

How to Split a String in Python

by [Martin Breuss](#) Feb 05, 2025 26m 4 Comments basics python

Mark as Completed



Share

Table of Contents

- [How to Split a String in Python Using .split\(\)](#)
- [Split With Different Delimiters Using sep](#)
- [Limit the Amount of Splits With maxsplit](#)
- [Go Backwards Through Your String Using .rsplit\(\)](#)
- [Split Strings by Lines With .splitlines\(\)](#)
- [Use re.split\(\) for Advanced String Splitting](#)
- [Conclusion](#)
- [Frequently Asked Questions](#)

Python's `.split()` method lets you divide a string into a list of substrings based on a specified delimiter. By default, `.split()` separates at whitespace, including spaces, tabs, and newlines. You can customize `.split()` to work with specific delimiters using the `sep` parameter, and control the amount of splits with `maxsplit`.


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

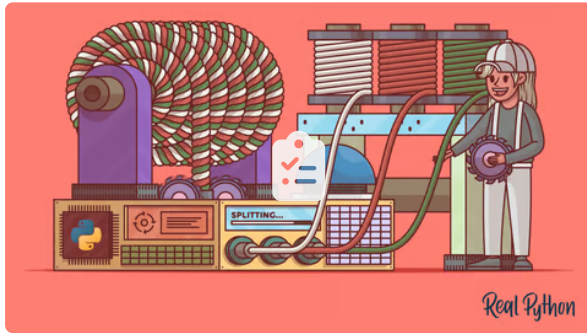
- You **split a string by spaces** in Python using `.split()` **without arguments**.
- Python's `.split()` method can split on **custom delimiters** when you pass a character or string as an argument.
- You **limit splits using** `maxsplit` to control the number of substrings Python extracts.
- `.splitlines()` **splits multiline strings** into individual lines, excluding or including line breaks with the `keepends` parameter.
- `re.split()` uses **regular expressions** for splitting strings based on complex patterns.

Exploring these methods will level up your text-processing capabilities and enable you to confidently tackle real-world data parsing challenges.

Help

Get Your Code: [Click here to download the free sample code](#) that shows you how to split strings in Python.

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



Interactive Quiz

[How to Split a String in Python](#)

In this quiz, you'll test your understanding of Python's `.split()` method. This method is useful for text-processing and data parsing tasks, allowing you to divide a string into a list of substrings based on a specified delimiter.

How to Split a String in Python Using `.split()`

The `.split()` method in Python is a versatile tool that allows you to divide a string into a list of substrings based on a specified delimiter. By default, `.split()` separates a string at each occurrence of whitespace, which includes spaces, tabs, and newlines. This makes it particularly handy when dealing with plain text where words are separated by spaces:

Python



```
>>> sentence = "Python is a versatile language."
>>> sentence.split()
['Python', 'is', 'a', 'versatile', 'language.']
```

As you can see, `.split()` divides the sentence into individual words, removing the spaces in the process. It’s important to note that `.split()` also strips newlines by default, which might not be immediately obvious:

Python



```
>>> text = """Line one
... Line two
... Line three"""

>>> text.split()
['Line', 'one', 'Line', 'two', 'Line', 'three']
```

The `.split()` method treats the newline characters as whitespace, and splits the text into individual words rather than lines. This is a crucial point to remember when [working with multiline strings](#).

The simplicity of `.split()` makes it a powerful tool for string manipulation. Whether you’re processing text files or parsing user input, using `.split()` can streamline your workflow.

When you’re faced with the task of cleaning up messy text, you may want to pair `.split()` with [.strip\(\)](#). You can learn more about using `str.strip().split()` by expanding the collapsible section below:

Using `str.strip().split()`

Show/Hide

When you use `.split()`, it returns a list of substrings. This means that you can iterate over the result, access individual elements using indexing, or [unpack the iterable](#) into separate variables.

For example, when you’re dealing with user input, you might want to extract specific pieces of information:

Python



```
>>> user_input = "Deborah Lopez 30"
>>> user_data = user_input.split()
>>> user_data
['Deborah', 'Lopez', '30']

>>> name, surname, age = user_data
>>> print(f"Welcome {name}! You're {age} years old.")
Welcome Deborah! You're 30 years old.
```

In this code snippet, you take a string containing a name, surname, and age, and split it into a list of three separate strings. Then, you unpack the list into three descriptive variables. Finally, you use an [f-string](#) to format the output.

