

Python String Formatting: Available Tools and Their Features

by Leodanis Pozo Ramos | Dec 01, 2024 | 22m | 26 Comments | [basics](#) [best-practices](#) [python](#)

Mark as Completed



Share

Table of Contents

- [Interpolating and Formatting Strings in Python](#)
- [Using F-Strings to Format Strings](#)
 - [Using the Formatting Mini-Language With F-Strings](#)
 - [Formatting Strings With F-Strings: A Practical Example](#)
- [Using str.format\(\) to Format Strings](#)
 - [Using the Formatting Mini-Language With .format\(\)](#)
 - [Formatting Strings With .format\(\): Practical Examples](#)
- [Formatting Strings With the Modulo Operator \(%\)](#)
 - [Using Conversion Specifiers](#)
 - [Using Conversion Flags](#)
- [Deciding Which String Formatting Tool to Use](#)
- [Conclusion](#)
- [Frequently Asked Questions](#)

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](#)

String formatting is essential in Python for creating dynamic and well-structured text by inserting values into strings. This tutorial covers various methods, including f-strings, the `.format()` method, and the modulo operator (%). Each method has unique features and benefits for different use cases. The string formatting mini-language provides additional control over the format, allowing for aligned text, numeric formatting, and more.


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

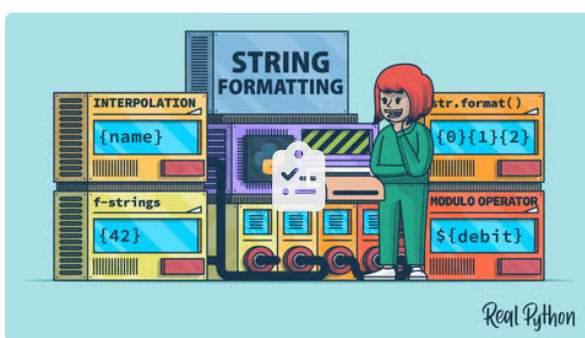
Help

- **String formatting in Python** involves inserting and formatting values within strings using interpolation.
- **Python supports different types of string formatting**, including f-strings, the `.format()` method, and the modulo operator (%).
- **F-strings** are generally the most readable and efficient option for eager interpolation in Python.
- **Python's string formatting mini-language** offers features like alignment, type conversion, and numeric formatting.
- While **f-strings** are more readable and efficient compared to `.format()` and the % operator, the `.format()` method supports lazy evaluation.

To get the most out of this tutorial, you should be familiar with Python's [string data type](#) and the available [string interpolation](#) tools. Having a basic knowledge of the string [formatting mini-language](#) is also a plus.

Get Your Code: [Click here to download the free sample code](#) you'll use to learn about Python's string formatting tools.

 **Take the Quiz:** Test your knowledge with our interactive “Python String Formatting: Available Tools and Their Features” quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

[Python String Formatting: Available Tools and Their Features](#)

You can take this quiz to test your understanding of the available tools for string formatting in Python, as well as their strengths and weaknesses. These tools include f-strings, the `.format()` method, and the modulo operator.

Interpolating and Formatting Strings in Python

[String interpolation](#) involves generating strings by inserting other strings or objects into specific places in a base string or template. For example, here's how you can do some string interpolation using an f-string:

Python

```
>>> name = "Bob"

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

In this quick example, you first have a Python [variable](#) containing a string object, "Bob". Then, you create a new string using an f-string. In this string, you insert the content of your name variable using a replacement field. When you run this last line of code, Python builds a final string, 'Hello, Bob!'. The insertion of name into the f-string is an interpolation.

Note: To dive deeper into string interpolation, check out the [String Interpolation in Python: Exploring Available Tools](#) tutorial.

When you do string interpolation, you may need to *format* the interpolated values to produce a well-formatted final string. To do this, you can use different string interpolation tools that support string formatting. In Python, you have these three tools:

1. [F-strings](#)
2. The [str.format\(\)](#) method
3. The [modulo operator \(%\)](#)

The first two tools support the string formatting mini-language, a feature that allows you to fine-tune your strings. The third tool is a bit old and has fewer formatting options. However, you can use it to do some minimal formatting.

Note: The built-in [format\(\)](#) function is yet another tool that supports the format specification mini-language. This function is typically used for date and number formatting, but you won't cover it in this tutorial.

In the following sections, you'll start by learning a bit about the string formatting mini-language. Then, you'll dive into using this language, f-strings, and the `.format()` method to format your strings. Finally, you'll learn about the formatting capabilities of the modulo operator.

Using F-Strings to Format Strings

Python [3.6](#) added a string interpolation and formatting tool called **formatted string literals**, or [f-strings](#) for short. As you've already learned, f-strings let you embed Python objects and expressions inside your strings. To create an f-string, you must prefix the string with an `f` or `F` and insert replacement fields in the string literal. Each replacement field must contain a variable, object, or expression:

Python

