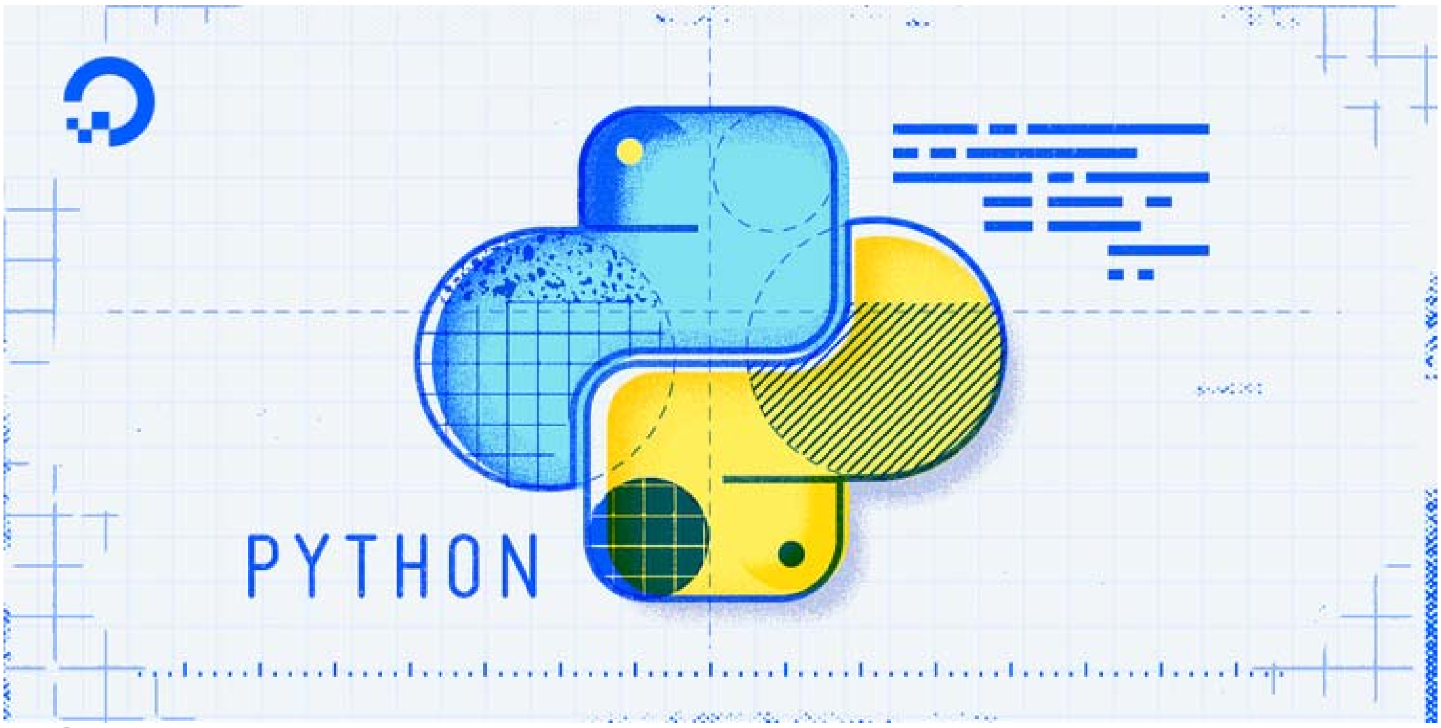
Subscribe



How To Use String Formatters in Python 3

60

Posted October 14, 2016 403.8k PYTHON DEVELOPMENT

By: Lisa Tagliaferri

Introduction

Python's `str.format()` method of the string class allows you to do variable substitutions and value formatting. This lets you concatenate elements together within a string through positional formatting.

This tutorial will guide you through some of the common uses of formatters in Python, which can help make your code and program more readable and user friendly.

