



Python's F-String for String Interpolation and Formatting

by Joanna Jablonski Nov 30, 2024 35m 69 Comments basics python

Mark as Completed



Share

Table of Contents

- [Interpolating and Formatting Strings Before Python 3.6](#)
 - [The Modulo Operator \(%\)](#)
 - [The str.format\(\) Method](#)
- [Doing String Interpolation With F-Strings in Python](#)
 - [Interpolating Values and Objects in F-Strings](#)
 - [Embedding Expressions in F-Strings](#)
- [Formatting Strings With Python's F-String](#)
- [Other Relevant Features of F-Strings](#)
 - [Using an Object's String Representations in F-Strings](#)
 - [Self-Documenting Expressions for Debugging](#)
 - [Comparing Performance: F-String vs Traditional Tools](#)
- [Upgrading F-Strings: Python 3.12 and Beyond](#)
 - [Using Quotation Marks](#)
 - [Using Backslashes](#)
 - [Writing Inline Comments](#)
 - [Deciphering F-String Error Messages](#)
- [Using Traditional String Formatting Tools Over F-Strings](#)
 - [Dictionary Interpolation](#)
 - [Lazy Evaluation in Logging](#)
 - [SQL Database Queries](#)
 - [Internationalization and Localization](#)
- [Converting Old String Into F-Strings Automatically](#)
- [Frequently Asked Questions](#)

Help

[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](#)

Python **f-strings** offer a concise and efficient way to interpolate variables, objects, and expressions directly into strings. By prefixing a string with `f` or `F`, you can embed expressions within curly braces (`{}`), which are evaluated at runtime.


This makes f-strings faster and more readable compared to older approaches like the [modulo \(%\) operator](#) or the string `.format()` method. Additionally, f-strings support advanced string formatting using Python's string [format mini-language](#).

By the end of this tutorial, you'll understand that:

- An f-string in Python is a string literal **prefixed with `f` or `F`**, allowing for the **embedding of expressions** within **curly braces** `{}`.
- To include dynamic content in an f-string, place your expression or variable inside the braces to **interpolate** its value into the string.
- An **f-string error** in Python often occurs due to syntax issues, such as **unmatched braces** or **invalid expressions** within the string.
- **Python 3.12** improved f-strings by allowing **nested expressions** and the use of **backslashes**.

This tutorial will guide you through the features and advantages of f-strings, including interpolation and formatting. By familiarizing yourself with these features, you'll be able to effectively use f-strings in your Python projects.

Get Your Code: [Click here to download the free sample code](#) that shows you how to do string interpolation and formatting with Python's f-strings.

 **Take the Quiz:** Test your knowledge with our interactive “Python F-Strings” quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

[Python F-Strings](#)

In this quiz, you'll test your knowledge of Python f-strings. With this knowledge, you'll be able to include all sorts of Python expressions inside your strings.

Interpolating and Formatting Strings Before Python 3.6

Before Python 3.6, you had two main tools for interpolating values, variables, and expressions inside string literals:

1. The [string interpolation operator \(%\)](#), or modulo operator
2. The [`str.format\(\)`](#) method

You'll get a refresher on these two string interpolation tools in the following sections. You'll also learn about the string formatting capabilities that these tools offer in Python.

The Modulo Operator (%)

The modulo operator (%) was the first tool for string interpolation and formatting in Python and has been in the language since the beginning. Here's what using this operator looks like in practice:

Python



```
>>> name = "Jane"

>>> "Hello, %s!" % name
'Hello, Jane!'
```

In this quick example, you use the % operator to interpolate the value of your name variable into a string literal. The interpolation operator takes two operands:

- A string literal containing one or more conversion specifiers
- The object or objects that you're interpolating into the string literal

The **conversion specifiers** work as replacement fields. In the above example, you use the %s combination of characters as a conversion specifier. The % symbol marks the start of the specifier, while the s letter is the **conversion type** and tells the operator that you want to convert the input object into a string.

If you want to insert more than one object into your target string, then you can use a [tuple](#). Note that the number of objects in the tuple must match the number of format specifiers in the string:

Python



```
>>> name = "Jane"
>>> age = 25

>>> "Hello, %s! You're %s years old." % (name, age)
'Hello, Jane! You're 25 years old.'
```

In this example, you use a tuple of values as the right-hand operand to %. Note that you've used a string and an integer. Because you use the %s specifier, Python converts both objects to strings.

You can also use dictionaries as the right-hand operand in your interpolation expressions. To do this, you need to create conversion specifiers that enclose key names in parentheses:

Python



```
>>> "Hello, %(name)s! You're %(age)s years old." % {"name": "Jane", "age": 25}
'Hello, Jane! You're 25 years old.'
```

This syntax provides a readable approach to string interpolation with the % operator. You can use descriptive key names instead of relying on the positional order of values.

