General

Conversion of values to Boolean values

In Python it is generally considered a better programming style (more *pythonic*) to rewrite the condition in the following code snippet

```
if x != 0:
    pass
```

in short as

```
if x:
    pass
```

This is possible, because the evaluation of the condition in an `if` statement or a `while` statement, implicitly converts the expression into a Boolean value. For most data types, all values are converted to the Boolean value `True`, except for a single value that is converted to `False`:

- for integers only 0 is converted to `False`
- for floats only 0.0 is converted to `False`
- for strings only the empty string (`''`) is converted to `False`
- for lists only the empty list (`[]`) is converted to `False`
- ...

As a result, you will encounter this shorthand notation very often in code examples that you find in books or online. So, even if you find the longer notation more readable, it is still necessary to understand the shorthand notation when trying to understand code examples that make use of it.

Also not that it is quite useless to write

```
if found == True:
    pass
```

as the variable `found` already references a Boolean value. Also in this case, it is shorter to write

```
if found:
    pass
```

Escaping literal backslashes

In Python a backslash is used inside a string to escape the next character in the string, so that this character loses its special meaning (for example to put a double quote inside a string that is enclosed itself in between a pair of double quotes) or to give a special meaning to the next character (such as the end of line that is represented by the character `'\n'` or a tab that is represented by the character `'\t'`).

Because this gives a special meaning to a backslash inside a string (used to escape characters), a literal backslash inside a string must be represented by two successive backslashes (`'\\'`), where the first backslash serves as the escape symbol and the second backslash is the character that is given its literal meaning by escaping it.

Representation of newlines

In Python you may represent the end of a line 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

In Python you can use the string method `strip` to remove leading and trailing whitespace (spaces, tabs and newlines). In case you only want to remove leading whitespace, you can use the string method `lstrip`. In case you only want to remove trailing whitespace, you can 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'
```

Align strings over a fixed number of positions

You can 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.

Iterate both the elements and their positions of sequence types

The built-in function `enumerate` can be used to request an iterator for a given sequence type (strings, lists, tuples, files, ...) that both returns the position and the value at that position for the next element of the

sequence type. The example below illustrates how this can be used to simultaneously iterate the positions of a string and the characters on those position.

```
>>> 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
```

You can also use this to simultaneously iterate over the characters 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 iterate over 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. One way to do this job makes use of the string method `split`. 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 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 needs to be split.

Because the string method `split` returns a list, you can directly iterate over 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
```

Convert letters to their ordinal value

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

```
>>> ord('?')
69
```

The interesting thing about these ordinal values, is that successive letters in the alphabet have successive numerical 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. With 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 26st letter of the alphabet
26
```

Vampire numbers

Specific information

Checking whether two strings contain the same characters (that are not necessarily in the same order) can easily be done by sorting both strings and comparing the sorted results with each other. Sorting two strings in Python can be done using the `sorted`-function. Calling `sorted_text = sorted(text)` on a string called `text` with contents `cab` yields a new string `sorted_text` with contents `abc`.