

PYTHON KEYWORDS

continue

Python Keywords

REFERENCE · CONTINUE

⏭️ Skipping to the Next Pass

Where `break` ends a loop, `continue` only ends the current pass and lets the loop carry on. It is the tool for "ignore this one": a blank line, an invalid entry, a record that fails a check. This page covers how `continue` behaves in `for` and `while` loops, the one trap that can turn it into an infinite loop, how it works with nested loops, the loop `else` and `try` / `except`, and how it works alongside `break`. Every example was run, and the output shown is the real output.

SYNTAX AT A GLANCE

```
for item in iterable:           # or: while condition:
    if skip_this_one:
        continue                # jump straight to the next pass
    ...                         # not reached when skipped
```

Legal only inside a `for` or `while` loop. Related keywords: `break`, `else`, `pass`.

⚡ COMING FROM JAVA / JAVASCRIPT

Same keyword, same job. `continue` skips the rest of the loop body and moves on to the next pass in Python, Java and JavaScript. Python has **no labelled continue** (`continue outer;`), so it always applies to the innermost loop. In JavaScript, `forEach` callbacks have no `continue` : you `return` early from the callback instead. One difference that matters more in Python: loops have an `else` block, and `continue` does *not* prevent it from running; only `break` does.

🚀 FASTAPI RELEVANCE

Batch processing is the everyday use. An endpoint that accepts a list of records (an uploaded CSV, a JSON array) can `continue` past rows that fail validation while collecting their errors, then return the good results and a list of problems in one response (Example 6), instead of failing the whole request because of one bad row. FastAPI's own request validation uses Pydantic for the same job, but plain loops like this are what you will write for anything it does not cover.

🧰 The Building Blocks

Eight pieces. The first two are the everyday uses, the third is the trap, and the rest are how it behaves around other features.

FOR LOOP

`continue` in a `for` loop

Skips the rest of the current pass and jumps straight to the next item. The loop itself carries on, unlike `break`, which ends it.

```
for n in [3, -1, 8, -5, 4]:
    if n < 0:
        continue
    print("positive:", n)

# Output:
# positive: 3
# positive: 8
# positive: 4
```

WHILE LOOP

Re-asking after bad input

In a `while` loop, `continue` jumps back to the top and the condition is checked again. A good fit for "that was not valid, ask again".

```
while True:
    text = input("Enter a number: ")
    if not text.isdigit():
        print("Digits only, try again")
        continue
    print("Thanks, you entered", int(text))
    break

# Sample run:
# Enter a number: abc
# Digits only, try again
# Enter a number:
# Digits only, try again
# Enter a number: 42
# Thanks, you entered 42
```

THE TRAP

continue can skip your counter

In a `while` loop, anything after the `continue` is skipped, including the line that updates the loop variable. If that line is skipped, the condition never changes and the loop never ends.

```
i = 0
while i < 6:
    if i % 2 == 0:
        continue      # i never changes: infinite loop!
    print(i)
    i += 1
```

STYLE

Guard clauses instead of nesting

Each `continue` says “not this one” and gets rid of a case early, so the main work stays at one indentation level. Compare the two versions: same output.

```
numbers = [4, -2, 7, 10, -3, 6]

for n in numbers:              # nested version
    if n > 0:
        if n % 2 == 0:
            print(n)

print("--")
for n in numbers:              # guard-clause version
    if n <= 0:
        continue
    if n % 2 != 0:
        continue
    print(n)

# Output:
# 4
# 10
# 6
# --
# 4
# 10
# 6
```

NESTED LOOPS

Only the innermost loop

Like `break`, `continue` belongs to the nearest loop around it. Here it skips column 1 on every row, and the outer loop is not affected.

```
for row in range(3):
    for col in range(3):
        if col == 1:
            continue
        print(row, col)

# Output:
# 0 0
# 0 2
# 1 0
# 1 2
# 2 0
# 2 2
```

ELSE CLAUSE

continue does not stop the else

The loop `else` runs when the loop ends without a `break`. A `continue` is not a break, so the `else` still runs.

```
for n in [1, 2, 3]:
    if n == 2:
        continue
    print(n)
else:
    print("loop finished normally")

# Output:
# 1
# 3
# loop finished normally
```

