Python `continue` Statement

1. What Is `continue` in Python?

The `continue` statement is used to **skip the rest of the current loop iteration** and immediately move to the **next iteration**.

Unlike `break`, `continue` does **not stop the loop**.

Basic example

```
for number in range(1, 6):
    if number == 3:
        continue

    print(number)
```

Output:

```
1
2
4
5
```

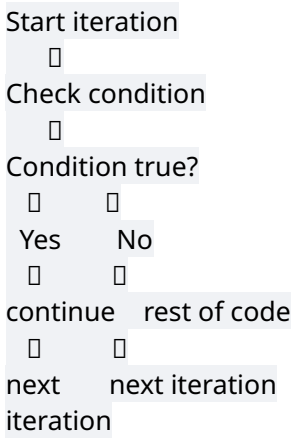
When `number` becomes `3`, Python executes `continue`.

It skips:

```
print(number)
```

for that iteration and moves to `4`.

Mental model



---

2. Why Do We Need `continue`?

Sometimes you want a loop to keep running but **ignore certain values**.

For example, suppose you want to print only positive numbers:

```
numbers = [10, -5, 20, -2, 30]

for number in numbers:
    if number < 0:
        continue

    print(number)
```

Output:

```
10
20
30
```

The negative numbers are skipped, but the loop continues processing the remaining numbers.

This is one of the most common uses of `continue`.

---

3. Syntax of `continue`

The syntax is simply:

```
continue
```

It must be used inside a loop.

Example:

```
for number in range(10):
    if number == 5:
        continue

    print(number)
```

---

4. `continue` With a `for` Loop

Consider:

```
for number in range(1, 6):
    print(number)
```

Output:

```
1
2
3
4
5
```

Now skip `3`:

```
for number in range(1, 6):
    if number == 3:
        continue

    print(number)
```

Output:

```
1
2
4
5
```

The loop itself never stopped.

Only the iteration where `number == 3` was skipped.

---

5. What Exactly Happens When `continue` Runs?

Look at this:

```
for number in range(1, 6):
    if number == 3:
```

```
        continue

    print("Number:", number)
```

Python processes it like this:

```
number = 1 → print
number = 2 → print
number = 3 → continue
number = 4 → print
number = 5 → print
```

At `3`, Python jumps back to the loop and starts the next iteration.

---

6. `continue` Does Not Stop the Loop

This is the biggest difference between `continue` and `break`.

```
for number in range(1, 6):
    if number == 3:
        continue

    print(number)
```

Output:

```
1
2
4
5
```

The loop continues after `3`.

Compare that with `break`:

```
for number in range(1, 6):
    if number == 3:
        break

    print(number)
```

Output:

```
1
2
```

Remember

```
break    → stop the loop
continue → skip this iteration
```

---

7. `continue` With a `while` Loop

`continue` can also be used with `while`.

```
number = 0

while number < 5:
    number += 1

    if number == 3:
        continue

    print(number)
```

Output:

```
1
2
4
5
```

When `number` becomes `3`, the current iteration is skipped.

---

8. Important `while` Loop Warning

With `while` loops, you must be careful about where you update the variable.

This can cause an infinite loop:

```
number = 0

while number < 5:
    if number == 3:
        continue

    number += 1
```

When `number` becomes `3`, `continue` runs before `number += 1`.

So `number` stays `3` forever.

The loop repeatedly reaches:

```
if number == 3:
    continue
```

Correct version

Update the variable before `continue`:

```
number = 0

while number < 5:
    number += 1

    if number == 3:
        continue

    print(number)
```

This is an important debugging concept.

---

9. Skipping Even Numbers

You can use `continue` to skip values you don't want.

```
for number in range(1, 11):
    if number % 2 == 0:
        continue

    print(number)
```

Output:

```
1
3
5
```

```
7
9
```

The even numbers are skipped.

---

10. Skipping Odd Numbers

The same idea works in reverse:

```
for number in range(1, 11):
    if number % 2 != 0:
        continue

    print(number)
```

Output:

```
2
4
6
8
10
```

---

11. `continue` With Strings

Suppose you want to print only names longer than three characters:

```
names = ["Ali", "Sara", "Hafsa", "Zain", "Ahmed"]

for name in names:
```

```
    if len(name) <= 3:
        continue

    print(name)
```

Output:

```
Sara
Hafsa
Ahmed
```

The shorter names are skipped.

