

Python while Loops: Repeating Tasks Conditionally

by [Leodanis Pozo Ramos](#) 📅 Mar 03, 2025 🕒 24m 💬 19 Comments 🏷️ [basics](#) [python](#)

Mark as Completed



🔗 Share

Table of Contents

- [Getting Started With Python while Loops](#)
- [Using Advanced while Loop Syntax](#)
 - [The break Statement: Exiting a Loop Early](#)
 - [The continue Statement: Skipping Tasks in an Iteration](#)
 - [The else Clause: Running Tasks at Natural Loop Termination](#)
- [Writing Effective while Loops in Python](#)
 - [Running Tasks Based on a Condition With while Loops](#)
 - [Using while Loops for an Unknown Number of Iterations](#)
 - [Removing Items From an Iterable in a Loop](#)
 - [Getting User Input With a while Loop](#)
 - [Traversing Iterators With while Loops](#)
 - [Emulating Do-While Loops](#)
- [Using while Loops for Event Loops](#)
- [Exploring Infinite while Loops](#)
 - [Unintentional Infinite Loops](#)
 - [Intentional Infinite Loops](#)
- [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: [Mastering While Loops](#)



Help

Python’s `while` loop enables you to execute a block of code repeatedly as long as a given condition remains true. Unlike `for` loops, which iterate a known number of times, `while` loops are ideal for situations where the number of iterations isn’t known upfront.

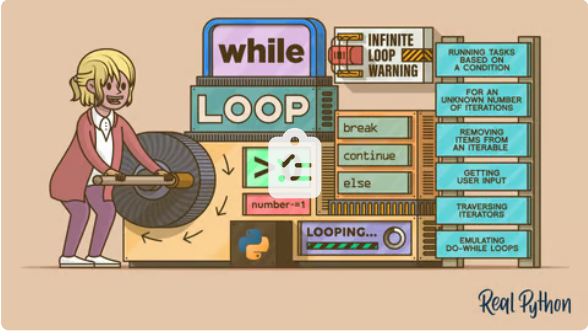
Loops are a pretty useful construct in Python, so learning how to write and use them is a great skill for you as a Python developer.

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

- **`while` is a Python keyword** used to initiate a loop that repeats a block of code as long as a condition is true.
- **A `while` loop works by evaluating a condition** at the start of each iteration. If the condition is true, then the loop executes. Otherwise, it terminates.
- **`while` loops are useful when the number of iterations is unknown**, such as waiting for a condition to change or continuously processing user input.
- **`while True` in Python creates an infinite loop** that continues until a `break` statement or external interruption occurs.
- **Python lacks a built-in `do-while` loop**, but you can emulate it using a `while True` loop with a `break` statement for conditional termination.

With this knowledge, you’re prepared to write effective `while` loops in your Python programs, handling a wide range of iteration needs.

Get Your Code: [Click here to download the free sample code](#) that shows you how to work with `while` loops in Python.



Take the Quiz: Test your knowledge with our interactive “Python while Loops: Repeating Tasks Conditionally” quiz. You’ll receive a score upon completion to help you track your learning progress:

Interactive Quiz
[Python while Loops: Repeating Tasks Conditionally](#)

In this quiz, you'll test your understanding of Python's `while` loop. This loop allows you to execute a block of code repeatedly as long as a given condition remains true. Understanding how to use `while` loops effectively is a crucial skill for any Python developer.

Getting Started With Python `while` Loops

In programming, loops are control flow statements that allow you to repeat a given set of operations a number of times. In practice, you’ll find two main types of loops:

1. [for loops](#) are mostly used to iterate a *known* number of times, which is common when you’re processing data collections with a specific number of data items.
2. [while loops](#) are commonly used to iterate an *unknown* number of times, which is useful when the number of iterations depends on a given condition.

Python has both of these loops and in this tutorial, you’ll learn about **`while` loops**. In Python, you’ll generally use `while` loops when you need to repeat a series of tasks an unknown number of times.

Python `while` loops are [compound statements](#) with a *header* and a *code block* that runs until a given condition becomes false. The basic syntax of a `while` loop is shown below:

Python Syntax

```
while condition:
    <body>
```

In this syntax, `condition` is an expression that the loop evaluates for its truth value. If the condition is true, then the loop body runs. Otherwise, the loop terminates. Note that the loop body can consist of one or more statements that must be indented properly.

