Python 3's f-Strings: An Improved String Formatting Syntax (Guide) – Real Python

realpython.com (https://realpython.com/python-f-strings/) · by Real Python

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 3's f-Strings: An Improved String Formatting Syntax** (https://realpython.com/courses/python-3-f-strings-improved-string-formatting-syntax/)

As of Python 3.6, f-strings are a great new way to format strings. Not only are they more readable, more concise, and less prone to error than other ways of formatting, but they are also faster!

By the end of this article, you will learn how and why to start using f-strings today.

But first, here's what life was like before f-strings, back when you had to walk to school uphill both ways in the snow.

"Old-school" String Formatting in Python

Before Python 3.6, you had two main ways of embedding Python expressions inside string literals for formatting: %-formatting and `str.format()`. You're about to see how to use them and what their limitations are.

Option #1: %-formatting

This is the OG of Python formatting and has been in the language since the very beginning. You can read more in the Python docs (https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting). Keep in mind that %-formatting is not recommended by the docs, which contain the following note:

> "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 (https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting))

How to Use %-formatting

String objects have a built-in operation using the `%` operator, which you can use to format strings. Here's what that looks like in practice:

>>>

```
>>> name = "Eric"
>>> "Hello, %s." % name
'Hello, Eric.'
```

In order to insert more than one variable, you must use a tuple of those variables. Here's how you would do that:

>>>

```
>>> name = "Eric"
>>> age = 74
>>> "Hello, %s. You are %s." % (name, age)
'Hello Eric. You are 74.'
```

Why %-formatting Isn't Great

The code examples that you just saw above are readable enough. However, once you start using several parameters and longer strings, your code will quickly become much less easily readable. Things are starting to look a little messy already:

>>>

```
>>> first_name = "Eric"
>>> last_name = "Idle"
>>> age = 74
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> "Hello, %s %s. You are %s. You are a %s. You were a member of %s." % (first_name, last_na
me, age, profession, affiliation)
'Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.'
```

Unfortunately, this kind of formatting isn't great because it is verbose and leads to errors, like not displaying tuples or dictionaries (https://realpython.com/python-dicts/) correctly. Fortunately, there are brighter days ahead.

Option #2: str.format()

This newer way of getting the job done was introduced in Python 2.6. You can check out A Guide to the Newer Python String Format Techniques (https://realpython.com/python-formatted-output/) for more info.

How To Use str.format()

`str.format()` is an improvement on %-formatting. It uses normal function call syntax and is extensible through the `__format__()` method (https://www.python.org/dev/peps/pep-3101/#controlling-formatting-on-a-per-type-basis) on the object being converted to a string.

With `str.format()`, the replacement fields are marked by curly braces:

>>>

```
>>> "Hello, {}. You are {}.".format(name, age)
'Hello, Eric. You are 74.'
```

You can reference variables in any order by referencing their index:

>>>

```
>>> "Hello, {1}. You are {0}.".format(age, name)
'Hello, Eric. You are 74.'
```

But if you insert the variable names, you get the added perk of being able to pass objects and then reference parameters and methods in between the braces:

>>>

```
>>> person = {'name': 'Eric', 'age': 74}
>>> "Hello, {name}. You are {age}.".format(name=person['name'], age=person['age'])
'Hello, Eric. You are 74.'
```

You can also use `**` to do this neat trick with dictionaries:

>>>

```
>>> person = {'name': 'Eric', 'age': 74}
>>> "Hello, {name}. You are {age}.".format(**person)
'Hello, Eric. You are 74.'
```

`str.format()` is definitely an upgrade when compared with %-formatting, but it's not all roses and sunshine.

Why str.format() Isn't Great

Code using `str.format()` is much more easily readable than code using %-formatting, but `str.format()` can still be quite verbose when you are dealing with multiple parameters and longer strings. Take a look at this:

>>>

```
>>> first_name = "Eric"
>>> last_name = "Idle"
>>> age = 74
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> print(("Hello, {first_name} {last_name}. You are {age}. " +
>>>        "You are a {profession}. You were a member of {affiliation}.") \
>>>        .format(first_name=first_name, last_name=last_name, age=age, \
>>>                profession=profession, affiliation=affiliation))
'Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.'
```

If you had the variables you wanted to pass to `.format()` in a dictionary, then you could just unpack it with `.format(**some_dict)` and reference the values by key in the string, but there has got to be a better way to do this.

f-Strings: A New and Improved Way to Format Strings in Python

The good news is that f-strings are here to save the day. They slice! They dice! They make julienne fries! Okay, they do none of those things, but they do make formatting easier. They joined the party in Python 3.6. You can read all about it in PEP 498 (https://www.python.org/dev/peps/pep-0498/), which was written by Eric V. Smith in August of 2015.

Also called "formatted string literals," f-strings are string literals that have an `f` at the beginning and curly braces containing expressions that will be replaced with their values. The expressions are evaluated at runtime and then formatted using the `__format__` protocol. As always, the Python docs (https://docs.python.org/3/reference/lexical_analysis.html#f-strings) are your friend when you want to learn more.

