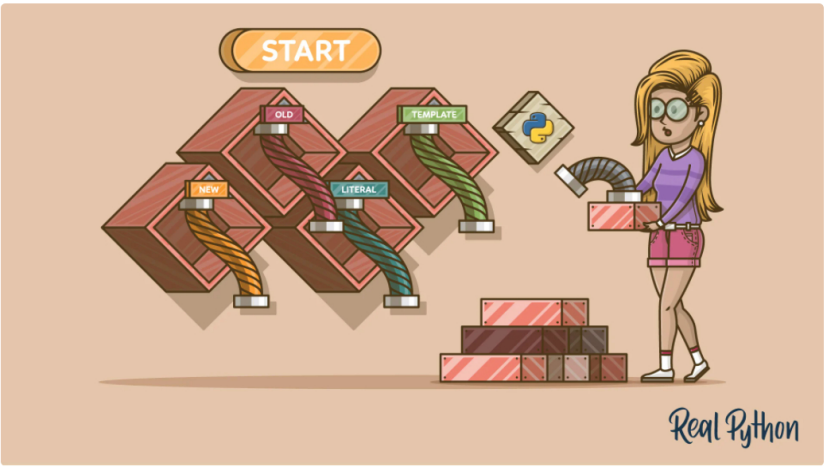


Python String Formatting Best Practices

by Dan Bader Jul 04, 2018 24 Comments basics best-practices python

Mark as Completed

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Python String Formatting Tips & Best Practices**

Remember the Zen of Python and how there should be “one obvious way to do something in Python”? You might scratch your head when you find out that there are *four* major ways to do string formatting in Python.

In this tutorial, you’ll learn the four main approaches to string formatting in Python, as well as their strengths and weaknesses. You’ll also get a simple rule of thumb for how to pick the best general purpose string formatting approach in your own programs.

Let’s jump right in, as we’ve got a lot to cover. In order to have a simple toy example for experimentation, let’s assume you’ve got the following variables (or constants, really) to work with:

Python >>>

```
>>> errno = 50159747054
>>> name = 'Bob'
```

Based on these variables, you’d like to generate an output string containing a simple error message:

Table of Contents
- #1 “Old Style” String Formatting (% Operator)
- #2 “New Style” String Formatting (str.format)
- #3 String Interpolation / f-Strings (Python 3.6+)
- #4 Template Strings (Standard Library)
- Which String Formatting Method Should You Use?
- Key Takeaways

Mark as Completed
Tweet Share Email

Recommended Video Course
Python String Formatting Tips & Best

```
Python >>>
'Hey Bob, there is a 0xbadc0ffee error!'
```

That error could really spoil a dev's Monday morning… But we're here to discuss string formatting. So let's get to work.

#1 "Old Style" String Formatting (% Operator)

Strings in Python have a unique built-in operation that can be accessed with the `%` operator. This lets you do simple positional formatting very easily. If you've ever worked with a `printf`-style function in C, you'll recognize how this works instantly. Here's a simple example:

```
Python >>>
>>> 'Hello, %s' % name
"Hello, Bob"
```

I'm using the `%s` format specifier here to tell Python where to substitute the value of `name`, represented as a string.

There are other format specifiers available that let you control the output format. For example, it's possible to convert numbers to hexadecimal notation or add whitespace padding to generate nicely formatted tables and reports. (See Python Docs: "printf-style String Formatting".)

Here, you can use the `%x` format specifier to convert an `int` value to a string and to represent it as a hexadecimal number:

```
Python >>>
>>> '%x' % errno
'badc0ffee'
```

The "old style" string formatting syntax changes slightly if you want to make multiple substitutions in a single string. Because the `%` operator takes only one argument, you need to wrap the right-hand side in a tuple, like so:

```
Python >>>
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

It's also possible to refer to variable substitutions by name in your format string, if you pass a mapping to the `%` operator:

