

Loops: while, for, range, break, continue & Patterns

Repeat work without repeating code. Learn while and for loops, range(), enumerate(), zip(), break, continue, the loop else clause, nested loops, accumulation, search and counting patterns, and common loop bugs; with an eye on when to replace loops with vectorised pandas/NumPy.

● Python classes 3-4

✦ Python 3.13

Beginner

🕒 110 min

By the end of this lesson you can

- Write while and for loops and choose between them
- Use range(), enumerate() and zip() for index, value and parallel iteration
- Control loops with break, continue and the loop-else clause; avoid infinite loops
- Apply accumulator, counter, search, nested-loop and pattern-printing recipes
- Recognise when a loop should be replaced by a built-in, comprehension or vectorised operation

Datasets used

superstore_sales.csv

Why loops?

A **loop** repeats a block of code. Processing 10,000 rows, trying each file in a folder, retrying a failed request, or checking each number in a list are all loops.

for vs while

for loop

- Iterates over a sequence (list, string, range, dict, file)
- Number of repeats is known from the data
- for city in cities:
- Safe: ends when the items run out

while loop

- Repeats while a condition is True
- Use when the number of repeats is unknown
- while balance > 0:
- Risk: infinite loop if the condition never becomes False

Two kinds of loop. for repeats over a known sequence; while repeats as long as a condition stays true.

The while loop

Python

```
count = 1
while count ≤ 5:
    print("Count:", count)
    count += 1           # without this line the loop would never end!
print("Done")
```

OUTPUT

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Done
```

Every while loop needs: **(1)** a starting value, **(2)** a condition, **(3)** a change inside the loop that eventually makes the condition False.

Python

```
# Compound growth: how many years until 1,00,000 doubles at 8%?
amount, years = 100000, 0
while amount < 200000:
    amount *= 1.08
    years += 1
print(f"{years} years → {amount:,.0f}")          # the 'rule of 72': 72 / 8 = 9 years
```

OUTPUT

10 years → 215,892

while True with break

Use an "infinite" loop that you leave deliberately when the input is right:

Python

```
attempts = ["abc", "-4", "12"]          # simulated user answers
i = 0
while True:
    answer = attempts[i]; i += 1
    if answer.isdigit() and int(answer) > 0:
        print("Accepted:", answer); break
    print("Invalid:", answer)
```

OUTPUT

Invalid: abc
Invalid: -4
Accepted: 12

The for loop

for item in sequence: takes each item in turn.

Python

```
for city in ["Pune", "Delhi", "Kochi"]:
    print(city.upper())
for ch in "SQL":
    print(ch, end=" ")
print()
```

OUTPUT

PUNE
DELHI
KOCHI
S Q L

range()

`range(stop)` , `range(start, stop)` , `range(start, stop, step)` produce integers (stop excluded). It is lazy: `range(1_000_000)` does not store a million numbers.

Python

```
print(list(range(5)), list(range(2, 8)), list(range(10, 0, -3)), list(range(0, 1)))  
total = 0  
for n in range(1, 101): total += n  
print("Sum 1..100 =", total, "| check:", sum(range(1, 101)), "| formula:", 100 * 101 // 2)
```

OUTPUT

```
[0, 1, 2, 3, 4] [2, 3, 4, 5, 6, 7] [10, 7, 4, 1] [0]  
Sum 1..100 = 5050 | check: 5050 | formula: 5050
```

enumerate() and zip()

Python

```
products = ["Pen", "Notebook", "Stapler"]  
prices = [10, 45, 180]  
for i, p in enumerate(products, start=1):           # index and value together  
    print(i, p)  
for p, price in zip(products, prices):              # walk two lists in parallel  
    print(f"{p:<10}-{price:>5}")  
print(dict(zip(products, prices)))                  # build a dictionary from two lists
```

OUTPUT

```
1 Pen  
2 Notebook  
3 Stapler  
Pen          10  
Notebook     45  
Stapler      180  
{'Pen': 10, 'Notebook': 45, 'Stapler': 180}
```

Avoid `for i in range(len(x))` when you only need the items; use `for item in x` or `enumerate`.

break, continue, else

Python

```
for n in range(1, 20):
    if n % 3 != 0: continue          # skip numbers not divisible by 3
    if n > 12: break                # leave the loop completely
    print(n, end=" ")
print()
for n in range(2, 6):
    if n == 9: break
else:                               # runs only if the loop finished WITHOUT break
    print("loop completed with no break")
```

OUTPUT

3 6 9 12

loop completed with no break

Keyword	Effect
<code>break</code>	Exit the nearest loop immediately
<code>continue</code>	Skip to the next iteration
<code>else</code> on a loop	Runs if the loop ended normally (no <code>break</code>): handy for "search and report not found"
<code>pass</code>	Do nothing (placeholder)

Loop patterns

Accumulate (sum / product / build a list)

Python