Here are some of the ways f-strings can make your life easier.

Simple Syntax

The syntax is similar to the one you used with `str.format()` but less verbose. Look at how easily readable this is:

>>>

```
>>> name = "Eric"
>>> age = 74
>>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

It would also be valid to use a capital letter `F`:

>>>

```
>>> F"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

Do you love f-strings yet? I hope that, by the end of this article, you'll answer `>>> F"Yes!"` (https://twitter.com/dbader_org/status/992847368440561664).

Arbitrary Expressions

Because f-strings are evaluated at runtime, you can put any and all valid Python expressions in them. This allows you to do some nifty things.

You could do something pretty straightforward, like this:

>>>

```
>>> f"{2 * 37}"
'74'
```

But you could also call functions. Here’s an example:

>>>

```
>>> def to_lowercase(input):
...     return input.lower()

>>> name = "Eric Idle"
>>> f"{to_lowercase(name)} is funny."
'eric idle is funny.'
```

You also have the option of calling a method directly:

>>>

```
>>> f"{name.lower()} is funny."
'eric idle is funny.'
```

You could even use objects created from classes with f-strings. Imagine you had the following class:

```
class Comedian:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __str__(self):
        return f"{self.first_name} {self.last_name} is {self.age}."

    def __repr__(self):
        return f"{self.first_name} {self.last_name} is {self.age}. Surprise!"
```

You’d be able to do this:

>>>

```
>>> new_comedian = Comedian("Eric", "Idle", "74")
>>> f"{new_comedian}"
'Eric Idle is 74.'
```

The `__str__()` and `__repr__()` methods (https://realpython.com/operator-function-overloading/) deal with how objects are presented as strings, so you’ll need to make sure you include at least one of those methods in your class definition. If you have to pick one, go with `__repr__()` because it can be used in place of `__str__()`.

The string returned by `__str__()` is the informal string representation of an object and should be readable. The string returned by `__repr__()` is the official representation and should be unambiguous. Calling `str()` and `repr()` is preferable to using `__str__()` and `__repr__()` directly.

By default, f-strings will use `__str__()` , but you can make sure they use `__repr__()` if you include the conversion flag `!r` :

>>>

```
>>> f"{new_comedian}"
'Eric Idle is 74.'
>>> f"{new_comedian!r}"
'Eric Idle is 74. Surprise!'
```

If you'd like to read some of the conversation that resulted in f-strings supporting full Python expressions, you can do so here (https://mail.python.org/pipermail/python-ideas/2015-July/034726.html).

Remove ads (https://realpython.com/account/join/)

Multiline f-Strings

You can have multiline strings:

>>>

```
>>> name = "Eric"
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> message = (
...     f"Hi {name}. "
...     f"You are a {profession}. "
...     f"You were in {affiliation}."
... )
>>> message
'Hi Eric. You are a comedian. You were in Monty Python.'
```

But remember that you need to place an `f` in front of each line of a multiline string. The following code won't work:

>>>

```
>>> message = (
...     f"Hi {name}. "
...     "You are a {profession}. "
...     "You were in {affiliation}."
... )
>>> message
'Hi Eric. You are a {profession}. You were in {affiliation}.'
```

If you don't put an `f` in front of each individual line, then you'll just have regular, old, garden-variety strings and not shiny, new, fancy f-strings.

If you want to spread strings over multiple lines, you also have the option of escaping a return with a `\` :

>>>

```
>>> message = f"Hi {name}. " \
...           f"You are a {profession}. " \
...           f"You were in {affiliation}."
...
>>> message
'Hi Eric. You are a comedian. You were in Monty Python.'
```

But this is what will happen if you use `"""` :

>>>

```
>>> message = f"""
...     Hi {name}.
...     You are a {profession}.
...     You were in {affiliation}.
... """
...
>>> message
'\n    Hi Eric.\n    You are a comedian.\n    You were in Monty Python.\n'
```

Read up on indentation guidelines in PEP 8 (https://pep8.org/).

Speed

The `f` in f-strings may as well stand for "fast."

f-strings are faster than both %-formatting and `str.format()`. As you already saw, f-strings are expressions evaluated at runtime rather than constant values. Here's an excerpt from the docs:

> "F-strings provide a way to embed expressions inside string literals, using a minimal syntax. It should be noted that an f-string is really an expression evaluated at run time, not a constant value. In Python source code, an f-string is a literal string, prefixed with `f`, which contains expressions inside braces. The expressions are replaced with their values." (Source (https://www.python.org/dev/peps/pep-0498/#abstract))

At runtime, the expression inside the curly braces is evaluated in its own scope and then put together with the string literal part of the f-string. The resulting string is then returned. That's all it takes.

Here's a speed comparison:

>>>

```
>>> import timeit
>>> timeit.timeit("""name = "Eric"
... age = 74
... '%s is %s.' % (name, age)""", number = 10000)
0.003324444866599663
```

>>>

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... '{} is {}.'.format(name, age)""", number = 10000)
0.004242089427570761
```

>>>

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... f'{name} is {age}.'""", number = 10000)
0.0024820892040722242
```

As you can see, f-strings come out on top.