Note: Extracting data pieces like this is useful when you’re dealing with structured data where you know the position of each element beforehand, and you can rely on consistent use of whitespace.

In this section, you’ve learned how to use Python’s `.split()` method to divide strings into smaller parts based on whitespace. This method is invaluable when working with plain text data, allowing you to extract and manipulate information. By understanding the default behavior of `.split()`, including its treatment of newlines, you’ll be well-equipped to handle a variety of string manipulation tasks in your Python projects.

In the next section, you’ll explore how to customize the behavior of `.split()` by specifying different delimiters, enabling you to tackle more complex string splitting scenarios.

Split With Different Delimiters Using `sep`

By default, `.split()` uses any whitespace as a delimiter to separate a string into a list of substrings. However, many real-world scenarios require splitting strings using other delimiters. This is where the `sep` parameter comes into play.

Imagine that you’re parsing data from a CSV (comma-separated values) file. CSV files are a popular format for storing tabular data, where each line represents a row and each value within a line is separated by a comma.

Note: In a real-world scenario, it’s best to use [Python’s `csv` library](#) when you’re [working with CSV files](#).

Trying to parse CSV files manually can be surprisingly tricky, as you might have to handle various edge cases, cross-platform newlines, and character encoding schemes.

To parse such data, you can specify a comma as the first argument to `.split()`. This will successfully split a string of comma-separated names:

Python

```
>>> csv_line = "Ankita,Vishal,Jane,Naga,Emily,Maria"
>>> csv_line.split(sep=",")
['Ankita', 'Vishal', 'Jane', 'Naga', 'Emily', 'Maria']
```

Here, the string `csv_line` contains several names separated by commas. By passing a comma (",") as the argument to `sep`, you instruct Python to use commas as the delimiter for splitting. As a result, `.split()` separates the string into a list of individual names. Note that `.split()` doesn’t include the delimiter in the output.

To consider another example, imagine that you have a line from a CSV file that contains product information:

Python

```
>>> csv_product_info = "Sweater,Blue,34.99,In Stock"
>>> fields = csv_product_info.split(",")
>>> print(fields)
['Sweater', 'Blue', '34.99', 'In Stock']
```

In this example, `csv_product_info` contains information about a product, such as its name, color, price, and availability. You split the string again on commas.

Note: In this second example, you pass ", " as a [positional argument](#) instead of as a [keyword argument](#) through `sep`.

In practice, you’ll see the separator string more often passed as a positional argument, but both ways lead to the same result.

By splitting the string on commas, you extract each piece of information into a list, which you can then access individually:

Python

```
>>> color = fields[1]
>>> print(color)
Blue
```

The `sep` parameter is not limited to commas. You can use any character—or even a sequence of characters—as a delimiter. Consider a scenario where your kids assembled a shopping list for you and used a semicolon followed by a closing parenthesis as item delimiters:

Python

```
>>> fruits = "Apple;)Orange;)Lemon;)Date"
>>> fruits.split(";)")
['Apple', 'Orange', 'Lemon', 'Date']
```

In this case, each fruit name is separated by an emoticon of a winking face. Looks like someone had a fun time doing data entry for this shopping list!

By specifying `";)"` as the `sep` argument, the `.split()` method will break the string into individual fruit names.

Passing a specific argument to `sep` can also help if you need to split a string only on a *specific* whitespace character. Remember that by default, `.split()` splits on any whitespace character.

Say that in your day job, you're working with TSV (tab-separated values) files that contain product information from a store's database, such as a product ID, an item description, and a price:

Python

```
>>> product_1 = "1\tapple\t0.99"
>>> product_1.split()
['1', 'apple', '0.99']
```

You want to split the data into separate pieces of information. For `product_1`, this works well by just relying on the default behavior of `.split()`. However, if the product name includes a whitespace character, then you'll run into trouble:

Python

```
>>> product_2 = "2\tapple juice\t2.00"
>>> product_2.split()
['2', 'apple', 'juice', '2.00']
```

Because the default behavior of `.split()` is to separate your string at *any* whitespace character, it also breaks `"apple juice"` into two separate items. This could mess up your whole pipeline, because different lines may produce a different amount of data and you can't rely on list indices anymore.

