Integers, Vari
Functions, Met
Attributes, Da
Types, Operato
Modularizatio
Exception Hand
Iterators, Par
Programming,
Debugging, Co

```
import math
def range_plus_plus(*args):
 match args:
  case []:
   return
  case [int(a), e, math.inf] if e is ...:
   while True:
    yield a
    a += 1
  case [e, int(a), *rest] if e is ...:
   yield from range(a + 1)
   yield from range_plus_plus(*rest)
  case [int(a), e, int(b), *rest] if e is ...:
   yield from range(a, b + 1)
```

2nd Edition
Updated and Revised

Python 3

The Comprehensive Guide

Johannes Ernesti
Peter Kaiser

Rheinwerk Computing

Contents

Chapter 12
Sequential Data Types

Sequential data types refers to a class of data types that manage sequences of similar or different *elements*. The elements stored in sequential data types have a defined order, and they can be accessed through unique indexes.

Python essentially provides the following five sequential types: `str`, `bytes`, `bytearray`, `list`, and `tuple` (see Table 12.1).

Data Type	Stores	Mutability	Section
`list`	Lists of any instances	Mutable	Section 12.3
`tuple`	Lists of any instances	Immutable	Section 12.4
`str`	Text as a sequence of letters	Immutable	Section 12.5
`bytes`	Binary data as a sequence of bytes	Immutable	Section 12.5
`bytearray`	Binary data as a sequence of bytes	Mutable	Section 12.5

Table 12.1: List of Sequential Data Types

12.1 The Difference Between Text and Binary Data

The `str` data type is intended for storing and processing text. Therefore, a `str` instance consists of a sequence of letters, spaces, punctuation marks, and line feeds; these are exactly the components that make up text in human language. It's remarkable that this also works with regional special characters, such as the German umlauts *ä*, *ü*, and *ö*.

In contrast, an instance of the `bytes` data type can store a binary data stream—that is, a sequence of bytes. The `bytearray` data type is also capable of storing binary data. However, `bytearray` instances are mutable, unlike `bytes` instances.

The structural separation of textual data and binary data is a feature that distinguishes Python from many other programming languages.

Note

The `str` and `bytes` data types represent one of the major differences between Python 2 and Python 3. In Python 2, there were two data types, `str` and `unicode`; `str` corresponded to the current `bytes` and `unicode` to the current `str`.

Because the old `str` data type often was used to store text strings, there were some stumbling blocks if you wanted to process special characters with Python programs.

Due to the `str` and `bytes` types in Python 3, the handling of strings is more clearly structured. However, you must pay attention to which data types the functions of the standard library expect as parameters or return as return values and then make conversions if necessary.

For more information, Section 12.5.10.

Instances of the `str` data type and of the `bytes` data type are immutable, so their value can't change after instantiation. Nevertheless, you can conveniently use and manipulate strings. When changes are made, the original string won't be changed because a new string is always created instead.

The `list` and `tuple` types can store sequences of any instances. The main difference between the two almost identical data types is that a list can be modified after it has been created, while a tuple does not allow the initial contents to be modified: `list` is a mutable and `tuple` an immutable data type.

Note

The distinction between mutable and immutable data types is very central in Python. Because mutability is the essential difference between the `list` and `tuple` data types, the question arises why the tuple as an immutable variant of the list is actually needed.

One reason is that tuples, because of their immutability, can be used in situations for which lists are not suitable. For example, tuples can be used as keys in a dictionary, while lists are forbidden there. This is due to the concept of *hashability*, which is closely related to immutability.

For each instance of a sequential data type, there is a basic set of operators and methods that's always available. For the sake of simplicity, we'll introduce this using `list` and `str` instances as general examples for a mutable and an immutable sequential data type. Later, we will dive deeper into specific aspects of the individual data types.

12.2 Operations on Instances of Sequential Data Types

The following operations are defined for all sequential data types (`s` and `t` are instances of the same sequential data type; `i`, `j`, `k`, and `n` are integers; and `x` is a reference to an instance; see Table 12.2).

Notation	Description	Section
`x in s`	Checks whether `x` is contained in `s`. The result is a truth value.	Section 12.2.1
`x not in s`	Checks if `x` is not contained in `s`. The result is a `bool` instance. Equivalent to `not x in s`.	Section 12.2.1
`s + t`	The result is a new sequence containing the concatenation of `s` and `t`.	Section 12.2.2
`s += t`	Creates the concatenation of `s` and `t` and assigns it to `s`.	Section 12.2.2
`s * n` or `n * s`	Returns a new sequence containing the concatenation of `n` copies of `s`.	Section 12.2.3
`s *= n`	Generates the product `s * n` and assigns it to `s`.	Section 12.2.3
`s[i]`	Returns the `i`th element of `s`.	Section 12.2.4
`s[i:j]`	Returns the section of `s` from `i` to `j`.	Section 12.2.5
`s[i:j:k]`	Returns the section of `s` from `i` to `j`, where only every `k`th element is considered.	Section 12.2.5
`len(s)`	Returns the number of elements of `s`.	Section 12.2.6
`max(s)`	Returns the largest element of `s`, provided that an ordering is defined for the elements.	Section 12.2.7
`min(s)`	Returns the smallest element of `s`, provided that an ordering is defined for the elements.	Section 12.2.7
`s.index(x[, i[, j]])`	Returns the index `k` of the first occurrence of `x` in the sequence `s` in the range $i \leq k < j$.	Section 12.2.8
`s.count(x)`	Counts how often `x` occurs in the sequence `s`.	Section 12.2.9

Table 12.2: Operations on Instances of Sequential Data Types

These operations are explained in detail ahead.

12.2.1 Checking for Elements

With the help of `in`, you can determine whether a certain element is contained in a sequence. For an instance of the `list` data type that contains both strings and numbers, it looks like this:

```
>>> lst = ["one", 2, 3.0, "four", 5, "six", "seven"]
>>> 3.0 in lst
True
>>> "four" in lst
True
>>> 10 in lst
False
```

Because the elements of a string are characters, you can use the operator to check whether a particular letter occurs in a string. The result is a truth value: `True` if the element is present and `False` if it isn't. In Python, you can represent characters using strings of the length 1:

```
>>> s = "This is our test string"
>>> "u" in s
True
>>> if "j" in s:
...     print("Yay, my favorite letter is included")
... else:
...     print("I don't like this string ...")
I don't like this string ...
```

You can also use the `in` operator to check whether a particular substring is contained in a string:

```
>>> s = "This is our test string"
>>> "is" in s
True
>>> "Hello" in s
False
```

This works only with strings—that is, instances of the types `str`, `bytes`, and `bytearray`. The `in` operator can't be used to check whether a sublist is contained in a `list` instance. The same applies to instances of the `tuple` type:

```
>>> [2,3] in [1,2,3,4]
False
```

To check the opposite—that is, whether an element is *not* contained in a sequence—you can use the `not in` operator. Its use is the same as that of the `in` operator, with the only difference being that it produces the negated result of the `in` operator:

```
>>> "a" in "Dentist appointment"
True
>>> "a" not in "Dentist appointment"
False
```

At this point, you'll rightly ask yourself why a separate operator has been defined for this purpose when you can negate any Boolean value with `not`. The following checks are equivalent:

```
>>> "n" not in "Python is great"
False
>>> not "n" in "Python is great"
False
```

The reason for this seemingly superfluous definition is to make it easier to read. The expression `x not in s` reads exactly like an English sentence, unlike `not x in s`, which is more difficult to read.[1]

12.2.2 Concatenation

To concatenate sequences, the `+` operator is used. In the following example, Mr. Miller's first and last name are concatenated along with a space character to form a new string:

```
>>> first_name = "Harold"
>>> last_name = "Miller"
>>> name = first_name + " " + last_name
>>> name
'Harold Miller'
```

Another way to concatenate strings is provided by the `+=` operator for augmented assignments:

```
>>> s = "music "
>>> t = "loudspeaker"
>>> s += t
>>> s
'music loudspeaker'
```

Here, `s += t` for immutable data types is to be read in the same way as `s = s + t` because a new instance is actually created with the value of `s + t`, which is then referenced by `s`. So after the operation `s += t`, the three instances `"music "`, `"loudspeaker"`, and `"music loudspeaker"` exist in the memory, whereas there is no reference to `"music "` anymore.

This doesn't apply to mutable data types such as `list`. Here, no further instance with the value `s + t` is created, but the instance `s` is changed.

1 In addition, for the interpretation of `not x in s`, you need to know the precedence of the two operators, `not` and `in`, respectively. If the `not` operator binds more strongly, then the expression would be evaluated like `(not x) in s`. If `in` has a higher priority, then the expression would be treated as `not (x in s)`. In fact, `in` binds stronger than `not`, making the latter interpretation the correct one.

In the following example, we investigate the effects of manipulating immutable data types based on the example of strings:

```
>>> s = "music "
>>> t = "loudspeaker"
>>> temp = s
>>> s += t
>>> s
'music loudspeaker'
>>> t
'loudspeaker'
>>> temp
'music '
```

Because a new `str` instance was created via the `s += t` statement, the `str` instance referenced by `temp` hasn't changed. This is different for the mutable `list` data type:

```
>>> s = [1,2]
>>> t = [3,4]
>>> temp = s
>>> s += t
>>> s
[1, 2, 3, 4]
>>> t
[3, 4]
>>> temp
[1, 2, 3, 4]
```

Here, `s` and `temp` refer to the same `list` instance even after the statement `s += t` because the existing list was modified and no new object was created. We say that operators exhibiting this behavior work *in place*. Incidentally, this also applies to the `*=` operator, which will be described in the following section.

12.2.3 Repetition

You can form the product of a sequence `s` with an integer `n`: `n * s` or `s * n`. The result is a new sequence containing `n` copies of `s` in succession:

```
>>> 3 * "abc"
'abcabcabc'
>>> "xyz" * 5
'xyzxyzxyzxyzxyz'
```

As is the case with concatenation, there is also an operator for the augmented assignment, `*=`:

```
>>> santa_claus = "ho"
>>> santa_claus *= 3
>>> santa_claus
'hohoho'
```

Lists of integers can be multiplied in the same way:

```
>>> [1,2] * 3
[1, 2, 1, 2, 1, 2]
```

Like `+=`, the `*=` operator works also *in place*. We explained what exactly this means in the previous section using the operator `+=`.

Note that the objects located in the list are not copied when the list is multiplied; the extra list elements are only additional references to the same instances. This is especially relevant if the list elements themselves are mutable objects—such as, in the following example, single-element lists:

```
>>> x = [["nice"], ["list"]] * 3
>>> x
[['nice'], ['list'], ['nice'], ['list'], ['nice'], ['list']]
>>> x[1][0] = "what?"
>>> x
[['nice'], ['what?'], ['nice'], ['what?'], ['nice'], ['what?']]
```

By multiplying the initial list, `x` contains three references each to the `['nice']` and `['list']` lists. As soon as you access and change the underlying instance via one of these references, you also change the other two associated entries of `x` via a side effect.

12.2.4 Indexing

As mentioned at the beginning of this chapter, sequences represent sequences of elements. Because these elements are stored in a specific order—for example, a string in which the order of the letters is arbitrary would make little sense as a store for text—you can assign an *index* to each element of the sequence. For this purpose, all elements of the sequence are numbered consecutively from front to back, with the first element getting the index `0`.

The `[]` operator allows you to access a specific element of the sequence by writing the corresponding index in the square brackets:

```
>>> alphabet = "abcdefghijklmnopqrstuvwxyz"
>>> alphabet[9]
'j'
>>> alphabet[1]
```

```
'b'
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[3]
4
```

To access the last or the `x`th element from the back, there is another indexing of the elements from the back to the front. The last element receives `-1` as index, the penultimate one `-2`, and so on. Table 12.3 illustrates the two types of indexing.

Index from Front	0	1	2	3	4	5
Elements	P	y	t	h	o	n
Index from Back	–6	–5	–4	–3	–2	–1

Table 12.3: Indexing from Front and Back

In the following example, we use negative indexes to access the second-to-last character of a string or the last element of a list:

```
>>> name = "Python"
>>> name[-2]
'o'
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[-1]
6
```

If you try to access a nonexistent element with an index, this will be acknowledged with an `IndexError`:

```
>>> too_short = "I'm too short"
>>> too_short[1337]
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    too_short[1337]
    ~~~~~~~~~^^^^^^
IndexError: string index out of range
```

12.2.5 Slicing

In addition to accessing individual elements of the sequence, you can also read entire subsequences using the `[]` operator. This is achieved by writing the beginning and the end of the relevant subsequence, separated by a colon, in the square brackets. The beginning is the index of the first element of the subsequence, and the end is the index of the first element that should no longer be included in the subsequence.

To extract the string `"NEEDLE"` from the string in the following example, we specify the index of the capital `"N"` and that of the first `"h"` after `"NEEDLE"`:

```
>>> s = "haystackNEEDLEhaystack"
>>> s[8]
'N'
>>> s[14]
'h'
>>> s[8:14]
'NEEDLE'
```

Figure 12.1 illustrates the access to the subsequence.

