

PYTHON KEYWORDS

while

Python Keywords

REFERENCE · WHILE

🔁 Looping While a Condition Holds

A `for` loop is for "do this for each item"; a `while` loop is for "keep doing this until something changes". It fits any job where you cannot say in advance how many passes you will need: reading input until it is valid, taking a number apart, working through a queue that may grow, narrowing a search range. This page covers how the condition is tested, what has to change inside the body, compound conditions, `while True`, the loop `else`, the walrus operator, and the mistakes that make a loop run forever. Every example below was run, and the output shown is the real output.

SYNTAX AT A GLANCE

```
while condition:                # tested before every pass
    ...                          # the body: must eventually make the condition False
else:                            # optional: runs when the condition becomes False
    print("condition became False")    # skipped if the loop ended with break
```

The condition is any expression; no parentheses are needed. Related keywords: `for`, `break`, `continue`, `else`, `if`, `True`.

COMING FROM JAVA / JAVASCRIPT

`while (cond) {}` **becomes** `while cond:`. Same idea, no parentheses or braces: the colon and the indentation define the block. `while True:` is `while (true)`. The differences: Python has **no** `do-while` (use `while True:` with a `break`), **no** `++` **or** `--` (`n += 1`), a loop can have an `else` block, and any value can be a condition (`while items:` loops while the list is non-empty). The Java idiom `while ((line = reader.readLine()) != null)` has a direct Python equivalent in the **walrus operator**: `while (line := next_line()) != "":`.

FASTAPI RELEVANCE

Waiting and polling is the everyday use in backend code: `while not job_done():` with a short sleep, working through a queue of tasks until it is empty, following a `next_page` link until there is none, or reading a stream of chunks until it ends. Always give such loops a limit or a timeout: a request handler stuck in a `while` loop blocks the worker that is serving it.

🧰 The Building Blocks

Eight pieces. The first three are the core behaviour, the next four are the variations you will use, and the last one is the mistake to avoid.

BASIC

A counting loop

The condition is checked *before* every pass. While it is true the body runs; the moment it is false the loop ends and the program carries on after it. If it is false at the very start, the body never runs at all.

```
count = 1
while count <= 3:
    print("pass", count)
    count += 1
print("done, count is", count)

# Output:
# pass 1
# pass 2
# pass 3
# done, count is 4
```

CHANGE

Something inside must change

A `while` loop only ends if the condition can become false. Whatever the condition depends on has to be updated inside the body, here by `n -= 1`.

```
n = 3
while n > 0:
    print(n)
    n -= 1
print("liftoff")

# Output:
# 3
# 2
# 1
# liftoff
```

TRUTHINESS

A collection as the condition

Any value can be a condition: an empty list, string, dictionary or `0` counts as false, anything else as true. `while stack:` means "while there is anything left".

```
stack = ["a", "b", "c"]
while stack:
    print(stack.pop())
print("empty now:", stack)

# Output:
# c
# b
# a
# empty now: []
```

AND / OR

A compound condition

The condition can combine tests with `and`, `or` and `not`. This loop stops when *either* the attempts run out *or* the thing is found, with no `break` needed.

```
tries = 0
found = False
while tries < 5 and not found:
    tries += 1
    found = (tries == 3)
print("tries:", tries, "found:", found)

# Output:
# tries: 3 found: True
```

WHILE TRUE

Loop until something inside says stop

`while True:` has no condition of its own, so it needs a `break` inside (see the `break` page). Use it when the stopping test fits better in the middle of the body than at the top.

```
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: quit
```

ELSE

The loop's else block

A `while` loop may have an `else` that runs when the condition becomes false. If the loop is left by a `break`, the `else` is skipped.

```
tries = 0
while tries < 3:
    tries += 1
    print("try", tries)
else:
    print("condition became False")

# Output:
# try 1
# try 2
# try 3
# condition became False
```

WALRUS

Assigning inside the condition

The *walrus operator* `:=` (Python 3.8 and later) assigns a value and tests it in one go, which suits "read until the end" loops. A plain `=` in a condition is a `SyntaxError`.

```
lines = iter(["alpha", "beta", "", "never reached"])
while (line := next(lines)) != "":
    print("read:", line)
print("stopped at the empty line")

# Output:
# read: alpha
# read: beta
# stopped at the empty line
```

THE TRAP

A loop that never ends

If nothing in the body changes the condition, the loop runs forever. Press `Ctrl+C` in the terminal to stop a runaway program. This is the most common `while` mistake.

```
n = 3
while n > 0:
    print(n)        # n never changes: infinite loop!
```

🔍 How It Works, Step by Step

1. Python evaluates the condition. If it is false, the loop is over and execution continues after the loop (after its `else` block, if there is one).

2. If it is true, the whole indented body runs once.
3. Control goes back to step 1 and the condition is evaluated again with whatever values the body has changed.
4. A `break` leaves at once, skipping the `else`. A `continue` jumps back to step 1 without finishing the body.

When to reach for it: when you cannot say in advance how many passes you need: reading input until it is valid, searching until a limit is reached, shrinking a range, working through a queue that may grow. If you already have a collection or a count to go through, a `for` loop is usually simpler and cannot get stuck.

🔬 In Context

Seven worked examples, from taking a number apart to a binary search. The last one shows a trap with floating-point numbers.