To avoid this issue, you can explicitly pass the whitespace character that you want `.split()` to operate on—in this case, the tab character (`\t`):

Python

```
>>> product_2 = "2\tapple juice\t2.00"
>>> product_2.split("\t")
['2', 'apple juice', '2.00']
```

By specifying the type of whitespace Python should split on, you can mitigate this issue and make sure that `.split()` only separates the TSV string when it encounters a tab character.

Note: Remember that if you're working with a TSV or CSV file in a real-world scenario, it's best to use [Python's csv library](#) instead of fiddling with `.split()`.

Sometimes, you might encounter data that's separated by a variety of inconsistently used delimiters. While `.split()` doesn't support this directly, you can tackle this task with the `re.split()` method from [Python's re module](#) that you'll learn about in the section on [advanced splitting](#).

Using the `sep` parameter allows you to handle a variety of string splitting scenarios with the `.split()` method. It's particularly useful when you're working with data separated by unusual but consistent delimiters, as it provides the flexibility needed to extract meaningful information.

Limit the Amount of Splits With `maxsplit`

Sometimes, you may need to limit the number of splits when working with strings in Python. This is especially useful when you need to extract a specific number of elements while preserving the remainder of the string as a single unit.

Python's `.split()` method provides an optional `maxsplit` parameter that allows you to specify the maximum number of splits to perform. Once Python reaches the specified number of splits, it returns the remaining part of the string as the final element in the list.

Consider a scenario where you're parsing a [log file](#). Each log entry contains a timestamp, log level, and a message. You want to extract the date, time, and log level, while keeping the entire rest of the message as a single string. You can achieve this using `maxsplit`:

```
Python                                                                    extract_log_info.py

log_line = "2025-01-15 08:45:23 INFO User logged in from IP 10.0.1.1"

date, time, log_level, message = log_line.split(maxsplit=3)

print(f>Date: {date}")
print(f>Time: {time}")
print(f>Log Level: {log_level}")
print(f>Message: {message}")
```

You pass the value `3` as an argument to the `maxsplit` parameter, which means that the method performs three splits on whitespace characters. This results in four elements: the date, time, log level, and the remainder of the message. This approach allows you to neatly separate the structured components from the unstructured message.

Note: Like in earlier examples, it's important to consider the consistency of your data. Using `maxsplit` like shown above requires that the delimiters and the number of expected splits are consistent across your dataset.

If the structure of your data varies, then you might need to handle exceptions or preprocess the data to ensure compatibility with your splitting logic.

If you want to pass an argument to `maxsplit` without specifying a separator, then you need to use `maxsplit` as a keyword argument. Otherwise, Python will throw a `TypeError`:

```
Python                                                                    [icon]

>>> "A B C".split(1)
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    "A B C".split(1)
    ~~~~~^
TypeError: must be str or None, not int
```

If you pass arguments positionally, then Python assigns the first value to `sep`, and that value needs to be a string for the method to work.

You *can* also pass an argument to `maxsplit` positionally, if you specify both possible arguments:

```
Python                                                                    [icon]

>>> "A B C".split(" ", 1)
['A', 'B C']
```


However, even when specifying a separator, it's more common to pass `maxsplit` as a keyword argument. This is because `maxsplit` is used less frequently, and adding the parameter's name into the method call improves readability:

Python



```
>>> "A B C".split(" ", maxsplit=1)
['A', 'B C']
```

Passing the value for `maxsplit` as a keyword argument helps to make your code more straightforward to read and understand, because it adds self-documenting information to the `.split()` method call.

Once you're familiar with how `maxsplit` works, you can also use it in combination with a related string method, `.rsplit()`.

