Python check if integer in range

In this post, we will write the program on Python to check if the input number is the best or not. The number is said to be a prime number if it can only be dealt by 1 and itself. For example, 13 is a prime number because it is only available to be dealt by 1 and 13, on the other hand, 12 is not a prime number because it is recognizable by 2, 4, 6 and the number itself. Check if the number is the best or not the A prime number is always positive, so we check it at the beginning of the program. We divide the input number by all numbers between 2 and (number – 1) to see if there are any positive dealers other than 1 and the number itself. If a dealer is found, we show that the number is not a prime number otherwise we show that the number is a prime number. We use a pause phrase in the loop to come out of the loop as soon as a positive dealer is found, as no further check is required. # Input from user number = int(input(Enter any number:)) # the prime number is always greater than 1, if number > 1 in the range(2, number): if (number % i) == 0: print(number, is not a prime number) break else: print(number, is a prime number) # if the given number is less than or equal to 1 # then it is not a prime number: Print(number, is not a prime number) Output: Related Posts: The following sections describe the standard types built into the interpreter. The most important completed types are numeric, sequences, mappings, categories, instances, and exceptions. Some collection lessons need to be converted. Methods that add, deassign, or rearrange their members and do not return a specific item will never restore the collection instance themselves, but nothing. Some actions are supported by multiple object types. in particular, practically all objects can be compared to equality, tested for boolean value, and converted to a string by using the (repr() function or a slightly different str() function). The latter function is implicitly used when an object is written with a output function. The boolean value of any object can be tested, it can be used in the if or when mode or operand of the logical actions below. By default, an object is considered true unless its class defines a __bool__() method that returns Untrue or a __len__() method that returns zero when called with an object. 1 Here are most of the built-in objects that are considered untrue: constants defined as untrue: None and Unreal. zero of any numeric types: 0, 0,0, 0j, decimal(0), Fraction(0, 1) blank sequences and collections: '', [], [], [], set(), range(0) Operations and built-in functions with boolean value always return 0 or Untrue untrue and otherwise mentioned or True, unless otherwise specified. (Important exception: Boolean actions or and always returns one of their operands.) These are boolean operations organized by rising priority: Result Notes x or y if x is false, then y, else x (1) x and y if x is false, then x, else y (2) not x if x if false, else False (3) Remarks: This is a short circuit operator, so it only evaluates the second argument if the first is false. This is a short circuit operator, so it only evaluates the second argument if the first is true. is no lower priority than non-boolean operators, so == b is not interpreted (a == b), and == no b is a syntax error. The python has eight reference operations. They all share the same priority (which is greater than boolean functions). Comparisons may be arbitrarily chained; for example, x < y <= z corresponds 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 untrue). This table summarizes comparison operations: Operation Meaning < is definitely less than <= smaller or equal to > absolutely greater than or equal to >= greater than or equal == equal to != equal to object credentials is not different types of object identifiers Different types of objects, except for different numeric types, are never compared to equals. The == operator is always specified, but some object types (for example, class objects) have the same as Operators <, <=, > and >= operators are defined only if they make sense. For example, they raise a TypeError exception when one of the arguments is a complex number. Non-identical instances of a class are usually compared to non-equal instances unless the category __eq__() method. Class instances cannot be prescribed in relation to other instances of the same class or other object types unless the class adequately defines methods __lt__(), __le__(), __gt__() and __ge__() (usually __lt__() and __eq__() are sufficient if you want the usual meanings of the reference operators). The operation of cannot be customized if the operators are not operators. they can also be applied to any two objects and never add an exception. Two other actions with the same syntactic priority are supported by types that are iterating or implement __contains__() method. There are three numeric types: integers, floating point numbers, and complex numbers. In addition, logical values are a subtype of integers. Integers have unlimited accuracy. Floating point numbers are usually executed using double C; information about the accuracy and internal representation of the floating point of the machine in which your program is used is sys.float_info. Complex numbers have an actual and ficttive section, which are each floating point. To extract these parts from the complex number z, use z.real and z.imag. (The standard library contains additional shares for numeric types. rational and decimal numbers. Decimal number if the accuracy of the floating point can be determined by the user.) configurable.) created with numeric literal or built-in functions and operators. Literals of an unadorned integer (including hem numbers, octal and binary numbers) produce integers. Numeric literals that contain decimal point or exponential yield floating points. When you add a j or J to a numeric literal, you get an imaginary number (a complex number with zero real parts) that you can add to an integer or float to get a complex number with actual and ficttive parts. Python fully supports mixed arithmetic: when a binary arithmetic operator has different numeric types, operand with a narrower type is widened to another, where the integer is narrower than the floating point, which is narrower than the complex. Comparing different types of numbers behaves as if the exact values of these numbers were being compared. 2 Constructors int(), float() and complex() can be used to produce certain types of numbers. All numerical types (except complex ones) support the following functions (for operational priorities, see Order of priority: Activity results Full documentation x + y x amounts - y x and y x*y*y x and y x*y* y product difference and sub-quantity of x and y x/y x and y x/y:: n (y) quotient x % y remaining x / y (2) -x negated +x x unchanged abs(x) absolute or equal to x abs() int(x) x converted to inte number (3)(6) int() float(x) x converted to float(6) float() complex(s), im with a real part, The fictional part. by default, zero. (6) complex() c.conjugate() conjugate from complex number c divmod(x, y) pair (x // y, x % y) (2) divmod() pow(x, y) x to power y (5) pow() x ** y x to power y (5) Notes: Also integer division. The result is an integer, although the result type may not be an integer. Result rounded always towards minus infinity: 1//2 is 0, (-1)//2 is -1, 1//(-2) is -1 and (-1)//(-2) is 0. Not complex numbers. Instead, convert to float using abs() vision if necessary. Conversion from a moving point to an integer may round or break in accordance with point C; see math.floor() and math.ceil() for well-defined conversions. float also accepts strings nan and inf with the optional prefix + or - no number (NaN) and positive or negative infinity. Python defines pow(0.0) and 0** as 0 1, as programming languages are common. Accepted numeric literals contain numbers 0-9 or any Unicode match (code points with Nd property). See list of code points with Nd property. All the numbers. Real types (int and float) also include the following features: For more information about numerical functions, see and cmath module. Bitwise operations only make sense The result of bit operations is calculated as if it were performed as a complement of two with an infinite number of character chips. Binary bit functionality priorities are all smaller than numerical functions and higher than comparisons. Unary operation ~ shares the same priority as other unassisted numerical functions (+ and -). This table lists bit-by-bit actions sorted into ascending priority: Operation Result Notes x | y bitwise or x and y (4) x ^ y bitwise exclusive or x and y (4) x & y bitwise and x and y (4) x << n x moved left n bit (1)(2) x >> n x moved correctly n bits (1)(3) ~x bits x inverted Notes: Negative rotations are illegal and cause ValueEror to be raised. The left shift with the n-bit corresponds to a multiplicing pow(2, n). The correct replacement with n bit corresponds to the floor division pow(2, n). Performing these calculations with at least one additional character extension section in a limited complement presentation (working bit width 1 + max(x.bit_length(), y.bit_length()) or more) is enough to get the same result as if there were an infinite number of character chips. The Int type executes the numbers. Integrated 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) <= abs(x) < 2**k. Equivalently, when abs(x) is small enough to have a correctly rounded logarithm, then k = 1 + int(log(abs(x), 2)). If x is zero, then x.bit_length() returns 0. Equivalent to: def bit_length(self): s = bin(self) # binary representation: bin(-37) --> '-0b100101' s = s.lstrip('-0b') # remove leading zeros and minus sign return len(s) # len('100101') --> 6 int.to_bytes(length, byteorder, *, signed=False)¶ Return 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() + 7) // 8 , byteorder='little') b'\xe8\x03' Integer is represented by length bytes. OverflowError is raised if the integer cannot be represented by the number of bytes provided. The byte order argument specifies the hyphen representing the integer. If the byte order is large, the most significant byte is at the beginning of the byte table. If the byte order is low, the most significant byte is at the end of the byte table. To request the host system in its original byte order, use as a hyphenate value The signed argument determines whether to use a replenishment of two represents an integer. If the signed is False and a negative integer is given, the overflow is increased. The default value for the signed one is Untrue. classmethod int.from_bytes(byteorder, *, signed=False)¶ Return the integer represented by the given byte array. >>> 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 Argument bytes must be either a byte-like object or a short byte. The byte order argument specifies the hyphen representing the integer. If the byte order is large, the most significant byte is at the beginning of the byte table. If the byte order is low, the most significant byte is at the end of the byte table. To request the original hyphen order of the host system, use sys.byteorder as the byteorder value. The signed argument indicates whether the duo's replenishment is used to represent an integer. int.as_integer_ratio()¶ Return an integer pair with the exact same ratio as the original integer and positive denominator. The integer ratio (integer) of integers is always an integer as an numerator and 1 as the denominator. The floating point type executes the numbers. With a real abstract basic class. float also has the following additional methods. float.as_integer_ratio()¶ Return an integer pair with the exact same ratio as the original floating point and positive denominator. Raise OverflowError infinities and ValueError on NaNs. float.is_integer()¶ Return true if the floating point instance is fixed and false otherwise: >>> (-2.0).is_integer() True >>> (3.2).is_integer() False Two methods support conversion to and from hexadecimal strings. Because Python floats are stored internally as binary numbers, converting a floating point to a decimal string or decimal string usually contains a minor rounding error. Instead, hexadecimal strings allow you to accurately display and define floating point numbers. This can be useful when debugging and in numerical work. float.hex()¶ Returns the floating point representation as a hexadecimal string. For finite floating points, this presentation always includes a leading 0x and a p and exponent after it. classmethod float.fromhex(s)¶ Class method to return the gradient represented by hexadecimal string s. Strings can contain spaces at the beginning and end. Note that float.hex() is an instance method, while float.fromhex() is a class method. The hexadecimal string is a format: [sign] ['0x'] integer ['. fraction] ['p' exponent], where the optional character can either + or , integer and fraction are the number of hexadecimal numbers and exponent is a decimal number with an optional optional Sign. The case is not significant and the number or fraction shall contain at least one hexadecimal number. This syntax is similar to the syntax defined in section 6.4.4.2 of the C99 standard and also the syntax used in Java 1.5. In particular, the output of the float.hex() file can be used as a hexadecimal number in the C or Java code, and the hexadecimal strings produced by the C %a-shaped character or Javan Double.toHexString are accepted with float.fromhex(). Note that the exponent is written with a decimal number rather than a hexadecimal number and that it provides power by telling 2 odds. For example, the hexadecimal string 0x3.a7p10 represents a floating point (3 + 10./16 + 7./16**2) * 2.0**10, or 3740.0: >>> float.fromhex ('0x3.a7p10') 3740.0 The application of the inverse conversion to 3740.0 gives a different hexade the same number: >>> float.hex(3740.0) '0x1.d380000000000000p+11' for numbers x and y, possibly different types, it is a requirement that hash(x) == hash(y) whenever x == y (see __hash__() method documentation for more information). Facilitates implementation and efficiency in a number of numerical types (including int, float, decimal number). Decimals and fractions. Fraction) The hash of python numerical types is based on a single mathematical function assigned to a rational number and thus applies to all instances of int and fractions. A fraction and all limited floating point and decimal occurrences. Decimal. Basically, this function is given with a reduction module P for fixed prime P. The value of the P is available to Python sys.hash_info. Details of CPython implementation: Currently used P = 2**31 - 1 for machines with 32-bit C-length and P = 2**61 - 1 for machines with 64-bit C-length. Here are the detailed rules: If x = m /n is a non-rational number and n is not distributed by P, specify the hash value (x) m* invmod(n, P) to %P, where invmod(n, P) gives the inverse n modulo P. If x = m / n is a non-rational number and n is disadarable by P (but m is not), n does not have an inverse modulo P and the above rule does not apply; In this case, set hash(x) to sys.hash_info.inf. If x = m / n is a negative rational number, set hash(x) to -hash(-x). If the resulting hashish is -1, replace it with -2. Certain 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 hashish have the same hash value.) The hash values of the complex z real and fictive parts are combined by calculating hash(z.real) + sys.hash_info.imag * hash(z.imag), reduced modulo 2**sys.hash_info.width width so that it is in the range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1)). If the result is -1, it is. -2. To clarify the above rules, here is an example of a Python code corresponding to the built-in hash, rational number, float or complex hashish: import sys, math def hash_fraction(m, n): Calculate the rational number m/n hashish. Assume that m and n are integers, n are positive. Corresponds to hashish (fractions). Fraction (m, n)). P = sys.hash_info.modulus # Remove common factors of P (Unnecessary, if m and n already koprime.) while m % P == n % P == 0: m, n = m // P, n // P if n % P == 0: hash_value = sys.hash_info.inf others: # Fermat's Little Theorem: pow(n, P-1, P) is 1, so # pow(n, P-2, P) gives inverse n modulo P. hash_value = (abs(m) % P) * pow(n, P - 2, P) % P if m < 0: hash_value = -hash_value if hash_value == -1: hash_value = -2 hash_value def hash_float(x): Calculate the floating point x's hash. 