```
>>> f"The number is {42}"
'The number is 42'

>>> a = 5
>>> b = 10
>>> f"{a} plus {b} is {a + b}"
'5 plus 10 is 15'
```

In the first example, you define an f-string that embeds the number 42 directly into the resulting string. In the second example, you insert two variables and an expression into the string.

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

Using the Formatting Mini-Language With F-Strings

When you use f-strings to create strings through interpolation, you need to use **replacement fields**. In f-strings, you can define a replacement field using curly brackets (`{}`) as in the examples below:

Python

```
>>> debit = 300.00
>>> credit = 450.00

>>> f"Debit: ${debit}, Credit: ${credit}, Balance: ${credit - debit}"
'Debit: $300, Credit: $450.0, Balance: $150.0'
```

Inside the brackets, you can insert Python objects and [expressions](#). In this example, you'd like the resulting string to display the currency values using a proper format. However, you get a string that shows the currency values with at most one digit on its decimal part.

To format the values and always display two digits on its decimal part, you can use a format specifier:

Python

```
>>> f"Debit: ${debit:.2f}, Credit: ${credit:.2f}, Balance: ${credit - debit:.2f}"
'Debit: $300.00, Credit: $450.00, Balance: $150.00'
```

In this example, note that each replacement field contains a string that starts with a colon. That's a format specifier. The `.2f` part tells Python that you want to format the value as a floating-point number (`f`) with two decimal places (`.2`).

Note: To learn more about the formatting mini-language and its syntax, check out the [Python's Format Mini-Language for Tidy Strings](#) tutorial. If you're interested in formatting floating-point numbers specifically, then you can also read [How to Format Floats Within F-Strings in Python](#).

The string formatting mini-language is a powerful tool with several cool features, including the following:

- Strings alignment
- Conversion between input objects' types

- Numeric values formatting
- Dynamic formatting

In the following sections, you'll explore how to use f-strings, the `.format()` method, and the string formatting mini-language to format the values that you insert into your strings through interpolation.

Formatting Strings With F-Strings: A Practical Example

You can use f-strings and the string formatting mini-language to format the values that you interpolate into your strings in multiple ways. To illustrate, say that you want to write a Python function to build a grades report for your students. The student data is in a dictionary that looks like the following:

Python



```
>>> student = {
...     "name": "John Doe",
...     "subjects": [
...         {
...             "name": "Mathematics",
...             "grade": 88,
...             "comment": "Excellent improvement.",
...         },
...         {
...             "name": "Science",
...             "grade": 92,
...             "comment": "Outstanding performance.",
...         },
...         {
...             "name": "History",
...             "grade": 78,
...             "comment": "Needs to participate more.",
...         },
...         {
...             "name": "Art",
...             "grade": 85,
...             "comment": "Very creative."
...         },
...     ],
... }
```

The dictionary has the `name` key to store the student's name. Then, it has the `subjects` key to hold a list of dictionaries containing the student's performance in each subject.

Your function may look like the following:

Python

grades_report_v1.py