---

12. `continue` for Filtering Data

One of the most useful mental models for `continue` is:

> **Ignore this item and keep processing everything else.**

Example:

```
numbers = [5, 12, 7, 20, 3, 15]

for number in numbers:
    if number < 10:
        continue

    print(number)
```

Output:

```
12
20
15
```

The loop acts like a simple filter.

---

13. Skipping Invalid Data

Imagine processing user records:

```
users = ["Ali", "", "Sara", "", "Hafsa"]

for user in users:
    if user == "":
        continue

    print("Processing:", user)
```

Output:

```
Processing: Ali
Processing: Sara
Processing: Hafsa
```

Empty values are ignored.

This pattern is useful when working with:

- user input
- API responses
- database records

- files
- lists of objects
- imported data

---

14. Skipping Invalid User Input

Suppose a program expects positive numbers:

```
numbers = [10, -2, 20, -5, 30]

for number in numbers:
    if number <= 0:
        continue

    print("Valid number:", number)
```

Output:

```
Valid number: 10
Valid number: 20
Valid number: 30
```

Instead of creating deeply nested conditions, `continue` lets you reject unwanted cases early.

---

15. `continue` and `if`

`continue` is commonly used inside an `if` statement.

```
for item in items:
```

```
    if unwanted_condition:
        continue

    # Process valid item
```

This creates a useful structure:

```
Check item
    
Unwanted?
    
Yes  continue
    
No
    
Process item
```

---

16. `continue` vs Nested `if`

Without `continue`:

```
numbers = [10, -5, 20, -2, 30]

for number in numbers:
    if number >= 0:
        print(number)
```

With `continue`:

```
numbers = [10, -5, 20, -2, 30]

for number in numbers:
```

```
    if number < 0:
        continue

    print(number)
```

Both work.

The second approach can become especially useful when there are **multiple reasons to skip an item**.

Example:

```
for number in numbers:
    if number < 0:
        continue

    if number == 0:
        continue

    print(number)
```

Only positive, non-zero numbers are processed.

---

17. Multiple `continue` Conditions

You can have more than one `continue` in a loop.

```
numbers = [10, 0, -5, 20, 0, 30]

for number in numbers:
    if number < 0:
        continue
```

```
    if number == 0:
        continue

    print(number)
```

Output:

```
10
20
30
```

A more compact version could be:

```
for number in numbers:
    if number <= 0:
        continue

    print(number)
```

---

18. `continue` With `range()`

`continue` works naturally with `range()`.

```
for number in range(1, 11):
    if number == 5:
        continue

    print(number)
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Notice that `range()` still generates the sequence.

`continue` only changes what happens during the current iteration.

---

19. `continue` With Nested Loops

Consider:

```
for i in range(3):
    for j in range(3):
        if j == 1:
            continue

        print(i, j)
```

Output:

```
0 0
0 2
1 0
1 2
```

```
2 0
2 2
```

The `continue` affects only the **innermost loop** where it appears.

It does not skip the outer loop.

Mental model

```
Outer loop
   
Inner loop
   
continue
   
Skip current inner iteration
   
Inner loop continues
```

---

20. `continue` Does Not Mean "Skip Everything"

Consider:

```
for number in range(1, 6):
    print("Start")

    if number == 3:
        continue

    print("End")
```

Output:

```
Start
End
Start
End
Start
Start
End
Start
End
```

For `3`:

```
Start
□
continue
□
next iteration
```

So `"End"` is skipped, but code before `continue` still runs.

---

21. `continue` vs `break`

This is one of the most important comparisons in Python.

Statement	Effect
`break`	Exits the loop completely
`continue`	Skips the current iteration

`return`	Exits the current function
`pass`	Does nothing

Example

```
for number in range(1, 6):
    if number == 3:
        continue

    print(number)
```

Result:

```
1
2
4
5
```

With `break`

```
for number in range(1, 6):
    if number == 3:
        break

    print(number)
```

Result:

```
1
2
```

Simple memory trick

BREAK
→
Leave

CONTINUE
→
Keep going

---

22. continue vs return

continue only affects the current loop iteration.

return exits the function.