```
Python >>>
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
...     "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0ffee error!'
```

This makes your format strings easier to maintain and easier to modify in the future. You don't have to worry about making sure the order you're passing in the values matches up with the order in which the values are referenced in the format string. Of course, the downside is that this technique requires a little more typing.

I'm sure you've been wondering why this `printf`-style formatting is called "old style" string formatting. It was technically superseded by "new style" formatting in Python 3, which we're going to talk about next.

#2 "New Style" String Formatting (`str.format`)

Python String Formatting Tips & Best Practices

#2 "New Style" String Formatting (`str.format`)

Python 3 introduced a new way to do string formatting that was also later back-ported to Python 2.7. This "new style" string formatting gets rid of the `%`-operator special syntax and makes the syntax for string formatting more regular. Formatting is now handled by calling `.format()` on a string object.

You can use `format()` to do simple positional formatting, just like you could with "old style" formatting:

```
Python >>>
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

Or, you can refer to your variable substitutions by name and use them in any order you want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to `format()`:

```
Python >>>
>>> 'Hey {name}, there is a 0x{errno:x} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

This also shows that the syntax to format an `int` variable as a hexadecimal string has changed. Now you need to pass a format spec by adding a `:x` suffix. The format string syntax has become more powerful without complicating the simpler use cases. It pays off to read up on this string formatting mini-language in the Python documentation.

In Python 3, this "new style" string formatting is to be preferred over `%`-style formatting. While "old style" formatting has been de-emphasized, it has not been deprecated. It is still supported in the latest versions of Python. According to this discussion on the Python dev email list and this issue on the Python dev bug tracker, `%`-formatting is going to stick around for a long time to come.

Still, the official Python 3 documentation doesn't exactly recommend "old style" formatting or speak too fondly of it:

> "The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the str.format() interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text." (Source)

This is why I'd personally try to stick with `str.format` for new code moving forward. Starting with Python 3.6, there's yet another way to format your strings. I'll tell you all about it in the next section.

#3 String Interpolation / f-Strings (Python 3.6+)

Python 3.6 added a new string formatting approach called formatted string literals or "f-strings". This new way of formatting strings lets you use embedded Python expressions inside string constants. Here's a simple example to give you a feel for the feature:

```
Python >>>
>>> f'Hello, {name}!'
'Hello, Bob!'
```

As you can see, this prefixes the string constant with the letter "`f`"—hence the name "f-strings." This new formatting syntax is powerful. Because you can embed arbitrary Python expressions, you can even do inline arithmetic with it. Check out this example:

```
Python >>>
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
```

```
'Five plus ten is 15 and not 30.'
```

Formatted string literals are a Python parser feature that converts f-strings into a series of string constants and expressions. They then get joined up to build the final string.

Imagine you had the following `greet()` function that contains an f-string:

Python >>>

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...
>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

When you disassemble the function and inspect what's going on behind the scenes, you'll see that the f-string in the function gets transformed into something similar to the following:

Python >>>

```
>>> def greet(name, question):
...     return "Hello, " + name + "! How's it " + question + "?"
```

The real implementation is slightly faster than that because it uses the `BUILD_STRING` opcode as an optimization. But functionally they're the same:

Python >>>

```
>>> import dis
>>> dis.dis(greet)
  2           0 LOAD_CONST               1 ('Hello, ')
              2 LOAD_FAST                0 (name)
              4 FORMAT_VALUE             0
              6 LOAD_CONST               2 ("! How's it ")
              8 LOAD_FAST                1 (question)
             10 FORMAT_VALUE             0
             12 LOAD_CONST               3 ('?')
             14 BUILD_STRING             5
             16 RETURN_VALUE
```

String literals also support the existing format string syntax of the `str.format()` method. That allows you to solve the same formatting problems we've discussed in the previous two sections:

Python >>>

```
>>> f"Hey {name}, there's a {errno:#x} error!"
"Hey Bob, there's a 0xbadc0ffee error!"
```

Python's new formatted string literals are similar to JavaScript's Template Literals added in ES2015. I think they're quite a nice addition to Python, and I've already started using them in my day to day (Python 3) work. You can learn more about formatted string literals in our in-depth Python f-strings tutorial.

#4 Template Strings (Standard Library)

Here's one more tool for string formatting in Python: template strings. It's a simpler and less powerful mechanism, but in some cases this might be exactly what you're looking for.

Let's take a look at a simple greeting example:

Python >>>

```
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

You see here that we need to import the `Template` class from Python's built-in `string` module. Template strings are not a core language feature but they're supplied by the `string` module in the standard library.