When you use the % operator for string interpolation, you can use conversion specifiers. They provide some string formatting capabilities that take advantage of **conversion types**, **conversion flags**, and some characters like the period (.) and the asterisk (*). Consider the following example:

Python



```
>>> "Balance: $%.2f" % 5425.9292
'Balance: $5425.93'

>>> print("Name: %s\nAge: %5s" % ("John", 35))
Name: John
Age:    35
```

In the first example, you use the %.2f conversion specifier to represent currency values. The f letter tells the operator to convert to a floating-point number. The .2 part defines the precision to use when converting the input. In the second example, you use %5s to align the age value five positions to the right.

Note: Formatting with the modulo operator is inspired by [printf\(\)](#) formatting used in [C](#) and many other programming languages.

Even though the % operator provides a quick way to interpolate and format strings, it has a few issues that lead to common errors. For example, it's difficult to interpolate tuples in your strings:

Python



```
>>> "The personal info is: %s" % ("John", 35)
Traceback (most recent call last):
...
TypeError: not all arguments converted during string formatting
```

In this example, the operator fails to display the tuple of data because it interprets the tuple as two separate values. You can fix this issue by wrapping the data in a single-item tuple:

Python

```
>>> "The personal info is: %s" % (("John", 35),)
"The personal info is: ('John', 35)"
```

This syntax fixes the issue, and now your string successfully shows the tuple of data. However, the syntax is hard to read, understand, and remember, isn't it?

Another issue with the % operator is its limited formatting capabilities and the lack of support for Python's [string formatting mini-language](#), which provides a powerful tool to format your strings.

The str.format() Method

The str.format() method is an improvement compared to the % operator because it fixes a couple of issues and supports the string formatting mini-language. With .format(), curly braces delimit the replacement fields:

Python

```
>>> name = "Jane"
>>> age = 25

>>> "Hello, {}! You're {} years old.".format(name, age)
"Hello, Jane! You're 25 years old."
```

For the .format() method to work, you must provide replacement fields using curly brackets. If you use empty brackets, then the method interpolates its arguments into the target string based on position.

You can manually specify the interpolation order by referencing the position of each argument to .format() using zero-based indices. For example, the code below switches the arguments to .format() in the target string:

Python

```
>>> "Hello, {1}! You're {0} years old.".format(age, name)
"Hello, Jane! You're 25 years old."
```

In this example, you use numeric indices to manually define the order in which you want to interpolate the values that you pass as arguments to .format().

You can also use keyword arguments in the call to the method and enclose the argument names in your replacement fields:

Python

```
>>> "Hello, {name}! You're {age} years old.".format(name="Jane", age=25)
"Hello, Jane! You're 25 years old."
```

This example showcases how .format() interpolates keyword arguments by their names into the target string. This construct considerably improves your code's readability compared to the previous example and to the examples using the % operator.

Finally, the .format() method allows you to use dictionaries to provide the values that you want to interpolate into your strings:

Python

```
>>> person = {"name": "Jane", "age": 25}

>>> "Hello, {name}! You're {age} years old.".format(**person)
"Hello, Jane! You're 25 years old."
```

In this example, you use a [dictionary](#) containing the data to interpolate. Then, you use the [dictionary unpacking operator \(**\) to provide the arguments to .format\(\)](#).

The .format() method supports **format specifiers**. These are strings that you insert into replacement fields to format the values that you want to interpolate. Consider the following examples:

Python



```
>>> "Balance: ${:.2f}".format(5425.9292)
'Balance: $5425.93'

>>> "{:=^30}".format("Centered string")
'====Centered string===='
```

In the first example, you use the `:.2f` format specifier. This specifier tells `.format()` to *format* the input value as a floating-point number with a precision of two. This way, you can represent currency values.

In the second example, you use the `:=^30` format specifier. In this case, you're telling `.format()` to format the input value using the `=` symbol as a filler character. The `^` symbol centers the input value by inserting `=` symbols on both sides to reach thirty characters.

Format specifiers provide a remarkable improvement over the limited formatting capabilities of the `%` operator. These specifiers have a straightforward syntax that makes up the string formatting mini-language. Thankfully, f-strings also support the string formatting mini-language, which is another cool feature of theirs. So, you won't have to use `.format()` if you don't need to.

In the upcoming sections, you'll write a few more examples of formatting strings using the mini-language with f-strings.

Doing String Interpolation With F-Strings in Python

F-strings joined the party in Python 3.6 with [PEP 498](#). Also called **formatted string literals**, f-strings are string literals that have an `f` before the opening quotation mark. They can include Python expressions enclosed in curly braces. Python will replace those expressions with their resulting values. So, this behavior turns f-strings into a string interpolation tool.