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): Calculate complex number z hashish. hash_value = hash_float(z) z.real) + sys.hash_info.imag * hash_float(z.imag) # make a signed reduction module 2**sys.hash_info.width M = 2**(sys.hash_info.width - 1) hash_value = (hash_value & (M -1)) - (hash_value & M) if hash_value == -1 : hash_value = -2 hash_value Python supports the concept of iteration over tanks. This will be done by two separate methods; They allow user-defined categories to support iteration. The sequences described below always support iteration methods. You must specify one method for container objects to provide iteration support: container.__iter__()¶ Restore iterator object. The object must support the iterator protocol described below. If the container supports different types of iteration, additional methods may be used to request iterators, in particular for the iteration type concerned. (An example of an object that supports multiple iteration shapes would be a wooden structure that supports both width-first and depth-first passages.) This method corresponds to the python tp_iter the structure of the type structure of the objects. Iterator objects must support two methods that together form an iterator protocol: iterator.__iter__()¶ Restore the iterator object itself. This is necessary in order to use both tanks and iterators in for and in sentences. This method corresponds to the python tp_iter the structure of the type structure of the objects. iterator.__next__()¶ Restore the next item from the store. If there are no other items, raise the StopIteration exception. This method corresponds to the tp_iternext the structure of the type structure of the python objects. Python defines multiple iterator objects iteration of common and specific sequence types, dictionaries, and other specialized forms. Certain types are not important besides implementing the iterator protocol. When the iterator __next__() method raises the StopIteration method, it shall continue to do so in subsequent calls. Implementations that do not comply with this feature are considered to have been violated. There are three sequence types: lists, many objects, and range objects. Other sequence types tailored to binary data and text string processing are described in separately described sections. The following table supports most sequence types, both convertable and unchanged. Collections.abc.Sequence ABC has been delivered to make it easier to implement these actions on custom sequence types. This table lists serial actions sorted in ascending priority. 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 the type and value constraints set by any s. In and non actions have the same priorities as comparison operations. + (concatenation) and * (playback) have the same priority as the corresponding numerical functions. 3 Operating result notes x s True if the s-section is equal to x, else False (1) x not in s False, if the s-section is equal to x, otherwise True (1) s + t s and t (6)(7) s*n or n*s corresponding to adding s to itself n times (2)(7) s[i] i, origin 0 (3) s[i :j] a slice of s i by j(3)(4) s[i:k] s-slice i to j by the length of stage k (3)(5) len(s) s min(s) of the smallest s.index(x[product; i[, j]]) index of the first occurrence of x as s (in index i or after and before index (j) (8) the total number of instances of s.count(x) x in sequences the same type also support comparisons. In particular, monarchs and catalogues are compared in dictionaries by comparing similar elements. This means that in a comparison, each element must be equal and two series must be of the same type and size. (For more information, see language reference comparisons.) Note: Although in general case, only simple isolation testing is used, some specialized sequences (such as str, bytes, and bytes) also use them for subordinate testing: Values below 0 are treated as 0 (in which case the empty series is of the same type as s). Note that items in series s are not copied; they are referred to several times. This often haunts new Python programmers; consider: >>> lists = [[]] * 3 >>> list [[], [], []] >>> lists[0].paste(3) >>> list [[3], [3], [3]] It has happened that [[]] is a one-part list containing an empty list, so all three elements of [[]] * 3 are references to this single blank list. Which edit element editing lists modify this single list. To create a list of different lists, follow these steps: >>> lists = [[] for i in range(3)] >>> lists[0].add(3) >>> lists[1].add(5) >>> lists[2].add(7) >>> list[1].more(5).) >>> lists [[3], [5], [7]] For more information, see How do I create a multidimensional list?. If i or j is negative, the index is proportional to the end of the series s: replaced by len(s) + i or len(s) + j. But note that -0 is still 0. S-slice from i to j is defined as a series of items with an index k with i <= k < j. If i or j is greater than len(s), use len(s). If i is omitted or None, use 0. If j is omitted or None, use len(s). If i is bigger than or equal to j, the slice is empty. The S slice from i to j in step k is defined as a series of items with an index of x = i + n*k with 0 <= n < (j-i)/k. In other words, the indices are i, i+k, i+2*k, i+3*k and so on, stopping when j is reached (but never j). When k is positive, i and j are reduced to lenses or lenses if they are larger. When k is negative, i and j decrease in length(s) to 1 if they are larger. If i or j are omitted or None, they become end values (determined by the character k of the mark). Note, k cannot be zero. If k is None, it will be treated like 1. Unchanging sequences together always lead to a new object. This means that building a sequence on a repeating chain has a quadrant runtime cost over the entire length of the series. To get linear runtime costs, you must switch to one of the options below: If you are merging str objects, you can create a list and use the end str.join() at the end or write to an io. An instance of StringIO and retrieve its value when it is complete, if you merge clean-up objects, you can use the clean-up.join() or io value accordingly. BytesIO, or you can locate with a bytearray object. Bytearray objects are convertible and are powerful overbooking mechanisms if they concatenate object, expand the list to other types instead, study the relevant class documentation Some sequence types (such as range) support only item sequences that follow specific models, and therefore do not support sequence concatenation or repetition. the index increases the ValueError value when x is not found in s. Not all implementations support the transmission of additional arguments (i) and (j). These arguments allow you to search for sub-sequences effectively. Passing on additional arguments is roughly equivalent to using the s[i:j].index(x) function only without copying any data, and when the returned index is relative to the beginning of the series and not to the beginning of the sector. The only action usually performed by unchanged sequence types and not implemented by the convertible sequence type, has the support of the built-in hash(). This support allows unchanged sequences, such as plural instances, to be used as dictation keys and stored in specified and frozen instances. appearances. The TypeError is represented by an unchanged sequence containing unchanged values. The actions in the following table are assigned to the types of sequence that you want to convert. Collections.abc.MutableSequence ABC has been provided to make it easier to implement these functions on custom sequence types. Table s contains an instance of the sequence you want to convert, t is any object you want to alter, and x is an arbitrary object that meets the type and value constraints set by any s (for example, bytearray only accepts integers that meet value limits of 0 <= x <= 255). Result of operation Comments s[i] = x point i of s is replaced by x s[i:j] = t s slice from i to j replaced by iterable t del s[i:j], which is the same as s[i:j] = [] s[i:j:k] = t is replaced by the elements of t(1) del s[i:j:k], end of series brackets x (same as s[len(s):len(s)] = [x]) s.clear() removes all items from s (same as del s[:]) (5) s.copy() creates a low copy of s (the same as s[:]) (5) s.extend(t) or s += t expand s with content t (mostly the same as s[len(s) :len(s)] = t) s *= n updates s and its contents are repeated n times (6) s.insert(i , x) adds the x to s in the directory i provided by the i (same as s[i:i] = [x]) s.pop([i]) and also removes it from s (2) s.remove(x) and removes the first item from the s[i] where s[i] is equal to x (3) s.reverse() translates stationary s-items (4) Remarks: t must be the same length as the slice to replace. The optional argument i is -1 by default, so by default, the last item is deleted and restored. remove() increases the ValueError value when x is not found in s. The inverse method () changes the financial sequence of the holding during the translation of a large sequence. If you want to remind users that it works with a side effect, it doesn't reverse order. clear() and copy() are included to bring them into line with the interfaces of complex containers that do not support slicing functions (such as dictation and set). copy() is not part of the collections.abc.MutableSequence ABC, but most concrete mutable sequence classes offer it. What's new in version 3.3: clear() and copy() methods. N is an integer or object that implements __index__(). Zero and negative values n clear sequence. Items in the series are not copied. they are referenced several times, as explained to s*n in common sequence functions. Lists are multifunctional sequences that are usually used to store collections of homogeneous objects (where the exact similarity varies depending on the application). Category list([iterable])¶ Lists can be generated in several ways: By using square brackets to describe an empty list: [] Using square brackets to separate items with commas: [a], [a, b, c] List use: [x x in iterable] With type constructor: or list (iterable) The constructor generates a list of items in the same order as the iterable items. iterable can be either a sequence, a iteration-enabled container, or an iterator object. If iterable is already a list, a copy similar to iterable[:] is made and returned. For example, list (abc) returns [a, b, c, and list ((1, 2, 3)) returns [1, 2, 3]. If no argument is entered, the constructor creates a new blank list[]. Many other functions also produce lists, including sorted () built-in. Lists perform all common and changed sequence actions. The lists also include the following additional method: sort(*, key=None, reverse=False)¶ This method sorts the list into place using only < cross-item comparisons. Exceptions are not muted - if the comparison operations fail, the entire sort operation fails (and the list is likely to remain in partially modified mode). Sorts() accepts two arguments that can only be moved with a keyword (keyword arguments only): the key specifies a single-argument function 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 counted once and then used throughout the sorting process. The Default None value means that list items are sorted directly without calculating a separate key value. Functools.cmp_to_key() add-in is available to convert a 2.x-style cmp function to a key action. inverse is a boolean value. If true, the items in the list are sorted as if each comparison had been completed. This method modifies the economic order of the farm when sorting in large order. If you want to remind users that it works by side effect, it does not return a sorted order (use sorted() to explicitly request a new sorted list instance). The sorting method () method is guaranteed to be stable. Sorting is stable if it ensures that the same number of comparing elements are not changed . For more information about sorting examples and a short sort guide, see Sort. CPython implementation information: During list sorting, the effect of converting or even checking a list is undefined. Python C implementation makes the list appear blank for duration and increases the ValueError if it detects that the list has been converted during sorting. Multiples are unchanged sequences that are usually used to store collections of heterogeneous data (such as two plurals produced by built-in enumeration (). Plural is also used in cases where an unchanged set of homogeneous data is required (such as allowing recording in a series or dictation instance). class plural([iterable])¶ Plural can be formed in ways: To describe an empty plural with parentheses: () () monotonous plural name comma: a or (a) Separating items with commas: a, b, c or (a, b, c) Using a built-in plural() or plural (iterable) the constructor builds a plural with the same parts and in the same order as iterable objects. iterable can be either a sequence, a iteration-enabled container, or iterator object. If iterable is already plural, it will be returned unchanged. For example, a plural (abc) returns (a, b, c) and a plural ([1, 2, 3]) returns (1, 2, 3). If no argument is entered, the constructor creates a new blank plural (). Note that the speck makes a tuple, not parentheses. Parentheses are optional, except in an empty plural case or when they are needed to avoid syntactic ambiguities. For example, f(a, b, c) is a function call with three arguments, while f(a, b, c)) is a function call whose only argument is a three-dum sum. The pits perform all common sequence functions. In heterogeneous data collections, where access by name is clearer than index usage, collections.namedtuple() may be a more suitable option than a simple plural object. The range type represents an unchanged number sequence and is commonly used to loops of loops for a certain period of time in loops. category range(stop)¶ category range(start, stop[, phase]) The arguments of the range constructor must be integers (either the built-in integer or any object that implements __index__ special method) If the phase argument is omitted, the default value is 1. If the initial arguments are omitted, that default value is 0. If the step is zero, ValueError is raised. In the positive step, the contents of the range r are determined by the formula r[i] = start + step*i, where i >= 0 and r[i] < stop. In a negative step, the contents of the range are still determined by the formula r[i] = start + step*i, but the restrictions are i >= 0 and r[i] > stop. The range object is empty if r[0] does not meet the value limit. Ranges support negative indexes, but are interpreted as an index at the end of a sequence defined by positive indexes. Absolute values greater than Sys.maxsize are allowed, but some features (such as len()) may increase OverflowError. Regional 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(0)) [] >>> list(range(1, 0)) [] Tracks perform all common sequence functions except concatenation and repetition (because range objects can only represent sequences that follow a strict formula and repetition chaining and concatenation usually violate this pattern). start¶ value (or 0 if parameter not provided) stops¶ stops¶ stop parameter stage value¶ The value of the step parameter (or 1 if no parameter is provided) The advantage of the range type over a regular list or plural is that the range object always takes up the same (small) amount of memory, regardless of the size of the range it represents (because it stores only the beginning, stop and step values, calculating individual items and intervals as needed). Range objects implement collections.abc.Sequence ABC and include features such as isolation tests, element index lookup, slicing, and support for negative indexes (see Sequence types - list, plural, 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] 18 Subjects in the test area in equality == and != compare them with the sequence wall. Therefore, two range objects are considered equal if they represent the same data series. (Note that two area objects that compare equally can have different start, stop, and step attributes, such as range(0) == range(2, 1, 3) or range(0, 3, 2) == range(0, 4, 2).) Changed in version 3.2: Implement series ABC. Supports slicing and negative indexes. Test int objects for membership in standard time instead of iterating all items. Changed in version 3.3: Specify '=' and'!