```
1 def build_student_report(student):
2     report_header = f"Progress Report. Student: {student['name']}"
3
4     total = sum(subject["grade"] for subject in student["subjects"])
5     average = total / len(student["subjects"])
6     average_report = f"Average: {average:.2f} / 100\n"
7
8     subject_report = "Course Details:\n"
9     for subject in student["subjects"]:
10         subject_report += (
11             f"{subject['name']:<15} "
12             f"Grade: {subject['grade']:3d} "
13             f"Comment: {subject['comment']}\n"
14         )
15
16     return f"""
17 {report_header}
18 {average_report}
19 {subject_report}
20 Thank you for reviewing the progress report.
21 """
```

In this function, you’re doing several things. Here’s a line-by-line breakdown:

- **Line 2** defines a heading for the report.
- **Line 4** [sums](#) all the grades.
- **Line 5** computes the average of the grades.
- **Line 6** defines the average grade subreport. In this part of the report, you use the `.2f` format specifier to express the average grade as a floating-point number with two decimal places.
- **Line 8** defines the first line of the subject subreport.
- **Line 9** starts a loop over the subjects.
- **Line 10** adds more information to the subject subreport.
- **Line 11** formats the subject’s name. To do this, you use the `<15` format specifier, which tells Python to align the name to the left within 15 characters.
- **Line 12** formats the grade. In this case, the `3d` format specifier tells Python to display the grade using up to three characters. A leading space is used if the grade doesn’t have three digits.
- **Line 13** adds the comment part.
- **Lines 16 to 21** build the final report and return it to the caller.

Now that you’ve gone through the complete code, it’s time to try out your function. Go ahead and run the following code:

Python

```
>>> from grades_report_v1 import build_student_report

>>> print(build_student_report(student))

Progress Report. Student: John Doe
Average: 85.75 / 100

Course Details:
Mathematics      Grade:  88 Comment: Excellent improvement.
Science          Grade:  92 Comment: Outstanding performance.
History          Grade:  78 Comment: Needs to participate more.
Art              Grade:  85 Comment: Very creative.

Thank you for reviewing the progress report.
```

Your report looks pretty nice! It provides general information about the student at the top and grade details for every subject.

Using `str.format()` to Format Strings

You can also use the [.format\(\)](#) method to format values during string interpolation. In most cases, you’d use the `.format()` method for lazy interpolation. In this type of interpolation, you define a template string in some part of your code and then interpolate values in another part:

Python

```
>>> number_template = "The number is {}"
>>> sum_template = "{0} plus {1} is {2}"

>>> number_template.format(42)
'The number is 42'

>>> a = 5
>>> b = 10
>>> sum_template.format(a, b, a + b)
'5 plus 10 is 15'
```

In both examples, you create a string template and then use the `.format()` method to interpolate the required values.

Using the Formatting Mini-Language With `.format()`

The `str.format()` method also supports the string formatting mini-language. This feature allows you to nicely format the interpolated values when you create strings through interpolation. To illustrate, consider the example you wrote in the section about using the [formatting mini-language with f-strings](#).

Here's how to do it with the `.format()` method:

Python



```
>>> debit = 300.00
>>> credit = 450.00

>>> template = "Debit: ${0:.2f}, Credit: ${1:.2f}, Balance: ${2:.2f}"
>>> template.format(debit, credit, credit - debit)
'Debit: $300.00, Credit: $450.00, Balance: $150.00'
```

Again, the resulting string displays the currency values using a proper format that shows two decimal places.

Formatting Strings With `.format()`: Practical Examples

Now it's time for a couple of practical examples of using the `.format()` method and the string formatting mini-language to format your strings.

For the first example, say that you need to create a sales report for your company. You'd like to create a report template and fill it with the appropriate data when someone requires it. In this situation, you can create the following report template:

Python

sales_report.py

```
REPORT_TEMPLATE = """
Monthly Sales Report
-----
Report Date Range: {start_date} to {end_date}

Number of Transactions: {sep:.>20} {transactions:,}
Average Transaction Value: {sep:.>11} ${avg_transaction:,.2f}

Total Sales: {sep:.>23} ${total_sales:,.2f}
"""
```

In this report template, you have a few format specifiers. Here's a summary of their specific meanings:

- `.>20` displays the interpolated value aligned to the right within a space of 20 characters. The dot after the colon works as the fill character. The other format specifiers, `.>11` and `.>23`, have a similar effect. Note that the field widths were chosen by trial and error to make the report line up nicely.
- `,` displays the preceding number using a comma as the thousands separator.
- `.2f` shows a value as a floating-point number using two decimal places and a comma as the thousands separator.

Now, you can code a function to generate the actual report:

Python

sales_report.py