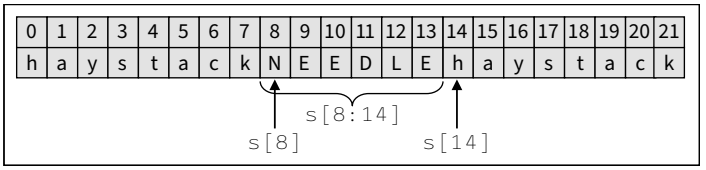


Figure 12.1: Extracting a Subsequence

You can extract parts of a list in a similar way:

```
>>> l = ["I", "am", "a", "list", "of", "strings"]
>>> l[2:5]
['a', 'list', 'of']
```

It's also possible to mix positive and negative indexes in this *slicing* process. For example, the following code section determines a subsequence without the first and last elements of the original sequence:

```
>>> string = "python"
>>> string[1:-1]
'ytho'
>>> l = ["I", "am", "a", "list", "of", "strings"]
>>> l[1:-1]
['am', 'a', 'list', 'of']
```

The indexes can also be omitted, which results in the maximum or minimum possible value being assumed. If you omit the start index, the zeroth element is assumed to be the first element of the subsequence, and if you omit the end index, all letters are copied to the end. For example, if you want to determine the first five letters of a string or all of them starting from the fifth character,[2] this is how to do it:

```
>>> s = "abcdefghijklmnopqrstuvwxyz"
>>> s[:5]
```

2 Note here again that the indexing of the characters of a string starts at 0.

```
'abcde'
>>> s[5:]
'fghijklmnopqrstuvwxyz'
```

Slicing for Copying

If you omit both indexes (`s[:]`), you can also create a genuine copy of the sequence because then all elements are copied, from the first to the last. To see this in action, first recall the behavior of the simple assignment:

```
>>> s1 = ["thesis"]
>>> s2 = s1
>>> s1 == s2
True
>>> s1 is s2
True
```

As expected, `s1` and `s2` reference the same instance, so they are identical. In the next example, we use slicing to create a real copy of the `["thesis"]` list in the memory.[3] This can be seen in the identity comparison with `is`:

```
>>> s1 = ["thesis"]
>>> s2 = s1[:]
>>> s1 == s2
True
>>> s1 is s2
False
```

Note

If you use an immutable data type such as `str` instead of a mutable data type such as `list` in the preceding example, the expression `s1 is s2` may evaluate to `True` in both cases. This is because for immutable data types, it makes no difference whether the instance is actually copied or the original instance is used: The value can't be changed anyway.

For this reason, an instance of an immutable data type isn't necessarily copied even when `[:]` is used; it is instead referenced one more time for efficiency.

If you use a string instead of a list, the differences compared to the previous example become clear:

```
>>> s1 = "Copy me"
>>> s2 = s1[:]
>>> s2 is s1
True
```

3 In real life, of course, you should never simply copy a PhD thesis!

Slicing with Steps

Slicing offers even more flexible options if you don't want to extract a whole subsequence, but only certain elements of this part. The *step* size allows you to specify how the indices are counted from the beginning to the end of a subsequence. The step size is specified after the trailing boundary, separated by another colon. For example, a step size of `2` ensures that only every other element is copied:

```
>>> digits = "0123456789"
>>> digits[1:10:2]
'13579'
```

The string containing every other element of `digits` starting from the first element—that is, the element with index zero—results in a new string with the odd digits. The boundary indexes can also be omitted in this extended notation. The following code therefore is equivalent to the previous example:

```
>>> digits = "0123456789"
>>> digits[1::2]
'13579'
```

A negative increment causes counting down from the start index to the end index, in which case the start index must reference an element of the sequence further back than the end index. With a step size of `-1`, for example, a sequence can be "flipped":

```
>>> name = "ytnoM Python"
>>> name[4::-1]
'Monty'
>>> name[::-1]
'nohtyP Monty'
```

With negative step sizes, the beginning and end of the sequence are swapped. Therefore, in the example `name[4::-1]`, not everything from the fourth to the last character is read, but only the part from the fourth to the first character.

Important for the handling of slicing is the fact that indexes that are too large or too small don't cause an `IndexError` as is the case when accessing single elements. Too large indexes are internally replaced by the maximum possible index and those that are too small by the minimum possible index. If both indexes are outside the valid range or if the start index is larger than the end index when the step size is positive, then an empty sequence is returned:

```
>>> s = "Much less than 1337 characters"
>>> s[5:1337]
'less than 1337 characters'
>>> s[-100:100]
'Much less than 1337 characters'
```

```
>>> s[1337:2674]
''
>>> s[10:4]
''
```

12.2.6 Length of a Sequence

The number of elements in a sequence defines the *length of the sequence*. The length of a sequence is a positive integer and can be determined via the built-in `len` function:

```
>>> string = "How long do you think I am?"
>>> len(string)
27
>>> len(["Hello", 5, 2, 3, "World"])
5
```

12.2.7 The Smallest and the Largest Element

To determine the smallest or largest element of a sequence, you can use the built-in `min` and `max` functions:

```
>>> l = [5, 1, 10, -9.5, 12, -5]
>>> max(l)
12
>>> min(l)
-9.5
```

However, these two functions are only useful if an *ordering* exists for the elements of the sequence. In Chapter 11, Section 11.7, on complex numbers, for example, the `complex` data type is described without an ordering. Likewise, it's not possible to compare entirely different data types, such as strings and numbers:

```
>>> l = [1,2, "world"]
>>> min(l)
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    min(l)
    ~~~^^^
TypeError: '<' not supported between instances of 'str' and 'int'
```

Nevertheless, `min` and `max` can be applied to strings in a meaningful way because for letters, their position in the alphabet[4] defines an ordering:

4 In fact, the alphabet represents only a part of this ordering. In Section 12.5.10, in connection with *encodings*, you will learn that every character—including control characters, punctuation marks, foreign language characters, and other symbols—has a unique index.

```
>>> max("who do you think will win")
'y'
>>> min("string")
'g'
```

12.2.8 Searching for an Element

You can use the `index` method of a sequence to determine the position of an element:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> digits.index(3)
2
>>> s = "Hello world"
>>> s.index("l")
2
```

To restrict the search to a subrange of the sequence, the method supports two optional parameters `i` and `j`, where `i` is the first index of the desired subsequence and `j` is the first index after the desired subsequence:

```
>>> sequence = [0, 11, 222, 3333, 44444, 3333, 222, 11, 0]
>>> sequence.index(222)
2
>>> sequence.index(222, 3)
6
>>> sequence.index(222, -5)
6
>>> "Hello World".index("l", 5, 100)
9
```

As with indexing elements of the sequence, negative values for `i` and `j` are counted from the end of the sequence. In the preceding example, `sequence.index(222, -5)` therefore was used to search in the `[44444, 3333, 222, 11, 0]` subsequence, starting at the fifth element from the back.

If the element `x` isn't contained in `s` or in the specified subsequence, then `index` results in a `ValueError`:

```
>>> s = [2.5, 2.6, 2.7, 2.8]
>>> s.index(2.4)
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    s.index(2.4)
    ~~~~~~~^^^^^
ValueError: 2.4 is not in list
```

12.2.9 Counting Elements

You can use `count` to determine how often a particular element `x` is contained in a sequence:

```
>>> s = [1, 2, 2, 3, 2]
>>> s.count(2)
3
>>> "Hello world".count("l")
3
```

The next section deals with operations that are only available for mutable sequences.

12.3 The list Data Type

In this section, you'll learn about the first mutable data type, *list*, in detail. Unlike the `str`, `bytes`, and `bytearray` sequential data types, which can only store elements (characters) of the same type, lists are suitable for managing any kind of instances, even of different data types. A list can therefore contain numbers, strings, or even other lists as elements.

You can create a new list by writing an enumeration of its elements in square brackets `[]`:

```
>>> l = [1, 0.5, "String", 2]
```

The list `l` now contains two integers—a float and a string.

Note

A list can also be generated via *list comprehension*. In that case, not all elements of the list are explicitly listed but are generated via a formation rule similar to a `for` loop. For example, the following list comprehension generates a list of the squares of the numbers from `0` to `9`:

```
>>> [i*i for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Section 12.3.7 describes list comprehensions in greater detail.

You can use unpacking when creating a list:

```
>>> [1, 2, *[3, 4]]
[1, 2, 3, 4]
```

For more information on unpacking, Section 12.4.1.

Because the list type named `list` within Python is a sequential data type, all the methods and procedures described in the last section can be applied to it. See Section 12.2 for a table of operations that are available for all sequential data types.

Unlike strings, the contents of a list can change even after it has been created, which is why several other operators and methods are available for it (see Table 12.4).

Operator	Effect	Section
`s[i] = x`	The element of `s` with index `i` is replaced by `x`.	Section 12.3.1
`s[i:j] = t`	The part `s[i:j]` is replaced by `t`. At the same time, `t` must be iterable.	Section 12.3.2
`s[i:j:k] = t`	The elements of `s[i:j:k]` are replaced by those of `t`.	Section 12.3.2
`del s[i]`	The `i`th element of `s` is removed.	Section 12.3.3
`del s[i:j]`	The part `s[i:j]` is removed from `s`. This is equivalent to `s[i:j] = []`.	Section 12.3.3
`del s[i:j:k]`	The elements of the subsequence `s[i:j:k]` are removed from `s`.	Section 12.3.3

Table 12.4: Operators for the list Data Type

We'll now explain these operators one by one with brief examples.

12.3.1 Changing a Value Within the List: Assignment via []

You can replace elements of a list with others if you know their index:

```
>>> s = [1, 2, 3, 4, 5, 6, 7]
>>> s[3] = 1337
>>> s
[1, 2, 3, 1337, 5, 6, 7]
```

However, this method isn't suitable for inserting additional elements into the list. Only existing elements can be replaced, while the length of the list remains unchanged.

12.3.2 Replacing Sublists and Inserting New Elements: Assignment via []

It's possible to replace a whole sublist with other elements. To do this, you must write the part of the list to be replaced like you did in the slicing process, but it must be on the left-hand side of an assignment:

```
>>> shopping_list = ["bread", "eggs", "milk", "fish", "flour"]
>>> shopping_list[1:3] = ["water", "beef"]
>>> shopping_list
['bread', 'water', 'beef', 'fish', 'flour']
```

The list to be inserted may have more or fewer elements than the part to be replaced and may even be completely empty.

You can specify a step size, just like in slicing. In the following example, every third element of the `s[2:11]` subsequence is replaced by the corresponding element from `["A", "B", "C"]`:

```
>>> s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> s[2:9:3] = ["A", "B", "C"]
>>> s
[0, 1, 'A', 3, 4, 'B', 6, 7, 'C', 9, 10]
```

If a step size is specified, then the sequence on the right-hand side of the assignment must have as many elements as the subsequence on the left-hand side. If that's not the case, a `ValueError` will be generated.

12.3.3 Deleting Elements and Sublists: del in Combination with []

To remove a single value from a list, you can use the `del` operator:

```
>>> s = [26, 7, 1987]
>>> del s[0]
>>> s
[7, 1987]
```

In this way, entire sublists also can be removed:

```
>>> s = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> del s[3:6]
>>> s
[9, 8, 7, 3, 2, 1]
```

For removing parts of a list, the step sequence of the slicing notation is also supported. In the following example, this removes all elements with an even index:

```
>>> s = ["a","b","c","d","e","f","g","h","i","j"]
>>> del s[::2]
>>> s
['b', 'd', 'f', 'h', 'j']
```

12.3.4 Methods of list Instances

Now that we've covered the operators for lists, let's turn to the methods of a list. In Table 12.5, `s` and `t` are lists, `i`, `j`, and `k` are integers, and `x` is an instance.[5]

5 *Warning:* If parameters are bracketed with square brackets in the left-hand column, this still means that they are optional parameters. These square brackets have nothing to do with creating a new list.

Method	Effect
`s.append(x)`	Appends `x` to the end of list `s`.
`s.copy()`	Creates a copy of list `s`.
`s.extend(t)`	Appends all elements of list `t` to the end of list `s`.
`s.insert(i, x)`	Inserts `x` at position `i` in list `s`. Then, `s[i]` has the value of `x`, with all subsequent elements moving up one place.
`s.pop([i])`	Returns the `i`th element of list `s` and removes it from `s`. If `i` isn't specified, the last element will be taken.
`s.remove(x)`	Removes the first occurrence of `x` from list `s`.
`s.reverse()`	Reverses the order of the elements in `s`.
`s.sort([key, reverse])`	Sorts list `s`.

Table 12.5: Methods of list Instances

Let's now take a more detailed look at the methods.

s.append(x)

The `append` method enables you to extend a list at the end by another element:

```
>>> s = ["There should be another string after me"]
>>> s.append("Here it is")
>>> s
['There should be another string after me', 'Here it is']
```

s.copy()

The `copy` method creates a copy of a list:

```
>>> s1 = [1, 2, 3]
>>> s2 = s1.copy()
>>> s2
[1, 2, 3]
```

Note that only the list itself is copied, not the elements it contains. This can lead to side effects; Section 12.3.6.

s.extend(t)

To append multiple elements to a list, you can use the `extend` method, which expects an iterable object—for example, another list—as the `t` parameter. As a result, all elements of `t` are appended to list `s`:

```
>>> s = [1, 2, 3]
>>> s.extend([4, 5, 6])
>>> s
[1, 2, 3, 4, 5, 6]
```

s.insert(i, x)

With `insert`, you can insert a new element into a list at any position. The first parameter, `i`, specifies the desired index of the new element, and the second, `x`, specifies the element itself:

```
>>> first_with_gap = [1, 2, 3, 5, 6, 7, 8]
>>> first_with_gap.insert(3, 4)
>>> first_with_gap
[1, 2, 3, 4, 5, 6, 7, 8]
```

If index `i` is too small, then `x` is inserted at the beginning of `s`; if it's too large, it's appended at the end like with `append`.

s.pop([i])

The counterpart to `append` and `insert` is `pop`. This method allows you to remove any element from a list based on its index. If the optional parameter isn't specified, then the last element of the list is removed. The removed element is returned by `pop`:

```
>>> s = ["H", "e", "l", "l", "o"]
>>> s.pop()
'o'
>>> s.pop(0)
'H'
>>> s
['e', 'l', 'l']
```

If you try to pass an invalid index or remove an element from an empty list, an `IndexError` will be generated.

s.remove(x)

If you want to remove an element with a certain value from a list, no matter what index it has, you can use the `remove` method. It removes the first element of the list which has the same value as `x`:

```
>>> s = ["W", "o", "o", "h", "o", "o"]
>>> s.remove("o")
>>> s
['W', 'o', 'h', 'o', 'o']
```

Attempting to remove an element that doesn't exist results in a `ValueError`.

s.reverse()

You can use `reverse` to reverse the order of elements in a list:

```
>>> s = [1, 2, 3]
>>> s.reverse()
>>> s
[3, 2, 1]
```

Unlike the `s[::-1]` slice notation, the reversal is done *in place*. Thus, no new `list` instance is created; instead, the old one is changed. This is particularly relevant if the list is very large and a copy would induce significant costs.