```
def process_numbers():
    for number in range(1, 6):
        if number == 3:
            continue

        print(number)

    print("Function still continues")

process_numbers()
```

Output:

```
1
2
4
```

```
5
Function still continues
```

With `return`:

```
def process_numbers():
    for number in range(1, 6):
        if number == 3:
            return

        print(number)

    print("Function still continues")
```

Output:

```
1
2
```

The function ends completely.

---

23. `continue` and Loop `else`

Python allows an `else` block after a loop.

A loop's `else` still executes if the loop completes normally, even if `continue` was used.

```
for number in range(1, 6):
    if number == 3:
        continue
```

```
    print(number)
else:
    print("Loop completed")
```

Output:

```
1
2
4
5
Loop completed
```

Why?

Because `continue` does not terminate the loop.

Compare:

```
for number in range(1, 6):
    if number == 3:
        break

    print(number)
else:
    print("Loop completed")
```

Here, the `else` block does not execute because `break` stopped the loop.

Remember

```
continue → loop continues → else can run
break    → loop stops     → else does not run
```

---

24. Real World Example: Processing haas.dev Resources

Imagine you have a list of resources and some are drafts.

```
resources = [
    {"title": "Python Variables", "published": True},
    {"title": "Python Conditions", "published": False},
    {"title": "Python Loops", "published": True},
    {"title": "Python Functions", "published": True}
]

for resource in resources:
    if not resource["published"]:
        continue

    print("Showing:", resource["title"])
```

Output:

```
Showing: Python Variables
Showing: Python Loops
Showing: Python Functions
```

The draft resource is skipped.

This is the kind of logic you'll use when working with real applications and data.

---

25. Real World Example: Processing API Data

Suppose an API returns users:

```
users = [
    {"name": "Ali", "active": True},
    {"name": "Sara", "active": False},
    {"name": "Hafsa", "active": True}
]

for user in users:
    if not user["active"]:
        continue

    print("Processing:", user["name"])
```

Output:

```
Processing: Ali
Processing: Hafsa
```

Inactive users are ignored.

The same idea appears in backend development, data processing, and application logic.

---

26. Real World Example: Skip Empty Input

```
while True:
    text = input("Enter something: ")

    if text == "":
        continue

    if text == "quit":
        break
```

```
    print("You entered:", text)
```

Here:

- Empty input → `continue`
- `"quit"` → `break`
- Anything else → process it

This demonstrates how `continue` and `break` can work together.

---

27. A Powerful Pattern: Validate Then Process

A clean programming pattern is:

```
for item in items:
    if invalid(item):
        continue

    process(item)
```

Instead of:

```
for item in items:
    if valid(item):
        process(item)
```

Neither is universally better, but `continue` can make complicated loops easier to read by handling invalid cases early.

---

28. Common Mistake: Thinking `continue` Stops the Loop

This:

```
for number in range(5):
    if number == 2:
        continue

    print(number)
```

does **not** stop at `2`.

It prints:

```
0
1
3
4
```

Only `2` is skipped.

---

29. Common Mistake: Using `continue` Outside a Loop

This is invalid:

```
if True:
    continue
```

`continue` must be inside a loop.

Correct:

```
for number in range(5):
```

```
    if number == 2:
        continue
```

---

30. Common Mistake: Creating an Infinite `while` Loop

Be careful:

```
number = 0

while number < 5:
    if number == 2:
        continue

    number += 1
```

When `number == 2`, the update never happens.

The program gets stuck.

A safe version:

```
number = 0

while number < 5:
    number += 1

    if number == 2:
        continue

    print(number)
```

Rule for `while`

Before using `continue`, ask:

> **Will the loop's condition-changing code still execute?**

If not, you may create an infinite loop.

---

31. Common Mistake: Putting Important Code After `continue`

Consider:

```
for number in numbers:
    if number < 0:
        continue

    process(number)
```

This is fine if negative numbers should be ignored.

But if you need some code to run for **every item**, don't place it after `continue`.

For example:

```
for number in numbers:
    if number < 0:
        continue

    total += number
```

Negative numbers never reach `total += number`.

Understand exactly which statements will be skipped.

---

32. Debugging `continue`

When a loop behaves unexpectedly, check:

1. Which condition triggers `continue`?

```
if number < 0:
    continue
```

Make sure this condition is correct.

2. What code comes after it?

Everything after `continue` in the current iteration is skipped.

3. Is this a `while` loop?

If yes, make sure your loop variable still changes.

4. Is the `continue` inside the intended loop?

Indentation matters:

```
for item in items:
    if condition:
        continue
```

---

33. Mini Quiz

Question 1

What does `continue` do?

A. Stops the loop

B. Stops the program

C. Skips the current iteration

D. Restarts Python

Answer: C

Question 2

What will this print?

