General

Representation of newlines

The end of a line can be represented as the string `'\n'`. For example, if you want to construct a string that represents multiple lines of text (a *multiline string*), you may use the following strategy.

```
>>> text = 'Here is a first line'
>>> text += '\n'
>>> text += 'And a second line'
>>> text
'Here is a first line\nAnd a second line'
>>> print(text)
Here is a first line
And a second line
```

Remove leading and/or trailing whitespace

Use the string method `strip` to remove leading and trailing whitespace (spaces, tabs and newlines). In case you only want to remove leading whitespace, use the string method `lstrip`. In case you only want to remove trailing whitespace, use the string method `rstrip`. You can also pass an argument to these string methods, that indicates which leading and/or trailing characters have to be removed. For more details about these string methods, we refer to The Python Standard Library.

```
>>> text = '  This is a text  '
>>> text.strip()
'This is a text'
>>> text.lstrip()
'This is a text  '
>>> text.rstrip()
'  This is a text'
>>> text2 = '===This is a text==='
>>> text2.lstrip('=')
'This is a text==='
```

Align strings over a fixed number of positions

Use the *format specifier* of the string method `format` to reserve a fixed number of positions to output a given text fragment. This is done by formatting the text as a string (indicated by the letter `s`), preceded by an integer that indicates the fixed number of positions that needs to be reserved for the string.

```
>>> f"{'abc':5s}"   # reserve 5 positions
'abc  '
```

In case the string is longer than the number of reserved positions, the placeholder is replaced by the entire string, and thus takes more than the reserved space. In case the string is shorter than the number of reserved positions, the string is left aligned by default. You can also explicitly define the type of alignment in the *format specifier*: left, right or centered. Python will automatically pad the string with additional spaces to the left and/or to the right.

```
>>> f"{'abc':<5s}"   # reserve 5 positions, left alignment
'abc  '
>>> f"{'abc':>5s}"   # reserve 5 positions, right alignment
'  abc'
>>> f"{'abc':^5s}"   # reserve 5 positions, centered alignment
' abc '
```

In case a centered alignment is chosen and the number of additional spaces is an odd number, Python prefers to add one more space to the right than to the left.

Read multiple lines from input

If you know in advance how many lines must be read from input, you may use the following strategy

```
>>> lines = 4
>>> for _ in range(lines):
...     line = input()
...     # process the line
...
```

If the number of lines that must be read from input is not known in advance, but you know for example that the last line is an empty line, you may use the following strategy

```
>>> line = input()
>>> while line:
...     # process the line
...     line = input()
...
```

Iterate the positions and the elements of a collection

The built-in function `enumerate` can be used to fetch an iterator for a collection (*iterable*: object of compound data types such as `str`, `list`, `tuple`, files, ...) that both returns the position and the next element in the collection. The example below illustrates how this can be used for a string to simultaneously traverse its characters and their positions.

```
>>> for index, character in enumerate('abc'):
...     print(f'index: {index}')
...     print(f'character: {character}')
...
index: 0
character: a
index: 1
character: b
index: 2
character: c
```

This can also be used to simultaneously traverse the characters at corresponding positions of two strings.

```
>>> first = 'abc'
>>> second = 'def'
>>> for index, character in enumerate(first):
...    print(f'{character}-{second[index]}')
...
a-d
b-e
c-f
```

However, in this case it is better to use the built-in function `zip`, which is especially equipped to traverse multiple iterable objects at once.

```
>>> first = 'abc'
>>> second = 'def'
>>> for character1, character2 in zip(first, second):
...    print(f'{character1}-{character2}')
...
```

```
a-d
b-e
c-f
```

Split strings into multiple parts

Sometimes you need to split a string into multiple parts. The string method `split` provides one way to do this. This method takes an optional string argument that indicates the sequence of characters (called the *separator*) that needs to be used to split the string on which the method is called.

```
>>> string = 'a-b-c-d'
>>> string.split('-')
['a', 'b', 'c', 'd']
```

By default, the method splits the string at all positions where the separator occurs in the string, and places each of the parts in a list (`list`) that is returned by the method. In case no argument is passed to the method, the string is split at each occurrence of a sequence of whitespace characters (spaces, tabs, newlines, ...). The string method also has another optional argument that you can use to indicate the maximal number of parts in which the string can be split.

Because the string method `split` returns a list, you can directly traverse the elements of the list using a `for`-loop.

```
>>> string = 'a-b-c-d'
>>> for element in string.split('-'):
...     print(element)
...
a
b
c
d
```

String repetition

To repeat a given string a fixed number of times, you can multiply that string with an integer. As with the multiplication of numbers, this uses the operator `*`. The order of the string and the integer does not matter in the multiplication

```
>>> 'a' * 3
'aaa'
>>> 5 * 'ab'
'ababababab'
```

This is very handy in case the number of repetitions of the string is not known beforehand, but for example can be retrieved from a variable.

```
>>> repetitions = 4
>>> repetitions * 'abc'
'abcabcabcabc'
```

Atbash

Convert letters to their ordinal value

Each character has a corresponding ordinal value (`int`). For example, the question mark (`?`) has ordinal value 63. The value can be easily obtained using the built-in function `ord`.

```
>>> ord('?')
63
```

The interesting thing about these ordinal values, is that successive letters in the alphabet have successive ordinal values. This holds for both uppercase and lowercase letters. If we know that the letter `a` has ordinal value 97, it immediately follows that the letter `b` must have ordinal value 98. Using this knowledge, we can easily determine the position of a letter in the alphabet.

```
>>> letter = 'd'
>>> ord(letter)
100
>>> ord('a')
97
>>> ord(letter) - ord('a') + 1   # d is the 4th letter of the alphabet
4
>>> ord('z') - ord('a') + 1      # z is the 26th letter of the alphabet
26
```

String methods `islower()` and `isupper()`

The string method `islower` checks if *all* characters of a string are lowercase letters. The string method `isupper` checks if *all* characters of a string are uppercase letters.

```
>>> 'ABCD'.islower()
False
>>> 'ABCD'.isupper()
True
>>> 'abcd'.islower()
True
>>> 'abcd'.isupper()
False
>>> 'AbCd'.islower()
False
>>> 'AbCd'.isupper()
False
```

Vampire numbers

Specific information

The goal is to check whether two strings contain the same characters. However, the characters of both strings are not necessarily in the same order. By first sorting the characters in both strings and then comparing the sorted strings, you can easily check whether both strings contain the same characters. Sorting the characters of a string can be done using the built-in function `sorted`.

Calling `sorted('abc'')` yields a new `list` containing the characters in lexicographic order: `'a'`, `'b'` and `'c'`.

```
>>> sorted('cab')
['a', 'b', 'c']
```