SCROLL TO TOP

Formatters work by putting in one or more **replacement fields** or placeholders — defined by a pair of curly braces {} into a string and calling the str.format() method. You'll pass into the ...rough in

Let's print out a string that uses a formatter:

```
print("Sammy has {} balloons.".format(5))
```

```
Output
Sammy has 5 balloons.
```

In the example above, we constructed a string with a pair of curly braces as a placeholder:

```
"Sammy has {} balloons."
```

We then added the `str.format()` method and passed the value of the integer `5` to that method. This places the value of `5` into the string where the curly braces were:

```
Sammy has 5 balloons.
```

We can also assign a variable to be equal to the value of a string that has formatter placeholders:

```
open_string = "Sammy loves {}."
print(open_string.format("open source"))
```

```
Output
Sammy loves open source.
```

In this second example, we concatenated the string `"open source"` with the larger string, replacing the curly braces in the original string.

Formatters in Python allow you to use curly braces as placeholders for values that you'll pass through with the `str.format()` method.

You can use multiple pairs of curly braces when using formatters. If we'd like to add another variable substitution to the sentence above, we can do so by adding a second pair of curly

```
new_open_string = "Sammy loves {} {}."                      #2 {} placeholders
print(new_open_string.format("open-source", "software"))    #Pass 2 strings into method, se
```

```
Output
Sammy loves open-source software.
```

To add another substitution, we added a second pair of curly braces into the original string. Then, we passed two strings into the `str.format()` method, separating them by a comma.

Following the same syntax, we can add additional substitutions:

```
sammy_string = "Sammy loves {} {}, and has {} {}."                      #4 {} placeholders
print(sammy_string.format("open-source", "software", 5, "balloons"))    #Pass 4 strings int
```

```
Output
Sammy loves open-source software, and has 5 balloons.
```

In `sammy_string` we added 4 pairs of curly braces as placeholders for variable substitution. We then passed 4 values into the `str.format()` method, mixing string and integer data types. Each of these values are separated by a comma.

Reordering Formatters with Positional and Keyword Arguments

When we leave curly braces empty without any parameters, Python will replace the values passed through the `str.format()` method in order. As we have seen, so far, a formatter construction with two empty curly braces with two values passed through will look like this:

```
print("Sammy the {} has a pet {}!".format("shark", "pilot fish"))
```

econd pair

The values that exist within the method look like this:

```
("shark", "pilot fish")
```

They are essentially the tuple data type and each individual value contained in the tuple can be called by its index number, which starts with the index number 0.

We can pass these index numbers into the curly braces that serve as the placeholders in the original string:

```
print("Sammy the {0} has a pet {1}!".format("shark", "pilot fish"))
```

In the above example, the output will be what we get without passing index numbers into the braces as we are calling the values in the tuple in order:

```
Output
Sammy the shark has a pet pilot fish!
```

But, if we reverse the index numbers with the parameters of the placeholders we can reverse the values being passed into the string:

```
print("Sammy the {1} has a pet {0}!".format("shark", "pilot fish"))
```

```
Output
Sammy the pilot fish has a pet shark!
```

If you call an index number of 2 in a tuple that has values at index positions 0 and 1, then you are calling on a value that is out of range. When you call an index number that is out of range, you'll receive an error message:

```
print("Sammy the {2} has a pet {1}!".format("shark", "pilot fish"))
```

and 1,

Let's add a few more placeholders and a few more values to pass to them, so we can understand how we can reorder formatters a little better. First, here is a new string with four placeholders:

```
print("Sammy is a {}, {}, and {} {}!".format("happy", "smiling", "blue", "shark"))
```

```
Output
Sammy is a happy, smiling and blue shark!
```

Without parameters, the values that are passed into the `str.format()` method are concatenated into the string in order.

The string values contained in the tuple correspond to the following index numbers:

"happy"	"smiling"	"blue"	"shark"
0	1	2	3

Let's use the index numbers of the values to change the order that they appear in the string:

```
print("Sammy is a {3}, {2}, and {1} {0}!".format("happy", "smiling", "blue", "shark"))
```

```
Output
Sammy is a shark, blue, and smiling happy!
```

Since we started with index number 3, we called the last value of `"shark"` first. The other index numbers included as parameters change the order of how the words appear within the original string.

In addition to positional arguments, we can also introduce keyword arguments that are called by their keyword name:

This example shows the use of a keyword argument being used with positional arguments. We can fill in the keyword argument `pr` alongside positional arguments, and can move these arguments around to change the resulting string:

```
print("Sammy the {pr} {1} a {0}.".format("shark", "made", pr = "pull request"))
```

```
Output
Sammy the pull request made a shark.
```

Positional and keyword arguments used with string formatters give us more control over manipulating our original strings through reordering.