12.3.5 Sorting Lists: s.sort([key, reverse])

The `sort` method can be used to sort a list according to certain criteria. If you call the method without parameters, then Python uses the normal comparison operators for sorting:

```
>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If a list contains elements for which no ordering is defined, such as instances of the `complex` data type, then calling `sort` without parameters will cause a `TypeError`:

```
>>> lst = [5 + 13j, 1 + 4j, 6 + 2j]
>>> lst.sort()
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    lst.sort()
    ~~~~~~~~^^
TypeError: '<' not supported between instances of 'complex' and
'complex'
```

To sort a list according to certain criteria, you can use the `key` parameter. The `sort` method expects a function in the `key` parameter, which is called before each comparison for both operands and therefore expects a parameter in its turn. In the result, not the operands but instead the corresponding return values of the passed function are compared directly.

In the following example, we'll sort a list of names by length. For this purpose, we'll use the built-in function `len`, which assigns its length to each name. In practice, this looks as follows:

```
>>> l = ["Catherine", "Peter", "Bob", "Michael", "Emily", "Ben"]
>>> l.sort(key=len)
>>> l
['Bob', 'Ben', 'Peter', 'Emily', 'Michael', 'Catherine']
```

Whenever the sorting algorithm compares two elements of the list, such as `"Michael"` and `"Bob"`, it compares not the elements themselves but the corresponding return values of the `key` function. In this example, `len("Michael")` and `len("Bob")`—that is, the numbers `7` and `3`—are thus compared. Consequently, the string `"Bob"` is to be placed before the string `"Michael"` in this case. Graphically, this example can be illustrated as shown in Figure 12.2.

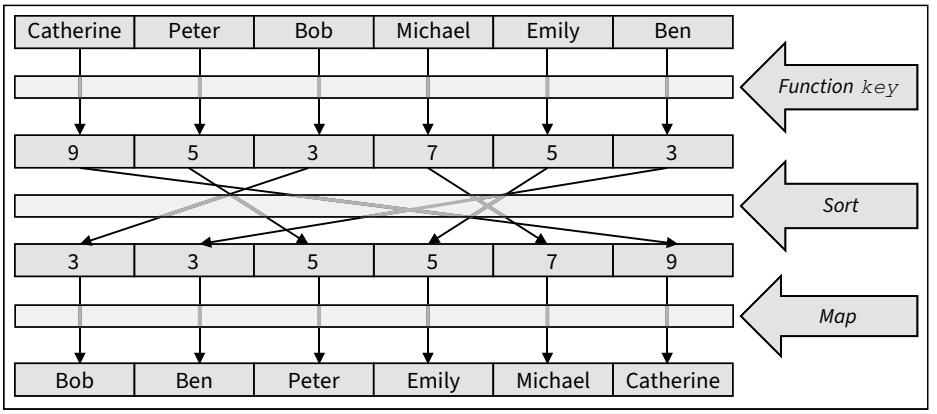


Figure 12.2: Sorting with Key

Of course, you can also pass more complex functions than the built-in `len` function. To learn how to define your own functions—for example, to use them with `sort`—see Chapter 17.

The last parameter, `reverse`, expects a Boolean value to be passed, indicating whether the sort order should be reversed:

```
>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort(reverse=True)
>>> l
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note

It should be noted here that `sort` is a function that accepts only keyword arguments. If you try to pass positional arguments, this will cause an error. In the following example, we'll again try to sort the name list by length. However, this time we'll use a positional argument to pass `len`:

```
>>> l = ["Catherine", "Peter", "Bob", "Michael", "Emily"]
>>> l.sort(len)
Traceback (most recent call last):
```

```
  File "<python-input-1>", line 1, in <module>
    l.sort(len)
    ~~~~~~^^^^^
TypeError: sort() takes no positional arguments
```

You may wonder how `sort` handles such values that are in the same place in the sort order. In the example shown previously, "Bob" and "Ben" with length 3 and "Peter" and "Emily" with length 5 each had the same value. In the next section, you'll learn about *stable sorting methods* and what that means for these values.

Stable Sorting Methods

An important property of `sort` is that it is a *stable sorting method*. Stable sorting methods are characterized by the fact that they don't swap the relative position of equivalent elements during sorting. Let's suppose you have the list of names shown in Table 12.6.

First Name	Last Name
Natalie	Smith
Matthew	Taylor
Felix	Crowe
Robert	Smith
Howard	Smith
Peter	Anderson

Table 12.6: List of Example Names

Say that it's your job to sort this list alphabetically by last name. Groups with the same last name should be sorted by their respective first names. To solve this problem, you can sort the list by the first names in the first step, which results in the arrangement shown in Table 12.7.

First Name	Last Name
Felix	Crowe
Howard	**Smith**
Matthew	Taylor
Natalie	**Smith**

Table 12.7: List of Names Sorted by First Name

First Name	Last Name
Peter	Anderson
Robert	**Smith**

Table 12.7: List of Names Sorted by First Name (Cont.)

As a result, we are now only interested in the positions of the three people whose last name is Smith. If you simply deleted all other names, the Smiths would be sorted correctly because their relative position was correctly established by the first sort run. Now the stability of the `sort` method comes into play, because when sorting by the last names again, this relative order is not affected. The end result would look as shown in Table 12.8.

First Name	Last Name
Peter	Anderson
Felix	Crowe
Howard	Smith
Natalie	Smith
Robert	Smith
Matthew	Taylor

Table 12.8: Fully Sorted List of Names

If `sort` were not stable, then there would be no guarantee that Howard would be ranked above Natalie and Robert.

12.3.6 Side Effects

In connection with the `list` data type in Python, there are a few peculiarities that aren't immediately obvious.[6]

You'll certainly remember the peculiarity of the `+=` operator, which was explained in Section 12.2.2 in connection with the term *in place*. Let's take another look at this behavior to provide some more comprehensive explanations.

First, `list` is a mutable data type; therefore, changes to a `list` instance always affect all references that point to it. Consider the following example, in which the `str` immutable data type is compared with `list`:

6 These special features basically apply to all mutable data types.

```
>>> a = "Hello "
>>> b = a
>>> b += "world"
>>> b
'Hello world'
>>> a
'Hello '
```

This example simply creates a `str` instance with the value `"Hello"` and has the two references `a` and `b` point to it. Then, the `+=` operator is used to append `"world"` to the string referenced by `b`. As you can see in the output—and as we expected—a new instance with the value `"Hello world"` is created and assigned to `b`, while `a` remains unaffected.

If you apply this example to lists, an important difference will emerge:

```
>>> a = [1337]
>>> b = a
>>> b += [2674]
>>> b
[1337, 2674]
>>> a
[1337, 2674]
```

Structurally, the code is similar to the `str` example—only this time the data type used is not `str`, but `list`. The interesting part is the output at the end, according to which `a` and `b` have the same value, although the operation was performed only on `b`. In fact, `a` and `b` refer to the same instance, which you can convince yourself of using the `is` operator:

```
>>> a is b
True
```

You should keep these side effects[7] in mind when working with lists and other mutable data types. If you want to make sure that the original list is not modified, you should create a genuine copy using slicing:

```
>>> a = [1337]
>>> b = a[:]
>>> b += [2674]
>>> b
[1337, 2674]
>>> a
[1337]
```

In this example, the list referenced by `a` has been copied, protecting it from indirect manipulation via `b`. In such cases, you must weigh resource consumption against protection

7 Side effects will play an important role in the context of functions in Chapter 17, Section 17.10.

against side effects as copies of the lists must be created in the memory. This consumes computing time and memory, especially with long lists, and can therefore slow down the program.

In the context of side effects, the elements of a list are also worth looking at: A list doesn't store instances per se, but only references to them. On the one hand, this makes lists more flexible and increases their performance, but on the other hand, it also makes them susceptible to side effects. Consider the following example, which may seem weird at first sight:

```
>>> a = [[]]
>>> a = 4 * a
>>> a
[[], [], [], []]
>>> a[0].append(10)
>>> a
[[10], [10], [10], [10]]
```

Initially, `a` references a list that contains another, empty list. During the subsequent multiplication by the factor `4`, the inner empty list is not copied but only referenced three more times. In the output, you therefore see the same list four times. Once this is understood, it is obvious why the `10` appended to the first element of `a` is also added to the other three lists: It is simply the same list. Figure 12.3 illustrates this.

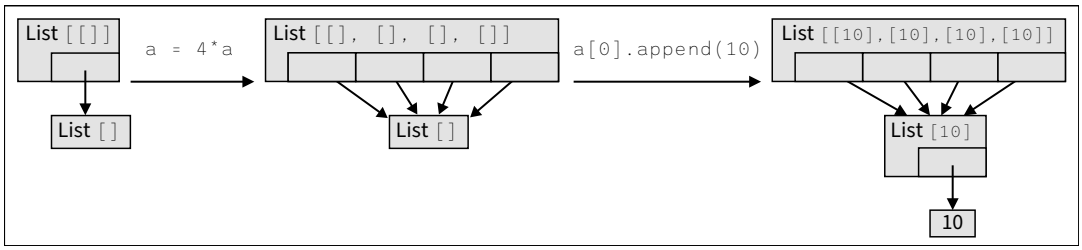


Figure 12.3: Side Effect when Multiple Elements Reference the Same List

It's also possible for a list to contain itself as an element:

```
>>> a = []
>>> a.append(a)
```

The result is an infinitely deep nesting as each list in turn contains itself as an element. Because only references need to be stored, this infinite nesting consumes very little memory and not, as one might initially assume, infinitely much. Nevertheless, such nestings bear the risk of endless loops if you want to process the contained data. For example, let's suppose you wanted to output such a list on the screen. This would result in an infinite number of opening and closing brackets. Nevertheless, it's possible to output such lists via `print`. Python checks whether a list contains itself, and then outputs three dots (`...`) instead of further nesting:

```
>>> a = []
>>> a.append(a)
>>> print(a)
[[...]]
```

Note that the notation with the three dots is not valid Python code for creating lists nested within themselves.

If you use lists yourself that might be recursive, you should equip your programs with queries to detect nesting of lists with themselves so that the program won't get stuck in an endless loop during processing.

12.3.7 List Comprehensions

It's a common problem for developers to want to create a new list from the elements of an existing list according to a certain calculation rule. Until now, you would have to do this awkwardly in a `for` loop. In the following example, we use this approach to generate a result list of the respective squares from a list of integers:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> result_lst = []
>>> for x in lst:
...     result_lst.append(x**2)
...
>>> result_lst
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python supports a more flexible syntax created specifically for this purpose: list comprehensions. From a list of integers, the following list comprehension generates a new list containing the squares of these numbers:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> [x**2 for x in lst]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A list comprehension is enclosed in square brackets and consists of an expression followed by any number of `for`/`in` sections. A `for`/`in` section draws on the syntax of the `for` loop and specifies which identifier is used to iterate over which list—in this case, the identifier `x` over the list `lst`. The specified identifier can be used in the expression at the beginning of the list comprehension. The result of a list comprehension is a new list that contains as elements the results of the expression in each iteration step. The list comprehension functionality presented previously can be summarized as follows: For each `x` element of `lst` list, take the square of `x` and insert the result into the result list.