=' to compare ranged objects based on the order of precedy they specify (instead of comparing the objects credentials). What's new in version 3.3: Start, stop, and step attributes. See also the Linspace recipe to show how to implement a lazy version of the collection suitable for floating point apps. Python text data is processed with str objects or strings. Strings are unchanged sequences of unicode code points. String literals are enclosed in many different ways: Individual quotation marks: allows embedded double quotation marks with quotation marks: allows embedded individuals in quotation marks. Triple quotation: "Three quotation marks", Three quotation marks Triple quoted strings can span multiple lines - all related spaces are included in the string literal. String literals that are part of a single expression and have only spaces between them are implicitly converted to a string litre. That is, (spam eggs) == spam. For more information about the different forms of string litres, including supported exhaust manifolds, and in the r(raw) prefix, which disables most exhaust sequence processing, see String and bytes. Strings can also be created from other objects in the str constructor. Because there is no separate character type, string indexing produces length strings 1. Therefore, for the non-empty string s[0] == s[0:1]. Multiple string type also does not exist, but or io. StringIO can be used to effectively generate strings from multiple fragments. Modified in version 3.3: Backward compatibility 2 series, u prefix is again a allowed string for literals. It does not affect the meaning of string litres and cannot be combined with the r prefix. class str(object='') ¶ class str(object=b'', encoding='utf-8', errors='strict') Return the string version of the object. If no object is provided, returns an empty string. Otherwise, the str() behavior depends on whether the encoding or errors are given as follows. If no encodings or errors are provided, the str(object) returns object.__str__(), which is an unofficial or fine printable string for the object. For string objects, this is the string itself. If the object does not __str__() method, the str() returns to the repr(object) method. If one or more encodings or errors are provided, the object must be a byte-like object (e.g. bytes or bytes). In this case, if the object is a bytes (or bytearray) object, str(bytes, encoding, errors) is matched bytes.decode(encoding, errors). Otherwise, the t?toting object on which the buffer object is based is obtained before calling bytes.decode(). For more information about buffer objects, see Binary Sequence Types bytes, bytes, memory view, and buffer protocol. Passing a T?t object to a str() function without encoding or error arguments is part of the first case where unofficial string prophecy is restored (see also the -b command-line option for Python). Example: >>> str(b'Zoot!') b'Zoot' For more information about the str class and its methods, see The Text Series Type - str and String Methods section below. To print formatted strings, see formatted string literals and format string syntax sections. See also the String Services section. Strings perform all common sequence operations and the additional methods described below. Strings also support two string formatting styles, one for high flexibility and customization (see str.format(), Format string syntax, and Custom String Formatting), and the other based on C-print style formatting, which handles a narrower range of types and is slightly harder to use correctly, but is often faster in the cases you work with (print-style string formatting). The Word Processing Services section of the standard library covers several other modules that offer a variety of text-related utilities (including regular expression support for the re module). str.big and capitalism()¶ Return a copy of the string with the first character in uppercase and the rest in lowercase letters. Changed in version 3.8: The first character is now written in header letters instead of uppercase. This means that characters such as digraphs have only a letter, and not the entire character. str.casefold()¶ Returns large and necklaceed copies of the string. Casefolded strings can be used to respond uninsorted. folding is similar to the väiijanne, but more aggressive because it is designed to eliminate all case differences in the string. For example, the German small letter ß corresponds to a ss. Since it is already small, a small () would do nothing about the ß; casefold() converts it to ss. The enclosure algorithm is described in section 3.13 of the Unicode standard. str.center(width[, fillchar])¶ Return centered on length width string. Fill is done with the specified fill law (the default is ASCII mode). The original string is returned if the width is less than or equal to the length(s). str.count(sub[, start[, end]])¶ Returns the number of non-overlapping instances of a substring in a substring in a range [start, end]. Optional arguments begin and end in the sector. str.encode(encoding=utf-8, errors=strict)¶ Return the encoded version of the string as a soother object. The default encoding is utf-8. errors can be used to determine another error handling template. The default error value is strict, which means that encoding errors raise the UnicodeError. Other possible values include skip, replace, xmlcharrefreplace, backslashreplace, and any other name registered with codecs.register_error(), see Error Handlers. For a list of possible encodings, see Standard encoding. By default, the error argument is not selected for best executions, but is used only for the first encoding error. Enable Python Development Mode or use the debugging build to check for errors. Changed in version 3.1: Support for added keyword arguments. Changed in version 3.9: Errors are now checked in development mode and debugging mode. str.endswith(suffix[, start[, end]])¶ Return true if the string ends with the specified suffix, otherwise restore your untrue. the suffixuffix can also be full of searched attachments. With an optional start, the test starts in that position. If the end is optional, stop comparing in this position. str.expandtabs(tabsize=8)¶ Return a copy of the string that replaces all tab characters with one or more spaces, based on the current column and the size of the given tab. Tabs display each tab-size character (the default is 8, where tab positions are in columns 0, 8, 16, and so on). To expand a string, the current column is set to zero and viewed one character at a time. If the character is a tab (\t), one or more spaces are added to the result until the current column is equal to the next tab stop. (The tab character itself is not copied.) If the character is a new row () or a return (\r), it is copied and the current column is reset to zero. All other characters are copied unchanged, and the current column is extended by one character, 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: sub can be found in the slice s[start:end]. Optional arguments begin and end in the sector. Return -1 if sub cannot be found. Note The Find() method should only be used if you want to know the location of the sub. To check whether substring is substring or not, use the in operator: >>> 'Py' in 'Python' True str.format(*args, **kwargs)¶ Perform a string formatting operation. The string in which this method is called can contain literal text or substitute fields separated by bindings '{}'. Each replacement field contains either the numerical index of the location argument or the name of the keyword argument. Returns a copy of a string in which each replacement field is replaced with the string value of the corresponding argument. >>> Sum 1 + 2 is {0}.format(1+2) 'Sum 1 + 2 is 3' See Format string syntax for description of the different formatting options that can be defined as formatting strings. Note When formatting a number (int, float, complex, decimal number). Decimals and subclasses) of type n (e.g. The {:n}.format(1234)), the function temporarily sets LC_CTYPE ranges to decimal_point and thousands_sep fields for LC_NUMERIC language code if they are not ASCII or longer than 1 byte, and the LC_NUMERIC language code is different from the LC_CTYPE language code. Other threads are affected by this temporary change. Changed in version 3.7: When you format a number with an n-type, the function LC_CTYPE the range to LC_NUMERIC in some cases. str.format_map(mapping)¶ Similar to str.format(**mapping), except that the mapping is used directly and does not copy to dictation. This is useful if, for example, mail merge is a dictation subcategory: >>> class default(dict): ... def __missing__(himself, key): ... recovery key ... >>> '{name} was born in {country}'.format_map(Default(name='Guido')) 'Guido was born in country' str.index(sub[, start[, end]])¶ Like find(), but increases ValueError when substring cannot be found. str.isalnum()¶ Return true if all characters in the string are alphanumeric and there is one or more characters, otherwise untrue. The character c is alphanumeric if any 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 alphabetical order and there is at least one character, otherwise untrue. Alphabetic characters are characters that are defined as Letter in the Unicode character database, that is, characters with the global category property Lm, Lt, Lu, Ll, or Lo. Note that this is different from the Alphabet property specified in unicode. str.isascii()¶ Return true if the string is empty or all characters in the string are ASCII, False otherwise. ASCII characters have code points in U+0000-U+007F. str.isdecimal()¶ Return true if all characters in the string are decimal numbers and there's at least one sign, False, by the way. Decimal characters are those that can be used to construct numbers in base number 10, such as U+0660, ARABIC-INDIC DIGIT ZERO. Formally, the decimal character is unicode global class Nd. str.isdigit()¶ Return true if all characters in the string are numbers and there is one or more characters, otherwise untrue. Numbers contain decimal characters and numbers that require special processing, such as compatibility index numbers. This covers numbers that cannot be used to make up numbers based on 10, such as Kharosthi numbers. Formally, a number is a character whose property is Numeric_Type=Number or Numeric_Type=Decimal. str.isidentifier()¶ Return true if the string is a valid id according to the language definition, section IDs, and keywords. Call the keyword.iskeyword() and test whether string is s is reserved id, such as def and class. Example: >>> on importing keywords iskeyword >>> 'hello'.isidentifier(), iskeyword('hello') True, False >>> 'def'.isidentifier(), iskeyword('def') True, True str.islower()¶ Return True, if all the letters in the string 4 are small and there is at least one letter, falsewise. str.isnumeric()¶ Return true if all characters in the string are numeric and there is at least one character, otherwise untrue. Numeric characters include numeric characters and all characters with Unicode numeric property, such as U+2155, VULGAR ONE FRACTION FIFTH. Formally, numeric characters are characters whose property is set to 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, otherwise untrue. Non-print characters are characters that are defined in the Unicode character database as Other or Separator, except for ASCII mode (0x20), which is considered printable. (Note that characters printed in this context are characters that should not be escaped when the string is restarted.) It does not affect the processing of strings written to sys.stdout or sys.stderr.) str.isspace()¶ Return true if the string contains only spaces and one or more characters, otherwise untrue. A character is a space if in a Unicode character database (see unicodedata), either its global category is Zs (Separator, Spacebar), or its bidirectional class is one of the WS, B, or S. str.istitle()¶ Return true if the string is a labeled string and has at least one character, for example, uppercase letters can only track letters and lowercase letters only. Return your untrue, by the way. str.isupper()¶ Return true if all uppercase letters 4 of the string are uppercase and there is at least one letter, in the case of the untrue. >>> 'BANANA'.isupper() True >>> 'banana'.isupper() False >>> 'baNana'.isupper() False >>> ' '.isupper() False str.join(iterable)¶ Return a string that is a string concatenate iterable. The TypeError is raised if there are non-string values in the iterer, including t?t objects. The delimiter between elements is a string that provides this method. str.ljust(width[, fillchar])¶ Return a string aligned in a length width. Fill is done with the specified fill law (the default is ASCII mode). The original string is returned if the width is less than or equal to the length(s). str.lower()¶ Return a copy of the string with all uppercase letters 4 are 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 that has been removed from the parent characters. A character argument is a string that specifies the set of characters to delete. If omitted or None, the character argument deletes spaces by default. Character argument is not a prefix. Instead, all combinations of its values have been stripped: >>> ' spacious '.lstrip() 'spacious ' >>> 'www.example.com'.lstrip('cmowz.') 'example.com' See the str.removeprefix() method that deletes one prefix string instead of all characters. For example: >>> 'Arthur: three!'.lstrip('Arthur: ') 'ee!' >>> 'Arthur: three!'. removeprefix('Arthur: ') 'three!' static str.makerans(x[, y[, z]])¶ This static method returns a translation table that can be used for str.translate(). If there is only one argument, it must be a dictionary that combines Unicode attributes (integers) or characters (strings of length 1) into Unicode, strings (arbitrary lengths), or None. Character keys are then converted to attributes. If there are two arguments, they must be strings of the same length, and in the resulting dictionary each x character is associated with the character in the same place in y. If there is a third argument, it must be a string whose characters must be combined as a result with None. str.partition(sep)¶ Share the string as soon as sep occurs and return the 3-plural that contains the separator, separator, and part after the separator. If the separator is not found, return the 3-plural that contains the string, followed by two empty strings. If the string begins with a prefix string, return the string[len(prefix):]. Otherwise, return a copy of the original string: >>> 'TestHook'.removeprefix('Test') 'Hook' >>> 'BaseTestCase'.removeprefix('Test') 'BaseTestCase' If the string ends in a suffix string and the suffix is not empty, return the string[:-len(suffix)]. Otherwise, return a copy of the original string: >>> 'MiscTests'.removesuffix('Tests') 'Misc' >>> 'TmpDirMixin' str.replace(old, new[, count])¶ Return a copy of the string with all instances replaced by a new one. If an optional number of arguments is entered, only the first quantity instances are overwritten. str.rfind(sub[, start[, end]])¶ Return the highest index of the string in which the substring is found so that the sub is contained in the s[start:end] string. Optional arguments begin and end in the sector. Return -1 if the sub fails. str.rindex(sub[, start[, end]])¶ Like rfind() but increases ValueError when substring cannot be found. str.rjust(width[, fillchar])¶ Return a string right-aligned with a length width. Fill is done with the specified fill law (the default is ASCII mode). The original string is returned if the width is less than or equal to the length(s). str.rpartition(sep)¶ Share the string in the last instance of sep and return a 3-plural that contains the separator, separator and section after the separator. If the separator is not found, return a 3-plural that contains two empty strings, followed by the string itself. str.rsplit(sep=None, maxsplit=-1)¶ Use the sep character as the separating string to return a list of words in a string. If maxsplit is given, maxsplit divisions at most are made, the most right-hand ones. If sep is not set or is none, any space string is a delimiter. With the exception of right-handing, rsplit() behaves like a split() described in detail below. str.rstrip([chars])¶ Return a copy of the string that