Here's a more detailed breakdown of this syntax:

- **while** is the keyword that initiates the loop header.
- **condition** is an expression evaluated for truthiness that defines the exit condition.
- **<body>** consists of one or more statements to execute in each iteration.

Here's a quick example of how you can use a `while` loop to iterate over a decreasing sequence of numbers:

Python

```
>>> number = 5

>>> while number > 0:
...     print(number)
...     number -= 1
...
5
4
3
2
1
```

In this example, `number > 0` is the loop condition. If this condition returns a false value, the loop terminates. The body consists of a call to `print()` that displays the value on the screen. Next, you decrease the value of `number`. This change will produce a different result when the loop evaluates the condition in the next iteration.

The loop runs while the condition remains true. When the condition turns false, the loop terminates, and the program execution proceeds to the first statement after the loop body. In this example, the loop terminates when `number` reaches a value less than or equal to 0.

If the loop condition doesn't become false, then you have a potentially infinite loop. Consider the following loop, and keep your fingers near the `^ Ctrl` + `C` key combination to terminate its execution:

Python

```
>>> number = 5
>>> while number != 0:
...     print(number)
...     number -= 1
...
5
4
3
2
1

>>> number = 5
>>> while number != 0:
...     print(number)
...     number -= 2
...
5
3
1
-1
-3
-5
-7
-9
-11
Traceback (most recent call last):
...
KeyboardInterrupt
```

In this example, the loop condition is `number != 0`. This condition works when you decrease `number` by 1. However, if you decrease it by 2, the condition may never become false, resulting in a potentially infinite loop. In such cases, you can usually terminate the loop by pressing `^ Ctrl` + `C`, which raises a `KeyboardInterrupt` exception on most operating systems.

Note that the `while` loop checks its condition first before anything else happens. If it's false to start with, then the loop body will never run:

Python



```
>>> number = 0
>>> while number > 0:
...     print(number)
...     number -= 1
... 
```

In this example, when the loop finds that `number` isn't greater than 0, it immediately terminates the execution without entering the loop body. So, the body never executes.

Now that you know the basic syntax of `while` loops, it's time to dive into some practical examples in Python. In the next section, you'll see how these loops can be applied to real-world scenarios.

Using Advanced `while` Loop Syntax

The Python `while` loop has some advanced features that make it flexible and powerful. These features can be helpful when you need to fine-tune the loop to meet specific execution flows.

So far, you've seen examples where the entire loop body runs on each iteration. Python provides two [keywords](#) that let you modify that behavior:

- **`break`:** Immediately terminates a loop. The program execution then proceeds with the first statement following the loop body.
- **`continue`:** Ends only the current iteration. The execution jumps back to the loop header, and the loop condition is evaluated to determine whether the loop will execute again.

Python's `while` loops also have additional syntax. They have an `else` clause that runs when the loop terminates naturally because the condition becomes true. In the following sections, you'll learn more about how the `break` and `continue` statements work, as well as how to use the `else` clause effectively in `while` loops.

The `break` Statement: Exiting a Loop Early

With the `break` statement, you can terminate the execution of a `while` loop and make your program continue with the first statement immediately after the loop body.

Here's a short script that demonstrates how the `break` statement works:

Python

`break.py`

```
number = 6

while number > 0:
    number -= 1
    if number == 2:
        break
    print(number)

print("Loop ended")
```

When `number` becomes 2, the `break` statement is reached, and the loop terminates completely. The program execution jumps to the call to `print()` in the final line of the script.

Running `break.py` from the command line produces the following output:

Shell



```
$ python break.py
5
4
3
Loop ended
```

The loop prints values normally. When `number` reaches the value of 2, then `break` runs, terminating the loop and printing `Loop ended` to your screen. Note that 2 and 1 aren't printed at all.

It's important to note that it makes little sense to have `break` statements outside of [conditionals](#). Suppose you include a `break` statement directly in the loop body without wrapping it in a conditional. In that case, the loop will terminate in the first iteration, potentially without running the entire loop body.

The `continue` Statement: Skipping Tasks in an Iteration

Next, you have the `continue` statement. With this statement, you can skip some tasks in the current iteration when a given condition is met.