```
for number in range(1, 6):
    if number == 4:
        continue

    print(number)
```

A.

```
1
2
3
```

B.

```
1
2
3
5
```

C.

```
4
5
```

D. Nothing

Answer: B

Question 3

What is the difference between `break` and `continue`?

Answer:

`break` exits the loop completely, while `continue` skips only the current iteration and allows the loop to continue.

---

34. 5 Minute Challenge

Create a program that prints numbers from `1` to `20`.

Requirements:

1. Use a `for` loop.
2. Skip all numbers divisible by `3`.
3. Use `continue`.
4. Print all other numbers.

Expected output:

```
1
2
4
5
7
8
10
11
13
14
16
17
19
20
```

Do not use multiple `if` blocks to manually list the numbers.

---

35. Exercises

Exercise 1: Skip Zero

Given:

```
numbers = [5, 0, 10, 0, 15]
```

Use `continue` to skip zero values.

Expected:

```
5
10
15
```

Exercise 2: Print Positive Numbers

Given:

```
numbers = [-5, 10, -2, 20, 0, 30]
```

Skip all values less than or equal to zero.

Expected:

```
10
20
30
```

Exercise 3: Skip Short Names

Given:

```
names = ["Ali", "Hafsa", "Sara", "Jo", "Ahmed"]
```

Skip names with fewer than four characters.

Expected:

```
Hafsa
Sara
Ahmed
```

Exercise 4: Skip Draft Resources

Given:

```
resources = [
    {"title": "Python Variables", "published": True},
    {"title": "Python Loops", "published": False},
    {"title": "Python Functions", "published": True}
]
```

Use `continue` to skip unpublished resources.

---

36. Mini Project: Data Cleaner

Build a small program that processes a list of values:

```
data = [10, -5, 20, 0, 15, -3, 30]
```

Requirements:

- Skip negative numbers.
- Skip zero.
- Process positive numbers.

- Print each valid number.
- Calculate the total of valid numbers.

Expected:

```
Valid: 10
Valid: 20
Valid: 15
Valid: 30

Total: 75
```

Use `continue` to ignore invalid values.

---

37. Interview Questions

Beginner

1. What is `continue` in Python?

`continue` skips the remaining statements in the current iteration and moves to the next iteration.

2. Does `continue` stop a loop?

No. The loop continues with its next iteration.

3. Where can `continue` be used?

Inside `for` and `while` loops.

Intermediate

4. What is the difference between `break` and `continue`?

`break` exits the loop completely. `continue` skips only the current iteration.

5. What happens to code after `continue`?

It is skipped for the current iteration.

6. Can `continue` cause an infinite loop?

Yes, particularly in `while` loops if the condition-changing code is placed after `continue` and therefore never executes.

Advanced

7. What happens to a loop's `else` block when `continue` is used?

`continue` does not prevent the loop from completing normally, so the `else` block can still execute.

8. In nested loops, which loop does `continue` affect?

It affects the innermost loop containing the `continue` statement.

---

38. Cheat Sheet

Skip one iteration

```
for item in items:
    if condition:
```

```
        continue

    process(item)
```

Skip even numbers

```
for number in range(1, 11):
    if number % 2 == 0:
        continue

    print(number)
```

Skip invalid values

```
for value in values:
    if value <= 0:
        continue

    process(value)
```

while loop

```
while condition:
    update()

    if skip_condition:
        continue

    process()
```

Remember

```
break
 Stop the loop

continue
```

 Skip this iteration

return
 Exit the function

pass
 Do nothing

---

39. Key Takeaways

- continue skips the current loop iteration.
- It does not stop the loop.
- It works with both for and while loops.
- Code after continue is skipped for that iteration.
- It is useful for filtering unwanted data.
- It is useful for ignoring invalid input.
- In nested loops, it affects the innermost loop.
- Unlike break, continue allows the loop to keep running.
- Be especially careful with continue inside while loops.
- A loop can still reach its else block after using continue.

The core idea:

Loop starts

Check condition

Should this item be skipped?

Yes continue

Next iteration

Visit haas.dev for more resources and guides.

haas.dev

https://dev-roast-app.vercel.app/resources