# os.path:

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.

# sys

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.

# re

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

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.

# math

pi

ceil()

floor()

# random

random.sample()

# statistics

median()

# itertools

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. The key is a function computing a key value for each element.

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

time()

ctime()

# glob

glob()

# site

getsitepackages()

# subprocess

run()

check\_output()

# functools

reduce()

# collections

Counter()

# inspect

getmodule()

# multiprocessing

cpu\_count()

# cProfile

run()

# socket

# hashlib

hashlib.sha256()

# traceback

traceback.print\_stack()

# http.client

http.client.HTTPConnection()

# numpy

matrix()

cumsum()

# numpy.linalg

inv()

pinv()

solve()

# sympy

Symbol()

# sympy.solvers.solveset

nonlinsolve()

# set

set([*iterable*])

add(*elem*)

Add element *elem* 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.

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

Add elements from all *others* to 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

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

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)*.

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()*.

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.

# 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