Go Backwards Through Your String Using `.rsplit()`

Python's string method `.rsplit()` allows you to split a string like `.split()` does, but instead of starting from the left, it starts splitting from the right. Without specifying `maxsplit`, `.split()` and `.rsplit()` produce identical results:

Python



```
>>> countdown = "3-2-1"
>>> countdown.split("-")
['3', '2', '1']
>>> countdown.rsplit("-")
['3', '2', '1']
```

However, in combination with `maxsplit`, this method comes in handy for string splitting tasks. For example, if you want to extract a filename from the end of a file path:

Python



```
>>> path = "/home/user/documents/tax.txt"
>>> directory, filename = path.rsplit("/", maxsplit=1)
>>> directory
'/home/user/documents'
>>> filename
'tax.txt'
```

Here, you use `.rsplit()` instead of `.split()` and pass 1 as the argument to `maxsplit`. This allows you to extract the filename while keeping the rest of the string as one element, which you assign to `directory`.

What happens if you use `.split()` instead of `.rsplit()` for this example, and why is it less robust? Think about what you expect the output to be, then click the *Show/Hide* toggle below to reveal the answer:

Splitting From Different Ends

Show/Hide

After slicing your file path with your precise split using `.rsplit()`, you now know *exactly* where you stored this year's tax documents. Just in time to submit your taxes and make your contribution to keep everything rolling smoothly.

Note: If you're working with file paths in a real-world scenario, then you should use [Python's `pathlib` module](#) instead of fiddling with `.split()` or `.rsplit()`.

The `pathlib` module handles paths in an [object-oriented](#) and cross-platform manner. This spares you from worrying about which directory separators to split on or how to reconstruct paths. Overall, it'll keep your code cleaner and more robust by offering convenient attributes and methods for working with file systems in a portable, reliable way.

The `maxsplit` parameter in Python's `.split()` and `.rsplit()` methods helps you to control string splitting operations. It provides a flexible way to manage the number of splits while preserving the integrity of the remaining string.

When you use `maxsplit` in `.rsplit()`, you can more quickly tackle string-splitting tasks when you're primarily interested in the final parts of a string.

Split Strings by Lines With `.splitlines()`

When processing text, you might often need to handle multiline text and split it into individual lines—for example, when you’re reading data from a file, processing user input, or dealing with text generated by an application. It’s *possible* to split text into multiple lines using `.split()` and specifying the line break character:

Python



```
>>> text = """Hello, World!
... How are you doing?
... """

>>> text.split("\n")
['Hello, World!', 'How are you doing?', '']
```

In this example, you use the newline character (`\n`) as a custom delimiter so that `.split()` only operates on line breaks, not on other whitespace characters.

While it works, you may have noticed that `.split()` adds an empty string when the text ends with a final newline. This may not always be what you want.

Additionally, splitting text into lines is a very common text processing task. Therefore, Python provides a dedicated string method called `.splitlines()` for it, which also avoids the awkward empty string in the final index.

The `.splitlines()` method splits a string at line boundaries, such as the newline characters (`\n`), carriage returns (`\r`), and some combinations like `\r\n`. It returns a list of lines that you can iterate over or manipulate further:

Python



```
>>> text = """Hello, world!
... How are you doing?
... """

>>> text.splitlines()
['Hello, world!', 'How are you doing?']
```

In this example, the string `text` contains again two lines with text, and a final newline. When you call `.splitlines()`, it returns a list with each line as a separate element. The final line break doesn’t result in an extra empty string element.

By default, `.splitlines()` doesn’t include line end characters in the resulting list elements. However, you can change this behavior by using the `keepends` parameter. Setting `keepends` to `True` will retain the line end characters in the resulting lines. This may be useful if you need to preserve the exact formatting of the original text:

Python



```
>>> text.splitlines(keepends=True)
['Hello, world!\n', 'How are you doing?\n']
```

With `keepends=True`, the output will include any existing newline characters at the end of each line.

Splitting strings by lines has numerous practical applications. Here are a few scenarios where `.splitlines()` can be particularly handy:

- **Reading small files:** When you read a file’s content into a string, you often want to process it line by line. Using `.splitlines()` allows you to quickly convert the entire content into a list of lines, helping you to [iterate](#) and analyze the data.
- **Processing logs:** Logs typically contain multiple lines of text, each representing a separate event or message. By splitting the log data into individual lines, you can efficiently parse and filter the information that you need.
- **Handling multiline user input:** If your application accepts multiline input from users, `.splitlines()` can help you break down and process each line separately, enabling more granular input validation and processing.