WITH TRY

continue inside except

A natural pairing: try to convert, and if it fails, report it and move on to the next item. The conversion code after the `try` only runs for good values.

```
raw = ["10", "x", "25", "", "7"]
total = 0
for item in raw:
    try:
        value = int(item)
```

ALTERNATIVE

A comprehension can replace it

When the loop only filters and collects, a list comprehension with an `if` says the same thing in one line. Keep `continue` for loops that do real work with each item.

```
numbers = [3, -1, 8, -5, 4]

positives = []
for n in numbers:
    if n < 0:
```

```
    except ValueError:
        print(f"skipping {item!r}")
        continue
    total += value
print("total:", total)

# Output:
# skipping 'x'
# skipping ''
# total: 42
```

```
        continue
    positives.append(n)

print(positives)
print([n for n in numbers if n >= 0])

# Output:
# [3, 8, 4]
# [3, 8, 4]
```

🔍 How It Works, Step by Step

1. The loop runs its body normally.
2. When Python reaches the `continue` line, it abandons the **rest of this pass**. Nothing after the `continue` in the loop body runs.
3. In a `for` loop, control moves to the next item (or ends the loop if there are none left). In a `while` loop, control goes back to the top and the condition is tested again.
4. The loop is **not** over. That is the whole difference from `break`.

When to reach for it: whenever some items should be ignored and you would otherwise have to wrap the whole body in an `if`: skipping blank lines, invalid entries, items that fail a check, or input that should be asked for again.

🧪 In Context

Real uses of `continue`, including the guessing game from the `break` page and a loop that uses both keywords together.

Example 1: Reading Settings: Skip Blank Lines and Comments

Typical text-file processing. Blank lines and lines starting with `#` are not data, so `continue` throws them away and the three lines that follow only ever see real settings.

```
lines = [
    "# settings",
    "name = demo",
    "",
    "debug = true",
    "   ",
    "# end",
]

for line in lines:
    line = line.strip()
    if not line or line.startswith("#"):
        continue
    key, value = line.split(" = ")
    print(key, "->", value)

# Output:
# name -> demo
# debug -> true
```

Example 2: continue in a while Loop, Done Safely

Adding up the odd numbers from 1 to 9. The fix for the infinite-loop trap is to update the counter before the `continue` can skip it: here `i += 1` is the first line of the body.

```
i = 0
total = 0
while i < 10:
    i += 1                # first, so continue cannot skip it
    if i % 2 == 0:
        continue
    total += i

print("sum of odd numbers up to 10:", total)

# Output:
# sum of odd numbers up to 10: 25
```

Example 3: The Guessing Game: Bad Input Does Not Cost an Attempt

Back to the game from the `break` page. A typo or an out-of-range number should not use up one of the five attempts. `continue` asks again without touching the counter, and `break` still ends the game on a correct guess. The two keywords work side by side.

```
import random

target = random.randint(1, 10)
attempts = 0

while attempts < 5:
    text = input("Guess (1-10): ")
    if not text.isdigit() or not 1 <= int(text) <= 10:
        print("Please enter a whole number from 1 to 10")
        continue              # not counted as an attempt
    attempts += 1
    guess = int(text)
    if guess == target:
        print(f"You got it in {attempts} attempts!")
        break
    print("Too low!" if guess < target else "Too high!")
else:
    print(f"Too bad! The number was {target}")

# Sample run:
# Guess (1-10): abc
# Please enter a whole number from 1 to 10
# Guess (1-10): 15
# Please enter a whole number from 1 to 10
# Guess (1-10): 5
# Too low!
# Guess (1-10): 9
# Too high!
# Guess (1-10): 7
# You got it in 3 attempts!
```

Look at the order inside the loop. The counter is only increased *after* the validation check, so a `continue` before that line leaves it alone. Moving `attempts += 1` to the top of the loop would charge an attempt for every typo.