has trailing characters removed. A character argument is a string that specifies the set of characters to delete. If omitted or None, the character argument deletes spaces by default. Character argument is not a suffixuffer; instead, all combinations of its values are stripped: >>> ' spacious '.rstrip() ' spacious' >>> 'mississippi'.rstrip('ipz') 'mississ' See str.removesuffix() method that deletes one suffix string instead of all characters. For example: >>> 'Monty Python'.rstrip(' Python') 'M' >>> 'Monty Python'.removesuffix(' Python') 'Monty' str.split(sep=None, maxsplit=-1)¶ Return the list of words in the string using sep.split as the delimiter string. If maxsplit is given, maxsplit shares will be made at most (he/she will not exceed maxsplit +1 elements in the list). If no maxsplit limit is specified or -1, the number of shares is not limited (all possible shares are made). If sep is entered, sequential delimiters are not grouped together and are considered to delimit empty strings (for example, '1,2'.split(',') returns ['1', '', '2']). The SEP argument can consist of several characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Sharing an empty string with the specified delimiter 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 the SEP is not specified or none, a different split algorithm shall apply: the splitting algorithm: a consecutive space is considered a single delimiter, and the result does not contain empty strings at the beginning or end if the string contains a space at the beginning or end. Therefore, sharing an empty string or a space-only string with the 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 the line list of rows that breaks the boundaries. Line breaks are not included in the resulting list unless keepends are provided and have not been provided. This method is divided into the following line borders. In particular, borders are a superset of universal new lines. Presentation Description Line Entry \r Line Break \r Line Break + Line Entry \v or \x0b Line Tab \f or \x0c Form Feed \x1c File Separator \x1d Group Separator \x1e Record Separator \x85 Next line (C1 control code) – Line separator – Paragraph Separator Changed in version 3.2: \v and \f added to the list of line boundaries. For example: >>> 'ab cde fg\rkl\r'.splitlines() ['ab c', '', '', 'de fg', 'kl'] >>> 'ab cde fg\rkl\r'.splitlines(keepends=True) ['ab c', '', '', 'de fg\r', 'kl\r'] Unlike split() when the separator string sep is given, this method returns an empty list to an empty string, and the breakage of the end line does not lead to an additional line: >>> .splitlines() [] >>> One line.splitlines() ['One line'] For comparison split(') gives: >>> ''.split('') [''] >>> 'Two rows '.split(') ['Two Lines', ''] str.startswith(prefix[, start[, end]])¶¶ Return true if the string starts with a prefix, otherwise return your untrue. With an optional start point, the test string starts in that position. If the string is optional, stop comparing the string in this position. str.strip([chars])¶ Return a copy of the string that has been removed from the signs and marks. A character argument is a string that specifies the set of characters to delete. If omitted or None, the character argument deletes spaces by default. Character argument is not a prefix or suffuffer. Instead, all combinations of its values have been stripped: >>> spacious .strip() spacious >>> www.example.com'.strip('cmowz.') 'example' of a 1990 string deletes 10 000 of the most sudden front end and end character arguments. Characters are removed from the front end until a string character that is not included in the character set is reached. Similar activity occurs at the end. Example: >>> comment_string = '#......'. Section 3.2.1 #32 ...'; >>> comment_string.strip ('.#! ') '3.2.1 Issue #32' str.swapcase()¶ Return a copy of the string in uppercase letters converted to lowercase and vice versa. Note that no keep it true that s.swapcase().swapcase() == s. str.title()¶ Restore Restore A version of the string in which the words start with a capital letter and the rest of the characters are lowercase. For example: >>> 'Hello world'.title() 'Hello World' The algorithm uses a simple language-independent definition of word as sequential letter groups. The definition works on many occasions, but it means that an apostrophe in contractions and possessive desires to form word boundaries, which may not be the desired result: >>> they are friends of Bill from the UK.title() They're Bill's Friends From The UK The way apostrophes are circulated can be built with regular expressions: >>> import re >>> defcase title(s): ... return re.sub(r[A-Za-z]+('[A-Za-z]+)?, ... lambda mo: mo.group(0).big and capitalising(), ... (s) ... >>> title case (they're friends of Bill's.) They're friends of Bill's. str.translate(table)¶ Return a copy of the string that each character is connected to through the given translation table. The table must be an object that performs indexing __getitem__(), usually through a mapping or sequence. Indexed, a Unicode forgery (integer), a table object can do one of the following: return a Unicode enlisting code or string and combine the character into one or more other characters. restore None to remove the character from the return string; or raise the Hakuerror exception to connect the character to itself. You can use the Str.maketrans() file to create a translation map of character and character mappings in different formats. See also the Codecs module for a more flexible approach to custom character mappings. str.upper()¶ Return a copy of the string with all uppercase capital letters. Note that s.upper().isupper() may be Untrue if s contains uninserted characters, or if the Unicode class of the resulting characters is not Lu (letter, uppercase), but, for example, Lt (letter, title). The upper case algorithm used is described in paragraph 3.13 of the Unicode standard. str.zfill(width)¶ To make a length width, return a copy of the remaining string filled with ASCII numbers '0'. The prefix (+/-) of the sign is treated by adding a fill after the sign and not earlier. The original string is returned if the width is less than or equal to the length(s). For example: >>> 42.zfill(5) '00042' >>> -42.zfill(5) '-0042' Note The formatting features described here show various quirks that lead to a number of common errors (such as not displaying plurals and dictions correctly). When you use a new formatted string literal, str.format() interface, or sample strings, you can avoid these errors. Each of these options offers its own trade-ins and simplicity, flexibility and/or ecksensity String objects have one unique built-in function: % of the operator (modulo). This is also called a string formatting or interpolation operator. % shape values entered (if shape string), % of conversion definitions in format are replaced by zero or more value elements. The effect is similar to sprintf() in C. If formatting requires one argument, values can be one non-plural object. 5 Otherwise, the values must be plural with exactly the number of items specified by the format string, or one mail merge object (for example, a dictionary). The transformation attribute contains two or more characters and must contain the following elements, which must occur in this order: the % character that represents the beginning of the attribute. A mail merge key (optional) consisting of parentheses in a series of characters (for example, (a name)). Conversion flags (optional) that affect the result of some conversion types. Minimum field width (optional). If it is set to *, the actual width is read as values in the next part of the plural, and the object you convert comes after the minimum field width and optional resolution. Accuracy (optional), given . (dot) and then accuracy. If set to *(asterisk), the actual resolution is read as values in the next part of the plural, and the value to be converted comes after the resolution. Length converter (optional). Type of conversion. When the correct argument is a dictionary (or other mail merge type), string formatting must have a parenthesive mail merge key in the dictionary that is inserted immediately after the % character. The mail merge key selects the value to format from the merge. For example: >>>('%(language)s is %(number)03d quotation types.' % ... {'language': Python, number: 2}) Python has 002 bid types. In this case* the attribute must not appear in the format (because they require sequential parameter lists). The characters on the conversion flag are: Flag Meaning '#' Value conversion uses an alternative form (if defined below). '0' The transformation is zero padded for numeric values. '-' The converted value is left adjusted (overrides the conversion '0' if both are given). ' ' (space) Zero is left before the positive number (or empty string) produced by the signed conversion. The symbol + or -) precedes the conversion (ignores the space flag). The length converter (h, l, or L) may exist, but it will be ignored because it is not necessary for Python — so%ld, for example, is identical to %d. Conversion types are: Conversion means notes d Signed integer decimal place. 'i' Signed integer decimal number. 'o' Signed octal value. 1) 'u' Outdated type – it is identical to d. (6) 'x' Signed hexadecimal (in lowercase). (2) 'X' Signed hexadecimal (iso letter). (2) 'e' Floating point in exponential form (lowercase). (3) 'E' floating point exponential shape (uppercase). (3) 'f' Floating point decimal number. (3) 'F' Floating point decimal number. (3) 'g' floating point format. Use shape, if the exponent is less than -4 or at least accuracy, the decimal format otherwise. (4) 'G' floating point format. Uses a large exponential format if the exponent is less than -4 or at least precision, decimal format. (4) 'c' A single character (accepts an integer or a string of one character). 'r' String (converts any Python object using repr()). (5) 's' String (converts any Python object using the str() statement). (5) 'a' String (converts any Python object with ascii(). (5) '%' No argument is converted, resulting in a % character. Remarks: An alternative form causes the front octal attribute ('0o') to be added before the first number. Depending on the alternative form, the front 0x or 0X (depending on whether the x or X format was used) is inserted before the first number. Because of an alternative form, the result always contains a decimal point, even if it is not followed by numbers. Accuracy determines the number of digits after the decimal point and the default value is 6. An alternate form causes the result to always contain a decimal point, and at the end zeros are not removed in the same way as they would otherwise be. Accuracy determines the number of significant digits before and after the decimal point and the default value is 6. If the resolution is N, the output is truncated to N characters. See PEP 237. Because Python strings have explicit lengths, %s conversions do not assume that \0 is the end of the string. Changed in version 3.1: %f conversion for numbers with an intent value exceeding 1e50 will no longer be replaced by %g conversion. The key built-in types for binary data processing are bytes and bytes. They are supported by a memory view that uses the buffer protocol to create memory for other binary objects without copying. The table module supports efficient storage of basic data types, such as 32-bit integers and IEEE754 dual-precision moving values. Objects in TT are unchanged single-tech sequences. Because many of the most important binary protocols are based on encoding ASCII text, the clean-up objects provide several methods that are valid only with ASCII-compliant data and are closely related to string objects in many other ways. Class study([source[, encoding[, errors]]])¶ First, the syntax of dense literals is much the same as the syntax of string literals, except that the b prefix is added: Individual quotation marks: b'allows still embedded double quotation marks: bsallii still embedded in individual quotation marks. Triple quotation: b'3 single quote', b3 double quotes Only ASCII characters are allowed in tate numbers in literary (regardless of the source code encoding indicated). All binary values with a value exceeding 127 shall be entered in the binary literaal values in an appropriate exhaust manifold Like string literals, literals in bytes can also use the r prefix to disable exhaust sequence processing. For more information about string and tatal numbers, see tataalien kirjaimellisia muotoja, mukaan lukien tuetut pakojaksot. 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 <= 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 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 whitespace being ignored. >>> bytes.fromhex('2Ef0 F1f2 ') b'.\xf0\xf1\xf2' Changed in version 3.7: bytes.fromhex() now skips all ASCII whitespace in the string, not just spaces. Käänteinen muunnosfunktio muuntaa titojen objektin heksadesimaaliedusttaan. heksa([sep[, bytes_per_sep]])¶ Palauta merkkijono-objekti, joka sisältää kaksi heksadesimaalilukua kutakin esiintymän tavua vastaan. >>> b'\xf0\xf1\xf2'.hex() 'f0f1f2' Jos haluat helpottaa kuusiomerkkijonon lukemista, voit määrittää tulosteeseen sisällytettävän yhden merkin erottimen sep-parametrin. Oletusarvon mukaan kunkin tavun välillä. Toinen valinnainen bytes_per_sep ohjaa välistystä. Positiiviset arvot laskevat erottimen sijainnin oikealta, negatiiviset arvot vasemmalta. >>> = b'\xf0\xf1\xf2' >>> value.hex('-') 'f0-f1-f2' >>> value.hex('_', 2) 'f0_f1f2' >>> b'UUDDLRLRAB'.hex(' ', -4) '55554444 4c524c52 4142' Muutettu versiossa 3.8: ti.hex() tukee nyt valinnaisia sep- ja bytes_per_sep-muuttujia erottimien asettamiseksi heksalähtöön. Koska tavuobjektit ovat kokonaislukusarjoja (jotka ovat kuin monikko), tavuobjektille b[0] tulee kokonaisluku, kun taas b[0:1] on tavukohde, jonka pituus on 1. (Tämä eroaa tekstimerkkijonoista, joissa sekä indeksointi että viipalointi tuottavat pituuden 1) Tavujen esitys käyttää literaalimuotoa (b'...'), koska se on usein hyödyllisempi kuin esimerkiksi tavut([46, 46, 46]). Voit aina muuntaa tavuobjektin kokonaislukuluetteloksi käyttämällä Note for Python 2.x users: The Python 2.x series allowed various implicit conversions between 8-bit strings (the closest thing 2.x offers to a built-in binary data type) and Unicode strings. This was a backward compatibility workaround that explains the fact that Python originally supported only 8-bit text and Unicode text was a later addition. In Python 3.x, these implicit conversions are gone - conversions between 8-bit binary data and Unicode text must be explicit, and inteations and string objects always compare inequality. bytearray objects are the mumbing equivalent of byte objects. category bytearray([source[, encoding[, errors]]])¶ There is no separate literal syntax for Bytearray objects, Instead, they are always created by calling the constructor: Create an empty instance: bytearray() Create a zero-fill instance at the given length: bytearray(10) From iteration of integers: bytearray(range(20)) Copy existing binary data through buffer protocol: bytearray(b'Hi!') Because bytearray objects are rotten, they support mutable sequence functionality in addition to the common syllables and bytearray operations described in bytes and Bytearray Operations. See also built-in byte sticker. Because two hexadecimal numbers correspond exactly one byte, hexadecimal numbers are a commonly used format for describing binary data. Thus, the bytearray type has an additional class method for reading data in this format: classmethod fromhex(string)¶ This bytearray class method returns the bytearray object and extracts the given string object. The string must contain two hexadecimal numbers per byte, and the ASCII space is ignored. >>> bytearray.fromhex('2Ef0 F1f2') bytearray(b'.