

PYTHON KEYWORDS

break

Python Keywords

REFERENCE · BREAK

Leaving a Loop Early

Your number-guessing game is the best possible introduction to `break`: a loop that has to stop the moment something happens, long before it would end by itself. This page covers what `break` does in `for` and `while` loops, how it behaves in nested loops, how it combines with the loop `else` block, how it differs from `return`, and the patterns you will use it for: guessing games, searches, menus and retry loops. Every example below was run, and the output shown is the real output.

SYNTAX AT A GLANCE

```
for item in iterable:           # or: while condition:
    if done:
        break                   # leave the loop now
else:                           # optional: runs only if no break happened
    print("loop finished without a break")
```

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

COMING FROM JAVA / JAVASCRIPT

Same keyword, same job. `break` leaves the nearest enclosing loop immediately, in Python, Java and JavaScript alike. The differences are around it. Python has **no labelled break** (`outer: for (...) { ... break outer; }`), so leaving two loops at once needs a flag or a function. Python loops can have an **`else` block** that runs only when no `break` happened, which Java and JavaScript lack. And Python has no `do-while`: `while True:` plus `break` is the idiom. JavaScript's `forEach` cannot be broken out of at all, whereas a Python `for` loop always can.

FASTAPI RELEVANCE

The `while True:` + `break` shape is how backend code talks to slow or unreliable services: keep asking an external API for the next page of results, and `break` when a page comes back empty; or retry a failed call a few times and `break` on the first success (Example 7). You will see both constantly in FastAPI services that call other services.

The Building Blocks

Eight pieces. The first three are the everyday uses; the rest are the behaviours that catch people out.

FOR LOOP

break in a for loop

Leaves the loop the moment it runs. Items still waiting in the list are never visited, and the rest of the loop body for the current pass is skipped too.

WHILE TRUE

The infinite loop and its exit

`while True:` never ends by itself, so it always needs a `break` somewhere inside. Use it when you cannot know in advance how many passes you need.

```
for n in [3, 8, 15, 4]:
    if n > 10:
        print("found one:", n)
        break
    print("checked", n)

# Output:
# checked 3
# checked 8
# found one: 15
```

```
while True:
    answer = input("Type quit to stop: ")
    if answer == "quit":
        break
    print("You typed", answer)

# Sample run:
# Type quit to stop: hello
# You typed hello
# Type quit to stop: again
# You typed again
# Type quit to stop: quit
```

WHILE CONDITION

An early exit from a normal while

A `while` with a real condition can still have a `break` for a second, unrelated reason to stop. Whichever happens first ends the loop.

```
total = 0
while total < 100:
    n = int(input("Add: "))
    if n < 0:
        print("Negative, stopping early")
        break
    total += n
print("total:", total)

# Sample run:
# Add: 40
# Add: 30
# Add: -1
# Negative, stopping early
# total: 70
```

NESTED LOOPS

Only the innermost loop

`break` leaves the *nearest* loop around it, not all of them. Here the inner loop stops at column 2 on every row, but the outer loop carries on.

```
for row in range(3):
    for col in range(3):
        if col == 2:
            break
        print(row, col)

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

ELSE CLAUSE

The loop's else block

A loop may have an `else` that runs only when the loop ends *without* a `break`. If a break fires, the else is skipped. Read it as “no break”.

```
for n in [3, 8, 15, 4]:
    if n > 100:
        print("found a big one")
        break
```

AFTER THE LOOP

The loop variable survives

After a `break` the loop variable still holds the value it had, so you can use it straight away. A loop that finishes normally leaves the last item there instead.

```
for n in [3, 8, 15, 4]:
    if n > 10:
        break
print("stopped at", n)
```

```
else:
    print("no number over 100")

# Output:
# no number over 100
```

```
# Output:
# stopped at 15
```

FUNCTION

break vs return

Inside a function, `return` also leaves any loop it is in, and sends a value back as well. Use `break` to leave a loop and carry on in the same block, and `return` to leave the whole function.

```
def first_negative(numbers):
    for n in numbers:
        if n < 0:
            return n
    return None

print(first_negative([4, 9, -2, 7, -8]))
print(first_negative([4, 9]))

# Output:
# -2
# None
```

ERROR

Outside a loop

`break` is only legal inside a `for` or `while`. An `if` on its own is not enough, and Python rejects the file before running anything.