Another difference is that template strings don't allow format specifiers. So in order to get the previous error string example to work, you'll need to manually transform the `int` error number into a hex-string:

```
Python >>>
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

That worked great.

So when should you use template strings in your Python programs? In my opinion, the best time to use template strings is when you're handling formatted strings generated by users of your program. Due to their reduced complexity, template strings are a safer choice.

The more complex formatting mini-languages of the other string formatting techniques might introduce security vulnerabilities to your programs. For example, it's possible for format strings to access arbitrary variables in your program.

That means, if a malicious user can supply a format string, they can potentially leak secret keys and other sensitive information! Here's a simple proof of concept of how this attack might be used against your code:

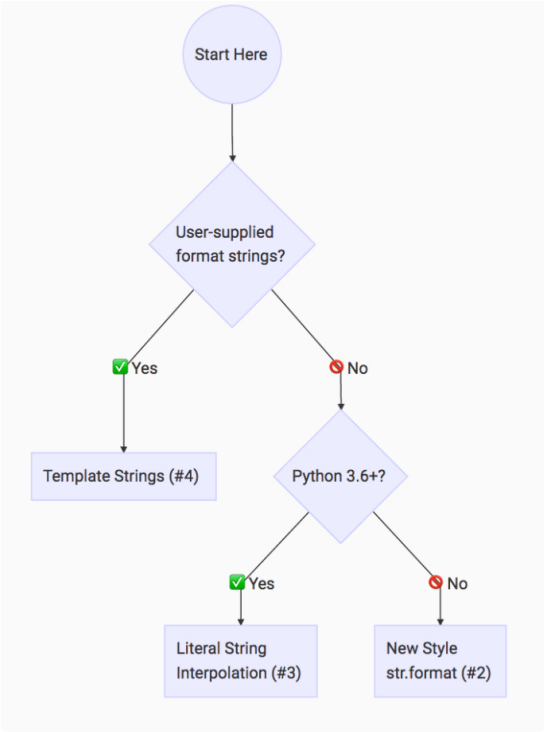
```
Python >>>
>>> # This is our super secret key:
>>> SECRET = 'this-is-a-secret'

>>> class Error:
...     def __init__(self):
...         pass

>>> # A malicious user can craft a format string that
>>> # can read data from the global namespace:
>>> user_input = '{error.__init__.__globals__[SECRET]}'

>>> # This allows them to exfiltrate sensitive information,
>>> # like the secret key:
>>> err = Error()
>>> user_input.format(error=err)
'this-is-a-secret'
```

See how a hypothetical attacker was able to extract our secret string by accessing the `__globals__` dictionary from a malicious format string? Scary, huh? Template strings close this attack vector. This makes them a safer choice if you're handling format strings generated from user input:

```
Python >>>
>>> user_input = '${error.__init__.__globals__[SECRET]}'
>>> Template(user_input).substitute(error=err)
ValueError:
"Invalid placeholder in string: line 1, col 1"
```

Which String Formatting Method Should You Use?

I totally get that having so much choice for how to format your strings in Python can feel very confusing. This is an excellent cue to bust out this handy flowchart infographic I've put together for you:



Python String Formatting Rule of Thumb (Image: Click to Tweet)

This flowchart is based on the rule of thumb that I apply when I'm writing Python:

> **Python String Formatting Rule of Thumb:** If your format strings are user-supplied, use Template Strings (#4) to avoid security issues. Otherwise, use Literal String Interpolation/f-Strings (#3) if you're on Python 3.6+, and "New Style" str.format (#2) if you're not.

Key Takeaways

- Perhaps surprisingly, there's more than one way to handle string formatting in Python.
- Each method has its individual pros and cons. Your use case will influence which method you should use.
- If you're having trouble deciding which string formatting method to use, try our *Python String Formatting Rule of Thumb*.

About **Dan Bader**

Dan Bader is the owner and editor in chief of Real Python and the main developer of the realpython.com learning platform. Dan has been writing code for more than 20 years and holds a master's degree in computer science.

» More about Dan

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Joanna

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: basics best-practices python

Recommended Video Course: Python String Formatting Tips & Best Practices

Keep Learning

Related Tutorial Categories: basics best-practices python

Recommended Video Course: Python String Formatting Tips & Best Practices