\xf0\xf1\xf2') Changed in version 3.7: bytearray.fromhex() now ignores all ASCII spaces in the string, not just spaces. A reverse transformation function exists to convert a bytearray object to a hexadecimalization object. hexa([sep[, bytes_per_sep]])¶ Return a string object containing two hexadecimal numbers for each byte of the instance. >>> bytearray(b'\xf0\xf1\xf2').hex() 'f0f1f2' Modified version 3.8: Similar to bytes.hex(), bytearray.hex() now supports optional sep and bytes_per_sep parameters to place separators between hexa-starting bytes. Because bytearray objects are integer sequences (which are like a list), an integer appears for bytearray b[0], while b[0:1] is a bytearray object with a length of 1. (This differs from text strings where both indexing and slicing produce length 1) The representation of Bytearray objects uses the literal format of bytearray (b'...')) because it is often more useful than, for example, bytearray([46, 46]). You can always convert a bytearray object to an integer list by using a list(b). Both bytes and bytearray objects support common sequence actions. They're not just interoperable. the same type, but with any object such as t?ti. Flexibility allows them to be mixed freely in action without causing errors. However, the return type of the result may depend on the operand order. Note Methods for bytes and bytearray objects do not accept strings as their argument, just as string methods do not accept bytes as their argument. For example, you must type: a = abc b = a.replace(a, f) and: a = babc b = a.replace(ba, bf) Some bytes and bytearray actions assume the use of ASCII-compliant binary formats, so they must be avoided when arbitrary binary data is used. These restrictions are set out below. Note Using these ASCII-based features to process binary data that is not stored in ASCII-based formats can lead to data corruption. The following methods for byte and byte objects can be used with arbitrary binary data. bytes.count(sub[, start[, end]])¶ bytearray.count(sub[, start[, end]])¶ Returns the subnumear of a range in the non-duplicate instance range [start, end]. Optional arguments begin and end in the sector. The subarray you are searching for can be any byte-like object or integer in ranges from 0 to 255. Changed in version 3.3: Also accept an integer between 0 and 255. If binary data starts with a prefix string, return t?ti[len(prefix):]. Otherwise, restore a copy of the original binary data: >>> b'TestHook'.removeprefix(b'Test') b'Hook' >>> b'BaseTestCase'.removeprefix(b'Test') b'BaseTestCase' Prefix can be any object similar to denses. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. If binary data ends in a suffuffror string and the suffy is not empty, restore the clean-up[:-len(suffuffer)]. Otherwise, return a copy of the original binary data: >>> b'MiscTests'.removesuffix(b'Tests') b'Misc' >>> b'TmpDirMixin'.removesuffix(b'Tests') b'TmpDirMixin' Suffix can be any byte-like item. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.decode(encoding=utf-8, errors=strict)¶ bytearray.decode(encoding=utf-8, errors=strict)¶ Return a string extracted from the given bytes. The default encoding is utf-8. errors can be issued to determine another error handling template. The default error value is strict, which means that encoding errors raise the UnicodeError. Other possible values include skip, replace, and any other codecs.register_error (), see Error Handlers. For a list of possible encodings, see Standard encoding. By default, an error argument is selected for best performances, but is used only for the first decoding error. Enable Python Development Mode or use the debugging build to check for errors. With. Passing an encoding argument to a str allows you to decod any object similar to a byte directly without having to make temporary bytes or bytearray objects. Changed in version 3.1: Added keyword argument support. Changed in version 3.9: Errors are now checked in development mode and debugging mode. bytes.endswith(suffix[, start[, end]])¶ bytearray.endswith(suffix[, start[, end]])¶ Return True if binary data ends with the specified suffix, otherwise restore your untrue. the suffuffuffle can also be full of searched attachments. With an optional start, the test starts in that position. If the end is optional, stop comparing in this position. Searched suffrites can be any form of. bytes.find(sub[, start[, end]])¶ bytearray.find(sub[, start[, end]])¶ Return the lowest index of data in the subfolder of the haircut so that the child value is included in s[start:end]. Optional arguments begin and end in the sector. Return -1 if sub cannot be found. The subarray you are searching for can be any byte-like object or integer in ranges from 0 to 255. Note The Find() method should only be used if you want to know the location of the sub. To check whether substring is substring or not, use the sub-operator: >>> b'Py' in b'Python' True Changed in version 3.3: Also accept an integer between 0 and 255 as the bottom. bytes.index(sub[, start[, end]])¶ bytearray.index(sub[, start[, end]])¶ Like find(), but increases valueerror when subordiment is not found. The subarray you are searching for can be any byte-like object or integer in ranges from 0 to 255. Changed in version 3.3: Also accept an integer between 0 and 255. bytes.join(iterable)¶ bytearray.join(iterable)¶ Restore bytes or bytearray object, which is a chaining of binary data sequences in iterable. The TypeError is raised if there are values in iterable that are not dim, including str objects. The delimiter between elements is the content of bytes or bytearray objects that provide this method. static bytes.maketrans(from, to)¶ static bytearray.maketrans(from, to)¶ This static method returns a translation table that can be used for bytes.translate(), which connects each string to the past in the same position on the item; both tins and other objects shall be tins and shall be of the same quality. bytes.partition(sep)¶ bytearray.partition(sep)¶ Share the sequence after the first occurrence of the SEP and return the 3-plural that contains the separator, separator itself, or the next part of its bytearray and separator. If the separator is not found, restore the 3-plural that contains a copy of the original series, followed by two empty bytes or bytearray objects. The separator you are searching for can be any object similar to t ti. bytes.replace(old, new[, count]¶ bytearray.replace(old, new[, count])¶ Restore from series all instances of the old subdillar that have been replaced by new ones. If an optional number of arguments is entered, only the first quantity instances are overwritten. The subar order you are searching for and replacing it can be any object similar to a density. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.rfind(sub[, start[, end]])¶ bytearray.rfind(sub[, start[, end]])¶ Return the highest index in the order in which the sub-sequence section is found so that the child is included in s[start:end]. Optional arguments begin and end in the sector. Return -1 if we fail. The subarray you are searching for can be any byte-like object or integer in ranges from 0 to 255. Changed in version 3.3: Also accept an integer between 0 and 255. bytes.rindex(sub[, start[, end]])¶ bytearray.rindex(sub[, start[, end]])¶ Like rfind() but increases ValueError when subsequence is not found. The subarray you are searching for can be any byte-like object or integer in ranges from 0 to 255. Changed in version 3.3: Also accept an integer between 0 and 255. bytes.rpartition(sep)¶ bytearray.rpartition(sep)¶ Share the sequence in the last instance of the SEP and return the 3-plural that contains the part in front of the separator, the separator itself or its copy of bytearray, and the section after the separator. If the separator is not found, return a 3-plural that contains two empty bytes or bytearray objects, followed by a copy of the original series. The separator you are searching for can be any object similar to t ti. bytes.startswith(prefix[, start[, end]])¶ bytearray.startswith(prefix[, start[, end]])¶ Return true if binary data starts with the specified prefix, otherwise restore your unreal. With an optional start, the test starts in that position. If the end is optional, stop comparing in this position. The prefix you are searching for (as a prefix) can be any object similar to t ti. bytes.translate(table, /, delete=b'')¶ bytearray.translate(table, /, delete=b'')¶ Restore a copy of the bytes or bytearray object from which all bytes in optional argument deletion are removed and the remaining bytes are mapped through a given translation table, which must be a 256-length byte object. You can use the bytes.maketrans() method to create a translation table. Set the table argument to None for translations that delete only characters: >>> b'read this short text'.translate(None, b'aeiou') b'rd ths shrt txt' Changed in version 3.6: deletion is now supported as a keyword argument. The following methods for bytes and bytearray objects have default behaviors that require the use of ASCII-compliant binary shapes, but can still be used with arbitrary binary data by passing appropriate arguments. that not all bytearray methods in this section and instead produces new objects. bytes.center(width[, fillbyte])¶ bytearray.center(width[, fillbyte])¶ Return a copy of the object centered on the length width. The fill is performed using the specified fill (the default is ASCII mode). The original series is returned to the T?etch objects if the width is less than or equal to the length(s). Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.ljust(width[, fillbyte])¶ bytearray.ljust(width[, fillbyte])¶ Return a copy of the object left aligned in length width order. The fill is performed using the specified fill (the default is ASCII mode). The original series is returned to the T?etch objects if the width is less than or equal to the length(s). Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.lstrip([chars]¶ bytearray.lstrip([chars]) ¶ Restore a copy of the series after deleting the specified benefit bytes. A character argument is a binary series that specifies the byte values to be removed - the name suggests that this method is usually used with ASCII characters. If omitted or None, the character argument deletes ASCII spaces by default. Character argument is not a prefix. Instead, all combinations of its values are deleted: >>> b' spacious .lstrip() b'spacious ' >>> b'www.example.com'.lstrip(b'cmowz.') b'example.com' The binary sequence of byte values to be deleted can be any byte-like object. For more information, see removeprefix() in a method that deletes one prefix string instead of all characters. For example: >>> b'Arthur: three!. lstrip(b'Arthur: ') b'ee!' >>> b'Arthur: three!. removeprefix(b'Arthur: ') b'three!' Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.rjust(width[, fillbyte])¶ bytearray.rjust(width[, fillbyte])¶ Return a copy of the right paragraph of the object aligned in length width order. The fill is performed using the specified fill (the default is ASCII mode). The original series is returned to the T?etch objects if the width is less than or equal to the length(s). Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.rsplit(sep=None, maxsplit=-1)¶ bytearray.rsplit(sep=None, maxsplit=-1)¶ Use the sep character as a separator string to share the binary order into sub-sequences of the same type. If maxsplit is given, maxsplit divisions at most are made, the most right-hand ones. If the SEP is not set to or none, the separator is a subordiform consisting exclusively of an ASCII space. With the exception of right-handing, rsplit() behaves like a split() described in Below. bytes.rstrip([chars]¶ bytearray.rstrip([chars])¶ Restore copy Sequence in which the specified post-tites have been deleted. A character argument is a binary series that specifies the byte values to be removed - the name suggests that this method is usually used with ASCII characters. If omitted or None, the character argument deletes ASCII spaces by default. Character argument is not a suffuffuffer; Instead, all combinations of its values are deleted: >>> b' spacious .rstrip() b spacious >>> b'mississippi'.rstrip(b'ipz') b'mississ' The binary sequence of byte values to be deleted can be any byte-like object. For more information, see Removesuffix() for a method that deletes one suffix string instead of all characters. For example: >>> b'Monty Python'.rstrip(b' Python') b'M' >>> b'Monty Python'.removesuffix(b' Python') b'Monty' Note The bytearray version of this method does not work - it always produces a new object even if no changes have been made. bytes.split(sep=None, maxsplit=-1)¶ bytearray.split(sep=None, maxsplit=-1)¶ Split the binary order into sub-sequences of the same type using the sep character as a separator string. If maxsplit is given and non-negative, maxsplit shares are made up to (hedgs + 1 elements are not more than in the list). If maxsplit is not set to -1, the number of shares is not limited (all possible shares are made). If sep is given, sequential delimiters are not grouped together and are considered to limit empty returns of subdious order (e.g. b'1,2'.split(b',') [b'1', b'', b'2']). The SEP argument can consist of a multibyte series (for example, b'1<>2<>3'.split(b'<>') returns [b'1', b'2', b'3']). Splitting a blank series with the specified separator returns [b''] or [bytearray(b'')] according to the type of object distributed. A SEP argument can be any object similar to a trick. 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 the sep is not specified or is none, another split algorithm is used: runs of consecutive ASCII spaces are considered as a single separator and the result does not contain empty strings at the beginning or end if the series has a leading or end space. Therefore, sharing an empty sequence or a series consisting solely of an ASCII space without 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])¶ Restore a copy of the sequence after deleting the specified front and post-bytes. A character argument is a binary series that specifies the byte values to be removed - the name suggests that this method is usually used with ASCII characters. If omitted or None, character argument deletes ASCII spaces by default. Sample Sample is not a prefix or a suffuffer; Instead, all combinations of its values are deleted: >>> b' spacious .strip() b'spacious >>> b'www.example.com'.strip(b'cmowz.') b'example' The binary order of byte values to be deleted can be any byte-like object. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. The following methods for bytes and bytearray objects use ASCII-compliant binary formats and should not be used in arbitrary binary data. Note that not all bytearray methods in this section work in place, but produce new objects. bytes.large and caliber()¶ bytearray.big()¶ Return a copy of the sequence in which each syllable is interpreted as an ASCII dress, and the first byte in uppercase and the rest in lowercase. Non-ASCII byte values are passed unchanged. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.expandtabs(tabsize=8)¶ bytearray.expandtabs(tabsize=8)¶ Restore a copy of a series where all ASCII tab characters are replaced by one or more ASCII spaces, based on the current column and the size of the given tab. Tab tabs display each tab test (the default is 8, where tab positions are in columns 0, 8, 16, and so on). To expand a series, the value in the current column is zero and the sequence is viewed one byte at a time. If the byte is an ASCII tab character (b'\t'), at least one space mark is added to the result until the current column is equal to the next tab stop. (The tab character itself is not copied.) If the current byte is an ASCII new row (b'') or line break (b'\r'), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column grows by one, regardless of how the byte value is displayed printed: >>> b'01\t0122\t0123\t01234'expand.expandtabs() b'01 0123 01234' >>> b'01\t0122\t0123\t01234'.expandtabs(4) b'01 0123 0123 01234' Note The bytearray version of this method does not work - it always produces a new object even if no changes have been made. bytes.isalnum()¶ bytearray.isalnum()¶ Return true if all bytes in the series are alphabetical ASCII or ASCII decimal numbers and the series is not empty, otherwise untrue. Alphabetical ASCII characters are the byte values in the series b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'. ASCII decimal places are byte values for series b'0123456789'. For example: >>> b'ABCabc1'.isalnum() True >>> b'ABC abc1'.isalnum() False bytes.isalpha()¶ bytearray.isalpha()¶ Return True if all the syllables in the series are alphabetical ASCII characters and the series is not empty, false by the way. The alphabetical ASCII character is the hyphen value of the series For example: >>> b'ABCabc'.isalpha() True >>> b'ABCabc1'.isalpha() False bytes.isascii()¶ bytearray.isascii()¶ Return True if the series is empty or all the syllables in the series are ASCII, Falsewise. ASCII acquaintances are in range 0-0x7F. bytes.isdigit()¶ bytearray.isdigit()¶ Return true if all bytes in the series are ASCII decimal numbers and the series is not empty, otherwise untrue. ASCII decimal places are byte values for series b'0123456789'. For example: >>> b'1234'.isdigit() True >>> b'1.23'.isdigit() False bytes.islower()¶ bytearray.islower()¶ Return True if the series contains at least one small ASCII character and no large ASCII characters, False by the way. For example: >>> b'hello world'.islower() True >>> b'Hello world'.islower() False small ASCII characters are a series of byte values b'abcdefghijklmnopqrstuvwxyz'. The large ASCII characters are the byte values of the series b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. bytes.isspace()¶ bytearray.isspace()¶ Return true if all bytes in the series are ASCII spaces and the order is not empty, otherwise untrue. ASCII spaces are a series of byte values b' \t\r\x0b\f' (space, tab, newline, transport return, vertical tab, form feed). bytes.istitle()¶ bytearray.istitle()¶ Return true if the series is an ASCII header case and the series is not empty, otherwise untrue. For more information about titlecase, see bytes.title(). For example: >>> b'Hello World'.istitle() True >>> b'Hello world'.istitle(False bytes.isupper()¶ bytearray.isupper()¶ Return True if the series contains at least one lower case letter with at least one alphabetical ASCII character and no small ASCII characters, false by the way. For example: >>> b'HELLO WORLD'.isupper() True >>> b'Hello world'.isupper() The wrong small ASCII characters are a series of byte values b'abcdefghijklmnopqrstuvwxyz'. The large ASCII characters are the byte values of the series b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. bytes.lower()¶ bytearray.lower()¶ Return a copy of the series in which all large ASCII characters are converted to the corresponding lowercase letters. For example: >>> b'Hello World'.lower() b'hello world' Lowercase ASCII characters are the byte values of the series b'abcdefghijklmnopqrstuvwxyz' The large ASCII characters are the byte values of the series b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.splitlines(keepends=False)¶ bytearray.splitlines(keepends=false)¶ Restore the list of binary order rows and trunch it at the boundaries of ASCII rows. This method uses a common newlines approach to line sharing. Line breaks are not included in the resulting list unless keepends are provided and have not been provided. Such as: b'ab cde fg\rkl\r'.splitlines() [b'ab c', b'', b'de fg', b'kl'] >>> b'ab cde fg\rkl\r'.splitlines(keepends=True) [b'ab c', b'', b'de fg\r', b'kl\r'] Unlike splitting() when separator string sep is given, this method returns an empty list for an empty string, and breaking the end line does not result in an extra line: >>> b.split(b') , b2 rows.split(b') ([b''], [b'Two rows' , b'']) >>> b.splitlines(), bOne line.splitlines() ([], [b'One line]) bytes.swapcase()¶ bytearray.swapcase()¶ Return a copy of the series in which all small ASCII characters are converted to the corresponding capital equivalent and vice versa. For example: >>> b'Hello World'.swapcase() b'hELLO wORLD' Small ascii characters are the byte values of the series b'abcdefghijklmnopqrstuvwxyz' The large ASCII characters are the byte values of the series b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Unlike str.swapcase(), bin.swapcase().swapcase() == binary versions binary. Case conversions are symmetrical in ASCII, although this usually does not apply to arbitrary Unicode code points. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.title()¶ bytearray.title()¶ Return the title version of the binary series, where the words start with a large ASCII dress and the rest of the characters are lowercase. There are no unconsordered byte values left. For example: >>> b'Hello world'.title() b'Hello World' Lowercase ASCII characters are the byte values of the series b'abcdefghijklmnopqrstuvwxyz' The large ASCII characters are the byte values of the series b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. All other byte values are uncons cocooned. The algorithm uses a simple language-independent definition of word as sequential groups of letters. The definition works on many occasions, but it means that the apostrophe in contractions and possessive desires creates word boundaries, which may not be the desired result: >>> bhe are friends of Bill from the UK.title() bThey'Re Bill's Friends From The UK The way apostrophes are circulated can be built with regular expressions: >>> import re >>> def title(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) ... >>> title case(bhe are friends of Bill's.) They're friends of Bill's. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.upper()¶ bytearray.upper()¶ Return a copy of the series in which all small ASCII characters are converted to matching uppercase letters. For example: >>> b'Hello World'.upper() b'HELLO WORLD' Lowercase ASCII characters are the byte values of the series b'abcdefghijklmnopqrstuvwxyz' The large ASCII characters are: values in b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Note The bytearray version of this method does not work in place - it always produces a new object even if no changes have been made. bytes.zfill(width)¶ bytearray.zfill(width)¶ To make a length width, return a copy of the remaining series with ASCII b'0' numbers. The prefix (b'+'/ b'-') is processed by adding a fill after and not before the sign. The original series of dense objects is restored if the width is less than or equal to the length(seq). For example: >>> b42.zfill(5) b'00042' >>> b-42.zfill(5) b'-0042' Note The bytearray version of this method does not work - it always produces a new item even if no changes have been made. Note The formatting features described here show various quirks that lead to a number of common errors (such as correctly displaying plurals and dictions). If the value you want to print can be plural or dictionary, wrap it in plural. Byte objects (bytes/bytes) have one unique built-in function: % of the operator (modulo). This is also called a sot formatting or interpolation operator. Given % shape values (where the shape is a dense object), % of the conversion definitions in the format are replaced by zero or more value elements. The effect is similar to sprintf() in C. If formatting requires one argument, values can be one non-plural object. 5 Otherwise, the values must be a plural with exactly the number of items specified by the formatting object, or one mail merge object (for example, a dictionary). The transformation attribute contains two or more characters and must contain the following elements, which must occur in this order: the % character that represents the beginning of the attribute. A mail merge key (optional) consisting of parentheses in a series of characters (for example, (a name)). Conversion flags (optional) that affect the result of some conversion types. Minimum field width (optional). If it is set to *, the actual width is read as values in the next part of the plural, and the object you convert comes after the minimum field width and optional resolution. Accuracy (optional), given . (dot) and then accuracy. If set to *(asterisk), the actual resolution is read as values in the next part of the plural, and the value to be converted comes after the resolution. Length converter (optional). Type of conversion. When the correct argument is a dictionary (or other mail merge type), the formatting of the tate object must have a mail merge key used in parentheses that is added to the dictionary immediately after the % character. The mail merge key selects the value to format from the merge. For example: >>>(b'%(language)s is %(number)03d quotation types. {b'language': bPython, bnumero: 2}) b'Python has a 002 citation In this case* the attribute must not appear in the format (because they require sequential parameter lists). The characters on the conversion flag are: Flag Meaning '#' Value conversion uses an alternative form (if defined below). '0' The transformation is zero padded for numeric values. '-' The converted value is left adjusted (overrides the conversion '0' if both are given). ' ' (space) Zero is left before the positive number (or empty string) produced by the signed conversion. The symbol + or -) precedes the conversion (ignores the space flag). The length converter (h, l, or L) may exist, but it will be ignored because it is not necessary for Python — so%ld, for example, is identical to %d. Conversion types are: Conversion means notes d Signed integer decimal place. 'i' Signed integer decimal number. 'o' Signed octal value. 1) 'u' Outdated type – it is identical to d. (8) 'x' Signed hexadecimal (small). (2) 'X' Signed hexadecimal (iso letter). (2) 'e' Floating point in exponential form (lowercase). (3) 'E' floating point exponential shape (uppercase). (3) 'f' Floating point decimal number. (3) 'F' Floating point decimal number. (3) 'g' floating point format. Uses lowercase letters in exponential format if the exponent is less than -4 or at least precision, decimal format. (4) 'G' floating point format. Uses a large exponential format if the exponent is less than -4 or at least precision, decimal format. (4) 'c' one byte (accepts integer or one-byte objects). 'b' (any object that follows the buffer protocol or is __bytes__()). (5) 's's' is an alias of the 'b' code and should only be used for python2/3 code criteria. (6) 'a' Ti's (converts any Python object using repr(obj).encodes ('ascii','backslashreplace)).' (5) 'r' 'r' is an alias of the 'a' code and should only be used for python2/3 basis. (7) '%' No argument is converted, resulting in a % character. Remarks: An alternative form causes the front octal attribute ('0o') to be added before the first number. Depending on the alternative form, the front 0x or 0X (depending on whether the x or X format was used) is inserted before the first number. Because of an alternative form, the result always contains a decimal point, even if it is not followed by numbers. Accuracy determines the number of digits after the decimal point and the default value is 6. An alternate form causes the result to always contain a decimal point, and at the end zeros are not removed in the same way as they would otherwise be. Accuracy determines the number of significant digits before and after the decimal point and the default value is 6. If the resolution is N, the output is truncated to N characters. b'%s' has expired but will not be removed During. b'%r' has expired but will not be removed during the 3.x series. See PEP 237. Note The bytearray version of this method does not work always produces a new object even if no changes have been made. See also PEP 461 - % adding formatting to bytes and bytearray memory view objects allows Python code to use without copying internal data from an object that supports the buffer protocol. Class Memory View(obj)¶ Creates a memory view that refers to an obj. obj that must support the buffer protocol. Built-in objects that support buffer protocol contain bytes and bytes. Memory view contains a concept element that is an atomic memory unit that is processed by the original object obj. Many simple types, such as bytes and bytes, have one byte element, but other types, such as array.matrix, can have larger elements. len(view) is as long as the enumeting length. If view.ndim = 0, the length is 1. If view.ndim = 1, the length is equal to the number of elements in the view. Larger measures are the same length as the presentation length of the nested list in the view. The Itemsize attribute has the number of for one element. Memory view supports retrieving and indexing its data. 1-D slicing leads to a sub-view: >>> v = memoryview(b'abcefg') >>> v[1] 98 >>> v[-1] 103 >>> v[1:4] <memory at= 0x7f3ddc9f4350=>>t ti(v[1:4]) b'bce' If the format is one of the original format attributes of the struct module, indexing with an integer or a plural of integers is also supported and returns one element of the correct type. 1-D memory views can be indexed by an integer or a plural of one integer. Multidimensional memory views can be indexed exactly by plurals of ndim integers, where ndim is the number of measures. Zero-dimensional memory views can be indexed with an empty plural. Here's an example that doesn't take the shape of a byte: >>> import matrix >>> a = array('l', [-1111111111, 22222222, -333333333, 44444444]) >>> m = memory view(a) >>> m[0] -11111111 >>> m[-1] 4444444444 >>> m[::2].tolist() [-11111111111, -33333333] If the underlying object needs to be written, the memory view supports a 1-D sector definition. Resizing is not allowed: >>> data = bytearray(b'abcefg') >>> v = memoryview(data) >>> v.readonly False >>> v[0] = ord(ord() b'z') >>> data bytearray(b'zbcefg') >>> v[1:4] = b'123' >>> data bytearray(b'z123fg') >>> v[2:3] = b'spam' Traceback (last call last): File <stdin> , line 1, <module>ValueError: memoryview assignment: lvalue and rvalue are different structures >>> v[2:6] = b'spam' >>> data bytearray(b'z1spam') One-dimensional memory views of formats B, b or c are also hashishable. Hash is defined as hash(m) == hash(m.tobytes()): >>> v = memoryview(b'abcefg') >>> == hash(b'abcefg') True >>> hash(v[2:2:2:4]) == hash(b'ce') True >>> hash(v[::-2]) == hash(b'abcefg'[::-2]) True Changed in version 3.3: Yksiulotteiset muistinäkymät voidaan nyt viipaloida. Yksiulotteiset muistinäkymät</module> </stdin> </memory> </memory> forms B, b or c are now hashable. Changed in version 3.5: Memory views can now be indexed by a plural of integers. there are several methods in the memory view: __eq__(exporter)¶ The memory view and the PEP 3118 exporter are equal if their shapes match and if all the corresponding values are equal when the operand formatting codes are interpreted by structural syntax. Subsets v and w of the design format strings currently supported by tolist(s) are equal, if v.tolist() == w.tolist(): >>> import table >>> 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 the design module does not support either format string, objects are always compared as unequal (although the shape strings and buffer content are identical): >>> from ctypes import BigEndianStructure, c_long >>> BEPoint(BigEndianStructure): ... _fields_ = [(x, c_long), (y, c_long)] ... >>> point = BEPoint(100, 200) >>> a = memoryview(point) >>> b = memoryview(point) >>> == point False >>> a == b False Note that , as with floating points, v is w does not mean v == w for memory view objects. Changed in version 3.3: Earlier versions compared raw memory that ignored the target format and the structure of the logical table. tobytes(order=None)¶ Reset buffer data to bytestring. This is equivalent to calling the t?t?t constructor for the memory view. >>> m = memoryview(babc) >>> m.tobytes() b'abc' >>> bytes(m) b'abc' For non-lonely arrays, the result is equal to the flattened listization in which all elements are converted to bytes. tobytes() supports all formatting strings, including those that are not in the schema module syntax. New in version 3.8: subscription may be {'C', 'F', 'A'}. When the order is C or F, the data in the original table is converted to a C or Fortran subscription. If the views converge, A returns an exact copy of the physical memory. In particular, the fortran order in memory is preserved. If the views do not converge, the data is first converted to C. order=None is the same as order='C'. hexa([sep[, bytes_per_sep]])¶ Return a string object containing two hexadecimal numbers for each byte in the buffer. >>> m = memoryview(babc) >>> m.hex() '616263' Modified in version 3.8: Similar to bytes.hex(), memoryview.hex() now supports optional sep and bytes_per_sep parameters to place separators between the tims for hex output. tolist()¶ Restore buffer elementtiluettelona. >>> muistinäkymä(b'abc).luettelo() [97, 98, 99] >>> tuontitaulukko >>> a = array.array('d', [1.1, 2.2, 3.3]) >>> m = memoryview(a) >>> m.tolist() [1.1, 2.2, 3.3] Muutettu Muutettu 3.3: tolist() now supports all single-character original shapes in design module syntax, as well as multidimensional presentations. toreadonly()¶ Restore the read-only version of the memoryview object. The original memory view object does not change. >>> m = memoryview(bytearray(b'abc)) >>> mm = m.toreadonly() >>> mm.tolist() [89, 98, 99] >>> mm[0] = 42 Traceback (last call): File , line 1, in TypeError: cannot edit read-only <stdin> <module> memory >>> m[0] = 43 >>> mm.tolist() [43, 98, 99] release()¶ Release buffer exposed to memory view object. Many objects take special action when view is taken (for example, bytearray would temporarily prohibit resizing); Therefore, calling release() is convenient to remove these restrictions (and release hanging resources) as soon as possible. After this method is called, all other actions in the view raise the valueeror (except for the publication itself() which can be called several times): >>> m = memoryview(b'abc') >>> m.release() >>> m[0] Traceback (last call last): File <stdin>, line 1, <module>ValueError: operation prohibited in released memory view object Context management protocol can be used for similar effect using phrase: >>> memoryview(b'abc') in m: ... m[0] ... 