In the following sections, you'll learn about f-strings and use them to interpolate values, objects, and expressions in your string literals.

Interpolating Values and Objects in F-Strings

F-strings make the string interpolation process intuitive, quick, and concise. The syntax is similar to what you used with `.format()`, but it's less verbose. You only need to start your string literal with a lowercase or uppercase `f` and then embed your values, objects, or expressions in curly brackets at specific places:

Python



```
>>> name = "Jane"
>>> age = 25

>>> f"Hello, {name}! You're {age} years old."
'Hello, Jane! You're 25 years old.'
```

Look how readable and concise your string is now that you're using the f-string syntax. You don't need operators or methods anymore. You just embed the desired objects or expressions in your string literal using curly brackets.

It's important to note that Python evaluates f-strings at runtime. So, in this example, both `name` and `age` are interpolated into the string literal when Python runs the line of code containing the f-string. Python can only interpolate these variables because you defined them *before* the f-string, which means that they must be in [scope](#) when Python evaluates the f-string.

Embedding Expressions in F-Strings

You can embed almost any Python expression in an f-string. This allows you to do some nifty things. You could do something pretty straightforward, like the following:

Python



```
>>> f"{2 * 21}"
'42'
```

When Python runs this f-string, it multiplies 2 by 21 and immediately interpolates the resulting value into the final string.

The example above is quite basic. However, f-strings are more powerful than that. You could also use other Python expressions, including [function](#) and method calls, and even [comprehensions](#) or other more complex expressions:

Python



```
>>> name = "Jane"
>>> age = 25

>>> f"Hello, {name.upper()}! You're {age} years old."
>Hello, JANE! You're 25 years old."

>>> f"{[2**n for n in range(3, 9)]}"
'[8, 16, 32, 64, 128, 256]'
```

In the first f-string, you embed a call to the `.upper()` string method in the first replacement field. Python runs the method call and inserts the uppercased name into the resulting string. In the second example, you create an f-string that embeds a [list comprehension](#). The comprehension creates a new list of powers of 2.

Formatting Strings With Python's F-String

The expressions that you embed in an f-string are evaluated at runtime. Then, Python formats the result using the `.__format__()` special method under the hood. This method supports the string formatting [protocol](#). This protocol underpins both the `.format()` method, which you already saw, and the built-in `format()` function:

Python



```
>>> format(5425.9292, ".2f")
'5425.93'
```

The `format()` function takes a value and a **format specifier** as arguments. Then, it applies the specifier to the value to return a formatted value. The format specifier must follow the rules of the string formatting mini-language.

Just like the `.format()` method, f-strings also support the string formatting mini-language. So, you can use format specifiers in your f-strings too:

Python



```
>>> balance = 5425.9292

>>> f"Balance: ${balance:.2f}"
'Balance: $5425.93'

>>> heading = "Centered string"
>>> f"{heading:=^30}"
'====Centered string===='
```

Note that the format specifiers in these examples are the same ones that you used in [the section on `.format\(\)`](#). In this case, the embedded expression comes before the format specifier, which always starts with a colon. This syntax makes the string literals readable and concise.

You can create a wide variety of format specifiers. Some common formats include currencies, dates, and the representation of numeric values. Consider the following examples of string formatting:

Python



```
>>> integer = -1234567
>>> f"Comma as thousand separators: {integer:,}"
'Comma as thousand separators: -1,234,567'

>>> sep = "_"
>>> f"User's thousand separators: {integer:{sep}}"
'User's thousand separators: -1_234_567'

>>> floating_point = 1234567.9876
>>> f"Comma as thousand separators and two decimals: {floating_point:,.2f}"
'Comma as thousand separators and two decimals: 1,234,567.99'

>>> date = (9, 6, 2023)
>>> f"Date: {date[0]:02}-{date[1]:02}-{date[2]}"
'Date: 09-06-2023'

>>> from datetime import datetime
>>> date = datetime(2023, 9, 26)
>>> f"Date: {date:%m/%d/%Y}"
'Date: 09/26/2023'
```

These examples show how flexible the format specifiers can be. You can use them to create almost any string format. Note how in the second example, you’ve used curly brackets to embed variables or expressions in your format specifiers. This possibility allows you to create dynamic specifiers, which is pretty cool. In the last example, you format a [datetime](#) which can be formatted with special [date format specifiers](#).

Other Relevant Features of F-Strings

So far, you’ve learned that f-strings provide a quick and readable way to interpolate values, objects, and expressions into string literals. They also support the string formatting mini-language, so you can create format specifiers to format the objects that you want to insert into your strings.

In the following sections, you’ll learn about a few additional features of f-strings that may be relevant and useful in your day-to-day coding.