Specifying Type

We can include more parameters within the curly braces of our syntax. We'll use the format code syntax `{field_name:conversion}`, where `field_name` specifies the index number of the argument to the `str.format()` method that we went through in the reordering section, and `conversion` refers to the conversion code of the data type that you're using with the formatter.

The conversion type refers to the the single-character type code that Python uses. The codes that we'll be using here are `s` for string, `d` to display decimal integers (10-base), and `f` which we'll use to display floats with decimal places. You can read more about the Format-Specification Mini-Language through Python 3's official documentation.

Let's look at an example where we have an integer passed through the method, but want to display it as a float by adding the `f` conversion type argument:

```
print("Sammy ate {0:f} percent of a {1}!".format(75, "pizza"))
```

```
Output
Sammy ate 75.000000 percent of a pizza!
```

In the example above, there are a lot of numbers displaying after the decimal point, but you can

e precision

nal you

would like to include.

If Sammy ate 75.765367% of the pizza, but we don't need to have a high level of accuracy, we can limit the places after the decimal to 3 by adding `.3` before the conversion type `f`:

```
print("Sammy ate {0:.3f} percent of a pizza!".format(75.765367))
```

```
Output
Sammy ate 75.765 percent of a pizza!
```

If we just want one decimal place, we can rewrite the string and method like so:

```
print("Sammy ate {0:.1f} percent of a pizza!".format(75.765367))
```

```
Output
Sammy ate 75.8 percent of a pizza!
```

Note that modifying precision will cause the number to be rounded.

Although we display a number with no decimal places as a float, if we try to change the float to an integer by using the `d` conversion type, we will receive an error:

```
print("Sammy ate {0:d} percent of a pizza!".format(75.765367))
```

```
Output
ValueError: Unknown format code 'd' for object of type 'float'
```

If you would like no decimal places to be shown, you can write your formatter like so:

```
print("Sammy ate {0:.0f} percent of a pizza!".format(75.765367))
```

own after

Padding Variable Substitutions

Because the placeholders are replacement fields, you can **pad** or create space around an element by increasing field size through additional parameters. This can be useful when we need to organize a lot of data visually.

We can add a number to indicate field size (in terms of characters) after the colon `:` in the curly braces of our syntax:

```
print("Sammy has {0:4} red {1:16}!".format(5, "balloons"))
```

```
Output
Sammy has    5 red balloons        !
```

In the example above, we gave the number `5` a character field size of 4, and the string `balloons` a character field size of 16 (because it is a long string).

As we see, by default strings are left-justified within the field, and numbers are right-justified. You can modify this by placing an alignment code just following the colon. `<` will left-align the text in a field, `^` will center the text in the field, and `>` will right-align it.

Let's left-align the number and center the string:

```
print("Sammy has {0:<4} red {1:^16}!".format(5, "balloons"))
```

```
Output
Sammy has 5    red     balloons    !
```

Now we see that `5` is left-aligned, providing space in the field before `red`, and `balloons` is centered in its field with space to the left and right of it.

By default, when we make a field larger with formatters, Python will fill the field with whitespace characters. We can modify that to be a different character by specifying the character we want it

```
print("{:*^20s}".format("Sammy"))
```

```
*******Sammy********
```

We are accepting the string being passed to `str.format()` in the index position of 0 since we did not specify otherwise, including the colon, and specifying that we will use `*` instead of space to fill up the field. We're centering the string with `^`, specifying that the field is 20 characters in size, and also indicating that we are working with a string conversion type by including `s`.

We can combine these parameters with other parameters we've used before:

```
print("Sammy ate {0:5.0f} percent of a pizza!".format(75.765367))
```

```
Output
Sammy ate    76 percent of a pizza!
```

In the parameters within the curly braces, we specified the index field number of the float and included the colon, indicated the size of the field number and included the full stop, wrote in the number of places after the decimal place, and then specified the conversion type of `f`.

Using Variables

So far, we have passed integers, floats, and strings into the `str.format()` method, but we can also pass variables through the method. This works just like any other variable.

```
nBalloons = 8
print("Sammy has {} balloons today!".format(nBalloons))
```

```
Output
Sammy has 8 balloons today!
```

We can use variables for both the original string and what is passed into the method :

```
print(sammy.format(nBalloons))
```

```
Sammy has 8 balloons today!
```

Variables can be easily substituted for each part of our formatter syntax construction. This makes it easier to work with when we are taking in user-generated input and assigning those values to variables.