97 >>> m[0] Traceback (last call last): File <stdin>, row 1, <module>ValueError: prohibited operation on a released memory view object(format[, format])¶ Enter a memory view in a new format or format. the default value for the byte_length //new_itemsize], which means that the result view is 1-D. The return value is a new memory view, but the buffer itself is not copied. Supported casts are 1D -> C-sided and C-aligned -> 1D. The destination shape is limited to one of the original shapes of the element in the design syntax. One shape must be a byte shape (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 'l' >>> 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, >>> 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 >>></module> </stdin> </module> </stdin> </module> </stdin> </module> </stdin> </stdin> 48 Cast 1D/unsigned long or 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]] Modified in version 3.5: The source format is no longer limited when it is in byte view. There are also several number attributes: obj¶ Memory view underlying item: >>> b = bytearray(b'xyz') >>> m = memoryview(b) >>> m.obj on b True nbytes¶ nbytes == product(shape) * itemsize =len(m.tobytes()). This is the clean-up mode that the matrix would use for a converge presentation. 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, 16.5]] >>> len(y) 3 >>> y.nbytes 96 readonly¶ A bool indicating whether the memory is read only. format¶ String containing the shape (design module style) of each element in the view. A memory view can be created from exporters with arbitrary formatting strings, but some methods (e.g. list()) are limited to the original single-element formats. Modified version 3.3: Shape B is now processed according to the syntax of the construction module. This means that memoryview(b'abc')[0] == b'abc'[0] == 97. itemsize¶ Size of each element in memory view in titonk: >>>, struct >>> m = memoryview(array.array('H', [32000, 32001, 32002])) >>> m.itemsize 2 >>> m[0] 32000 >>> struct.calcsize('H') == m.itemsize True ndim¶ Integer indicating the dimensions of a multidimensional matrix represented by memory. shape¶ Plural integers that give the memory format as an array of N dimensions. Changed in version 3.3: Blank plural nothing when ndim = 0. Steps¶ Plural integers that are nism-length and give size to bytes so that you can use each element in each dimension of the table. Changed in version 3.3: Blank plural nothing when ndim = 0. suboffsets¶ Used internally for PIL-style arrays. A value is just information. c_contiguous¶ Bool, indicating if the memory is C-aligned. f_contiguous¶ Bool, indicating if the memory is fortran's converge. A side-sided¶ Bool indicating if the memory is converge. A bulk object is an unmanaged collection of separate hashable objects. Common uses include: Remove duplicates from a sequence and calculate mathematical functions such as intersection, join, difference, and symmetric differences. (For other containers, see built-in dictation, catalogue and plural plural and collection module.) Like other collections, sets support in x set, len(set), and x set. Because sets are an unsorted collection, they do not store the location or order in which the element is inserted. Therefore, the sets do not support indexing, slicing, or other sequence-like activities. Currently, there are two built-in bulk types, defined and frozen. The set type must be converted — content can be changed, for example, by using add() and remove(). Because it is measurable, it has no hash value and cannot be used as a dictionary key or as part of another. Frozenset type is unchanged and hashable - its contents cannot be changed after it is created; Therefore, it can be used as a dictionary key or as part of another set-up. Non-empty sets (no frozen sets) can be created by placing a composite densified element list of cuffs, such as : {'jack', 'sjoerd'}, in addition to the bulk constructor. The constructors for both categories work in the same way: class set([iterable]¶ class frozenset([iterable])¶ Restore a new set or frozen set item whose elements are taken from iterable. The elements of the set must be hashable. For innermost sets to be series sets, they must be frozenset objects. If iterable is not specified, a new blank set is returned. There are several ways to create sets: Use a comma-delimited list of elements in series: {'jack', 'sjoerd'} Use bulk understanding: {c for c in 'abracadabra' if c is not 'abc'} Use type constructor: set(), set('foobar'), set(['a', 'b', 'foo']) Sets and frozenset instances provide the following functions: len(s) Return the number of elements in series (cardinality). x in s Test x for membership s. x no s Test x non-membership s. isdisjoint(other)¶ Return True if the series does not share elements with others. Series are detached if and only if their intersection is an empty set. issubset(other)¶ set <= other Test whether everyone in the set element is other. set < test whether the series is the correct subset, that is, set <= other and set != other. issuperset(other)¶ set >= other Test if all element of another element is in the series. set > test whether the series is the correct superset of others, i.e. set >= other and set != other. union(*others)¶ set | Other | ... Restore a new set with elements from the series and all the others. cutting (*others)¶ set & other & ... Restore a new set with a set and all other common elements. difference(*other)¶ set - other - ... Restore a new set with elements that do not exist in others. symmetric_difference(other)¶ set ^ other Restore new set with elements either in a set or other, but not both. copy()¶ Restore a low copy of the series. Please note that non-union(), cut(), symmetric_difference(), issubset() and issuperset() methods other than accept all iterable methods as an argument. Inches Inches their operator-based counterparties require their arguments to be series. This excludes structures prone to errors, such as set ('abc') & 'cbs' in favour of a more readable set ('abc') crossing ('cbs'). Support for both the set and frozenset is set to determine comparisons. Two sets are equal if and only if each series contains all the elements (each is a subset of the other). A set is less than the second set if and only if the first set is the correct subset of the second set (is a subset, but not equal). The set is larger than the second series, if and only if the first series is the right superset of the second set (is a superset, but is not as large). Bulk instances are compared to frozen instances based on their members. For example, set('abc') == frozenset('abc') returns True and so does set('abc') on the set([frozenset('abc')]). Subset and equality comparisons are not generalised as an overall order function. For example, two non-non-floating series are not equal and are not subsets of each other, so all subsequent twists and turns are untrue:<b, a==b, or= a=>b. Because the sets specify only a partial order (subset relationships), the result of the list.sort() method is not specified for serial lists. Definition elements, such as vocabulary keys, must be shareable. Binary functions that mix bulk instances with frozensets return the type of first operand. For example: frozenset(ab) | set ('bc') returns an instance of frozenset. The following table lists functions that do not apply to unchanged frozenset instances: update(*others)¶ set |= other | ... Update the set by adding elements from everyone else. intersection_update(*other)¶ set &= other & ... Update the set and keep only that and all other elements. difference_update(*other)¶ set -= other | ... Update the set and remove the elements found in others. symmetric_difference_update(other)¶ set ^= other Update set and keep only the elements found in both series, but not both. add(elem)¶ Adds an element elem to the series. Remove the element elem from the series. Raises the KeyError key if elem is not included in the series. discard(elem)¶ Remove the element elem from the series if it exists. pop()¶ Remove and restore an arbitrary element from the series. Raises the KeyError key if the series is empty. clear()¶ Remove all elements from the series. Note that non-operator versions of the update(), intersection_update(), difference_update() and symmetric_difference_update() methods accept all iterable methods as an argument. Note that the elem __contains__(), remove() and discard() methods can be a set. To support the search for the corresponding frozenset, a temporary gesture is created. The mail merge object combines hashable values with arbitrary objects. Mappings mutation objects. There is currently only one standard definition type, vocabulary. (Other containers have a built-in list, bulk and plural categories, and collections </b,> </b,> The keys to the dictionary are almost arbitrary values. Non-distributed values, i.e. values that contain lists, dictions, or other convertible types (compared by value or object credentials), must not be used for keys. The numeric types used in keys follow the normal rules of numeric comparison: if two numbers are equal to (such as 1 and 1.0), they can be used among themselves to index the same dictionary entry. (Note, however, that because computers store floating point numbers in approximately numbers, it is usually not wise to use them as vocabulary keys.) Dictionaries can be created by placing a comma-delimited key list: value pairs inside supports, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'} or dictation converter. class dict(**kwarg)¶ class dict(mapping, **kwarg) class dict(iterable, **kwarg) Restore a new dictionary formatted from an optional location argument and possibly an empty set of keyword arguments. There are several ways to create dictionaries: Use a comma-delimited key list: value pairs inside supports: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'} Use dictation comprehension: {}, {x: x** for 2 x in range(10)} Use type constructor: dict(), dict([('foo', 100), ('bar', 200)]), dict(foo=100, bar=200) If no location argument is provided, an empty dictionary is created. If a location argument is provided and is a mail merge object, the dictionary is created with the same pairs of key values as the mail merge object. Otherwise, the location argument must be an iterable object. Each iterable must itself be iterable with exactly two objects. The first object in each item becomes the key to the new dictionary and the value corresponding to the second object. If the key occurs more than once, the last value of the key becomes the corresponding value for the new dictionary. If keyword arguments are provided, keyword arguments and their values are added to the dictionary created from the location argument. If the key you are inserting already exists, the keyword argument value replaces the value in the location argument. To illustrate, the following examples return a dictionary equal to {one: 1, two: 2, three: 3}: >>> a = dict(1=1, two=2, three=3) >>> b = {'one': 1, 'two': 2, 'three': 3} >>> c = dict(zip('one', two, three, [1, 2, 3])) >>> d = dictation([[(two, 2), (one, 1), (three, 3)]) >>> e = dictation({'three' : 3, one: 1, two: >>> f = dict({'one: 1 , 'three': 3}, two=2) >>> a == b == c == d == e == f True Providing keyword arguments, because only keys that are valid Python IDs work in the first example. Otherwise, any valid keys can be used. These are functions that are supported by vocabularies (and therefore also by custom should be supported): list(d) Restore list of all keys used in dictionary d. len(d) Restore Restore d. d[key] Use the key to reset the d-item. Raises the KeyError key if the key is not on the map. If a subclass in the dictation class defines __missing__() and the key does not exist, d[key] calls that method with the key key as the argument. The d[key] action then returns or raises what __missing__(key) call returns or raises. No other actions or methods are called __missing__(). If __missing__() is not specified, KeyError is raised. __missing__(s) shall be a method; it cannot be an instance variable: >>> class Counter(dict): ... def __missing__ (himself, key): ... return 0 >>> c = Counter() >>> c['red'] 0 >>> c['red'] += 1 >>> c['red'] 1 The example above shows part of the collection implementation. Counter. Collections.defaultdict uses a different __missing__ different methods. d[key] = Set d[key] to. del d[key] Remove d[key] from d. Raises the KeyError key if the key is not on the map. d Return with true if d has a key, otherwise Untrue. key is not d Similar to non-key d. iter(d) Return iterator over dictionary keys. This is an iter(d.keys()) shortcut. clear()¶ Remove all items from the dictionary. copy()¶ Return a low copy of the dictionary. classmethod fromkeys(iterable[, value])¶ Create a new dictionary with iterable keys and values. fromkeys() is a class method that returns a new dictionary. the default value is None. All values refer to only one instance, so it usually doesn't make sense for a value to be a convertable object, such as an empty list. If you want to get separate values, use dictation understanding instead. get(key[, default])¶ Reset the key value if the key is in the dictionary, otherwise the default value. If no default value is entered, its default value is None, so this method will never raise KeyError. items()¶ Restore a new view of the pairs of objects in the dictionary ((key, value). See instructions for view objects. keys()¶ Restore a new view of dictionary keys. See instructions for view objects. pop(key[, default])¶ If the key is in the dictionary, delete it and reset its value, reset the default setting. If no default value is entered and the key does not exist in the dictionary, KeyError starts. popitem()¶ Remove and restore (key, value) a couple of dictionary words. Pairs are returned in LIFO order. popitem() is useful when iterated destructively over a dictionary, as is often the case with set algorithms. If the dictionary is empty, calling popitem() raises the KeyError key. Modified in version 3.7: Lifo subscription is now guaranteed. In earlier versions, popitem() returns an arbitrary key/value pair. reverse(d) Return the inverse iterator over the dictionary keys. This is an inverse (d.keys()) shortcut. setdefault(key[, default])¶ If the key is in the dictionary, return its value. If not, add with a default value, and then reset the default value. default is None. update([other])¶ Update Update key/value pairs from other keys that are replacing existing keys. Return nothing. update() accepts either another dictionary object or iterable for key/value pairs (plural or other length iterables). If keyword arguments are specified, the dictionary is updated with these key/value pairs: d.update(red=1, blue=2). values()¶ Returns a new view of dictionary values. See instructions for view objects. A equality comparison between one dict.values() view and another always returns The Unre. This also applies when comparing the dict.values() with itself: >>> d = {'a': 1} >>> d.values() == d.values() False d | others Create a new dictionary with a mapped key and a value of d, and others, both of which must be dictionary. Other values take precedence over d and other wrench. d |= other Update Dictionary d with other keys and values, which can be either a mapping or iteration of key/value pairs. Other