Using an Object’s String Representations in F-Strings

Python’s f-strings support two flags with special meaning in the interpolation process. These flags are closely related to how Python manages the [string representation](#) of objects. These flags are:

Flag	Description
!s	Interpolates the string representation from the <code>.__str__()</code> method
!r	Interpolates the string representation from the <code>.__repr__()</code> method

The `.__str__()` [special method](#) generally provides a user-friendly string representation of an object, while the `.__repr__()` method returns a developer-friendly representation. To illustrate how these methods work under the hood, consider the following class:

Python

```
# person.py

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"I'm {self.name}, and I'm {self.age} years old."

    def __repr__(self):
        return f"{type(self).__name__}(name='{self.name}', age={self.age})"
```

This class has two [instance attributes](#), `.name` and `.age`. The `.__str__()` method returns a string that consists of an informative message for users of your class. This message should be useful for end users rather than developers.

Note: To dive deeper into the `.__str__()` and `.__repr__()` methods, check out [When Should You Use. `.__repr__\(\)` vs `.__str__\(\)` in Python?](#)

In contrast, the `.__repr__()` method returns a string that's a developer-friendly representation of the object. In short, the representation tells the developer how the current instance was created. Ideally, the developer should be able to copy this string representation and create an equivalent object.

How does this discussion about string representation affect f-strings? When you create your f-strings, you can choose which string representation to use with the `!r` and `!s` flags:

Python



```
>>> from person import Person

>>> jane = Person("Jane Doe", 25)

>>> f"{jane!s}"
"I'm Jane Doe, and I'm 25 years old."

>>> f"{jane!r}"
"Person(name='Jane Doe', age=25)"
```

In the first f-string, you use the `!s` tag to interpolate the string representation that `.__str__()` returns. In the second f-string, you use the `!r` flag to interpolate the developer-friendly string representation of your current object.

These two flags are pretty relevant for you as a Python developer. Depending on your code's intended audience, you can decide which one to use. In general, it should be the one that provides more value to your users.

It's important to note that the `%` operator also supports equivalent conversion types, `s` and `r`, which work the same as the `!s` and `!r` flags in f-strings.

Self-Documenting Expressions for Debugging

F-strings have another cool feature that can be useful, especially during your [debugging](#) process. The feature helps you self-document some of your expressions. For example, say that you're dealing with a minor bug or issue in your code, and you want to know the value of a variable at a given moment in the code's execution.

For this quick check, you can insert a call to [print\(\)](#) like the following:

Python



```
>>> variable = "Some mysterious value"

>>> print(f"{variable = }")
variable = 'Some mysterious value'
```


You can use a variable name followed by an equal sign (=) in an f-string to create a [self-documented expression](#). When Python runs the f-string, it builds an expression-like string containing the variable's name, the equal sign, and the variable's current value. This f-string feature is useful for inserting quick debugging checks in your code.

Note that the whitespaces around the equal sign aren't required but they are reflected in the output:

Python



```
>>> print(f"{variable=}")
variable='Some mysterious value'

>>> print(f"{variable= }")
variable= 'Some mysterious value'

>>> print(f"{variable =}")
variable = 'Some mysterious value'
```

Even though the whitespaces aren't required, they can improve your code's readability and the output's format.

Comparing Performance: F-String vs Traditional Tools

F-strings are a bit faster than both the modulo operator (%) and the `.format()` method. That's another cool characteristic. In the script below, you use the [timeit](#) module to measure the execution time that it takes to build a string using the modulo operator, the `.format()` method, and an f-string:

Python

```
# performance.py

import timeit

name = "Linda Smith"
age = 40
strings = {
    "Modulo operator": "'Name: %s Age: %s' % (name, age)",
    ".format() method": "'Name: {} Age: {}'.format(name, age)",
    "f_string": "f'Name: {name} Age: {age}'",
}

def run_performance_test(strings):
    max_length = len(max(strings, key=len))
    for tool, string in strings.items():
        time = timeit.timeit(
            string,
            number=1000000,
            globals=globals()
        ) * 1000
        print(f"{tool}: {time:>{max_length - len(tool) + 6}.2f} ms")

run_performance_test(strings)
```

In this script, the `run_performance_test()` function takes care of measuring the execution time of the three different string interpolation tools. The `timeit.timeit()` function inside the [for loop](#) runs each interpolation tool a million times and returns the total execution time.

Then, the function prints the result to the screen. Note how your f-string in the call to `print()` takes advantage of format specifiers to conveniently format the code's output.

If you [run the script](#) from your command line, then you'll get an output similar to the following. Of course, the numbers will be different for you:

Shell



```
$ python performance.py
Modulo operator:    90.98 ms
.format() method: 144.69 ms
f_string:          87.08 ms
```

This output shows that f-strings are a bit faster than the % operator and the .format() method, which is the slowest tool because of all the required function calls. So, f-strings are readable, concise, and also fast.