Example 4: Break and Continue Together: A Price Total

Entering prices until the user types `done`. `break` is the exit; `continue` rejects an entry that is not a number or is negative and asks again. Neither bad entry changes the total.

```
total = 0
while True:
    entry = input("Price (or done): ")
    if entry == "done":
        break
    try:
        price = float(entry)
    except ValueError:
        print("Not a number, skipped")
        continue
    if price < 0:
        print("Negative price, skipped")
        continue
    total += price

print(f"Total: {total:.2f}")

# Sample run:
# Price (or done): 4.50
# Price (or done): abc
# Not a number, skipped
# Price (or done): -3
# Negative price, skipped
# Price (or done): 2.25
# Price (or done): done
# Total: 6.75
```

Example 5: continue vs break vs pass

The three often get mixed up. Each loop below prints the numbers 1 to 5 and does something different at 3. `pass` is a placeholder that does nothing at all, so the rest of the body still runs.

```
for n in range(1, 6):
    if n == 3:
        continue
    print(n, end=" ")
print("<- continue skips 3")

for n in range(1, 6):
    if n == 3:
        break
    print(n, end=" ")
print("<- break stops at 3")

for n in range(1, 6):
    if n == 3:
        pass
    print(n, end=" ")
print("<- pass changes nothing")

# Output:
# 1 2 4 5 <- continue skips 3
```

```
# 1 2 <- break stops at 3
# 1 2 3 4 5 <- pass changes nothing
```

Example 6: Skipping Bad Records in a Batch

Processing a list of records where some are broken. `continue` keeps the good ones flowing, and the bad ones are collected so they can be reported at the end, a pattern you will use whenever a whole batch should not fail because of one bad row.

```
rows = [
    {"name": "Ann", "age": "34"},
    {"name": "", "age": "29"},
    {"name": "Cyd", "age": "abc"},
    {"name": "Dee", "age": "41"},
]

good, errors = [], []
for number, row in enumerate(rows, start=1):
    if not row["name"]:
        errors.append(f"row {number}: missing name")
        continue
    if not row["age"].isdigit():
        errors.append(f"row {number}: age is not a number")
        continue
    good.append((row["name"], int(row["age"])))

print("good:", good)
print("errors:", errors)

# Output:
# good: [('Ann', 34), ('Dee', 41)]
# errors: ['row 2: missing name', 'row 3: age is not a number']
```

📋 Quick Reference

WHAT YOU WANT	WRITE	NOTES
Skip the rest of this pass	`continue`	*Works in `for` and `while`. The loop itself carries on.*
Skip items that fail a check	`if bad: continue`	*Keeps the main work at one indentation level.*
Ask again after invalid input	`continue` inside `while True`	*Do the validation before the counter is updated.*
Skip a failed conversion	`except ValueError: continue`	*The code after the `try` only runs for good values.*
Keep a counter safe in a while loop	update it *before* the `continue`	*Otherwise the loop may never end.*
End the loop instead of skipping	`break`	*See the `break` page.*
Do nothing, keep the rest of the body	`pass`	*A placeholder, not a loop control.*
Filter into a new list	`[x for x in xs if cond]`	*Replaces a filter-only loop with a `continue`.*
continue outside any loop	`SyntaxError`	*Raised before the program runs at all.*

⚠ Gotchas for Java / JavaScript Programmers:

`continue` in a `while` loop can skip the update: *if the line that changes the loop variable (`i += 1`) comes after the `continue` , it is skipped and the loop never ends. In a `for` loop this cannot happen, because `for` advances itself. Update first, or use a `for` .*

`continue` is not `break` : *`break` ends the loop; `continue` only ends the current pass. If the loop stops when you expected it to carry on (or the other way round), check which one you wrote.*

There is no labelled `continue` : *Java and JavaScript allow `continue outer;` to jump to the next pass of an outer loop. In Python, `continue` always belongs to the innermost loop. To skip to the next outer item, break out of the inner loop (and check a flag), or put the inner part in a function that returns.*

JavaScript's `forEach` has no `continue` : *inside the callback you use `return` to skip an item. A Python `for` loop is a real loop, so `continue` and `break` both work directly.*

The loop `else` still runs after a `continue` : *only `break` prevents it. If the `else` is meant to mean "nothing was found", make sure a `continue` is not hiding the case you care about.*

Do not overuse it: *several `continue` lines in a row can make a loop harder to follow than a plain `if` . For a simple filter, a list comprehension is clearer, and a loop with one or two guard clauses is fine.*

Put validation before side effects: *anything that changes state (a counter, a total, a list) should come after the checks that may `continue` . Otherwise a rejected item has already left its mark, as in Example 3 where the attempt counter would have been charged for every typo.*