The next script is almost identical to the one in the previous section except for the `continue` statement in place of `break`:

```
Python                                                                    continue.py

number = 6

while number > 0:
    number -= 1
    if number == 2:
        continue
    print(number)

print("Loop ended")
```

The output of `continue.py` looks like this:

```
Shell                                                                    >
$ python continue.py
5
4
3
1
0
Loop ended
```

This time, when `number` is 2, the `continue` statement terminates that iteration. That's why 2 isn't printed. The control then returns to the loop header, where the condition is re-evaluated. The loop continues until `number` reaches 0, at which point it terminates as before.

The `else` Clause: Running Tasks at Natural Loop Termination

Python allows an optional `else` clause at the end of `while` loops. The syntax is shown below:

```
Python Syntax

while condition:
    <body>
else:
    <body>
```

The code under the `else` clause will run only if the `while` loop terminates naturally without encountering a `break` statement. In other words, it executes when the loop condition becomes false, and only then.

Note that it doesn't make much sense to have an `else` clause in a loop that doesn't have a `break` statement. In that case, placing the `else` block's content after the loop—without indentation—will work the same and be cleaner.

When might an `else` clause on a `while` loop be useful? One common use case for `else` is when you need to break out of the loop. Consider the following example:

Python

connection.py

```
import random
import time

MAX_RETRIES = 5
attempts = 0

while attempts < MAX_RETRIES:
    attempts += 1
    print(f"Attempt {attempts}: Connecting to the server...")
    # Simulating a connection scenario
    time.sleep(0.3)
    if random.choice([False, False, False, True]):
        print("Connection successful!")
        break
    print("Connection failed. Retrying...")
else:
    print("All attempts failed. Unable to connect.")
```

This script simulates the process of connecting to an external server. The loop lets you try to connect a number of times, defined by the `MAX_RETRIES` [constant](#). If the connection is successful, then the `break` statement terminates the loop.

Note: In the example above, you used the `random` module to simulate a successful or unsuccessful connection. You'll use this module in a few other examples throughout this tutorial. To learn more about random data, check out the [Generating Random Data in Python \(Guide\)](#) tutorial.

When all the connection attempts fail, the `else` clause executes, letting you know that the attempts were unsuccessful. Go ahead and run the script several times to check the results.

Writing Effective `while` Loops in Python

When writing `while` loops in Python, you should ensure that they're efficient and readable. You also need to make sure that they terminate correctly.

Normally, you choose to use a `while` loop when you need to repeat a series of actions until a given condition becomes false or while it remains true. This type of loop isn't the way to go when you need to process all the items in an iterable. In that case, you should use a [for](#) loop instead.

In the following sections, you'll learn how to use `while` loops effectively, avoid infinite loops, implement control statements like `break` and `continue`, and leverage the `else` clause for handling loop completion gracefully.

Running Tasks Based on a Condition With `while` Loops

A general use case for a `while` loop is waiting for a resource to become available before proceeding to use it. This is common in scenarios like the following:

- Waiting for a [file](#) to be created or populated
- Checking whether a [server is online](#)
- Polling a [database](#) until a record is ready
- Ensuring a [REST API](#) is responsive before making requests

Here's a loop that continually checks if a given file has been created:

Python

check_file.py

```
import time
from pathlib import Path

filename = Path("hello.txt")

print(f"Waiting for {filename.name} to be created...")

while not filename.exists():
    print("File not found. Retrying in 1 second...")
    time.sleep(1)

print(f"{filename} found! Proceeding with processing.")
with open(filename, mode="r") as file:
    print("File contents:")
    print(file.read())
```

The loop in this script uses the `.exists()` method on a `Path` object. This method returns `True` if the target file exists. The `not` operator negates the check result, returning `True` if the file doesn't exist. If that's the case, then the loop waits for one second to run another iteration and check for the file again.

If you run this script, then you'll get something like the following:

Shell



```
$ python check_file.py
Waiting for hello.txt to be created...
File not found. Retrying in 1 second...
File not found. Retrying in 1 second...
File not found. Retrying in 1 second...
...
```

In the meantime, you can open another terminal and create the `hello.txt` file. When the loop finds the newly created file, it'll terminate. Then, the code after the loop will run, printing the file content to your screen.