Upgrading F-Strings: Python 3.12 and Beyond

Now that you've learned why f-strings are great, you're probably eager to get out there and start using them in your code. However, you need to know that f-strings up to Python [3.11](#) have a few limitations regarding the expressions that you can embed in curly brackets and a few other details.

Fortunately, Python [3.12](#) lifted those limitations by removing the old f-string parser and providing a new implementation of f-strings based on the [PEG parser](#) of Python [3.9](#). In the following sections, you'll learn about the limitations and how Python 3.12 fixed them.

Using Quotation Marks

Python supports several different types of quotation marks as delimiters in string [literals](#). You can use single (') and double quotes ("). You can also use triple single ('''') and triple double quotes ("""). All these string delimiters work for f-strings as well. This feature allows you to insert quotation marks in f-strings. It also lets you introduce string literals in the embedded expressions and even create nested f-strings.

A typical use case of using different quotation marks in an f-string is when you need to use an apostrophe or access a dictionary key in an embedded expression:

Python



```
>>> person = {"name": "Jane", "age": 25}

>>> f"Hello, {person['name']}! You're {person['age']} years old."
"Hello, Jane! You're 25 years old."
```

In this example, you have a dictionary with a person's data. To define the f-string, you use double quotes. To access the dictionary key, you use single quotes. In the "You're" contraction, you use a single quote as an apostrophe.

So, where's the quote-related limitation of f-strings up to Python 3.11? The problem is that you can't reuse quotation marks in an f-string:

Python



```
>>> f"Hello, {person["name"]}!"
File "<input>", line 1
    f"Hello, {person["name"]}!"
                      ^^^^
SyntaxError: f-string: unmatched '['
```

In this example, when you try to reuse double quotes to access the dictionary key, your f-string fails, and Python raises a [SyntaxError](#) exception.

Fortunately, the new f-strings in Python 3.12 solved this issue, allowing you to reuse quotes:

Python



```
>>> # Python 3.12

>>> person = {"name": "Jane", "age": 25}
>>> f"Hello, {person["name"]}!"
'Hello, Jane!'
```

In this example, you reuse the double quotes in your embedded expressions, and the f-string works correctly. The limitation is gone. However, it may not be clear if reusing quotations in this example is cleaner than differentiating nested strings with different quotation marks.

There's another f-string limitation that's closely related to quotation marks. You can only nest as many f-strings as there are quote delimiters in Python:

Python

```
>>> f"""{
...     f' '{
...         f"{f'{42}'}"
...     }' '
... }"""
'42'

>>> f"""{
...     f' '{
...         f"{f' '{f'{42}'}' }'"
...     }' '
... }"""
File "<stdin>", line 1
    (f'{f' '{f'{42}'}' }')
        ^
SyntaxError: f-string: f-string: unterminated string
```

The number of nesting levels in an f-string up to Python 3.11 is limited by the available string delimiters, which are `"`, `'`, `"""`, and `'''`. So, you only have four delimiters that you can use to differentiate your levels of nesting.

In Python 3.12, this limitation is removed because you can reuse quotation marks:

Python

```
>>> # Python 3.12

>>> f"{
...     f"{
...         f"{
...             f"{
...                 f"Deeply nested f-string!"
...             }"
...         }"
...     }"
... }"
'Deeply nested f-string!'
```

Before the new f-string implementation, there was no formal limit on how many levels of nesting you could have. However, the fact that you couldn't reuse string quotes imposed a natural limit on the allowed levels of nesting in f-string literals. Starting with Python 3.12, you can reuse quotes, so there are no limits for nesting f-strings.

Using Backslashes

Another limitation of f-strings before 3.12 is that you can't use backslash characters in embedded expressions. Consider the following example, where you try to [concatenate strings](#) using the newline (`\n`) escape sequence:


Python

```
>>> words = ["Hello", "World!", "I", "am", "a", "Pythonista!"]

>>> f"{'\n'.join(words)}"
File "<input>", line 1
    f"{'\n'.join(words)}"
        ^
SyntaxError: f-string expression part cannot include a backslash
```

In this example, you get a `SyntaxError` because f-strings don't allow backslash characters inside expressions delimited by curly brackets.

Again, the new f-string implementation that comes with Python 3.12 solves the issue:

```
Python 
```

```
>>> # Python 3.12

>>> words = ["Hello", "World!", "I", "am", "a", "Pythonista!"]


>>> f"{'\n'.join(words)}"
'Hello\nWorld!\nI\nam\na\nPythonista!'

>>> print(f"{'\n'.join(words)}")
Hello
World!
I
am
a
Pythonista!
```

The new f-string implementation lifted the limitation of using backslash characters in embedded expressions, so you can now use escape sequences in your f-strings.