```
# ...

def build_sales_report(sales_data, report_template=REPORT_TEMPLATE):
    total_sales = sum(sale["amount"] for sale in sales_data)
    transactions = len(sales_data)
    avg_transaction = total_sales / transactions

    return report_template.format(
        sep=".",
        start_date=sales_data[0]["date"],
        end_date=sales_data[-1]["date"],
        total_sales=total_sales,
        transactions=transactions,
        avg_transaction=avg_transaction,
    )
```

This function takes the sales data and the report template as arguments. Then, it computes the required values and calls `.format()` on the template. Go ahead and give this function a try by running the following code:

Python



```
>>> from sales_report import build_sales_report

>>> sales_data = [
...     {"date": "2024-04-01", "amount": 100},
...     {"date": "2024-04-02", "amount": 200},
...     {"date": "2024-04-03", "amount": 300},
...     {"date": "2024-04-04", "amount": 400},
...     {"date": "2024-04-05", "amount": 500},
... ]

>>> print(build_sales_report(sales_data))
```

```
Monthly Sales Report
-----
Report Date Range: 2024-04-01 to 2024-04-05

Number of Transactions: ..... 5
Average Transaction Value: ..... $300.00

Total Sales: ..... $1,500.00
```

Cool! You have a nicely formatted sales report. You can experiment on your own and tweak the format specifiers further as an exercise. For example, you can improve the date format. However, note that dates have their own formatting, which you can learn about in the [Formatting Dates](#) section of the tutorial about Python's string formatting mini-language.

You can also take advantage of the `.format()` method when your data is stored in dictionaries. For example, here's how you can update the `build_student_report()` function using the `.format()` method:

Python

grades_report_v2.py

```
SUBJECT_TEMPLATE = "{name:<15} Grade: {grade:3} Comment: {comment}"
REPORT_TEMPLATE = """
Progress Report. Student: {name}
Average: {average:.2f} / 100

Course Details:
{subjects_report}

Thank you for reviewing the progress report.
"""

def build_student_report(student):
    data = {
        "name": student["name"],
        "average": sum(subject["grade"] for subject in student["subjects"])
        / len(student["subjects"]),
        "subjects_report": "\n".join(
            SUBJECT_TEMPLATE.format(**subject)
            for subject in student["subjects"]
        ),
    }


    return REPORT_TEMPLATE.format(**data)
```

In the first lines of code, you create string templates to display the information about each subject and also the final report. In `build_student_report()`, you create a dictionary with the required data to build the student report. Next, you interpolate the data into the report template using `.format()` with the dictionary as an argument.

Note that to fill the string templates, you use the `**` operator to unpack the data from the input dictionary.

Formatting Strings With the Modulo Operator (%)

Strings in Python have a built-in operation that you can access with the modulo operator (%). This operator lets you do positional and named string interpolation. If you’ve ever worked with the `printf()` function in [C](#), then the syntax will be familiar. Here’s a toy example:

```
Python 

>>> name = "Bob"

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

The `%s` substring is a **conversion specifier** that works as a replacement field. It tells Python where to substitute the value of `name`, represented as a string.

Using Conversion Specifiers

To build a conversion specifier, you need two or more characters. Here’s a quick summary of the accepted characters and their corresponding order in the specifier:

1. **The % character** marks the start of the specifier.
2. **An optional mapping key in parentheses** allows you to use named replacement fields like `(name)`.
3. **An optional conversion flag** affects how some conversion types display.
4. **An optional minimum field width** allows you to define the number of characters to display.
5. **An optional precision** consists of a dot character (.) followed by the desired precision.
6. **An optional length modifier** is an `l` or `h` for long and short integers.
7. **A conversion type** specifies how the output string will be formatted, mimicking different data types.

Several conversion types are available for the modulo operator in Python. They allow you to control the output’s format in some way. For example, you can convert numbers to hexadecimal notation or add whitespace padding to generate nicely formatted tables and reports.