Using while Loops for an Unknown Number of Iterations

`while` loops are also great when you need to process a stream of data with an unknown number of items. In this scenario, you don't know the required number of iterations, so you can use a `while` loop with `True` as its control condition. This technique provides a Pythonic way to write loops that will run an unknown number of iterations.

To illustrate, suppose you need to read a temperature sensor that continuously provides data. When the temperature is equal to or greater than 28 degrees Celsius, you should stop monitoring it. Here's a loop to accomplish this task:

Python

temperature.py

```
import random
import time

def read_temperature():
    return random.uniform(20.0, 30.0)

while True:
    temperature = read_temperature()
    print(f"Temperature: {temperature:.2f}°C")

    if temperature >= 28:
        print("Required temperature reached! Stopping monitoring.")
        break

    time.sleep(1)
```

In this loop, you use `True` as the loop condition, which generates a continually running loop. Then, you read the temperature sensor and print the current temperature value. If the temperature is equal to or greater than 25 degrees, you break out of the loop. Otherwise, you wait for a second and read the sensor again.

Removing Items From an Iterable in a Loop

[Modifying a collection during iteration](#) can be risky, especially when you need to remove items from the target collection. In some cases, using a `while` loop can be a good solution.

For example, say that you need to process a list of values and remove each value after it's processed. In this situation, you can use a `while` loop like the following:

Python



```
>>> colors = ["red", "blue", "yellow", "green"]

>>> while colors:
...     color = colors.pop(-1)
...     print(f"Processing color: {color}")
...
Processing color: green
Processing color: yellow
Processing color: blue
Processing color: red
```

When you evaluate a list in a [Boolean context](#), you get `True` if it contains elements and `False` if it's empty. In this example, `colors` remains true as long as it has elements. Once you remove all the items with the `.pop()` method, `colors` becomes false, and the loop terminates.

Getting User Input With a `while` Loop

Getting user input from the command line is a common use case for `while` loops in Python. Consider the following loop that takes input using the built-in [input\(\)](#) functions. The loop runs until you type the word "stop":

Python

user_input.py

```
line = input("Type some text: ")

while line != "stop":
    print(line)
    line = input("Type some text: ")
```

The `input()` function asks the user to enter some text. Then, you assign the result to the `line` [variable](#). The loop condition checks whether the content of `line` is different from "stop", in which case, the loop body executes. Inside the loop, you call `input()` again. The loop repeats until the user types the word `stop`.

This example works, but it has the drawback of unnecessarily repeating the call to `input()`. You can avoid the repetition using the [walrus operator](#) as in the code snippet below:

Python



```
>>> while (line := input("Type some text: ")) != "stop":
...     print(line)
...
Type some text: Python
Python
Type some text: Walrus
Walrus
Type some text: stop
```

In this updated loop, you get the user input in the `line` variable using an assignment expression. At the same time, the expression returns the user input so that it can be compared to the sentinel value "stop".

Traversing Iterators With `while` Loops

Using a `while` loop with the built-in `next()` function may be a great way to fine-control the iteration process when working with [iterators and iterables](#).

To give you an idea of how this works, you'll rewrite a `for` loop using a `while` loop instead. Consider the following code:

Python

for_loop.py

```
requests = ["first request", "second request", "third request"]

print("\nWith a for loop")
for request in requests:
    print(f"Handling {request}")

print("\nWith a while loop")
it = iter(requests)
while True:
    try:
        request = next(it)
    except StopIteration:
        break

    print(f"Handling {request}")
```

The two loops are equivalent. When you run the script, each loop will handle the three requests in turn:

Shell



```
$ python for_loop.py

With a for-loop
Handling first request
Handling second request
Handling third request

With a while-loop
Handling first request
Handling second request
Handling third request
```

Python's `for` loops are quite flexible and powerful, and you should generally prefer `for` over `while` if you need to iterate over a given collection. However, translating the `for` loop into a `while` loop, like above, gives you even more flexibility in how you handle the iterator.

Emulating Do-While Loops

A [do-while loop](#) is a control flow statement that executes its code block at least once, regardless of whether the loop condition is true or false.

If you come from languages like [C](#), [C++](#), [Java](#), or [JavaScript](#), then you may be wondering where Python's do-while loop is. The bad news is that Python doesn't have one. The good news is that you can emulate it using a `while` loop with a `break` statement.