Writing Inline Comments


F-strings up to Python 3.11 don't allow you to use the `#` symbol in embedded expressions. Because of that, you can't insert comments in embedded expressions. If you try to do it, then you'll get a syntax error:

```
Python 
```

```
>>> employee = {
...     "name": "John Doe",
...     "age": 35,
...     "job": "Python Developer",
... }

>>> f"""Storing employee's data: {
...     employee['name'].upper() # Always uppercase name before storing
... }"""
File "<stdin>", line 3
    }"""
    ^
SyntaxError: f-string expression part cannot include '#'
```

When you use `#` to introduce a comment in an f-string, you get a `SyntaxError`. Fortunately, the new f-strings in Python 3.12 also fix this problem:

```
Python 
```

```
>>> # Python 3.12

>>> employee = {
...     "name": "John Doe",
...     "age": 35,
...     "job": "Python Developer",
... }

>>> f"Storing employee's data: {
...     employee["name"].upper() # Always uppercase name before storing
... }"
"Storing employee's data: JOHN DOE"
```

Now you can add inline comments if you ever need to clarify something in the embedded expressions of an f-string. Another improvement is that you can add line breaks inside the curly braces, similar to what you can do inside parentheses outside f-strings. You don't even need to use the triple-quoted multiline strings to do this.

Deciphering F-String Error Messages

Python's new PEG parser opens the door to many improvements in the language. From the user's perspective, one of the most valuable improvements is that you now have [better error messages](#). These enhanced error messages weren't available for f-strings up to Python 3.11 because they didn't use the PEG parser. So, the error messages related to f-strings were less specific and clear.

Python 3.12 came along to fix this issue, too. Take a look at the following example, which compares the error message for an incorrect f-string in both 3.11 and 3.12:

Python

```
>>> # Python 3.11
>>> f"{42 + }"
File "<stdin>", line 1
    (42 + )
        ^
SyntaxError: f-string: invalid syntax

>>> # Python 3.12
>>> f"{42 + }"
File "<stdin>", line 1
    f"{42 + }"
        ^
SyntaxError: f-string: expecting '=', or '!', or ':', or '}'
```

The error message in the first example is generic and doesn't point to the exact location of the error within the offending line. Additionally, the expression is surrounded by parentheses, which adds noise to the problem because the original code doesn't include parentheses.

In Python 3.12, the error message is more verbose. It signals the exact location of the problem in the affected line. Additionally, the exception message provides some suggestions that might help you fix the issue.

In this specific example, the suggestions aren't that useful because they focus on an operator that's possibly wrong. However, having the exact location where the problem happened gives you a strong clue. You have a missing operand in the embedded expression.

Using Traditional String Formatting Tools Over F-Strings

Even though f-strings are a pretty cool and popular Python feature, they're not the one-size-fits-all solution. Sometimes the modulo operator (%) or the `.format()` method provides a better solution. Sometimes, they're your only option. It all depends on your specific use case.

In the following sections, you'll learn about a few situations where f-strings may not be the best option. To kick things off, you'll start with a use case that's closely related to your code's readability. That's when you want to interpolate values from a dictionary into a given string.

Dictionary Interpolation

Interpolating dictionary values into a string may be a common requirement in your code. Because you now know that f-strings are neat, you may think of using them for this task. You end up with a piece of code that looks like the following:

Python

```
>>> person = {"name": "Jane Doe", "age": 25}

>>> f"Hello, {person['name']}! You're {person['age']} years old."
"Hello, Jane Doe! You're 25 years old."
```

That's great! The code works just fine. However, it doesn't look clean because of all those dictionary key lookups embedded in the string. The f-string looks cluttered and may be hard to read. How about using the `.format()` method?

Here's a new version of your code:

Python



```
>>> "Hello, {name}! You're {age} years old.".format(**person)
Hello, Jane Doe! You're 25 years old."

>>> "Hello, {name}!".format(**person)
'Hello, Jane Doe!'
```

In this example, you use direct names instead of dictionary lookups in the replacement fields. The only additional requirement is that you need to use the dictionary unpacking operator (`**`) in the call to `.format()`. Now, the string looks cleaner and is also a bit shorter than the version using an f-string.

As an additional gain, it's important to note that the number of replacement fields in the string doesn't have to match the number of keys in the input dictionary. The `.format()` method will ignore unnecessary keys.

You also have the option of using the modulo operator, though:

Python



```
>>> "Hello, %(name)s! You're %(age)s years old." % person
Hello, Jane Doe! You're 25 years old."

>>> "Hello, %(name)s!" % person
'Hello, Jane Doe!'
```

This time, the string is even shorter. You use direct names in the replacement fields and don't have to use the dictionary unpacking operator because the modulo operator unpacks the dictionary for you. However, some may say that the replacement fields aren't that readable and that the modulo operator has limited formatting capabilities.