```
if True:
    break
# SyntaxError: 'break' outside loop
```

🔍 How It Works, Step by Step

1. The loop runs its body normally, pass after pass.
2. When Python reaches the `break` line, it stops the loop **at once**. Nothing after the `break` in that pass runs, and the loop's condition or its remaining items are never looked at again.
3. If the loop has an `else` block, it is **skipped**.
4. Execution continues with the first line after the loop, at the loop's own indentation.

When to reach for it: whenever the answer is found, or the job is done, before the loop would end by itself: searching, validating input, menus, retry loops, and every `while True` loop, which has no other way out.

🧪 In Context

The first three examples are the guessing game from class, built three ways. The rest are the patterns `break` is used for in real programs.

Example 1: The Number Guessing Game (while True Version)

The game from class. The loop runs until the guess is right, and `break` is the only exit. The counter goes up before the check, so the winning guess is counted too. The hints are what make this a binary search in disguise: each answer halves the range.

```
import random

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

while True:
    guess = int(input("Guess a number from 1 to 10: "))
    tries += 1
    if guess == target:
        print(f"You got it in {tries} tries!")
        break
    elif guess < target:
        print("Too low!")
    else:
        print("Too high!")

# Sample run:
# Guess a number from 1 to 10: 5
# Too low!
# Guess a number from 1 to 10: 9
# Too high!
# Guess a number from 1 to 10: 7
# You got it in 3 tries!
```

Two bugs from class are worth remembering here. Wrapping `input()` in `print()` prints `None`, because `print()` returns nothing. And `input()` always gives back a *string*, so comparing it to the integer `target` is always `False`: the `int()` around it is what fixes that.

Example 2: Limited Attempts With a for Loop and a found Flag

Five attempts only. The `break` is still needed: without it the loop would keep asking for guesses even after a correct one. The `found` flag records why the loop ended, so the failure message can be printed once, after the loop, and not on every wrong guess.

```
import random

target = random.randint(1, 10)
found = False
```

```
for attempt in range(1, 6):
    guess = int(input(f"Attempt {attempt} of 5: "))
    if guess == target:
        print(f"You got it! The number was {target}")
        found = True
        break
    elif guess < target:
        print("Too low!")
    else:
        print("Too high!")

if not found:
    print(f"Too bad! The number was {target}")

# Sample run:
# Attempt 1 of 5: 3
# Too low!
# Attempt 2 of 5: 9
# Too high!
# Attempt 3 of 5: 5
# Too low!
# Attempt 4 of 5: 8
# Too high!
# Attempt 5 of 5: 1
# Too low!
# Too bad! The number was 7
```

`range(1, 6)` gives 1, 2, 3, 4, 5: the end value is excluded, so five passes need `6`.

Example 3: The Same Game With for/else: No Flag Needed

Python lets the loop answer the question itself. The `else` block runs only if the loop used up all five attempts without a `break`, which is exactly the "not found" case. Same behaviour as Example 2, one variable fewer.

```
import random

target = random.randint(1, 10)

for attempt in range(1, 6):
    guess = int(input(f"Attempt {attempt} of 5: "))
    if guess == target:
        print(f"You got it! The number was {target}")
        break
    elif guess < target:
        print("Too low!")
    else:
```

```
        print("Too high!")
else:
    print(f"Too bad! The number was {target}")

# Sample run:
# Attempt 1 of 5: 4
# Too low!
# Attempt 2 of 5: 9
# Too high!
# Attempt 3 of 5: 7
# You got it! The number was 7
```

Example 4: Searching: Stop at the First Match

Looking for the position of the first name that starts with `D`. Without the `break` the loop would carry on to the end, which wastes time on a long list and, if there were several matches, would report the last one instead of the first.

```
names = ["Ann", "Bo", "Cyd", "Dee", "Dan", "Eli"]
position = -1

for i, name in enumerate(names):
    if name.startswith("D"):
        position = i
        break

print("first D at index", position)

# Output:
# first D at index 3
```

Example 5: A Menu That Runs Until You Quit

The classic use of `while True`. The menu is shown again and again, and choosing option 3 is the only way out. Anything unexpected falls into the final `else` and the loop simply goes round again.