This is the simplest form of list comprehension. The `for`/`in` section can be extended by a condition so that only certain elements are taken over into the new list. For example, you could extend the preceding list comprehension to form only the squares of even numbers:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> [x**2 for x in lst if x%2 == 0]
[4, 16, 36, 64]
```

For this purpose, the `for`/`in` section is extended by the `if` keyword, which is followed by a condition. Only if this condition returns `True` will the calculated element be included in the result list. This form of list comprehension thus can be described as follows: For each `x` element of list `lst`—if `x` is an even number—take the square of `x` and insert the result into the result list.

The next example is supposed to use a list comprehension to add two three-dimensional vectors represented as lists. Vectors are added coordinate by coordinate—in this case, element by element:

```
>>> v1 = [1, 7, -5]
>>> v2 = [-9, 3, 12]
>>> [v1[i] + v2[i] for i in range(3)]
[-8, 10, 7]
```

For this purpose, a list of indexes generated by `range` is iterated in the list comprehension. In each run, the respective coordinates are added and appended to the result list.

We already mentioned that a list comprehension can have any number of `for`/`in` sections. These can be considered similar to nested `for` loops. In the following sections, we'll describe an example in which this property is useful. First, we define two lists:

```
>>> lst1 = ["A", "B", "C"]
>>> lst2 = ["D", "E", "F"]
```

Now, a list comprehension is supposed to create a list containing all possible letter combinations that can be formed by first choosing a letter from `lst1` and then choosing one from `lst2`. The combinations should appear as tuples in the list:

```
>>> [(a,b) for a in lst1 for b in lst2]
[('A', 'D'), ('A', 'E'), ('A', 'F'), ('B', 'D'), ('B', 'E'), ('B', 'F'),
('C', 'D'), ('C', 'E'), ('C', 'F')]
```

This list comprehension can be described as follows: For each `a` element of list `lst1`, go over all `b` elements of `lst2` and insert the `(a,b)` tuple into the result list in each case.

List comprehensions provide an interesting and elegant way to write complex operations in a space-saving manner. Many problems for which list comprehensions are used could also

be solved by the built-in `map` or `filter` functions (see Chapter 17, Section 17.14) or by a combination of the two, but list comprehensions are often more readable and lead to clearer source code.

12.4 Immutable Lists: tuple

The list is a very flexible sequential data type that can accommodate arbitrarily long sequence of elements. This section describes the *tuple*, a data type closely related to the list.

Unlike the list, the tuple is immutable, which opens the possibility of calculating *hash values* for tuples. This property is important for the use of tuples in combination with data types like the dictionary or the set (see Chapter 13), which are based on hash tables. Based on the condition that hash values can also be calculated for all elements of a tuple, it may therefore be used as a key in a dictionary or as an element in a set, which differentiates it from a list.

The concept of a tuple is related to that of the list, but due to its immutability, only the basic set of operations for sequential data types is available for `tuple` instances, as described in Section 12.2.

To create new `tuple` instances, you must use parentheses, which—like lists—contain the elements of the tuple, separated by commas:

```
>>> a = (1, 2, 3, 4, 5)
>>> a[3]
4
```

An empty tuple is defined by two parentheses `()` without content. A special case involves tuples with only one element. If you try to create a tuple with only one element in the way described previously, the program will behave differently than intended:

```
>>> no_tuple = (2)
>>> type(no_tuple)
<class 'int'>
```

With `(2)`, no new `tuple` instance is created because the parentheses are already used in this context for arithmetic operations with integers. Hence tuple literals containing only a single element are written with an additional comma after the element:

```
>>> a_tuple = (2,)
>>> type(a_tuple)
<class 'tuple'>
```

12.4.1 Packing and Unpacking

You can omit the enclosing parentheses in a `tuple` definition. Nevertheless, the references separated by commas are combined into a `tuple`, which is referred to as *tuple packing* or, more simply, *packing*:

```
>>> date = 7, 26, 1987
>>> date
(7, 26, 1987)
```

The `*` operator can be used to include the elements of a sequence during packing:

```
>>> date = 7, 26, 1987
>>> name = "Bill"
>>> date, name
((7, 26, 1987), 'Bill')
>>> *date, *name
(7, 26, 1987, 'B', 'i', 'l', 'l')
>>> *date, name
(7, 26, 1987, 'Bill')
```

Conversely, it's also possible to *unpack* the values of a tuple:

```
>>> date = 7, 26, 1987
>>> (month, day, year) = date
>>> month
7
>>> day
26
>>> year
1987
```

This process is referred to as *tuple unpacking* or simply *unpacking*, and again the parentheses can be omitted. By combining packing and unpacking, you can very elegantly swap the values of two variables without a helper variable or combine several assignments in one line:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Unpacking is not limited to the `tuple` data type; it works for sequential data types in general. In this case, the term *sequence unpacking* is also commonly used:

```
>>> a, b, c = "abc"
>>> a
'a'
```

If applied properly, the use of this feature can contribute to the readability of programs as the technical detail of caching data moves to the background of the actual intent of swapping values.

Unpacking can also be used to read values at the beginning and end of a sequence. Consider the following example:

```
>>> numbers = [11, 18, 12, 15, 10]
>>> eleven, *others, ten = numbers
>>> eleven
11
>>> ten
10
>>> others
[18, 12, 15]
```

If an asterisk (`*`) is prefixed to a reference when unpacking, all other values of the sequence are stored in it. In the preceding example, the first value of `numbers` is stored in `eleven` and the last value in `ten`. The numbers in between are collected into `others`.

Any number of other references may precede and follow the asterisked entry. In particular, the first or last entry can have an asterisk:

```
>>> numbers = [11, 17, 17, 19, 10]
>>> *something, nineteen, ten = numbers
>>> nineteen
19
>>> ten
10
>>> eleven, *blah_blah_blah = numbers
>>> eleven
11
>>> blah_blah_blah
[17, 17, 19, 10]
```

There can always only be exactly one reference with an asterisk in an assignment with unpacking. This makes sense, as otherwise ambiguities can arise.

Note

In general, you should be careful when using unpacking for unordered data types. In the following example, the order of the elements `1`, `2`, and `3` depends on the order in which the set `{3,1,2}` is iterated over:

```
>>> a, b, c = {3, 1, 2}
>>> a, b, c
(1, 2, 3)
```

Because this order is an implementation detail, it may differ between different versions of Python, or even between different runs of the same program. For more details on sets, see Chapter 13, Section 13.2.

12.4.2 Immutable Doesn't Necessarily Mean Unchangeable!

Although `tuple` instances are immutable, the values of the elements contained during their generation may change. When a new tuple is created, the references it is to store are specified. If such a reference points to an instance of a mutable data type, such as a list, its value can still change:

```
>>> a = ([],)
>>> a[0].append("And yet it moves!")
>>> a
(['And yet it moves!'],)
```

The immutability of a tuple thus refers only to the references it contains and explicitly not to the instances behind it.

Thus, the fact that tuples are immutable is no guarantee that elements won't change once the tuple has been created.

12.5 Strings: str, bytes, bytearray

This section describes the handling of strings in Python and, in particular, the properties of the `str`, `bytes`, and `bytearray` data types provided for this.

As you have already learned, strings are sequences of characters. This means that all operations for sequential types are available to them.[8]

8 Because `bytearray` is a mutable data type, unlike `str` and `bytes`, many of the methods of the `list` data type are supported in addition to the basic set of operations for sequential data types presented in this chapter. For a listing of the methods provided by `bytearray` as a mutable sequential data type, see Chapter 19, Section 19.11.3.

The characters an instance of the `str` data type can store are letters, punctuation, spaces, German umlauts, and other special characters. In contrast, the `bytes` and `bytearray` data types are intended for storing binary data. For this reason, instances of the `bytes` and `bytearray` data types consist of a sequence of individual bytes—that is, integers from 0 to 255.

For the time being, we'll deal only with `str` instances as dealing with `str` is not much different from dealing with `bytes`. Only when you convert `str` to `bytes` and vice versa are there some stumbling blocks, which are described in Section 12.5.10 about character sets and special characters.

To create new `str` instances, the following literals are available:

```
>>> string1 = "I was defined with double quotation marks"
>>> string2 = 'I was defined with single quotes'
```

The desired content of the string is written between the quotation marks, but it mustn't contain any line feeds (in the following example, Enter was pressed at the end of the first line):

```
>>> s = "First line
  File "<unknown>", line 1
    s = "First line
        ^
SyntaxError: unterminated string literal (detected at line 1)
```

String constants that can also extend over several lines are enclosed in `"""` or `'''`:[9]

```
>>> string3 = """First line!
... Ooops, another line"""
```

If two string literals immediately follow each other or are separated by spaces, Python joins them to form a string:

```
>>> string = "First part" "Second part"
>>> string
'First partSecond part'
```

As you can see in the example, the spaces between the literals are no longer present when concatenating.

This type of concatenation is very suitable for splitting long or unwieldy strings into several program lines without storing the line feeds and spaces in the result, as would be the case with strings containing `"""` or `'''`. To achieve this separation, you must write the string parts in parentheses:

9 You'll often come across strings in Python code that describe classes, functions, or modules. For these docstrings, the literal with three quotes is usually used. For more information on docstrings, see Chapter 37, Section 37.1.

```
>>> a = ("Imagine a terribly "
...      "complicated string that "
...      "in no case can be written in one "
...      "line.")
>>> a
'Imagine a terribly complicated string that in no case can be written in
one line.'
```

As you can see, the string was stored as if it had been defined in a single line.

The creation of `bytes` instances works the same way as the creation of `str` instances described previously. The only difference is that you have to prepend a lowercase `b` to the string literal to obtain a `bytes` string:

```
>>> string1 = b"I am bytes!"
>>> string1
b'I am bytes!'
>>> type(string1)
<class 'bytes'>
```

The other types of string creation work in a similar way as `bytes`. Note, however, that you may only use ASCII characters within `bytes` literals (Section 12.5.10).

To create a new instance of the `bytearray` type, you can use the built-in `bytearray` function:

```
>>> string1 = b"Hello world"
>>> string2 = bytearray(string1)
>>> string2
bytearray(b'Hello world')
```

This way, a `bytearray` instance can be created that adopts its value from an existing `bytes` instance.

If you pass an integer *k* as a parameter to the built-in `bytearray` function, a new `bytearray` of the length *k* is created, where each of the bytes is assigned the value zero:

```
>>> bytearray(7)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00')
```

In the next section, you'll learn what the `\x00` representation of these characters is all about. This topic will be discussed again in Section 12.5.10 in the context of string encoding.

The relationship between the `bytes` and `bytearray` data types is similar to the relationship between the `tuple` and `list` data types. Whereas the `bytes` data type represents an

immutable sequence of byte values, a `bytearray` may be modified with the same operations you already know from lists:

```
>>> b = bytearray(b"Hello world")
>>> b[:5] = b"Bye"
>>> b
bytearray(b'Bye world')
>>> b.append(ord(b"!"))
>>> b
bytearray(b'Bye world!')
>>> b.extend(b"!!!")
>>> b
bytearray(b'Bye world!!!!')
```

In the following sections, we will discuss several topics that shed further light on how to work with strings in Python. After discussing control characters in Section 12.5.1, we will discuss the various methods available when working with strings, grouped by topic. At the end of the chapter, we will deepen your knowledge of strings via more detailed sections on string formatting (Section 12.5.9) and on character sets and special characters (Section 12.5.10).

12.5.1 Control Characters

There are special text elements that control the text flow and can't be displayed on the screen as individual characters. These *control characters* include the line feed, the tabulator, and the backspace. Such characters are represented within string literals by means of special strings: *escape sequences*. Escape sequences are introduced by a backslash (\), followed by the identifier of the desired special character. For example, the `"\n"` escape sequence represents a line break:

```
>>> a = "First line\nSecond line"
>>> a
'First line\nSecond line'
>>> print(a)
First line
Second line
```

Note the difference between the output with `print` and without `print` in interactive mode: the `print` statement converts the control characters to their screen representation (i.e., `"\n"` starts a new line), while the output without the `print` statement displays a string literal with the escape sequences of the special characters on the screen.

For control characters, Python offers the escape sequences listed in Table 12.9.

Escape Sequence	Meaning
`\a`	Bell (BEL) generates a signal tone.
`\b`	Backspace (BS) resets the output position by one character.
`\f`	Formfeed (FF) creates a form feed.
`\n`	Linefeed (LF) sets the output position to the next line.
`\r`	Carriage return (CR) sets the output position to the beginning of the line.
`\t`	Horizontal tab (TAB) has the same meaning as the Tab key.
`\v`	The vertical tab (VT) is used for vertical indentation.
`\"`	Double quote.
`\'`	Single quote.
`\\`	Backslash that should really appear as such within the string.

Table 12.9: Escape Sequences for Control Characters

Control characters date from the time when output was mainly processed via printers. For this reason, some of these characters have little practical significance today.

The escape sequences for single and double quotes are necessary because Python uses these characters as delimiters for string literals. If the type of quote used to delimit a string is to occur as a character within that string, then the corresponding quote must be specified there as an escape sequence:

```
>>> a = "The following single quote does not need to be encoded ' "
>>> b = "This double quote does \" "
>>> c = 'This is also true in strings with single quotes " '
>>> d = 'Here an escape sequence must be used \' '
```

In Section 12.5.10, we'll return to escape sequences and use them to encode special characters such as umlauts or the euro sign.

The automatic replacement of escape sequences is sometimes annoying, especially if many backslashes are to occur in a string. For this purpose, Python provides the `r` and `R` prefixes, which can be prepended to a string literal. These prefixes mark the literal as a *raw string*, which results in all backslashes being transferred one by one into the result string:

```
>>> "A \tstring with \\ many \nescape sequences\t"
'A \tstring with \\ many \nescape sequences\t'
>>> r"A \tstring with \\ many \nescape sequences\t"
'A \\tstring with \\\\ many \\nescape sequences\\t'
```

```
>>> print(r"A \tstring with \\ many \nescape sequences\t")
A \tstring with \\ many \nescape sequences\t
```

As you can see from the double backslashes in the literal of the result and the output via `print`, the escape sequences were not interpreted as masked special characters.

By using the `rb` or `br` prefix, you can create raw strings of the `bytes` type.

Note

To facilitate the migration from Python 2 to Python 3, you can optionally create `str` instances with or without a prefixed `u`. These `u` literals were used in Python 2 to create `unicode` instances, the counterpart of the `str` data type in Python 3:

```
>>> u"The goshawk is the bird of the year 2015"
'The goshawk is the bird of the year 2015'
```

When we talk about *white spaces* ahead, we're referring to all kinds of characters between words that are not displayed as separate characters. White spaces are the characters listed in Table 12.10.

String Literal	Name
`" "`	Blank character
`"\n"`	Newline
`"\v"`	Vertical tab
`"\t"`	Horizontal tab
`"\f"`	Formfeed
`"\r"`	Carriage return

Table 12.10: List of White Space Characters

12.5.2 Splitting Strings

To split strings into several parts according to certain rules, the methods listed in Table 12.11 are available.

Method	Description
`s.split([sep, maxsplit])`	Splits `s` on occurrence of `sep`. The search starts at the beginning of the string.

Table 12.11: String Methods for Separating Strings

Method	Description
`s.rsplit([sep, maxsplit])`	Splits `s` on occurrence of `sep`. The search starts at the end of the string.
`s.splitlines([keepends])`	Splits `s` on occurrence of new lines.
`s.partition(sep)`	Splits `s` into two parts at the first occurrence of `sep`. In addition to the products of this separation, the resulting tuple contains the separator `sep`, if it occurs in `s`.
`s.rpartition(sep)`	Splits `s` into two parts at the last occurrence of `sep`. In addition to the products of this separation, the resulting tuple contains the separator `sep`, if it occurs in `s`.

Table 12.11: String Methods for Separating Strings (Cont.)

The `split` and `rsplit` methods split a string into its words and return them as a list. Here, the `sep` parameter specifies the string that separates the words, and `maxsplit` enables you to restrict the number of splits. If you don't specify `maxsplit`, the string will be split as many times as `sep` occurs in it. Any remainder is inserted as a string into the resulting list. Note that `split` starts splitting at the beginning of the string, while `rsplit` starts at the end:

```
>>> s = "1-2-3-4-5-6-7-8-9-10"
>>> s.split("-")
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
>>> s.split("-", 5)
['1', '2', '3', '4', '5', '6-7-8-9-10']
>>> s.rsplit("-", 5)
['1-2-3-4-5', '6', '7', '8', '9', '10']
```

If several separators follow each other, they won't be combined, but separated again each time:

```
>>> s = "1---2-3"
>>> s.split("-")
['1', '', '', '2', '3']
```

If `sep` isn't specified, the two methods behave differently. First all white spaces at the beginning and end of the string are removed, and then the string is split based on white spaces, this time combining consecutive separators into one:

```
>>> s = "  Any \t\t sentence with \n\r\t whitespaces"
>>> s.split()
['Any', 'sentence', 'with', 'whitespaces']
```

Calling `split` entirely without parameters is very useful to split a text string into its words, even if they aren't separated only by spaces.

The `splitlines` method splits a string into its individual lines and returns a list containing the lines. The `"\n"` Unix line feeds, `"\r\n"` Windows line feeds, and `"\r"` Mac line feeds are interpreted as separators:

```
>>> s = "Unix\nWindows\r\nMac\rLast line"
>>> s.splitlines()
['Unix', 'Windows', 'Mac', 'Last line']
```

If the separating line feeds at the ends of the lines are to be retained, the value `True` must be passed for the optional `keepends` parameter.

The `partition` method splits a string at the first place where the passed separator string `sep` occurs, and it returns a tuple consisting of the part before the separator string, the separator string itself, and the part after. The `rpartition` method works the same way, except that it uses the last occurrence of `sep` in the source string as the separator:

```
>>> s = "www.python-book.com"
>>> s.partition(".")
('www', '.', 'python-book.com')
>>> s.rpartition(".")
('www.python-book', '.', 'com')
```

12.5.3 Searching for Substrings

To determine the position and the number of occurrences of a string in another string or to replace parts of a string, the methods listed in Table 12.12 are available.

Method	Description
`s.find(sub, [start, end])`	Searches for the `sub` string in the string `s`. The search starts at the beginning of the string.
`s.rfind(sub, [start, end])`	Searches for the `sub` string in the string `s`. The search starts at the end of the string.
`s.index(sub, [start, end])`	Searches for the `sub` string in the string `s`. The search starts at the beginning of the string. If `sub` is not present in `s`, an exception is raised.
`s.rindex(sub, [start, end])`	Searches for the `sub` string in the string `s`. The search starts at the end of the string. If `sub` is not present in `s`, an exception is raised.
`s.count(sub, [start, end])`	Counts the occurrences of `sub` in `s`.

Table 12.12: String Methods for Searching in Strings

The optional `start` and `end` parameters of the five methods are used to narrow the search range. If you specify `start` or `end`, only the substring `s[start:end]` will be considered.

Note

Remember: When slicing a string `s` using `s[start:end]`, a substring is created that starts at `s[start]` and no longer contains the element `s[end]`.

To find out whether and, if so, at what position a particular string occurs in another, Python provides the `find` and `index` methods, along with their counterparts, `rfind` and `rindex`. The `find` method returns the index of the first occurrence of `sub` in `s`, while `rfind` correspondingly returns the index of the last occurrence. If `sub` isn't contained in `s`, `find` and `rfind` will return `-1`:

```
>>> s = "Let's see where the 'e' occurs in this string"
>>> s.find("e")
1
>>> s.rfind("e")
21
```

The `index` and `rindex` methods work in the same way, but they generate a `ValueError` if `sub` is not contained in `s`:

```
>>> s = "This string is about to be searched"
>>> s.index("is")
2
>>> s.index("not available")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    s.index("not available")
    ~~~~~~~^^^^^^^^^^^^^^^^
ValueError: substring not found
```

The reason for these almost identical methods is that error messages may be handled more elegantly than invalid return values.[10]

The number of occurrences of one substring within another can be determined via `count`:

```
>>> "Peter Piper picked a peck of pickled peppers".count("p")
7
```

12.5.4 Replacing Substrings

The methods listed in Table 12.13 can be used to replace certain parts or letters of strings with others.

10 You can find a more detailed description of this in Chapter 20.

Method	Description
`s.replace(old, new, [count])`	Replaces the occurrences of `old` in the string `s` with `new`.
`s.lower()`	Replaces all uppercase letters in `s` with corresponding lowercase letters.
`s.upper()`	Replaces all lowercase letters in `s` with corresponding uppercase letters.
`s.swapcase()`	Replaces all uppercase letters in `s` with corresponding lowercase letters and, vice versa, all lowercase letters with corresponding uppercase letters.
`s.capitalize()`	Replaces the first letter of `s` with the corresponding uppercase letter and all following uppercase letters with corresponding lowercase letters.
`s.casefold()`	Works similarly to `s.lower()` but additionally substitutes special characters. One such example is the German letter ß, which is substituted with ss. The behavior of `casefold` is defined in the Unicode standard. It's intended for comparisons between strings in which case is not important.
`s.title()`	Changes the case of `s` so that words are all lowercase, except for the respective initial letter.
`s.expandtabs([tabsize])`	Indents `s` by replacing tabs (`"\t"`) with white spaces.

Table 12.13: String Methods for Replacing Substrings

The `replace` method returns a string in which all occurrences of `old` have been replaced by `new`:

```
>>> false = "Python is not great!"
>>> true = false.replace("not", "really")
>>> true
'Python is really great!'
```

The `count` parameter can be used to limit the number of replacements:

```
>>> s = "Please replace only the first four e's"
>>> s.replace("e", "E", 4)
"PlEasE rEplacE only the first four e's"
```

An interesting special case arises when the empty string `""` is passed as the first argument `old`. Then the string passed as `new` is inserted between all characters and at the beginning and end:

```
>>> s = "abcdefg"
>>> s.replace("", "--")
'--a--b--c--d--e--f--g--'
```

The `lower` method replaces all uppercase letters of a string with the corresponding lowercase letters and returns the result string:

```
>>> s = "FIRST ALL BIG AND THEN ALL SMALL!"
>>> s.lower()
'first all big and then all small!'
```

With `upper`, you can achieve exactly the opposite effect.

The `swapcase` method changes the case of all letters in a string by replacing all uppercase letters with the corresponding lowercase letters and vice versa:

```
>>> s = "iF ALL eNGLISH WORDS WERE WRITTEN LIKE THIS ..."
>>> s.swapcase()
'If all English words were written like this ...'
```

The `capitalize` method returns a copy of the source string with the first character converted to an uppercase letter, if that's possible:

```
>>> s = "everything small ... yet ;)"
>>> s.capitalize()
'Everything small ... yet ;)'
```

The `title` method creates a string where the first letter of each word is uppercase and the remaining letters are lowercase, as is common in English for titles:

```
>>> s = "i Am nOt rEAlly a tITLe yET"
>>> s.title()
'I Am Not Really A Title Yet'
```

Using `expandtabs`, you can have all tab characters (`"\t"`) of a string replaced by spaces. The optional `tabsize` parameter specifies how many spaces should be inserted for a tab. If `tabsize` is not specified, then eight spaces will be used:

```
>>> s = ("\tThis could be source code\n" +
...      "\t\tOne level further down")
>>> print(s.expandtabs(4))
    This could be source code
        One level further down
```

12.5.5 Removing Prefixes and Suffixes

The `strip` methods allow you to remove unwanted characters at the beginning or end of a string (see Table 12.14).

Method	Description
`s.strip([chars])`	Removes occurrences of the characters from `chars` at the beginning and end of string `s`. If `chars` is not specified, `strip` removes whitespace characters such as spaces or tabs.
`s.lstrip([chars])`	Like `strip`, but only characters at the beginning of string `s` are removed.
`s.rstrip([chars])`	Like `strip`, but only characters at the end of string `s` are removed.
`s.removeprefix(prefix)`	Removes the string `prefix` at the beginning of string `s`.
`s.removesuffix(suffix)`	Removes the string `suffix` at the end of string `s`.

Table 12.14: String Methods for Removing Certain Characters at Beginning or End

The `strip` method removes unwanted characters at both sides of the string. The `lstrip` method removes only the characters on the left and `rstrip` only the characters on the right.

For the optional `chars` parameter, you can pass a string containing the characters to be removed. If you don't specify `chars`, all white spaces will be deleted:

```
>>> s = "    \t\n  \rSurrounded by whitespaces   \t\t\r"
>>> s.strip()
'Surrounded by whitespaces'
>>> s.lstrip()
'Surrounded by whitespaces   \t\t\r'
>>> s.rstrip()
'    \t\n  \rSurrounded by whitespaces'
```

For example, to remove all surrounding digits, you could proceed as follows:

```
>>> digits = "0123456789"
>>> s = "3674784673546Hidden between numbers3425923935"
>>> s.strip(digits)
'Hidden between numbers'
```

The `removeprefix` and `removesuffix` methods were added to the language scope with Python 3.9 and are thus only available in more recent language versions. You can remove a specified prefix or suffix from a string if it occurs:

```
>>> s = "PREFIX: This is my message (SUFFIX)"
>>> s.removeprefix("PREFIX: ")
'This is my message (SUFFIX)'
>>> s.removesuffix(" (SUFFIX)")
'PREFIX: This is my message'
>>> s.removesuffix("DOESNOTOCCUR")
'PREFIX: This is my message (SUFFIX)'
```

12.5.6 Aligning Strings

The methods listed in Table 12.15 create a string of a given length and align the source string in it in a particular way.

Method	Description
`s.center(width, [fillchar])`	Centers `s` in the resulting string
`s.ljust(width, [fillchar])`	Left-aligns `s` in the resulting string
`s.rjust(width, [fillchar])`	Right-aligns `s` in the resulting string
`s.zfill(width)`	Right-aligns `s` by filling in zeros on the left

Table 12.15: String Methods for Alignment

You can use the `width` parameter to specify the desired length of the new string. The optional `fillchar` parameter of the first three methods must be a string of length 1 and specifies the character to be used for filling up to the passed length. By default, spaces are used for filling:

```
>>> s = "Align me"
>>> s.center(50)
'                     Align me                     '
>>> s.ljust(50)
'Align me                                          '
>>> s.rjust(50, "-")
'------------------------------------------Align me'
```

If the length of `s` is greater than the value of the `width` parameter, a copy of `s` is returned because, in this case, there is not enough space for alignment.

The `zfill` method is a special case of `rjust` and is intended for strings that contain numeric values. Calling the `zfill` method creates a string of length `width` in which the source string is right-aligned and the left-hand side is filled with zeros:

```
>>> "13.37".zfill(20)
'00000000000000013.37'
```

12.5.7 String Tests

The methods listed in Table 12.16 return a truth value that states whether the contents of the string have a particular property. For example, via `islower`, you can check if all letters in `s` are lowercase.

Method	Description
`s.isalnum()`	`True` if all characters in `s` are letters or digits
`s.isalpha()`	`True` if all characters in `s` are letters
`s.isascii()`	`True` if `s` contains only characters from the ASCII character set*
`s.isdigit()`	`True` if all characters in `s` are digits
`s.islower()`	`True` if all letters in `s` are lowercase
`s.isupper()`	`True` if all letters in `s` are uppercase
`s.isspace()`	`True` if all characters in `s` are white spaces
`s.istitle()`	`True` if all words in `s` are capitalized
`s.startswith(prefix, [start, end])`	`True` if `s` starts with the string `prefix`
`s.endswith(suffix, [start, end])`	`True` if `s` ends with the string `suffix`

* In Section 12.5.10, you'll learn more about character sets and their meaning for special characters in strings.

Table 12.16: Methods for String Tests

Because the first seven methods in the table are very similar, one example should suffice at this point:

```
>>> s = "1234abcd"
>>> s.isdigit()
False
>>> s.isalpha()
```

```
False
>>> s.isalnum()
True
```

To check whether a string begins or ends with a certain string, you can use the `startswith` or `endswith` methods. The optional `start` and `end` parameters limit the query to the range `s[start:end]`—as was already the case with the search-and-replace methods:

```
>>> s = "www.python-book.com"
>>> s.startswith("www.")
True
>>> s.endswith(".com")
True
>>> s.startswith("python", 4)
True
```

12.5.8 Concatenating Elements in Sequential Data Types

Concatenating a list of strings using a separator is a common task. For this purpose, Python provides the `join` method (see Table 12.17).

Method	Description
`s.join(seq)`	Concatenates the elements of the `seq` sequence into a new string in which `s` is the separator

Table 12.17: String Method for Concatenating Multiple Elements with Separator String

The `seq` parameter can be any iterable object, but all its elements must be strings. The elements of `seq` are concatenated with `s` as a separator. In the following example, several names are concatenated, separated by commas:

```
>>> contact_list = ["Mickey", "Minnie", "Donald", "Daisy"]
>>> ", ".join(contact_list)
'Mickey, Minnie, Donald, Daisy'
```

If a string is passed for `seq`, the result will be the concatenation of all letters, each separated by `s`:

```
>>> sentence = "Unintelligible sentence"
>>> "...um...".join(sentence)
'U...um...n...um...i...um...n...um...t...um...e...um...l...um...l...um..
.i...um...g...um...i...um...b...um...l...um...e...um... ...um...s...um
...e...um...n...um...t...um...e...um...n...um...c...um...e'
```

The `join` method is often used to concatenate the elements of a sequence without a separator. In this case, you call the `join` method of the empty string:

```
>>> "".join(["www", ".", "python-book", ".", "com"])
'www.python-book.com'
```

The following section deals with the topic of string formatting.

12.5.9 Formatting Strings

You will often want to customize your screen outputs in a certain way. For example, to display a three-column table of numbers, you must insert spaces—depending on the length of the numbers—so that the items in the individual columns are displayed underneath each other. Customizing the output is also necessary if you want to output an amount of money stored in a `float` instance that has more than two decimal places.

To solve these kinds of problems, you can use the `format` method of the `str` data type. Using `format`, you can replace placeholders in a string with specific values. These placeholders are enclosed by curly brackets and can be both numbers and names. In the following example, we'll replace the placeholders `{0}` and `{1}` with two numbers:

```
>>> "It's {0}:{1}".format(13, 37)
"It's 13:37"
```

If numbers are used as placeholders, they must be numbered consecutively, starting at 0. Then they are replaced in order by the parameters passed to the `format` method—the first parameter replaces `{0}`, the second parameter replaces `{1}`, and so on.

It's also possible to have this numbering done implicitly by not writing anything between the curly brackets. Python then numbers the placeholders automatically, starting at `0`:

```
>>> "It's {}:{}".format(13, 37)
"It's 13:37"
```

Names can also be used as placeholders. In this case, you must pass the values as keyword parameters to the `format` method:

```
>>> "It's {hour}:{minute}".format(hour=13, minute=37)
"It's 13:37"
```

All character strings that can also be used as variable names in Python can be used as names for the placeholders. In particular, your placeholder names shouldn't start with digits; otherwise, Python will try to interpret them as integers.

You can also mix numbered placeholders with symbolic placeholders:

```
>>> "It's {hour}:{0}".format(37, hour=13)
"It's 13:37"
```

This mixing of symbolic and numbered placeholders also works together with implicit numbering. In that case, Python numbers the placeholders automatically, starting at `0`:

```
>>> "{h}oz. yeast, {}oz. flour, {w}fl.oz. water, {}oz. salt".format(
...     2, 15, h=0.2, w=3)
'0.2oz. yeast, 2oz. flour, 3fl.oz. water, 15oz. salt'
```

Instead of the numerical values used in the previous examples, you can generally use any objects as values, so long as they can be converted to a string.[11] In the following code snippet, different types of data are passed to the `format` method:

```
>>> "List: {0}, String: {string}, Complex number: {1}".format(
...     [1,2], 13 + 37j, string="Hello world")
'List: [1, 2], String: Hello world, Complex number: (13+37j)'
```

To insert the same value multiple times in a string, you can use a placeholder multiple times:

```
>>> "{h}{um} yeast, {}{um} flour, {w}{uv} water, {}{um} salt".format(
...     2, 15, h=0.2, w=3, um='oz.', uv='fl.oz.')
'0.2oz. yeast, 2oz. flour, 3fl.oz. water, 15oz. salt'
```

Here `um` and `uv` stand for *unit for mass* and *unit for volume*.

If you want to prevent a curly bracket from being interpreted as a delimiter of a placeholder, you must use two brackets in a row. As a result, these double brackets are replaced by single ones:

```
>>> "Unformatted: {{NoPlaceholder}}. Formatted: {v}.".format(
...     v="only a test")
'Unformatted: {NoPlaceholder}. Formatted: only a test.'
```

Note

The `format` method and the closely related f-strings are now the standard for string formatting, replacing the `%` formatting operator commonly used in older versions of the language. Although the `%` operator still works for downward compatibility reasons, its use is not recommended; f-strings or the `format` method should be used instead.

For more details on how the `%` operator works, you can refer to Python's online documentation.

11 For more details on how this conversion works internally and how it can be influenced, see Chapter 19, Section 19.11.

String Interpolation: f-Strings

In Python 3.6, a special string literal was introduced that further simplifies the formatting of strings. As an example, let's define a time by creating the `hour` and `minute` variables:

```
>>> hour = 13
>>> minute = 37
```

You already know two ways to format the time as a string using the `format` string method:

```
>>> "It's {}.{}".format(hour, minute)
"It's 13.37"
>>> "It's {hour}.{minute}".format(hour=hour, minute=minute)
"It's 13.37"
```

In the first variant, the values are inserted into the string based on their position in the parameter list of the `format` call. This implicit assignment of placeholders to values results in compact code, but it can become confusing in more complex cases, especially when you insert new placeholders later. The second variant establishes an explicit assignment between placeholders and variables via a temporary identifier, which we also called `hour` or `minute` in the preceding example. This results in unambiguous, but also inconveniently redundant code.

Using *f-strings*, you can formulate the example unambiguously and compactly at the same time. An f-string is the special string literal `f""`, which automatically replaces placeholders with instances of the same name:

```
>>> f"It's {hour}:{minute}"
"It's 13:37"
```

This technique is also known as *string interpolation*. Another important property of f-strings is that any expressions may be placed inside the placeholders, and their value is then inserted into the string at this point:

```
>>> f"It's soon {hour + 1}:{minute + 1}"
"It's soon 14:38"
>>> f"{60 * hour + minute} minutes of the day have already passed"
'817 minutes of the day have already passed'
>>> f"It's around {hour if minute < 30 else hour + 1}:00"
"It's around 14:00"
>>> f"It's {bin(hour)}:{bin(minute)}"
"It's 0b1101:0b100101"
```

Note

In older versions of Python, expressions allowed within f-strings were subject to syntactic restrictions. For example, it was not possible to use a string literal with the same

quotation marks that were used for the enclosing f-string itself. Multiline expressions, comments, and escape sequences also were not allowed.

With Python 3.12, these restrictions were lifted, so any valid Python expression is now allowed within f-strings:

```
>>> f"{", ".join([
...     "A",  # This is
...     "B",  # just a
...     "C",  # multi-line test
... ])}"
'A, B, C'
```

Ahead, we will consistently use f-strings in our examples, which is common in modern Python code.

Accessing Attributes and Elements

Beyond simply replacing placeholders, you can also access attributes of the passed value in the string format. To do this, you must write the respective attribute, separated by a period, after the placeholder name, just as you would for normal attribute access in Python.

The following example outputs the imaginary and real parts of a complex number in this way:

```
>>> c = 15 + 20j
>>> f"Real part: {c.real}, Imaginary part: {c.imag}"
'Real part: 15.0, Imaginary part: 20.0'
```

As you can see, attribute access also works with numbered placeholders.

In addition to accessing attributes of the value to be formatted, the `[]` operator also can be used. For example, it enables you to output specific elements of a list:

```
>>> my_list = ["I'm first!", "No, I'm first!"]
>>> f"{my_list[1]}. {my_list[0]}"
"No, I'm first!. I'm first!"
```

Even if you haven't learned about other data types that support the `[]` operator at this point, its use in format strings is not limited to sequential data types. In particular, this type of access is interesting in the case of dictionaries, which we'll describe in Chapter 13, Section 13.1.

When using explicit or implicit numbering of placeholders in the `format` method, access to attributes and the `[]` operator continue to work:

```
>>> "Attribute: {.imag}, list element: {[1]}".format(1+4j, [1,2,3])
'Attribute: 4.0, list element: 2'
```

```
>>> "Attribute: {0.imag}, list element: {1[1]}".format(1+4j, [1,2,3])
'Attribute: 4.0, list element: 2'
```

The following sections describe how you can influence the replacement itself.

Formatting the Output

Up to this point, we've only used format strings to replace placeholders with specific values, without specifying the rules according to which the replacement is to be made. To do that, you can specify *format specifiers* separated from the placeholder name by a colon. For example, to output a float rounded to two decimal places, you would use the format specification `.2f`:

```
>>> amount = 13.37690
>>> f"Amount: {amount:.2f} €"
'Amount: 13.38 €'
```

The effect of the format specification depends on the data type that is passed as the value for the respective placeholder. We'll take a closer look at the formatting options for Python's built-in data types ahead.

Note that all formatting specifications are optional and independent of each other. For this reason, they can also occur individually.

Before we look at the formatting options in detail, we'd like to briefly show you the basic structure of a format specification:

```
[[fill]align][sign][z][#][0][minimumwidth][,][.precision][type]
```

The square brackets indicate that their content is optional. All these fields will be described individually in the following sections.

Setting the Minimum Width: minimumwidth

If a simple integer is used as format specification, it specifies the minimum width the replaced value should have. For example, if you want to output a table of names and make sure that everything is flush underneath, you can achieve this as follows:

```
>>> table = [
...     ("First name", "Last name"),
...     ("Daniel", "Rodriguez"),
...     ("Linda", "Moore"),
...     ("Steve", "Bishop"),
...     ("Kaylee", "Harris"),
... ]
>>> for line in table:
...     print(f"{line[0]:15}|{line[1]:15}")
```

In this tiny program, we format the two columns with a width of 15 characters each. The output thus looks as follows:

```
First name     |Last name
Daniel         |Rodriguez
Linda          |Moore
Steve          |Bishop
Kaylee         |Harris
```

If a value is longer than the minimum width, the width of the inserted value is adjusted to the value and not cut off:

```
>>> long = "I'm longer than two characters!"
>>> f"{long:2}"
"I'm longer than two characters!"
```

Determining the Alignment: align

When you specify the minimum width of a field, you can determine the alignment of the value in case it doesn't fill the entire width—as was the case in the first of the previous two examples.

For example, to align an amount of money to the right as usual, you must place a > sign in front of the minimum width:

```
>>> final_price = 443
>>> f"Final price: {final_price:>5} €"
'Final price:   443 €'
```

Table 12.18 describes four alignment types.

Sign	Meaning
<	The value is inserted left-justified in the reserved space. This is the default behavior unless an alignment is specified.
>	The value is inserted right-justified in the reserved space.
=	Makes sure that for numeric values the sign is always at the beginning of the inserted value; only after that does an alignment to the right take place (see the example ahead). This specification is only useful for numeric values and causes a ValueError for other data types.
^	The value is inserted centered in the reserved space.

Table 12.18: Alignment Types

An example should clarify the not quite intuitive effect of the `=` alignment type. The position of the sign is interesting here:

```
>>> t = -12.5
>>> f"Temperature: {t:10}"
'Temperature:      -12.5'
>>> f"Temperature: {t:=10}"
'Temperature: -     12.5'
```

Note that an alignment specification has no effect if the inserted value is as long as or longer than the relevant minimum width.

Fill Characters: fill

Before you specify the alignment, you can specify the character you want to use for filling the excess characters during the alignment process. By default, the space character is used for this purpose. However, you can use any character you like:

```
>>> text = "Hello world"
>>> f"{text:-^25}"
'-------Hello world-------'
```

Here, the string `"Hello World"` was inserted centered and surrounded by minus signs.

Treating Signs: sign and z

Between the minimum width specification and the alignment specification, you can specify how to deal with the sign of a numeric value. The three possible formatting characters are described in Table 12.19.

Sign	Meaning
`+`	A sign must be indicated for both positive and negative numerical values.
`-`	This sign is indicated only for negative numbers. This is the default behavior.
Blank character	Using the blank character, you can ensure that a space is inserted instead of a sign for positive numerical values. Negative numbers get a minus as the sign with this setting.

Table 12.19: Different Types of Treating Signs

Let's demonstrate the handling of signs with a few simple examples:

```
>>> value = 135
>>> f"Cost: {value:+} €"
'Cost: +135 €'
```

```
>>> f"Cost: {-value:+} €"
'Cost: -135 €'
>>> f"Cost: {value:-} €"
'Cost: 135 €'
>>> f"Cost: {value: } €"
'Cost:  135 €'
>>> f"Cost: {-value: } €"
'Cost: -135 €'
```

As already mentioned, the `=` alignment specification only makes sense when used with a sign:

```
>>> f"Cost: {-value:=+10} €"
'Cost: -      135 €'
```

As you can see, in the preceding example, the minus sign is inserted at the beginning of the reserved space; only after that is the number `135` aligned to the right.

In addition to formatting signs via `sign`, Python 3.11 allows you to use format specification `z` to control which sign should be displayed for a "negative zero." Such a negative zero value occurs when a negative floating-point number is rounded to zero:

```
>>> very_small = -0.001
>>> f"{very_small:.2f}"
'-0.00'
```

If `z` is specified, then a positive sign is assumed in such cases:

```
>>> f"{very_small:z.2f}"
'0.00'
```

Types of Number Representation: type

To be able to further customize the output for numerical values, there are various output types that are inserted at the very end of the format specification. For example, the `b` type specification outputs integers in binary notation:

```
>>> f"Funny Bits: {109:b}"
'Funny Bits: 1101101'
```

Python provides a total of eight possible type specifications for integers, which are listed in Table 12.20.

Sign	Meaning
`b`	The number is output in binary notation.

Table 12.20: Integer Output Types

Sign	Meaning
c	The number is interpreted as a Unicode character. For more information on Unicode, see Section 12.5.10.
d	The number is output in decimal notation. This is the default behavior.
o	The number is output in octal notation.
x	The number is output in hexadecimal notation, using lowercase letters for the digits a to f.
X	Like x, but with uppercase letters for the digits from A to F.
n	Like d, but it tries to use the usual character for the region to separate numbers (e.g., a dot versus a comma as a thousands separator).

Table 12.20: Integer Output Types (Cont.)

There's another alternative mode for the output of integers, which you can activate by writing a hash (#) between the minimum width and the sign. In this mode, outputs in numeral systems with bases other than 10 are identified by appropriate prefixes:

```
>>> value = 109
>>> f"{value:#b} vs. {value:b}"
'0b1101101 vs. 1101101'
>>> f"{value:#o} vs. {value:o}"
'0o155 vs. 155'
>>> f"{value:#x} vs. {value:x}"
'0x6d vs. 6d'
```

There are also various output types for floats, which are listed in Table 12.21.

Sign	Meaning
e	The number is output in scientific notation, using a lowercase *e* to separate the mantissa and exponent.
E	Like e, but with a capital *E* as separator.
f	The number is output as a decimal number with a decimal point.
g	The number is output as for f if it isn't too long. For numbers that are too long, the e type is automatically used.
G	Like g, except that the E type is used for numbers that are too long.
n	Like g, but it tries to use a separator adapted to the region.

Table 12.21: Output Types for Floats

Sign	Meaning
`%`	The numerical value is first multiplied by 100 and then output, followed by a percent sign.
Not specified	Like `g`, but at least one decimal place is specified.

Table 12.21: Output Types for Floats (Cont.)

The following example illustrates the formatting for floats:

```
>>> number = 123.456
>>> f"{number:e}"
'1.234560e+02'
>>> f"{number:f}"
'123.456000'
>>> f"{number:n}"
'123.456'
>>> f"{number / 1000:%}"
'12.345600%'
```

Precision with Floats: precision

It's possible to specify the number of decimal places when outputting floats. To do this, you write the relevant number, separated by a period, between the minimum length and the output type, as we did in our introductory example:

```
>>> value = 13.37690
>>> f"Amount: {value:.2f} €"
'Amount: 13.38 €'
```

The excess decimal places won't be truncated during formatting but rounded.

Note that in this example the minimum length hasn't been specified, and therefore the format specification starts with a period.

As a final formatting option, `0` can be inserted just before the minimum width. This zero causes the excess space to be filled with zeros and the sign to be inserted at the beginning of the reserved space. Thus, this mode is equivalent to the alignment type `=` and the fill character `0`:

```
>>> number = 23
>>> f"The following holds true: {number:05} == {number:0=5}."
'The following holds true: 00023 == 00023.'
```

Thousands Separation: The , and _ Options

If the , option is set, then blocks of thousands are separated by commas. Meanwhile, _ allows you to set the underscore as a thousand separator:

```
>>> amount = 12345678900
>>> f"A lot of money: {amount:,d} €"
'A lot of money: 12,345,678,900 €'
>>> f"A lot of money: {amount:_d} €"
'A lot of money: 12_345_678_900 €'
```

The thousand separators can also be specified when formatting floating point numbers. Since Python 3.14, different separators can be specified for the integer and fractional parts:

```
>>> amount = 12345.678900
>>> f"Less money: {amount:,f} €"
'Less money: 12,345.678900 €'
>>> f"Less money: {amount:_f} €"
'Less money: 12_345.678900 €'
>>> f"Less money: {amount:_._f} €"
'Less money: 12_345.678_900 €'
>>> f"Less money: {amount:_.,f} €"
'Less money: 12_345.678,900 €'
>>> f"Less money: {amount:,.,f} €"
'Less money: 12,345.678,900 €'
```

Self-Documenting Expressions in f-Strings

Python 3.8 introduced *self-documenting expressions* in f-strings, which are used to easily format debug outputs. Such outputs are inserted when testing programs to observe the values of certain variables. A typical debug output might look like this:

```
>>> variable_1 = 12
>>> variable_2 = 17
>>> print(f"variable_1={variable_1}, variable_2={variable_2}")
variable_1=12, variable_2=17
```

Such outputs are often inserted into the program code for a short time during debugging and removed later. For this reason, it can be very annoying to have to write the variable names twice for a clean debug output: once for the output of the variable name itself and once for the output of the value of the variable.

A self-documenting expression provides help in this context. It makes it possible to formulate the preceding example in a compact way:

```
>>> print(f"{variable_1=}, {variable_2=}")
variable_1=12, variable_2=17
```

Note the use of the equal sign, which indicates a self-documenting expression, instead of the usual colon in the placeholder. In the formatted result, in addition to the values of the variables, their identifiers are automatically inserted so that it's always clear how the output values are to be assigned.

As with regular placeholders in f-strings, self-documenting expressions don't necessarily have to consist of only one identifier, as the following example shows:

```
>>> import math
>>> f"{math.cos(math.pi)=}"
'math.cos(math.pi)=-1.0'
```

12.5.10 Character Sets and Special Characters

In the previous sections, you learned about the `str` data type for text and `bytes` data type for byte sequences. These two data types are very similar, not least because both can be interpreted as a string and instantiated via a corresponding literal:

```
>>> "Python"
'Python'
>>> b"Python"
b'Python'
```

In this case, we have instantiated the `Python` string once as text and once as a byte sequence, which at first seems to make no difference. Internally, however, very different representations of the string are generated in the two cases, which you'll notice in particular if your string contains special characters:

```
>>> "Püthøn"
'Püthøn'
>>> b"Püthøn"
  File "<unknown>", line 1
    b"Püthøn"
    ^^^^^^^^^
SyntaxError: bytes can only contain ASCII literal characters
```

Whereas `str` literals may contain any special characters without a problem, this is apparently not possible for `bytes` literals without further ado. Also, the conversion between `str` and `bytes` instances isn't possible automatically—which is why, for example, a `bytes` string can't be appended to a `str` string:

```
>>> "P" + b"ython"
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    "P" + b"ython"
    ~~~~^~~~~~~~~~
TypeError: can only concatenate str (not "bytes") to str
```

In general, text data in Python programs should be represented by `str` instances whenever possible as these can represent any special character without a problem. Nevertheless, it's often necessary to represent strings explicitly as byte sequences. This is especially the case when text data is read or output via binary interfaces. Such binary interfaces can be enforced, for example, by file operations, direct memory access, network connections, databases, or external libraries.

In Figure 12.4, this principle is illustrated using two Python programs that exchange strings via a byte-oriented communication channel, such as a network connection. Internally, both programs use `str` instances and can thus process any special characters. For communication, the sender must *encode* its strings into byte sequences, while the receiver *decodes* the received byte sequences back into `str` instances. For this process, both sides agree on the `"utf8"` encoding.

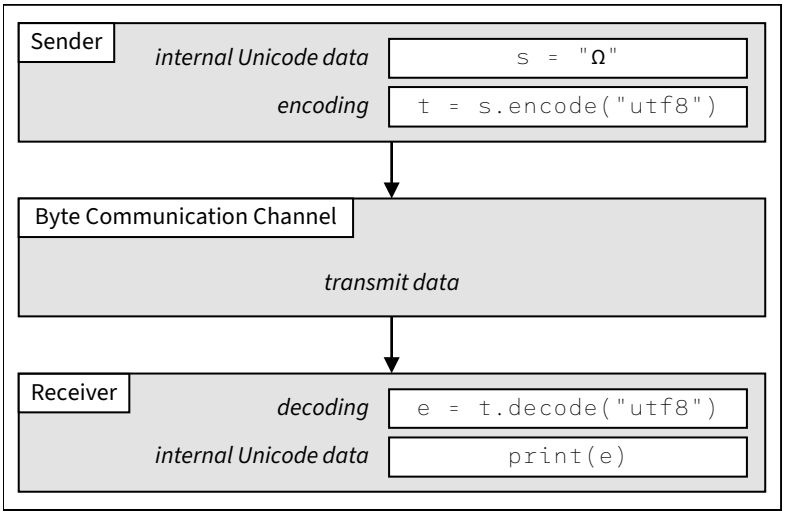


Figure 12.4: Encoding and Decoding Character Strings for Transmission Purposes in Byte-Oriented Communication Channels

In this section, we'll first look at how the `str` and `bytes` data types represent strings internally, then discuss how a conversion between the two worlds is possible.

Character Encoding

First, we want to develop an idea of how a computer handles strings internally. To start, it can be said that a computer doesn't actually know any characters at all as only bits and numerical values based on them can be represented in its memory. To still produce screen output or perform other operations with characters, defined translation tables are available, called *codepages*, which assign a certain number to each letter. This assignment means that, in addition to letters and digits, punctuation marks and special characters are also mapped. There are also nonprintable *control characters*, such as the tabulator or the line feed (see Table 12.9).

The best-known and most important character set is the *ASCII character* set,[12] which implements a seven-bit character encoding. This means that each character utilizes seven bits of memory; therefore, a total of 27 (128) different characters can be mapped. The definition of the ASCII character set is based on the alphabet of the English language, which doesn't contain special characters such as *ä*, *ø*, or *ü*. To be able to represent such special characters, the ASCII code was extended by increasing the storage space for a character by one bit so that 28 (256) different characters can be stored.

This extension of the ASCII character set results in space for 128 additional special characters. The codepage used determines which interpretation these further places have in concrete terms. Which codepage is to be used depends on the configuration of the respective computer. In particular, operating system and regional settings affect the selection of the codepage.

Encoding and Decoding

The `bytes` data type represents a sequence of bytes that, depending on the codepage used, can represent a character string. Python allows the programmer to instantiate `bytes` strings directly via a special literal, provided that the characters contained in the literal are limited to the ASCII character set as the lowest common denominator of all character sets. This explains the behavior observed previously when instantiating `bytes` strings with and without special characters:

```
>>> b"Python"
b'Python'
>>> b"Püthøn"
  File "<unknown>", line 1
    b"Püthøn"
    ^^^^^^^^^
SyntaxError: bytes can only contain ASCII literal characters
```

If you still want to represent the string `Püthøn` as a `bytes` string, such as to output it as part of a binary I/O operation, you must first create a `str` string and then *encode* it. This can be done using the `encode` method of the `str` data type, which requires the character set to be used for encoding to be specified when it's called:

```
>>> str_string = "Püthøn"
>>> bytes_string = str_string.encode("iso-8859-15")
>>> bytes_string
b'P\xfcth\xf8n'
```

In the `iso-8859-15` encoded `bytes` string, all characters not included in the ASCII character set are represented as escape sequences. These are introduced by the `\x` prefix followed by a two-digit hexadecimal number indicating the numeric value of the encoded character.

12 ASCII is an abbreviation for American Standard Code for Information Interchange.

In the preceding example, the special characters ü and ø are represented with the escape sequences \xfc and \xf8, which stand for the numeric decimal values 252 and 248.

In the process of *decoding*, a bytes string is transformed into a str string using the decode method. Also in this case, the character set to be used must be specified:

```
>>> bytes_string.decode("iso-8859-15")
'Püthøn'
```

Figure 12.5 summarizes the processes of encoding and decoding and the corresponding conversions between str and bytes.

The iso-8859-15 codepage (also referred to as *Latin-9*) used in this case covers all important characters for Western Europe, such as umlauts, accents, and the euro sign. In addition to iso-8859-15, there is a plethora of codepages with different objectives and widely varying degrees of dissemination. Modern operating systems are usually based on Unicode and the associated UTF-8 codepage, which we'll look at in more detail in the following sections.

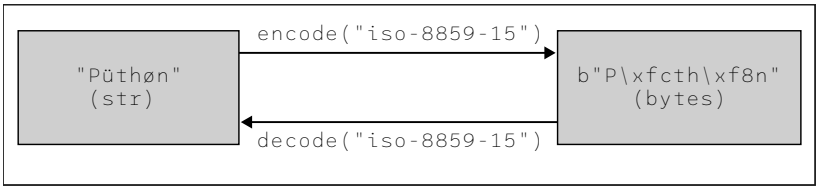


Figure 12.5: Encoding and Decoding of Character String

The Universal Character Set Unicode

A major drawback of the character-encoding approach described so far is the limitation on the number of characters. Imagine a string containing an elaboration on authors from different language areas with original citations. You would quickly reach the limit of eight-bit encoding due to the many different alphabets and wouldn't be able to digitize the work. Or imagine you wanted to encode a text in Chinese: This would become impossible due to the more than 100,000 characters in that language.

An obvious solution to this problem is to increase the memory space per character, but this introduces new drawbacks. For example, if you use 16 bits for each character, then the number of characters is still limited to 65,536. It can be assumed that languages will continue to develop and that this number will therefore no longer be sufficient in the long run.[13] Furthermore, in the 16-bit example, the memory requirements for a string would double because twice as many bits would be used for each character as with extended ASCII encoding, and this despite the fact that a large part of all texts consists mainly of a small subset of all available characters.

13 It is indeed the case that 16 bits are already insufficient to encode all the characters of written language.

A long-term solution to the encoding problem was finally devised by the Unicode standard, which provides variable encoding lengths for individual characters. In principle, *Unicode* is a huge table that assigns a number, called a *code point*, to each known character. This table is maintained and constantly expanded by the Unicode Consortium, a nonprofit institution. Code points are usually written as "U+*x*", where *x* is the hexadecimal representation of the code point.

What's really new about Unicode is the *Unicode Transformation Format* (UTF) procedure, which can represent code points by byte sequences of different lengths. There are several of these transformation formats, but the most important and widely used one is *UTF-8*. UTF-8 uses up to seven bytes to encode a single character, with the actual length depending on the frequency of the character in texts. For example, all characters of the ASCII standard can be encoded with one byte each, which has the same numerical value as the corresponding ASCII encoding of the character. This procedure ensures that every string encoded with ASCII is also a valid UTF-8 code as UTF-8 is compatible with ASCII. At this point, we won't look further into the technical implementation of Unicode; rather, we'll examine how you can use Unicode in Python.

Unicode in Python

In Python, there's a clear separation between binary data (`bytes` data type) and text data (`str` data type) at the level of data types for strings. So long as you work with text data, with the `str` data type, you usually don't need to worry about character encoding. The explicit use of a special character within a string literal via its Unicode code point is an exception.

As you've already seen in the example at the beginning of this section, you can encode special characters in string literals using escape sequences. We used escape sequences that start with `\x`. However, these sequences are only suitable for characters that use one of the first 256 code points. For any Unicode character, such as the euro symbol (€; code point 8364, hexadecimal 0x20ac), there are escape sequences that are introduced with `\u`:

```
>>> s = "\u20ac"
>>> print(s)
€
```

In addition to the code point, a Unicode character has a unique name by which it can be identified. The `\N` escape sequence allows you to use special characters via their Unicode name in string literals:

```
>>> "\N{euro sign}"
'€'
```

In the example, the `\N` escape sequence was used to insert the euro sign based on its Unicode name, *euro sign*, within a string literal.

Note

The popular emojis used in chat applications are also defined in the Unicode standard and can be used in Python programs via their respective code points or via their unique names.

The following code enables you to decorate your program outputs with a beautiful Python emoji, for example:

```
>>> "\N{Snake}"
'🐍'
```

Note that the specific appearance of an emoji is not standardized and depends on the system font used in each case.

The built-in `chr` and `ord` functions enable you to convert Unicode code points and the corresponding characters into each other:

```
>>> chr(8364)
'€'
>>> ord("€")
8364
```

For encoding or decoding with UTF-8, the `"utf-8"` character set is specified when calling the `encode` and `decode` methods:

```
>>> str_string ="Püthøn"
>>> bytes_string = str_string.encode("utf-8")
>>> bytes_string
b'P\xc3\xbcth\xc3\xb8n'
>>> bytes_string.decode("utf-8")
'Püthøn'
```

Other Character Sets

So far, we've only dealt with the two encoding methods: `"iso-8859-15"` and `"UTF-8"`. There are quite a few other methods besides these two, many of which are natively supported by Python. Each of these encodings has a name in Python that you can pass to the `encode` and `decode` methods. Table 12.22 shows a few of these names as examples.

Name in Python	Properties
`"ascii"`	Encoding using the ASCII table; English alphabet, English digits, punctuation marks and control characters; one byte per character

Table 12.22: Four Encodings Supported by Python

Name in Python	Properties
`"utf-8"` or `"utf8"`	Encoding for all Unicode code points; downward compatible with ASCII; variable number of bytes per character
`"iso-8859-15"`	Encoding of the characters for Western Europe as standardized by the International Organization for Standardization (ISO) in 1999
`"cp1252"`	Encoding for Western Europe used by older Windows versions; in addition to ASCII characters, support for European special characters, especially the euro sign; downward compatible with ASCII; one byte per character

Table 12.22: Four Encodings Supported by Python (Cont.)

Note that Unicode and UTF-8 have now become the standard in many places and have displaced other character encodings. Thanks to this, developers can worry increasingly less about adequate character encoding than they had to in the past.

If you try to decode a `bytes` string with the wrong character set, the result depends on whether or not the code points used in the string are also defined in the character set used for decoding. If they are, nonsense will be decoded; if not, a `UnicodeDecodeError` will be generated:

```
>>> bytes_string.decode("utf-8")
'Püthøn'
>>> bytes_string.decode("iso-8859-15")
'PÃŒthÃžn'
>>> bytes_string.decode("ascii")
Traceback (most recent call last):
  File "<python-input-6>", line 1, in <module>
    bytes_string.decode("ascii")
    ~~~~~~~~~~~~~~~~~~~^^^^^^^^^
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 1:
ordinal not in range(128)
```

The `encode` and `decode` methods provide an optional `errors` parameter that defines the procedure to follow in such error cases. The most important values for `errors` are summarized in Table 12.23.

Value	Meaning
`"strict"`	Default setting. Any character that can't be encoded will cause an error.

Table 12.23: Values for errors Parameter of Encode and Decode

Value	Meaning
`"ignore"`	Characters that can't be encoded are ignored.
`"replace"`	Characters that can't be encoded are replaced by a placeholder: during encoding by the question mark `"?"` and during decoding by the `U+FFFD` Unicode character.
`"xmlcharrefreplace"`	Nonencodable characters are replaced with their XML entity.* (Only possible with `encode`.)
`"backslashreplace"`	Nonencodable characters are replaced by a backslashed `\x` escape sequence.

* This is a special formatting method for displaying special characters in XML files. For more information about XML files, see Chapter 32, Section 32.6.

Table 12.23: Values for errors Parameter of Encode and Decode (Cont.)

Let's look again at the decoding of the `bytes` string `bytes_string` from the last example with different values for `errors`:

```
>>> bytes_string.decode("ascii", "ignore")
'Pthn'
>>> bytes_string.decode("ascii", "replace")
'P??th??n'
>>> bytes_string.decode("ascii", "backslashreplace")
'P\\xc3\\xbcth\\xc3\\xb8n'
```

To avoid having to work around encoding problems with these tools up front, you should always use universal encoding schemes such as UTF-8 whenever possible.

Encoding Declaration for Python Source Code

By default, Python assumes that program files are stored in UTF-8 encoding, which is especially relevant when special characters are used in string literals, identifiers, or comments. The UTF-8-encoded storage of Python program files is also the default behavior of established Python editors and development environments.

If you want to deviate from this default behavior, you can insert an *encoding declaration* in the header of a Python program file. This is a line that identifies the encoding in which the program file was saved. An encoding declaration is usually located directly below the shebang line[14] or in the first line of the program file and looks like this:

```
# -*- coding: cp1252 -*-
```

In this case, the obsolete `cp1252` Windows character set was used.

14 The meaning of a shebang line is explained in Chapter 4, Section 4.1.1.

Note

In Python 3, encoding declarations are needed only in the rare cases in which you don't want to use the common default encoding, UTF-8.

Note, however, that Python 2 assumes ASCII encoding of program files by default. Without an encoding declaration, no special characters can be used in string literals or comments here.

12.5.11 Template Strings

With f-strings, Python provides a simple syntax for string interpolation, which is very common in practice. Since Python 3.14, *template strings* (also known as *t-strings*) are available, which can be understood as a generalization of f-strings and address a specific problem: It is not possible to tell from a string whether it was created by interpolation and which values were inserted.

To illustrate the problem, let's write a simple logging function that outputs a message with a timestamp on the screen:

```
>>> import datetime
>>> def log(m):
...     t = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
...     print(f"[{t}]: {m}")
```

We use the `datetime` module to add formatted date and time information, then output the message to the screen using `print`. For more information on `datetime`, see Chapter 15, Section 15.2.

The `log` function now can be used in a program to output a variety of status information on the screen. We would naturally use f-strings to embed the values of relevant variables in a message—for example:

```
>>> log("This is a test")
[2026-01-06 12:15:48]: This is a test
>>> p = 12
>>> q = 5.7
>>> log(f"The values are: {p=} {q=}")
[2026-01-06 12:16:58]: The values are: p=12 q=5.7
```

The problem with using f-strings in this context, as mentioned at the beginning, is that the `log` function has no way of recognizing whether a log message is the result of string interpolation or not. In particular, this means that information about the inserted values that goes beyond their string representation is lost.

As a simple example, let's extend the log function so that the types of the values embedded in the messages are automatically added. To do this, you can use the template strings introduced in Python 3.14.

A template string is created like an f-string, using a special string literal with the prefix *t*:

```
>>> t"The values are: {p=} {q=}"
Template(strings=('The values are: p=', ' q=', ''),
interpolations=(Interpolation(12, 'p', 'r', ''), Interpolation(5.7, 'q',
'r', '')))
```

Like an f-string, a template string performs string interpolation and supports the entire formatting syntax known from f-strings. Unlike the f-string, however, the t-string literal does not generate a string, but rather a `Template` instance. This contains the individual components of the interpolation: the string components (`strings`) contained in the literal and the values inserted with the interpolation (`interpolations`).

A `Template` instance cannot be converted directly into a string; it must be processed by a function in an application-specific manner. In the following example, we write an alternative function `log`, which receives a `Template` string and outputs it on the screen together with a timestamp and supplemented data types:

```
>>> from string.templatelib import Interpolation
>>> def log(template):
...     parts = []
...     for item in template:
...         if isinstance(item, Interpolation):
...             parts.append(f"{item.value} ({type(item.value)})")
...         else:
...             parts.append(item)
...     t = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
...     print(f"[{t}]:", "".join(parts))
```

Here, we go through the individual components of string interpolation and assemble the log message piece by piece from these components. If a component is an interpolation value, we add the corresponding data type to this value.

Apart from expecting a t-string, the new `log` function does not differ from its predecessor in terms of usage. At the same time, it allows for a much more flexible design of log messages:

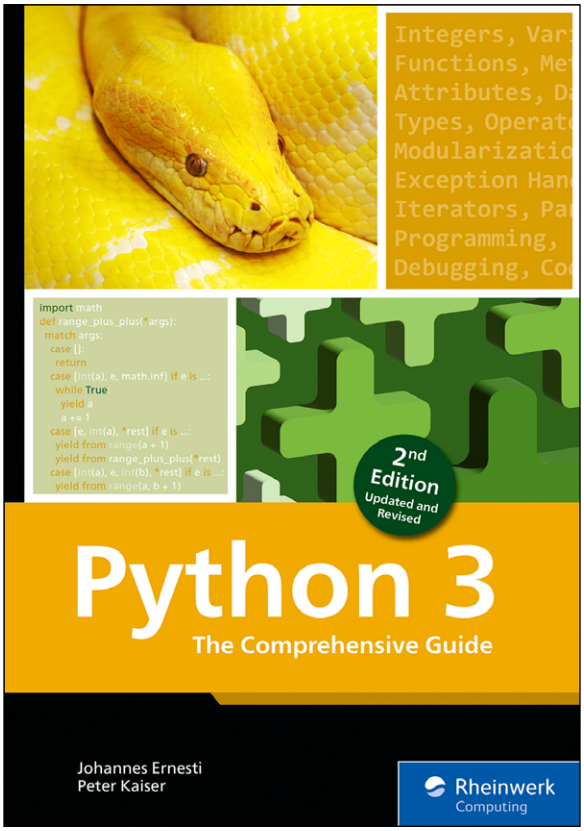
```
>>> log(t"The values are: {p=} {q=}")
[2026-01-06 12:27:12]: The values are: p=12 (<class 'int'>) q=5.7
(<class 'float'>)
```

This example looked at how template strings work using a logging function, but other interesting use cases arise in which careless concatenation of string components can pose a security risk—for example, when composing SQL queries.

Note

Structural pattern matching (see Chapter 25) is well suited for working with `Template` instances. The `log` function can be written more elegantly in this way:

```
>>> def log(template):
...     parts = []
...     for item in template:
...         match item:
...             case Interpolation(value):
...                 parts.append(f"{value} ({type(value)})")
...             case str() as s:
...                 parts.append(s)
...     t = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
...     print(f"[{t}]:", "".join(parts))
```



Johannes Ernesti, Peter Kaiser

Python 3

The Comprehensive Guide

- The complete Python 3 handbook
- Learn basic Python principles and work with functions, methods, data types, and more
- Walk through GUIs, network programming, debugging, optimization, and other advanced topics

rheinwerk-computing.com/6243

The Authors

Johannes Ernesti and Peter Kaiser are research scientists at DeepL. Johannes is a graduate mathematician and received his doctorate in applied mathematics from the Karlsruhe Institute of Technology (KIT). Peter has a degree in computer science and received a doctorate in humanoid robotics from the Karlsruhe Institute of Technology (KIT).

ISBN 978-1-4932-2800-3 • 1048 pages • 08/2026

E-book: $64.99 • **Print book:** $69.95 • **Bundle:** $79.99

Rheinwerk Publishing