Consider the following example, which takes user input in a loop:

Python



```
>>> while True:
...     number = int(input("Enter a positive number: "))
...     print(number)
...     if not number > 0:
...         break
...
Enter a positive number: 1
1
Enter a positive number: 4
4
Enter a positive number: -1
-1
```

Again, this loop takes the user input using the built-in `input()` function. The input is then converted into an integer number using [int\(\)](#). If the user enters a number that's 0 or lower, then the `break` statement runs, and the loop terminates.

Note that to emulate a do-while loop in Python, the condition that terminates the loop goes at the end of the loop, and its body is a `break` statement. It's also important to emphasize that in this type of loop, the body runs at least once.

Using while Loops for Event Loops

Another common and extended use case of `while` loops in Python is to create [event loops](#), which are infinite loops that wait for certain actions known as [events](#). Once an event happens, the loop dispatches it to the appropriate [event handler](#).

Some classic examples of fields where you'll find event loops include the following:

- [Graphical user interface \(GUI\)](#) frameworks
- [Game](#) development
- [Asynchronous](#) applications
- [Server](#) processes

For example, say that you want to implement a number-guessing game. You can do this with a `while` loop:

Python

guess.py

```
from random import randint

LOW, HIGH = 1, 10

secret_number = randint(LOW, HIGH)
clue = ""

# Game loop
while True:
    guess = input(f"Guess a number between {LOW} and {HIGH} {clue} ")
    number = int(guess)
    if number > secret_number:
        clue = f"(less than {number})"
    elif number < secret_number:
        clue = f"(greater than {number})"
    else:
        break

print(f"You guessed it! The secret number is {number}")
```

In this example, the game loop runs the game logic. First, it takes the user's guess with `input()`, converts it into an integer number, and checks whether it's greater or less than the secret number. The `else` clause executes when the user's guess is equal to the secret number. In that case, the loop breaks, and the game shows a winning message.

Exploring Infinite while Loops

Sometimes, you might write a `while` loop that doesn't naturally terminate. A loop with this behavior is commonly known as an **infinite loop**, although the name isn't quite accurate because, in the end, you'll have to terminate the loop somehow.

You may write an infinite loop either *intentionally* or *unintentionally*. Intentional infinite loops are powerful tools commonly used in programs that need to run continuously until an external condition is met, such as game loops, server processes, and event-driven apps like GUI apps or asynchronous code.

In contrast, unintentional infinite `while` loops are often the result of some kind of logical issue that prevents the loop condition from ever becoming false. For example, this can happen when the loop:

- Fails to update a condition variable
- Uses a condition that's logically flawed

In these cases, the loop erroneously continues to run until it's terminated externally.

In the following sections, you'll learn about both types of infinite loops and how to approach them in your code. To kick things off, you'll start with unintentional infinite loops.

Unintentional Infinite Loops

Suppose you write a `while` loop that never ends due to an internal error. Getting back to the example in the initial section of this tutorial, you have the following loop that runs continuously:

Python

```
>>> number = 5
>>> while number != 0:
...     print(number)
...     number -= 2
...
5
3
1
-1
-3
-5
-7
-9
-11
Traceback (most recent call last):
...
KeyboardInterrupt
```

To terminate this code, you have to press `^Ctrl + C`, which interrupts the program’s execution from the keyboard. Otherwise, the loop would run indefinitely since its condition never turns false. In real-world code, you’d typically want to have a proper loop condition to prevent unintended infinite execution.

In most cases like this, you can prevent the potentially infinite loop by fixing the condition itself or the internal loop logic to make sure the condition becomes false at some point in the loop’s execution.

In the example above, you can modify the condition a bit to fix the issue:

Python

```
>>> number = 5
>>> while number > 0:
...     print(number)
...     number -= 2
...
5
3
1
```

This time, the loop doesn’t go down the 0 value. Instead, it terminates when `number` has a value that’s equal to or less than 0.

Alternatively, you can use additional conditionals in the loop body to terminate the loop using `break`:

Python

```
>>> number = 5
>>> while number != 0:
...     if number <= 0:
...         break
...     print(number)
...     number -= 2
...
5
3
1
```

This loop has the same original loop condition. However, it includes a failsafe condition in the loop body to terminate it in case the main condition fails.

Figuring out how to fix an unintentional infinite loop will largely depend on the logic you’re using in the loop condition and body. So, you should analyze each case carefully to determine the correct solution.