To examine a practical situation, you’ll take a closer look at using `.splitlines()` to parse a multiline log file:

```
log_data = """2025-01-15 08:45:23 INFO User logged in
2025-01-15 09:15:42 ERROR Failed to connect to server
2025-01-15 10:01:05 WARNING Disk space running low"""

log_lines = log_data.splitlines()

for line in log_lines:
    if "ERROR" in line:
        print(line)
```

In this example, `log_data` contains several log entries, each on a new line. By splitting the string into lines, you can iterate over each entry. You then use [Python's membership operator](#) to search for a specific keyword, which allows you to filter messages based on severity and display only error messages.

The `.splitlines()` method is a convenient tool for working with multiline strings in Python. Whether you need to handle text files, logs, or user input, it provides a straightforward way to split strings by line boundaries and manipulate the resulting data. By leveraging the `keepends` parameter, you can further control the output to suit your specific needs.

Note: If you need to iterate over lines in a file, it's best to use `.splitlines()` only for small files. The method reads the entire file into memory, which will impact performance when you're working with larger files. In such cases, it's best to directly [iterate over the file object](#) to access each line. Python [evaluates this operation lazily](#) because the file object itself is a lazy iterator that [yields](#) the data on demand.

As you continue to work with text data in Python, keep `.splitlines()` in your toolkit for situations where you need to split text into separate lines.

Use `re.split()` for Advanced String Splitting

When you need to divide strings based on more complex splitting criteria, you'll need a more powerful tool. This is where the `re.split()` function from [Python's re module](#) shines. It allows you to use regular expressions for splitting strings, enabling you to handle patterns that `.split()` can't easily address.

The `re.split()` function takes a regular expression pattern as its first argument and the target string as its second argument. You can use this function to split strings based on complex criteria, such as multiple, inconsistently used delimiters:



```
>>> import re

>>> shopping_list = "Apple:Orange|Lemon-Date"
>>> re.split(r"[:|-]", shopping_list)
['Apple', 'Orange', 'Lemon', 'Date']
```

In this example, the regular expression `[:|-]` specifies that Python should split the string at any occurrence of a colon, vertical bar, or minus sign. As you can see, the `re.split()` function provides a concise way to handle cases that involve multiple delimiters.

But that's not all! With `re.split()` you have access to the full power of regular expressions. So, consider a more complex scenario where your shopping list user input got completely out of hand, but you still need to extract the relevant information from it.

You're now dealing with a shopping list that uses a varying number of delimiters, along with the possibility of different types of delimiters. There may even be whitespace around the delimiters—but maybe not! Finally, the list even contains one delimiter made up of several different characters (AND). You're starting to wonder whether you should have introduced more [input validation](#) when building your shopping list app!

To add some more data for you to split on, this messy shopping list also hides information about *how many* of the items you need. Amazingly, you can handle all of this with an elegant pattern that you pass to `re.split()`:




```
>>> import re

>>> shopping_list = "Apple ::::3:Orange | 2|||Lemon --1 AND Date :: 10"
>>> pattern = r"\s*(?:[:|\-|+|AND)\s*"

>>> re.split(pattern, shopping_list)
['Apple', '3', 'Orange', '2', 'Lemon', '1', 'Date', '10']
```

The pattern that you pass to `re.split()` handles a complex string splitting scenario where you'd be hopelessly lost and in the weeds when using the string method `.split()`. Here's a list of the regex constructs that you used to make this split happen:

- `\s`: Matches any whitespace character.
- `*`: Matches zero or more occurrences of the preceding.
- `(?:)`: Creates an alternation group, for example `(?:abc|def)`, that matches any of the patterns `abc` or `def` in their entirety. In your specific example, this allows you to treat `AND` as a single delimiter to split on.
- `[]`: Creates a character set that matches any one of the characters inside the square brackets.
- `+`: Matches one or more occurrences of the preceding.

When you arrange these different regex constructs into the concise pattern shown above, you can split your messy shopping list into useful substrings. Thanks to `re.split()` you'll have your morning smoothie for the rest of the week!