Example 1: Taking a Number Apart: Sum of Digits

There is no way to know how many digits a number has until you take them off, which is a natural fit for `while`. `n % 10` is the last digit and `n //= 10` removes it. The loop ends when nothing is left.

```
n = 4821
total = 0

while n > 0:
    digit = n % 10
    total += digit
    n //= 10

print("sum of digits:", total)

# Output:
# sum of digits: 15
```

Example 2: Asking Again Until the Input Is Valid

Ask once before the loop, then keep asking while the answer is bad. The key line is the second `input()` inside the loop: without it the condition checks the same bad value forever.

```
age = input("Enter your age: ")
while not age.isdigit():
    print("Digits only, please")
    age = input("Enter your age: ")

print("Age recorded:", int(age))

# Sample run:
# Enter your age: abc
# Digits only, please
# Enter your age:
# Digits only, please
# Enter your age: 37
# Age recorded: 37
```

Example 3: The Guessing Game With a Condition-Based Limit

The same game as on the `break` page, but the attempt limit lives in the loop condition, and the `else` reports the failure. A correct guess still needs a `break` to leave early.

```
import random

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

while attempts < 5:
    guess = int(input("Guess a number from 1 to 10: "))
    attempts += 1
    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 a number from 1 to 10: 3
# 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 attempts!
```

Example 4: Working Through a Queue That Can Grow

The body changes the thing the condition tests. Here finishing the `report` task adds a new task to the queue, so the number of passes was not known when the loop started. A `for` loop over a list that is being changed would be fragile; `while queue:` is not.

```
queue = ["email", "report", "backup"]

while queue:
    task = queue.pop(0)
    print("doing", task)
    if task == "report":
        queue.append("report-review")

print("all done")

# Output:
# doing email
# doing report
# doing backup
# doing report-review
# all done
```

Example 5: Until a Quantity Is Small Enough

How many halvings does it take to get from 100 down to 1 or less? You cannot compute that by counting items, so the loop runs until the condition fails and the counter records how long it took.

```
amount = 100.0
days = 0
```

```
while amount > 1:
    amount /= 2
    days += 1

print(f"after {days} days only {amount} is left")

# Output:
# after 7 days only 0.78125 is left
```

Example 6: Binary Search: Shrinking a Range

The loop condition `low <= high` says "there is still somewhere to look". Each pass halves the range, so 100 numbers need at most 7 guesses. The `else` runs only when the range has run out without finding it.

```
target = 73
low, high = 1, 100
guesses = 0

while low <= high:
    mid = (low + high) // 2
    guesses += 1
    if mid == target:
        print(f"found {target} in {guesses} guesses")
        break
    elif mid < target:
        low = mid + 1
    else:
        high = mid - 1
else:
    print("not found")

# Output:
# found 73 in 6 guesses
```

Example 7: A Float Trap: Never Test a Float for Equality

Adding `0.1` ten times does not give exactly `1.0`, so a loop that waits for `x != 1.0` to become false steps over it and keeps going. The `steps` limit is only there to stop the demonstration. Use `<` or `<=` for floats, or count with whole numbers.

```
x = 0.0
steps = 0

while x != 1.0 and steps < 15:
    x += 0.1
    steps += 1

print("steps:", steps, "x:", x)

# the safe version
x = 0.0
while x < 0.99:
    x += 0.1
print("stopped at", round(x, 1))
```

```
# Output:
# steps: 15 x: 1.5000000000000002
# stopped at 1.0
```

📋 Quick Reference

WHAT YOU WANT	WRITE	NOTES
Repeat while a condition is true	`while condition:`	*Checked before every pass, so the body may run zero times.*
Loop until something inside says stop	`while True:` + `break`	*Python has no `do-while`; this is the idiom.*
Keep going while a collection has items	`while items:`	*Empty list, string, dict, set and `0` are false.*
Two reasons to stop	`while a and not b:`	*Or use a `break` for the second reason.*
Count up or down	`n += 1` / `n -= 1`	*No `++` or `--` in Python.*
Assign and test in one step	`while (x := read()) != "":`	*Walrus operator, Python 3.8+. Plain `=` is a SyntaxError here.*
Run something when the condition fails	`while ...: ... else:`	*Skipped if the loop ended with `break`.*
Skip to the next pass	`continue`	*Update the loop variable before it, or the loop may never end.*
Stop a runaway loop	`Ctrl+C`	*In the terminal; raises `KeyboardInterrupt`.*
Known number of passes	`for i in range(n):`	*Usually simpler and cannot get stuck.*

⚠ Gotchas for Java / JavaScript Programmers:

A `while` loop is only as good as what changes inside it: *if the condition depends on a variable, something in the body must update that variable, or the loop runs forever. This is the number-one `while` bug in every language, but it is easy to hit when you first translate "keep checking" into Python.*

The update line must be *inside* the loop: *indentation is the block in Python. If `n -= 1` is indented one level too little, it sits after the loop, and the loop body never changes `n`. There are no braces to show you which block a line belongs to.*

There is no `do-while`: *when you need "run once, then test", write `while True:` and put an `if ...: break` where the test belongs, or ask once before the loop and again at the end of the body (as in Example 2).*

No `++` or `--`: *`count++` is a `SyntaxError`. Write `count += 1`. And `count += 1` is a statement, not an expression, so it cannot go inside the condition; use `:=` if you need to assign there.*

Never compare floats with `==` or `!=` in a loop condition: *`0.1` added ten times is `0.9999999999999999`, not `1.0`, so the loop in Example 7 steps right over its stopping value. Use `<` or `<=`, or count whole numbers.*

Truthiness works differently from Java and JavaScript: *`while count:` stops when `count` reaches `0`, and `while text:` stops on an empty string. That is handy for `while items:`, but if a legitimate value can be `0` or `""`, test explicitly (`is not None`, `!= ""`).*

The `else` belongs to the loop, not to an `if`: *it runs when the condition becomes false, and is skipped when the loop is left by a `break`. Java and JavaScript have nothing like it. Read it as "no break".*

You cannot put `=` in a condition: *`while x = 5:` is a `SyntaxError`, which protects you from the classic C-style typo, but `while x == 5:` and `while (x := f()) == 5:` are fine.*