```
while True:
    print("1) Add   2) Show   3) Quit")
    choice = input("Choose: ")
    if choice == "3":
        print("Goodbye")
        break
    elif choice == "1":
        print("Adding...")
    elif choice == "2":
```

```
        print("Showing...")
    else:
        print("Invalid choice")

# Sample run:
# 1) Add   2) Show   3) Quit
# Choose: 1
# Adding...
# 1) Add   2) Show   3) Quit
# Choose: x
# Invalid choice
# 1) Add   2) Show   3) Quit
# Choose: 2
# Showing...
# 1) Add   2) Show   3) Quit
# Choose: 3
# Goodbye
```

Example 6: Leaving Two Loops at Once

Python has no `break outer;`. Two clean ways round it: a variable that the outer loop checks after the inner loop ends, or moving the loops into a function so that `return` leaves both at once. The function version is usually the tidier of the two.

```
grid = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Way 1: check a variable after the inner loop
found = None
for r, row in enumerate(grid):
    for c, value in enumerate(row):
        if value == 5:
            found = (r, c)
            break           # leaves the inner loop only
    if found is not None:
        break               # leaves the outer loop

print("way 1:", found)

# Way 2: return leaves every loop in the function
def find(grid, target):
    for r, row in enumerate(grid):
        for c, value in enumerate(row):
            if value == target:
                return (r, c)
    return None
```

```
print("way 2:", find(grid, 5))

# Output:
# way 1: (1, 1)
# way 2: (1, 1)
```

Example 7: A Retry Loop

Try something up to five times, stop at the first success, and give up if it never works. The shape is the same as the guessing game: a `for` over the attempts, a `break` on success, and an `else` for failure.

```
def flaky_call(attempt):
    # pretend the service only starts working on the 3rd try
    return attempt >= 3

for attempt in range(1, 6):
    if flaky_call(attempt):
        print(f"Succeeded on attempt {attempt}")
        break
    print(f"Attempt {attempt} failed, retrying...")
else:
    print("Giving up after 5 attempts")

# Output:
# Attempt 1 failed, retrying...
# Attempt 2 failed, retrying...
# Succeeded on attempt 3
```

📋 Quick Reference

WHAT YOU WANT	WRITE	NOTES
Leave the nearest loop now	`break`	*Works in `for` and `while`. Everything after it in that pass is skipped.*
Loop that only ends via break	`while True:`	*Python has no `do-while`: loop forever, test inside, break when satisfied.*
Run code only if no break happened	`for ...: ... else:`	*Also works on `while`. The `else` is skipped when a `break` fires.*
Count attempts in a while loop	`tries += 1`	*Put it before the check if the winning pass should count.*

WHAT YOU WANT	WRITE	NOTES
Limit a loop to N attempts	`for i in range(1, N + 1):`	*The end of `range` is excluded, so add 1.*
Leave two loops at once	`return` inside a function	*Or a flag checked after the inner loop. There is no labelled break.*
Leave a function from inside a loop	`return value`	*Leaves every loop and sends the value back.*
Loop variable after a break	still holds its last value	*Handy for "which item stopped us?".*
break outside any loop	`SyntaxError`	*Raised before the program runs at all.*

⚠ Gotchas for Java / JavaScript Programmers:

`break` leaves only one loop: *in nested loops it ends the innermost loop and the outer one carries on. There is no `break outer;` as in Java or JavaScript, so use a flag, or move the loops into a function and `return` (Example 6).*

The `else` after a loop reads backwards: *it does not mean "if the loop did not run". It runs when the loop finished without a `break`. Java and JavaScript have nothing like it, so say "no break" to yourself when you read one.*

A `while True` with no reachable `break` never ends: *the program just sits there. Press `Ctrl+C` in the terminal to stop it, then check that something inside the loop can actually trigger the `break`.*

Code after the check does not run on the winning pass: *if `tries += 1` came after the `if guess == target: ... break`, the winning guess would never be counted and "You got it in 3 tries" would say 2. Increment before the test.*

`input()` returns a string: *`input()` gives back text, so `guess == target` compares a string with an integer and is always `False`. Wrap it in `int()`. And `int()` crashes with `ValueError` on something like `abc`; the `continue` page shows how to re-ask instead.*

There is no `do-while`: *when you need "do this at least once, then check", write `while True:` and put an `if ...: break` where the real condition would go.*

`break` is not used with `match`: *in Java and JavaScript `break` ends each `case` of a `switch`. Python's `match` statement does not fall through, so no `break` is needed (or allowed) there unless you are inside a loop.*