values take precedence over d and other wrench. Dictionaries compare in the same way if and only if they have the same (key, value) pairs (regardless of order). Subscription comparisons ('<', '<=', '>=', '>') increase TypeEror. Dictionaries maintain the order in which they are inserted. Note that updating the key does not affect the order. After deletion, the added keys are added to the end. >>> d = {one: 1, two: 2, three: 3, four: 4} >>> d {'one': 1, two: 2, three: 3, four: 4} >>> list(d) ['one', 'two', 'three', 'four'] >>> list(d.values()) [1, 2, 3, 4] >>> d [one] = 42 >>> d {'one': 42, two: 2, three: 3, four: 4} >>> del d[two] >>> d[two] = None >>> d {'one': 42, three: 3, four: 4, two: None} Changed in version 3.7: Dictionary order is guaranteed as insertion order. This activity was CPython's implementation detail 3.6. Dictionaries and dictionary views can be reversed. >>> d = {one: 1, two: 2, three: 3, four: 4} >>> d {'one': 1, two: 2, three: 3, four: 4} >>> list(reversed(d)) ['four', 'three', two, one] >>> list(reversed(d.values()) [4, 3, 2, 1] >>> list(reversed(d.items())) [(four, 4), (three, 3), (two, 2), (one, 1)] Changed in version 3.8: Dictionaries are now reversible. Objects that return Dict.keys(),dict.values() and dict.items() are view objects. They provide a dynamic view of dictionary entries, which means that when the dictionary changes, the view reflects these changes. Glossary views can be iterated to produce their data and support membership tests: len(dictview) Reset the number of entries in the dictionary. iter(dictview) Return the iterator over the keyes, values, or objects in the dictionary (presented as plurals of (key, value). Iterate keys and values This allows you to create (value, key) pairs zip(): pairs = zip(d.values(), d.keys()). Another way to create the same list is pairs = = (k) (k, v) d.items()]. Iteration of views during the insertion or deletion of entries in the dictionary may raise RuntimeError or fail to iteration all entries. Amended in version 3.7: The vocabulary order is guaranteed to be the order of addition. x in dictview Return True view if x is in keys, values, or items in the underlying dictionary (in the latter case, x must be a plural (key, value). reversed(dictview) Return the inverse iterator over the keys, values, or objects in the dictionary. The view is iterated in reverse order of the entrance. Changed in version 3.8: Dictionary views are now reversible. Key views are set because their entries are unique and shareable. If all values are hashable values so that (key, value) pairs are unique and shareable, the Items view is also defined. (Value views are not treated as set-like because entries are usually not unique.) In views such as set, all actions assigned to abstract base class collections.abc.Set are available (for example, ==, < or ^). Example of using dictionary view: >>> container = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500} >>> keys = astiat.keys() >>> values = dishes.values() >>> # iteration >>> n = 0 >>> val values: ... n += val >>> print(n) 504 >>> # keys and values are iterated in the same order (order of addition) >>>(keys) ['eggs', 'sausage', 'bacon', 'spam'] >>> list (values) [2, 1, 1, 500] >>> # view objects >>> are dynamic and reflect dictation changes in del containers ['eggs'] >>> del dishes['sausage'] >>> list(keys) ['bacon', 'spam'] >>> # set operations >>> keys & {'eggs', 'bacon', 'salad'} {'bacon'} >>> keys ^ {'sausage', 'juice'} {'juice', 'sausage', 'bacon', 'spam'} Python statement supports the concept of runtime context defined by the context manager. This is done by using a pair of methods that allows user-defined categories to specify the time-of-visit context that is entered before the sentence body is run and deleted when the sentence ends: contextmanager.__enter__()¶ Enter the time context and restore either this object or another object associated with the time context. The value returned by this method is bound by this context control to the passpric ID of the sentences. An example of context management that returns itself is a file object. File objects return to __enter__() to allow an open() expression to be used as a context expression in an expression. An example of context management returning a related object is an object returned by the decimal.localcontext() function. These managers the active decimal context as a copy of the original decimal context, and then return the copy. This allows you to make changes to the current decimal context clause in connection with the body without affecting the code outside the statement. contextmanager.__exit__(exc_type, exc_val, exc_tb)¶ Exit the running time context and return boolean boolean value indicate whether any exception has occurred should be muted. If an exception occurred while executing the statement, the arguments contain an exception type, value, and trace information. Otherwise, all three arguments are None. If you return an actual value from this method, the statement blocks the exception and continues to run with the statement immediately after the sentence. Otherwise, the exception will continue to spread after this method has been performed. Exceptions that occur during this method overwrite any exceptions that have occurred in the body of the statement. The exception transmitted should never be explicitly changed - instead, this method should return the wrong value indicating that the method has been successfully completed and does not want to prevent the exception presented. This allows the context management code to easily identify whether __exit__() method has actually failed. Python configures multiple contextual administrations to support easy thread synchronization, fast closing of files or other objects, and simpler handling of the active decimal context. Certain types are not specifically addressed after the context management protocol is implemented. See examples in the contextlib module. Python generators and contextlib.contextmanager decorator provide a convenient way to implement these protocols. If the generator function is decorated with contextlib.contextmanager décor, it returns a context manager who implements the necessary __enter__() and __exit__() methods, rather than an iterator produced by an uncoded generator function. Note that there is no specific location for these methods in the type structure of python objects in the python/C API. Extension types that want to configure these methods must provide them as normal Python-easy-to-use methods. Compared to the overheads of setting up a runtime context, the search for a single class dictionary is low. GenericAlias objects are created by subjuncting a class (usually a container), such as a list[int]. They are primarily intended for type notes. Typically, the order for container objects calls __getitem__(s) statement. However, an order for categories of some containers can call __class_getitem__ () entry in the category. The class __class_getitem__() should return the GenericAlias object. The GenericAlias object acts as a proxy for generic types and implements parameterized generic drugs - a specific generic instance that provides the types of storage elements. The type revealed by the user of the GenericAlias object can be used for types. GenericAlias and used for isinstance() inspections. It can also be used to create GenericAlias objects directly. T[X, Y, ...] Create a genericalias named GenericAlias that represents the type contains X, Y and other elements depending on the T used. For example, the function is waiting for a list of floating point elements: def average(values: list[float]) -> float: return sum(values) / len(values) Another example of object aggregation using dictation, a common type waiting for two type parameters representing the key type and the value type. In this example, the function waits for dictation with keys of type str and values of type int: def send_post_request(url: str, body: dict[str, int]) -> None: ... Builtin functions are stance() and issubclass() do not accept GenericAlias for their second argument: >>> isinstance([1, 2], list[str]) Traceback (last call last): File <stdin>, line 1, <module>TypeError: isinstance() argument 2 cannot be parameterized generic Python run time does not force type notes. This extends to common types and their type parameters. When you create an object from GenericAlias, the container elements were not checked according to their type. Esimerkiksi seuraavaa koodia ei suositella, mutta se suoritetaan virheettä: >>> t = list[str] >>> t([1, 2, 3]) [1, 2, 3] Lisäksi parametroidut geneeriset parametrit poistavat tyyppiparametrit objektin luomisen aikana: >>> t = list[str] >>> type(t) <class 'types.genericalias'=>>>> l = t() >>> type(l) <class 'list'=>Calling repr() tai str() yleisellä tyypillä näyttää parametroidun tyypin: >>> repr(list[int]) list[int] >>> str(list[int]) [int]' Geneeristen lääkkeiden __getitem__()-menetelmä aiheuttaa poikkeuksen virheiden, kuten dict[str][str]: >>> dict[str][str] Traceback (viimeisin kutsu viimeinen): File <stdin>, line 1, in <module>TypeError: Dict[str] tyyppimuuttujia ei kuitenkaan ole jäljellä , but such expressions apply when type variables are used. The index must have as many elements as the genericAlias object __args__. >>> TypeVar >>> Y = TypeVar ('Y') >>> dict[str, Y][int] dict[str, int] All parameterized parallel attributes implement read-only attributes. genericalias.__origin__¶ This property refers to an unprofitable global category: >>> list[int].__origin__ genericalias.__args__¶ This property is a plurality passed to the original __class_getitem__() class of the <class 'list'=>global store (possibly length 1): >>> dict[str, list[int]].__args__ (<class 'str'=>, list[int]) genericalias.__parameters__¶ This attribute is a lazily comical plurality (possibly blank) of unique type variables found in __args__: >>> Type Import TypeVar >>> T = TypeVar('T') >>> list[T].__parameters__ (~T,) The interpreter supports several other objects in objects. Most of these only one or two actions. The only special function of the module is the use of attributes: m.name a module where m is a module and the name uses the name specified in the m symbol table. Module attributes can be specified. (Note that</class> </class> </module> </stdin> </class> </class> </module> </stdin> </stdin> </stdin> the import description is not strictly a function of the module object; import foo does not require the existence of a module object called foo, but requires an (external) definition of a module called foo somewhere.) The specific attribute of each module is __dict__. This dictionary contains the symbol table for the module. Editing this dictionary changes the symbol table in the module, but direct definition of the __dict__ attribute is not possible (you can type m.__dict__['a'] = 1, which sets the value to m.a. 1, but you cannot enter m.__dict__ = {}). Editing and __dict__ is not recommended. Modules built into the interpreter are written as follows: <module 'sys'= (built-in)=>. If they are downloaded from a file, they are written as <module 'os'= from= '/usr/local/lib/pythonx.y/os.pyc'=>. See objects, values, and types, and their class definitions. Function objects are created by using function definitions. The only action of a function object is to call it func(list of arguments). Functional objects really have two flavors: built-in functions and user-defined actions. Both support the same action (to call the function), but the implementation is different, so different object types. For more information, see Function definitions. Methods are functions called using the attribute definition definition. There are two flavors: built-in methods (such as adding lists() and class instance methods. Built-in methods are described by the types that support them. If you use a method (a function defined in the category namespace) through an instance, you will receive a special object: a bound method (also called an instance method). When called, it adds a self-argument to the list of arguments. Bound methods have two specific read-only attributes: m.__self__ object in which the method operates, and m.__func__ the function that implements the method. Playing m(arg-1, arg-2, ..., arg-n) is fully equivalent to calling m.__func__(m.__self__, arg-1, arg-2, ..., arg-n). Method objects bound like function objects support arbitrary attributes. However, because method attributes are actually stored in the underlying function object (meth.__func__), it is prohibited to set method attributes for bound methods. If an attribute is attempted for the method, the AttributeError attribute is raised. To set a method attribute, you must explicitly set it for the underlying function object: >>> C: ... def method (itself): ... Pass... >>> c = C() >>> c.method.whoami = 'my name is method' # cannot set traceback (last call last): File <stdin>, line 1, <module>in AttributeError: 'method' object is not specified as 'whoami' >>> c.method.__func__.whoami = 'my name is method' >>> c.method.whoami 'my is method' Katso Standard type hierarchy lisätietoja. Toteutus käyttää koodiobjekteja</module> </stdin> </module> </module> </module> pseudo-compiled executable Python code, such as function body. They differ from function objects because they do not contain a reference to their global execution environment. The built-in compile() function returns code objects and can be extracted from function objects by using __code__ function. See also code module. A code object can be executed or evaluated by passing it (instead of a source string) to exec() or eval() built-in functions. For more information, see Standard type hierarchy. Type objects represent different object types. The object type is used by the built-in function type(). There are no special functions for the types. Standard module types specify the names of all completed standard types. The types are typed as:<class 'int'=>. This object is returned by functions that do not explicitly return a value. It doesn't support special operations. There is exactly one null object named None (built-in name). type (None)() produces the same singleton. It's written under the name None. Slicing uses this object in general (see Slices). It doesn't support special operations. There is exactly one ellipsis object called Ellipsis (built-in name). type(Ellipsis)() produces Ellipsis singleton. It is written as Ellipsis or .... This object is returned from comparisons and binary actions when prompted to work on types that they do not support. For more information, see Comparisons. There's exactly one unreal artifact. type(NotImplemented)() produces a singleton instance. It's written out. Boolean values are two standard objects, Unreal and True. They are used to represent logical values (although other values can also be considered unreal or truth). In numeric contexts (for example, when used as an aritmetic operator argument), they work according to integers 0 and 1. The built-in bool() function of a function can be used to convert any value to a boolean value if the value can be interpreted as a boolean value (see the boolean test above). They're written as untrue and true. Implementation adds multiple read-only attributes to multiple object types if they are relevant. Some of these are not reported by the dir() built-in function. object.__dict__¶ A dictionary or other mail merge object used to store an object (writeable) attribute. instance.__class__¶ The category to which the class instance belongs. class.__bases__¶ Plural of class object base categories. definition.__name__¶ The name of the class, function, method, graph, or generator instance. definition.__qualname__¶ The approved name of the class, function, method, graph, or generator instance. class.__mro__¶ This attribute is a plural of categories that must be considered when searching for basic categories during method resolution. class.mro()¶ This method can be </class> </class> metaclass to adjust the precision sequence of the method for its instances. It is called a category in an instant, and its result is __mro__. class.__subclasses__()¶ Each category has a list of weak references to its immediate subcategories. This method returns a list of all these references that are still alive. The list is in the order in which it was set up. Example: >>> int.__subclasses__() [<class 'bool'=>] Footnotes </class>