Here’s a summary of the conversion types currently available in Python:

Conversion Type	Description
d	Signed integer decimal
i	Signed integer decimal
o	Signed octal value
x	Signed hexadecimal with lowercase prefix
X	Signed hexadecimal with uppercase prefix
e	Floating-point exponential format with lowercase e
E	Floating-point exponential format with uppercase E
f	Floating-point decimal format
F	Floating-point decimal format
g	Floating-point format
G	Floating-point format
c	Single character (accepts integer or single character string)
r	String as per calling repr() .
s	String as per calling str() .
a	String as per calling ascii() .
%	A percentage character (%) in the result if no argument is converted

With all these conversion types, you can process your interpolated values to display them using different representation types.

Note: It’s important to note that f-strings and the `.format()` method also support the conversion types listed above. For details on this topic, check out the [Converting Between Type Representations](#) section in the *Python’s Format Mini-Language for Tidy Strings* tutorial.

Here are a few examples of how to use some of the above specifiers in your strings:

Python

```
>>> "%d" % 123.123 # As an integer
'123'

>>> "%o" % 42 # As an octal
'52'

>>> "%x" % 42 # As a hex
'2a'

>>> "%e" % 1234567890 # In scientific notation
'1.234568e+09'

>>> "%f" % 42 # As a floating-point number
'42.000000'
```

In these examples, you’ve used different conversion types to display values using different type representations. Now, check out the examples below to see other formatting options in action:

Python

```
>>> # Named replacement fields
>>> jane = {"first_name": "Jane", "last_name": "Doe"}
>>> "Full name: %(first_name)s %(last_name)s" % jane
'Full name: Jane Doe'

>>> # Minimal width of 15 chars
>>> "%-15f" % 3.1416 # Aligned to the left
'3.141600      '
>>> "%15f" % 3.1416 # Aligned to the right
'          3.141600'

>>> # Dynamic width
>>> width = 15
>>> "%-*f" % (width, 3.1416)
'3.141600      '
>>> "%*f" % (width, 3.1416)
'          3.141600'

>>> # Precision
>>> "%.2f" % 3.141592653589793
'3.14'
>>> "%.4f" % 3.141592653589793
'3.1416'
>>> "%.8f" % 3.141592653589793
'3.14159265'
```

In the first example, you use named replacement fields in parentheses and a dictionary to provide the values you want to interpolate. In the second example, you provide a minimum width for your string in characters.

The third example is a dynamic variation of the second. Note how the `*` symbol allows you to insert the desired width dynamically.

Finally, you use the precision option to display a floating-point number using different precisions. You use two, four, and eight digits after the decimal separator, respectively.

Unfortunately, the modulo operator doesn’t support the string formatting mini-language, so if you use this tool to interpolate and format your strings, then your formatting options are more limited than when you use f-strings or `format()`.

Using Conversion Flags

The modulo operator also supports what’s known as **conversion flags**. Here’s a quick summary of the currently available flags:

Conversion Flag	Description
'#'	Displays numeric values using alternate forms
'0'	Adds zero padding for numeric values
'-'	Left-justifies the value (overrides the '0' conversion if both are given)
' ' (space)	Adds a space before a positive number
'+'	Adds a sign character ('+' or '-') before the value

These flags help you apply some additional formatting options to your strings. Consider the following quick examples:

Python

```
>>> "%o" % 10
'12'
>>> "%#o" % 10
'0o12'

>>> "%x" % 31
'1f'
>>> "%#x" % 31
'0x1f'
```

In these examples, you demonstrate the effect of the # flag, which prepends the appropriate prefix to the input number depending on the base you use. This flag is mostly used with integer values.

Here are some more examples of using different flags:

Python

```
>>> "%05d" % 42
'00042'

>>> "%10d" % 42
'          42'
>>> "%-10d" % 42
'42          '

>>> "% d" % 42
' 42'
>>> "% d" % -42
'-42'

>>> "%+d" % 42
'+42'
>>> "%+d" % -42
'-42'
```

In the first example, you add zero padding to the input value. Next, you have an example of how to use the minus sign to align a value to the left in a width of ten characters.