Intentional Infinite Loops

Intentionally infinite `while` loops are pretty common and useful. However, writing them correctly requires ensuring proper exit conditions, avoiding performance issues, and preventing unintended infinite execution.

For example, you might want to write code for a service that starts up and runs forever, accepting service requests. *Forever*, in this context, means until you shut it down.

The typical way to write an infinite loop is to use the `while True` construct. To ensure that the loop terminates naturally, you should add one or more `break` statements wrapped in proper conditions:

Python Syntax

```
while True:
    if condition_1:
        break
    ...
    if condition_2:
        break
    ...
    if condition_n:
        break
```

This syntax works well when you have multiple reasons to end the loop. It's often cleaner to break out from several different locations rather than try to specify all the termination conditions in the loop header.

To see this construct in practice, consider the following infinite loop that asks the user to provide their password:

Python

password.py

```
MAX_ATTEMPTS = 3

correct_password = "secret123"
attempts = 0

while True:
    password = input("Password: ").strip()
    attempts += 1

    if password == correct_password:
        print("Login successful! Welcome!")
        break

    if attempts >= MAX_ATTEMPTS:
        print("Too many failed attempts.")
        break
    else:
        print(f"Incorrect password. {MAX_ATTEMPTS - attempts} attempts left.")
```

This loop has two exit conditions. The first condition checks whether the password is correct. The second condition checks whether the user has reached the maximum number of attempts to provide a correct password. Both conditions include a `break` statement to finish the loop gracefully.

Conclusion

You've learned a lot about Python's `while` loop, which is a crucial control flow structure for iteration. You've learned how to use `while` loops to repeat tasks until a condition is met, how to tweak loops with `break` and `continue` statements, and how to prevent or write infinite loops.

Understanding `while` loops is essential for Python developers, as they let you handle tasks that require repeated execution based on conditions.

In this tutorial, you've learned how to:

- Understand the **syntax** of Python `while` loops
- **Repeat** tasks when a **condition is true** with `while` loops

- Use `while` loops for tasks with **an unknown number of iterations**
- Control loop execution with `break` and `continue` statements
- **Avoid unintended infinite loops** and **write correct ones**

This knowledge enables you to write dynamic and flexible code, particularly in situations where the number of iterations is unknown beforehand or depends on one or more conditions.

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

Frequently Asked Questions

Now that you have some experience with Python `while` loops, 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 a `while` loop in Python?

Show/Hide

How does a `while` loop differ from a `for` loop?

Show/Hide

How can you prevent an infinite loop in Python?


Show/Hide

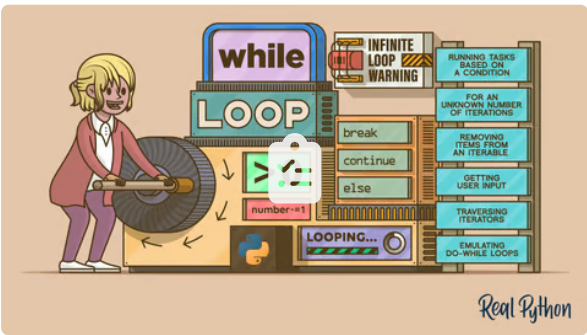
What is the purpose of the `break` statement in a `while` loop?

Show/Hide

Can you use an `else` clause with a `while` loop in Python?

Show/Hide

 **Take the Quiz:** Test your knowledge with our interactive “Python while Loops: Repeating Tasks Conditionally” quiz. You’ll receive a score upon completion to help you track your learning progress:



Interactive Quiz

[Python while Loops: Repeating Tasks Conditionally](#)

In this quiz, you'll test your understanding of Python's while loop. This loop allows you to execute a block of code repeatedly as long as a given condition remains true. Understanding how to use while loops effectively is a crucial skill for any Python developer.

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: [Mastering While Loops](#)

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](#)



[Geir Arne](#)



[Joanna](#)



[John](#)



[Kyle](#)



[Philipp](#)

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: [Mastering While Loops](#)

Related Tutorials:

- [Python for Loops: The Pythonic Way](#)
- [Python Exceptions: An Introduction](#)
- [Defining Your Own Python Function](#)
- [Lists vs Tuples in Python](#)
- [Strings and Character Data 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.