So, what version do you prefer? Share your thoughts in the comments!

Lazy Evaluation in Logging

Providing [logging](#) messages is a common example of those use cases where you shouldn't use f-strings or `.format()`. The `logging` module runs string interpolation [lazily](#) to optimize performance according to the selected logging level.

For example, you may have a hundred debugging messages but only ten warning messages in your code. If you use an f-string or the `.format()` method to construct your logging messages, then Python will interpolate all the strings regardless of the logging level that you've chosen.

However, if you use the `%` operator and provide the values to interpolate as arguments to your logging functions, then you'll optimize the interpolation process. The `logging` module will only interpolate those strings that belong to the current and higher logging levels.

Consider the following example:

Python



```
>>> import logging
>>> msg = "This is a %s message!"

>>> logging.warning(msg, "WARNING")
WARNING:root:This is a WARNING message!

>>> logging.debug(msg, "DEBUGGING")
```

In this example, you use the modulo operator syntax to create the logging message. Then, you pass the value that you want to interpolate as an argument to the logging functions. Because `WARNING` is the default logging level, only the messages at this level and higher will be logged. That's why the `debug()` function doesn't generate any output.

In the above call to `debug()`, the string interpolation never happens because you're using a higher logging level. However, if you use `.format()` like in the code below, then the interpolation will always happen:

Python



```
>>> msg = "This is a {} message!"

>>> logging.debug(msg.format("DEBUGGING"))
```

If you call `debug()` a million times inside a loop, then Python will eagerly evaluate its argument, and the interpolation will happen a million times. This behavior will add performance overhead to your code. That's why the `logging` module does the interpolation lazily.

The lazy nature of how `logging` does string formatting can make a difference, and it's only possible using the modulo operator.

SQL Database Queries

Using any string interpolation tool is a bad idea when you're building SQL queries with dynamic parameters. In this scenario, interpolation tools invite [SQL injection attacks](#).

To illustrate the problem, say that you're working with a [PostgreSQL](#) database using the [Psycopg 2](#) adapter, and you want to run a query to get all the users with a given role or set of privileges. You come up with one of the following queries:

Python

```
import psycopg2

connection = psycopg2.connect(
    database="db",
    user="user",
    password="password"
)
cursor = connection.cursor()

role = "admin"

query_modulo = "SELECT * FROM users WHERE role = '%s'" % role
query_format = "SELECT * FROM users WHERE role = '{role}'".format(role=role)
query_f_string = f"SELECT * FROM users WHERE role = '{role}'"

cursor.execute(query_modulo)
cursor.execute(query_format)
cursor.execute(query_f_string)
```

All of these strings directly insert the query parameter into the final query without any validation or security check. If you run any of these queries using the `.execute()` method, then the database won't be able to perform any security checks on the parameters, which makes your code prone to SQL injection attacks.

Note: The code in the above example doesn't run because of the missing `psycopg2` library and the assumption of a certain database structure and setup. It's a demonstrative code example only.

To avoid the risk of SQL injection, you can use the `%` operator syntax to build the query template and then provide the query parameter as the second argument to the `.execute()` method in a tuple or list:

Python

```
query_template = "SELECT * FROM users WHERE role = %s"

cursor.execute(query_template, (role,))
```

In this example, you use the `%` operator syntax to create the query template. Then, you provide the parameters as an independent argument to `.execute()`. In this case, the database system will use the specified type and value of `role` when executing the query. This practice offers protection against SQL injection.

Note: You should only use the modulo operator syntax in the string literal that represents the query template. You shouldn't use the operator and the actual sequence of parameters to build the final query. Just let `.execute()` do the hard work and build the final query for you in a safer way.

In short, you must avoid using any string interpolation tool to build dynamic queries beforehand. Instead, use the `%` operator syntax to build the query template and pass the query parameters to `.execute()` in a sequence.

Internationalization and Localization

When you want to provide [internationalization and localization](#) in a Python project, the `.format()` method is the way to go:

Python



```
>>> greeting_template = "{greeting} Pythonista!"

>>> greeting_en = "Good Morning!"
>>> greeting_es = "¡Buenos días!"
>>> greeting_fr = "Bonjour!"

>>> for greeting in (greeting_en, greeting_es, greeting_fr):
...     print(greeting_template.format(greeting=greeting))
...
Good Morning! Pythonista!
¡Buenos días! Pythonista!
Bonjour! Pythonista!
```

You can support multiple languages using string templates. Then, you can handle localized string formatting based on the user's locale. The `.format()` method will allow you to dynamically interpolate the appropriate strings depending on the user's language selection.