Using Formatters to Organize Data

Formatters can be seen in their best light when they are being used to organize a lot of data in a visual way. If we are showing databases to users, using formatters to increase field size and modify alignment can make your output more readable.

Let's look at a typical for loop in Python that will print out `i`, `i*i`, and `i*i*i` in the range from 3 to 12:

```
for i in range(3,13):
    print(i, i*i, i*i*i)
```

```
Output
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
11 121 1331
12 144 1728
```

While the output is organized in a way, the numbers overflow into each other's columns, making the bottom of the output less readable. If you are working with a bigger data set with many small and big numbers, this can pose a problem.

Let's use formatters to give more space to these numbers:

ith the working with integers. In this example, we accommodated for the size of each expected output, giving 2 extra character spaces for each, depending on the maximum possible number size, so our output looks like this:

```
Output
  3    9    27
  4   16    64
  5   25   125
  6   36   216
  7   49   343
  8   64   512
  9   81   729
 10  100  1000
 11  121  1331
 12  144  1728
```

We can specify a consistent field size number in order to have even columns, making sure that we accommodate the larger numbers:

```
for i in range(3,13):
    print("{:6d} {:6d} {:6d}".format(i, i*i, i*i*i))
```

```
Output
     3      9     27
     4     16     64
     5     25    125
     6     36    216
     7     49    343
     8     64    512
     9     81    729
    10    100   1000
    11    121   1331
    12    144   1728
```

We can also manipulate the alignment of the columns by adding `<`, `^`, and `>` for text alignment, change `d` to `f` to add decimal places, change field name index numbers, and more to ensure

Conclusion

and
passing
variable substitutions into a string, and are useful for making sure output is readable and user friendly.

By: Lisa Tagliaferri

Upvote (60) Subscribe

Tutorial Series

How To Code in Python 3

Python is an extremely readable and versatile programming language. Written in a relatively straightforward style with immediate feedback on errors, Python offers simplicity and versatility, in terms of extensibility and supported paradigms.

Show Tutorials

and more!

REDEEM CREDIT

Related Tutorials

How To Build a Neural Network to Recognize Handwritten Digits with TensorFlow

How to Install, Run, and Connect to Jupyter Notebook on a Remote Server

How To Set Up a Jupyter Notebook with Python 3 on Debian 9

How To Set Up Django with Postgres, Nginx, and Gunicorn on Debian 9

How To Install the Anaconda Python Distribution on Debian 9

5 Comments

Leave a comment...

Log In to Comment

thor77 *April 23, 2017*

at:

```
print("Sammy ate {0:.d} percent of a pizza!".format(75.765367))
```

The Exception given by you doesn't correspond to this code, though, Python first complains about the missing precision, because you're giving a dot (`.`) before the format code (`d`).

```
ValueError: Format specifier missing precision
```

Corrected:

```
print("Sammy ate {0:d} percent of a pizza!".format(75.765367))
```

ltagliaferri MOD *April 23, 2017*
1 Thanks for finding that typo, I've updated the tutorial!

vernaleisti *March 4, 2018*
0 If I have to first calculate how much space I need for some string or number how can I put the calculated number in my code? I can't put variables inside these { } ...

MikePillows *April 19, 2018*
0 Hi, vernaleisti!

I had to deal with the same situation; here's what I came up with after fiddling around for a while.

The string I wanted to use for a print statement, with placeholders, looked something like this:

```
"Some text: '{list_element:{max_word_length}}'. Some other text."
```

I concluded I needed to build it by "nesting" str.format() method calls and put together the string parts by using a str.join() method call.

```
    for element in my_list:
        print(("".join(["Some text: '{",
                         "list_element:{max_size}".format(max_size=max_size),
                         "}'. Some other text."])
              ).format(list_element=element)
             )
```

Would be:

```
Some text: 'foo   '. Some other text.
Some text: 'bar   '. Some other text.
Some text: 'foobar'. Some other text.
```

Hope it helps!

P.S. Thank you, Lisa, for a useful article! :)

MisaAG *May 13, 2018*
0 Thank you for such a clear tutorial!