The space flag allows you to add a space before positive numbers. This space disappears when the value is negative. Finally, you use the plus sign so the string always displays whether the input value is positive or negative.

Deciding Which String Formatting Tool to Use

You’ve learned about three different tools for string formatting up to this point. Having several choices for one task can be confusing. In the end, what tool should you use?

If you want readable syntax, good performance, and you’re doing eager interpolation, then f-strings are for you. On the other hand, if you need a tool for doing lazy string interpolation, then the `.format()` method is the way to go.

In contrast, the modulo operator (%) is an old-fashioned tool not commonly used in modern Python. You could say that this tool is almost dead. However, you may find it in legacy Python code, so it’s good to know how it works.

The following table compares the three tools using several comparison criteria:

Feature	F-strings	.format()	%
Readability	High	Medium	Low
Supports lazy evaluation	⊖	✓	✓
Supports dictionary unpacking	⊖	✓	✓
Supports the format mini-language	✓	✓	⊖

Feature	F-strings	.format()	%
Supports conversion types	✓	✓	✓
Supports conversion flags	✓	✓	✓

F-strings are the clear winner in terms of readability. However, they don’t allow you to do lazy interpolation. There’s no way to use an f-string to create a reusable string template that you can interpolate later in your code. If you want a universal tool with all the features, then the `.format()` method is the way to go.

Conclusion

As you’ve learned in this tutorial, Python has several tools that allow you to format your strings during interpolation. In modern Python, you’ll see and use f-strings or the `.format()` method most of the time. In legacy code, you can see the modulo operator being used.

In this tutorial, you’ve learned how to:

- Format your strings using **f-strings** for eager interpolation
- Use the `.format()` method to format your strings lazily
- Work with the **modulo operator (%)** for string formatting
- Decide which interpolation and **formatting tool** to use

Additionally, you’ve learned the basics of Python’s string formatting mini-language and how to use it, along with f-strings and the `.format()` method. Now, you have enough knowledge to start giving your strings a nice and professional format.

Frequently Asked Questions

Now that you have some experience with string formatting tools in Python, 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 string formatting in Python?

Show/Hide

How do you format a string in Python?

Show/Hide

What are the different types of string formatting in Python?

Show/Hide


Which string formatting method is best in Python?

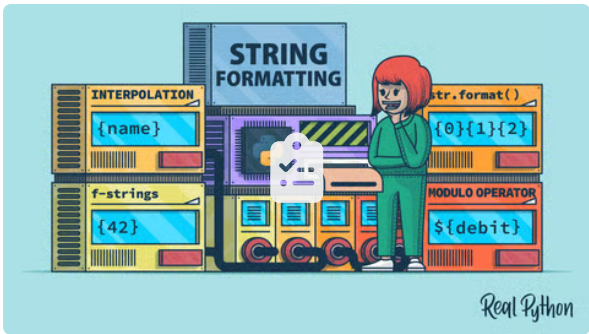
Show/Hide

What features does Python's string formatting mini-language offer?

Show/Hide

Get Your Code: [Click here to download the free sample code](#) you’ll use to learn about Python’s string formatting tools.

 **Take the Quiz:** Test your knowledge with our interactive “Python String Formatting: Available Tools and Their Features” quiz. You’ll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Python String Formatting: Available Tools and Their Features

You can take this quiz to test your understanding of the available tools for string formatting in Python, as well as their strengths and weaknesses. These tools include f-strings, the .format() method, and the modulo operator.

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 String Formatting Tips & Best Practices](#)

About Leodanis Pozo Ramos



Leodanis is a self-taught Python developer, educator, and technical writer with over 10 years of experience.

[» More about Leodanis](#)

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](#)



[Brenda](#)



[Bartosz](#)



[Dan](#)



[Geir Arne](#)



[Joanna](#)

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](#) [best-practices](#) [python](#)

Recommended Video Course: [Python String Formatting Tips & Best Practices](#)

Related Tutorials:

- [Python's F-String for String Interpolation and Formatting](#)
- [Python's Format Mini-Language for Tidy Strings](#)
- [A Guide to Modern Python String Formatting Tools](#)
- [Python Exceptions: An Introduction](#)
- [How to Split a String 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.