Converting Old String Into F-Strings Automatically

If you're working on porting a legacy codebase to modern Python, and one of your goals is to convert all your strings into f-strings, then you can use the [flynt](#) project. This tool allows you to convert traditional strings into f-strings quickly.

To use `flynt`, you need to [pip](#) install it first:

Shell



```
$ python -m pip install flynt
```

This command downloads and installs `flynt` in your current Python environment. Once you have it installed, you can use the command against your code files. For example, say that you have the following Python file:

Python

```
# sample.py

name = "Jane"
age = 25

print("Hello, %s! You're %s years old." % (name, age))
```

If you want to update this file and start using f-strings instead of the `%` operator, then you can just run the following command:

Shell



```
$ flynt sample.py
Running flynt v.1.0.1
Using config file at ../pyproject.toml

Flynt run has finished. Stats:

Execution time:                0.002s
Files checked:                 1
Files modified:                1
Character count reduction:      78 (98.73%)

Per expression type:
Old style (`%`) expressions attempted:  1/2 (50.0%)
No `.format(...)` calls attempted.
No concatenations attempted.
No static string joins attempted.
F-string expressions created:          1
...
```

This command tells `flynt` to update the content of your `sample.py` file by replacing strings that use the `%` operator and the `.format()` method with equivalent f-strings. Note that this command will modify your files in place. So, after running the command, `sample.py` will look something like the following:

Python

```
# sample.py

name = "Jane"
age = 25

print(f"Hello, {name}! You're {age} years old.")
```

That’s cool, isn’t it? You can also run `flynt` against a complete directory containing a large Python codebase. It’ll scan every file and convert the old strings into f-strings. So, the tool is quite useful if you’re modernizing your codebase.


Frequently Asked Questions

Now that you have some experience with Python f-strings, you can use the questions and answers below to check your understanding and recap what you’ve learned.

These FAQs are related to the most important concepts you’ve covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer:

What is an f-string in Python?	Show/Hide
How do you write an f-string in Python?	Show/Hide
How do you format numbers in f-strings?	Show/Hide
Can you embed expressions in f-strings?	Show/Hide
What are the advantages of f-strings compared to traditional tools?	Show/Hide
What limitations did f-strings have before Python 3.12?	Show/Hide

How did you do? Would you like to take a quick quiz to evaluate your new skills? If so, then click the link below:

 **Take the Quiz:** Test your knowledge with our interactive “Python F-Strings” quiz. You’ll receive a score upon completion to help you track your learning progress:




Interactive Quiz
[Python F-Strings](#)

In this quiz, you'll test your knowledge of Python f-strings. With this knowledge, you'll be able to include all sorts of Python expressions inside your strings.

Mark as Completed



 Share

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](#)

About **Joanna Jablonski**



Joanna is the Executive Editor of Real Python. She loves natural languages just as much as she loves programming languages!

» [More about Joanna](#)

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:



[Adriana](#)



[Aldren](#)



[Brenda](#)



[Bartosz](#)



[David](#)



[Dan](#)



[Geir Arne](#)



[Jim](#)



[Kate](#)



[Leodanis](#)



[Martin](#)



[Pavel](#)

What Do You Think?

Rate this article:



[LinkedIn](#) [Twitter](#) [Bluesky](#) [Facebook](#) [Email](#)

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.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next [“Office Hours” Live Q&A Session](#). Happy Pythoning!

Keep Learning

Related Topics: [basics](#) [python](#)

Recommended Video Course: [Python 3's F-Strings: An Improved String Formatting Syntax](#)

Related Tutorials:

- [Python Exceptions: An Introduction](#)
- [Logging in Python](#)
- [A Guide to Modern Python String Formatting Tools](#)
- [Python's Format Mini-Language for Tidy Strings](#)
- [Dictionaries in Python](#)

Learn Python

[Start Here](#)
[Learning Resources](#)
[Code Mentor](#)
[Python Reference](#)
[Support Center](#)

Courses & Paths

[Learning Paths](#)
[Quizzes & Exercises](#)
[Browse Topics](#)
[Workshops](#)
[Books](#)

Community

[Podcast](#)
[Newsletter](#)
[Community Chat](#)
[Office Hours](#)
[Learner Stories](#)

Membership

[Plans & Pricing](#)
[Team Plans](#)
[For Business](#)
[For Schools](#)
[Reviews](#)

Company

[About Us](#)
[Team](#)
[Sponsorships](#)
[Careers](#)
[Press Kit](#)
[Merch](#)



[Privacy Policy](#) · [Terms of Use](#) · [Security](#) · [Contact](#)

♥ Happy Pythoning!

© 2012–2025 DevCademy Media Inc. DBA Real Python. All rights reserved.
REALPYTHON™ is a trademark of DevCademy Media Inc.