However, that wasn't always the case. When they were first implemented, they had some speed issues (https://stackoverflow.com/questions/37365311/why-are-literal-formatted-strings-so-slow-in-python-3-6-alpha-now-fixed-in-3-6) and needed to be made faster than `str.format()` . A special `BUILD_STRING` opcode (https://bugs.python.org/issue27078) was introduced.

Remove ads (https://realpython.com/account/join/)

Python f-Strings: The Pesky Details

Now that you've learned all about why f-strings are great, I'm sure you want to get out there and start using them. Here are a few details to keep in mind as you venture off into this brave new world.

Quotation Marks

You can use various types of quotation marks inside the expressions. Just make sure you are not using the same type of quotation mark on the outside of the f-string as you are using in the expression.

This code will work:

>>>

```
>>> f"{'Eric Idle'}"
'Eric Idle'
```

This code will also work:

>>>

```
>>> f'{"Eric Idle"}'
'Eric Idle'
```

You can also use triple quotes:

>>>

```
>>> f"""Eric Idle"""
'Eric Idle'
```

>>>

```
>>> f'''Eric Idle'''
'Eric Idle'
```

If you find you need to use the same type of quotation mark on both the inside and the outside of the string, then you can escape with `\`:

>>>

```
>>> f"The \"comedian\" is {name}, aged {age}."
'The "comedian" is Eric Idle, aged 74.'
```

Dictionaries

Speaking of quotation marks, watch out when you are working with dictionaries. If you are going to use single quotation marks for the keys of the dictionary, then remember to make sure you're using double quotation marks for the f-strings containing the keys.

This will work:

>>>

```
>>> comedian = {'name': 'Eric Idle', 'age': 74}
>>> f"The comedian is {comedian['name']}, aged {comedian['age']}."
The comedian is Eric Idle, aged 74.
```

But this will be a hot mess with a syntax error (https://realpython.com/invalid-syntax-python/):

>>>

```
>>> comedian = {'name': 'Eric Idle', 'age': 74}
>>> f'The comedian is {comedian['name']}, aged {comedian['age']}.'
  File "<stdin>", line 1
    f'The comedian is {comedian['name']}, aged {comedian['age']}.'
                                    ^
SyntaxError: invalid syntax
```

If you use the same type of quotation mark around the dictionary keys as you do on the outside of the f-string, then the quotation mark at the beginning of the first dictionary key will be interpreted as the end of the string.

Braces

In order to make a brace appear in your string, you must use double braces:

>>>

```
>>> f"{{70 + 4}}"
'{70 + 4}'
```

Note that using triple braces will result in there being only single braces in your string:

>>>

```
>>> f"{{{70 + 4}}}"
'{74}'
```

However, you can get more braces to show if you use more than triple braces:

>>>

```
>>> f"{{{{70 + 4}}}}"
'{{70 + 4}}'
```

Backslashes

As you saw earlier, it is possible for you to use backslash escapes in the string portion of an f-string. However, you can't use backslashes to escape in the expression part of an f-string:

>>>

```
>>> f"{\"Eric Idle\"}"
  File "<stdin>", line 1
    f"{\"Eric Idle\"}"
                      ^
SyntaxError: f-string expression part cannot include a backslash
```

You can work around this by evaluating the expression beforehand and using the result in the f-string:

>>>

```
>>> name = "Eric Idle"
>>> f"{name}"
'Eric Idle'
```

Go Forth and Format!

You can still use the older ways of formatting strings, but with f-strings, you now have a more concise, readable, and convenient way that is both faster and less prone to error. Simplifying your life by using f-strings is a great reason to start using Python 3.6 if you haven't already made the switch. (If you are still using Python 2, don't forget that 2020 (https://pythonclock.org/) will be here soon!)

According to the Zen of Python (https://www.python.org/dev/peps/pep-0020/), when you need to decide how to do something, then "[t]here should be one– and preferably only one –obvious way to do it." Although f-strings aren't the only possible way for you to format strings, they are in a great position to become that one obvious way to get the job done.

Further Reading

If you'd like to read an extended discussion about string interpolation, take a look at PEP 502 (https://www.python.org/dev/peps/pep-0502/). Also, the PEP 536 draft (https://www.python.org/dev/peps/pep-0536/) has some more thoughts about the future of f-strings.

Free PDF Download: Python 3 Cheat Sheet (https://realpython.com/bonus/python-cheat-sheet-short/)

For more fun with strings, check out the following articles:

- Python String Formatting Best Practices (https://realpython.com/python-string-formatting/) by Dan Bader
- Practical Introduction to Web Scraping in Python (https://realpython.com/python-web-scraping-practical-introduction/) by Colin OKeefe

Happy Pythoning!

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 3's f-Strings: An Improved String Formatting Syntax** (https://realpython.com/courses/python-3-f-strings-improved-string-formatting-syntax/)