The power of `re.split()` lies in its ability to utilize the full range of regular expression capabilities, such as character sets, groups, and metacharacters. You can define intricate patterns to split strings in ways that go beyond simple character delimiters. When you're dealing with complex string-splitting requirements, especially with multi-delimiter scenarios, using `re.split()` is the right tool to handle the job effectively.

If you frequently need to split strings following complex patterns, then you may want to continue learning about [regular expressions in Python](#) to understand their full capabilities. This will provide you with the knowledge to leverage regular expressions for a wide range of text processing tasks.

Conclusion

Splitting strings into smaller, more manageable parts is a fundamental skill for data processing and data analysis tasks. You can work with the string method `.split()` for basic scenarios or use powerful tools like the `split()` function from Python's `re` module for complex splitting patterns.

In this tutorial, you've learned how to:

- Use `.split()` to **break strings** down by **whitespace**
- Provide a **custom delimiter**, like commas, tabs, and semicolons, with the `sep` parameter
- Control splitting behavior with `maxsplit` to **limit the number of substrings** you extract
- **Split by lines** using `.splitlines()`, with or without line breaks
- **Take advantage of regular expressions** with `re.split()` for advanced or multi-delimiter scenarios

Keep practicing these techniques, and you'll be able to handle a wide range of real-world text-parsing challenges. You can also continue to explore Python's other powerful [string methods](#) as you continue to sharpen your programming and [text processing](#) skills. The more you practice, the better you'll become at writing clean, efficient, and powerful code for your string manipulation needs.

Get Your Code: [Click here to download the free sample code](#) that shows you how to split strings in Python.

Frequently Asked Questions

Now that you have some experience with splitting strings 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.

How do you split a string by spaces in Python?

Show/Hide

Can Python's .split() method use custom delimiters?

Show/Hide

Can .split() take two arguments?

Show/Hide

How do you limit splits using maxsplit in Python?

Show/Hide

What does .splitlines() do in Python?

Show/Hide

What does .split("t") do in Python?


Show/Hide

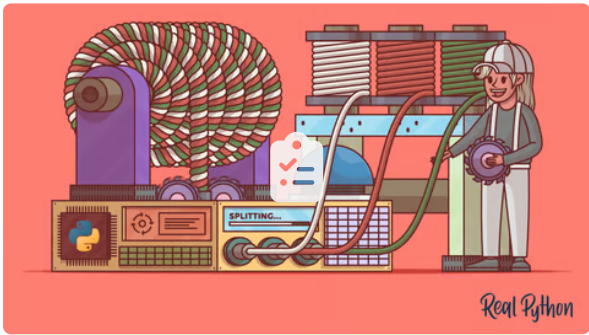
How do you split a string into words while keeping whitespace?

Show/Hide

What does .strip().split() do in Python?

Show/Hide

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




Interactive Quiz

[How to Split a String in Python](#)

In this quiz, you'll test your understanding of Python's .split() method. This method is useful for text-processing and data parsing tasks, allowing you to divide a string into a list of substrings based on a specified delimiter.

Mark as Completed



 Share

About **Martin Breuss**



Martin is Real Python's Head of Content Strategy. With a background in education, he's worked as a coding mentor, code reviewer, curriculum developer, bootcamp instructor, and instructional designer.

[» More about Martin](#)

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](#)



[Brad](#)



[Brenda](#)



[Bartosz](#)



[Dan](#)



[Geir Arne](#)



[Joanna](#)



[Kyle](#)

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](#)

Related Tutorials:

- [How to Join Strings in Python](#)
- [Python for Loops: The Pythonic Way](#)
- [Strings and Character Data in Python](#)
- [Nested Loops in Python](#)
- [Python's F-String for String Interpolation and Formatting](#)

- [Start Here](#)
[Learning Resources](#)
[Code Mentor](#)
[Python Reference](#)
[Support Center](#)
- [Learning Paths](#)
[Quizzes & Exercises](#)
[Browse Topics](#)
[Workshops](#)
[Books](#)
- [Podcast](#)
[Newsletter](#)
[Community Chat](#)
[Office Hours](#)
[Learner Stories](#)
- [Plans & Pricing](#)
[Team Plans](#)
[For Business](#)
[For Schools](#)
[Reviews](#)
- [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.

