Theory: String formatting

29 minutes 11 / 12 problems solved

There are certain situations when you want to make your strings kind of "dynamic", i.e. make them change depending on the value of a variable or expression. For example, you want to prompt the user for their name and print the greeting to them with the name they entered. But how can you embed a variable into a string? If you do it in the most intuitive way, you'll be disappointed:

```
a = input()
print('Hello, a')
```

The output will be:

```
Hello, a
```

Luckily, Python offers a large variety of methods to format the output the way you want and we'll concentrate on the main two:

- Formatted string literals
- The `str.format()` method

Earlier **the % operator** was in use. This built-in operator derived from C-language and was used in some situations by following the scheme:

string % value

Thus, the variable to the right of `%` was included in the string to the left of it. If we'd wanted to divide 11 by 3, the `/` operator would have returned a float number with a lot of decimal places.

```
print(11 / 3)  # 3.6666666666666665
```

With `%` character, we could control the number of decimal places, for example, reduce their number to 3 or 2:

```
print('%.3f' % (11/3))  # 3.667
print('%.2f' % (11/3))  # 3.67
```

For every operation, you had to know plenty of specifiers, length modifiers and conversion types. A huge variety of extra operators led to some common errors. That's why more modern and easy to use operators were introduced. Progress never stops, you know!

Formatting your strings also makes your code look more readable and easily editable.

§1. The str. format() method

The operation of the method is already described in its name: in the **string part**, we introduce curly braces as placeholders for variables enlisted in the **format part**:

```
print('Mix {}, {} and a {} to make an ideal omelet.'.format('2 eggs', '30 g of milk', 'pinch of salt'))
```

The expressions are put instead of braces in the order they were mentioned:

```
Mix 2 eggs, 30 g of milk and a pinch of salt to make an ideal omelet.
```

You can use the same variable in one string more than once if it's necessary. Furthermore, you can address the objects by referring to their **positions** in curly braces (as usual, ***the numbering starts from zero***). Attention: the order

Current topic:

- ✓ String formatting (Stage 1) 7★

Topic depends on:

- ✓ Quotes and multi-line strings (Stage 1) 12★
- ✓ Taking input (Stage 1) 17★

Topic is required for:

- ✓ Class vs instance 3★
- ✓ Methods (Stage 1) 3★
- Datetime parsing and formatting
- Time module
- Working with CSV

Table of contents:

you choose can be very important. The following two codes:

```
print('{0} in the {1} by Frank Sinatra'.format('Strangers', 'Night'))
```

and

```
print('{1} in the {0} by Frank Sinatra'.format('Strangers', 'Night'))
```

will have different outputs:

```
Strangers in the Night by Frank Sinatra
```

```
Night in the Strangers by Frank Sinatra
```

The second output sounds really weird, doesn't it?

If you've mentioned more variables than needed in the format part, the extra ones just will be ignored.

We can also use keywords to make such string more readable. Don't forget, that you can easily break the lines! For example:

```
print('The {film} at {theatre} was {adjective}!'.format(film='Lord of the Rings',
                                                        adjective='incredible',
                                                        theatre='BFI IMAX'))
```

Note that you can also mix the order as you want if you use keywords. Here's the formatted string:

```
The Lord of the Rings at BFI IMAX was incredible!
```

Also, you can combine both positional and keyword arguments:

```
print('The {0} was {adjective}!'.format('Lord of the Rings', adjective='incredible'))
# The Lord of the Rings was incredible!
```

Keep tabs on the order of your arguments, though:

```
print('The {0} was {adjective}!'.format(adjective='incredible', 'Lord of the Rings'))
# SyntaxError: positional argument follows keyword argument
```

The last code snippet resulted in `SyntaxError`, since positional arguments are to be mentioned first.

> Remember as a Python rule that keyword arguments are always written **after** positional, or non-keyword, arguments.

§2. Formatted string literals

Formatted string literals (or, simply, **f-strings**) are used to embed the values of expressions inside string literals. This way is supposed to be the easiest one: you only need to put **f** before the string and put the variables you want to embed into the string in curly braces. They are also the newest feature among all string formatting methods in Python.

```
name = 'Elizabeth II'
title = 'Queen of the United Kingdom and the other Commonwealth realms'
reign = 'the longest-lived and longest-reigning British monarch'
f'{name}, the {title}, is {reign}.'
```

If you print this short string, you'll see an output that is five times longer than its representation in code:

```
Elizabeth II, the Queen of the United Kingdom and the other Commonwealth realms, i
s the longest-lived and longest-reigning British monarch.
```

You can also use different formatting specifications with f-literals, for example rounding decimals would look like this:

```
hundred_percent_number = 1823
needed_percent = 16
needed_percent_number = hundred_percent_number * needed_percent / 100


print(f'{needed_percent}% from {hundred_percent_number} is {needed_percent_number}
')
# 16% from 1823 is 291.68


print(f'Rounding {needed_percent_number} to 1 decimal place is {needed_percent_num
ber:.1f}')
# Rounding 291.68 to 1 decimal place is 291.7
```

You can read more about Format Specification Mini-Language in Python in the official documentation.

Maybe, you'll think that these methods are not important and overrated, but they give you the opportunity to make your code look fancy and readable.

Report a typo

894 users liked this theory. **21** didn't like it. **What about you?**

😍 🙂 😐 ☹️ 😡

Start practicing

This content was created over 1 year ago and updated 6 days ago. Share your feedback below in comments to help us improve it!

Comments (13) Hints (0) Useful links (2) Show discussion