```
orders = [1200, 450, 3000, 780, 2200]
total, large = 0, []
for amount in orders:
    total += amount
    if amount >= 1000: large.append(amount)
print(total, large, total / len(orders))
```

OUTPUT

7630 [1200, 3000, 2200] 1526.0

Count and find the maximum

Python

```
sales = [("North", 120), ("South", 95), ("East", 140), ("West", 88)]
best_region, best_value = None, float("-inf")
count_above_100 = 0
for region, value in sales:
    if value > best_value: best_region, best_value = region, value
    if value > 100: count_above_100 += 1
print(best_region, best_value, count_above_100)
```

OUTPUT

East 140 2

(`max(sales, key=lambda x: x[1])` does the same in one line, Lesson 10.)

Search with early exit

Python

```
def find_first(items, target):
    for i, x in enumerate(items):
        if x == target: return i          # stop at the first match
    return -1
print(find_first(["a", "b", "c"], "c"), find_first(["a", "b"], "z"))
```

OUTPUT

2 -1

Nested loops

Python

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i * j, end="\t")
    print()
```

OUTPUT

1	2	3
2	4	6
3	6	9

Python

```
# Pattern: right triangle of stars
for row in range(1, 6): print("*" * row)
```

OUTPUT

```
*
**
***
****
*****
```

Nested loops multiply the work: $1,000 \times 1,000 = 1,000,000$ iterations. Be careful with big data.

Reading until a sentinel

Python

```
readings = iter([12, 15, 11, -1, 99])          # -1 marks the end
total = 0
for r in readings:
    if r == -1: break
    total += r
print("Total before sentinel:", total)
```

OUTPUT

```
Total before sentinel: 38
```

Data examples

Looping over rows (and why it is slow)

Python

```
import pandas as pd, time
s = pd.read_csv("superstore_sales.csv")
t0 = time.perf_counter()
loss_lines = 0
for profit in s["profit"]:
    if profit < 0: loss_lines += 1
t1 = time.perf_counter()
vec = (s["profit"] < 0).sum()
t2 = time.perf_counter()
print(f"loop: {loss_lines} loss-making lines in {1000 * (t1 - t0):.2f} ms | vectorised: {vec} in {1000 * (t2 - t1):.2f} ms")
```

OUTPUT

loop: 1682 loss-making lines in 0.84 ms | vectorised: 1682 in 0.34 ms

Same answer; the vectorised version is usually many times faster and shorter. **Rule:** in pandas/NumPy avoid Python loops over rows; use column operations, `groupby`, `np.where`. Use loops for files, APIs, orchestration, and small collections.

Summarising with a loop

Python

```
totals = {}
for region, category, sales in zip(s["region"][:2000], s["category"][:2000], s["sales"][:2000]):
    key = (region, category)
    totals[key] = totals.get(key, 0) + sales
for k in sorted(totals)[::-1]: print(k, f"{totals[k]:.0f}")
```

OUTPUT

```
('East', 'Furniture') 2,148,878
('East', 'Office Supplies') 717,588
('East', 'Technology') 10,862,196
('North', 'Furniture') 2,782,383
('North', 'Office Supplies') 645,662
('North', 'Technology') 13,766,572
```

This is a hand-made `GROUP BY` : you now see what `groupby` does behind the scenes.

Common loop bugs

Bug	Symptom	Fix
Missing update in <code>while</code>	Runs forever (press ■ to interrupt)	Change the condition variable inside the loop
Off-by-one with <code>range</code>	Misses the last item	Remember stop is excluded: <code>range(1, n + 1)</code>
Modifying a list while iterating over it	Skipped items	Iterate over a copy <code>for x in items[:]</code> or build a new list
<code>for i in range(len(a))</code> then using a wrong index	<code>IndexError</code>	Use <code>enumerate / zip</code>
Re-using a loop variable after the loop	Holds the last value	Avoid relying on it
Resetting the accumulator inside the loop	Wrong totals	Initialise before the loop
String <code>+=</code> in a loop	Slow for big text	<code>"".join(parts)</code>

TIP

Debug with print or a debugger

Print the loop variable and key values each iteration (`print(i, value, total)`), or use the debugger in VS Code/Jupyter (`%debug`). Test the loop on a tiny example by hand first.

Summary

KEY TAKEAWAYS

- `while` repeats while a condition holds (make sure something changes!); `for` iterates over any sequence.
- `range(start, stop, step)` generates integers, stop excluded; `enumerate` gives index and value; `zip` iterates in parallel.
- `break` exits, `continue` skips to the next item, `loop else` runs when no `break` happened.
- Patterns: accumulator, counter, max/min tracking, search with early exit, nested loops, sentinel.
- For tables of data prefer **vectorised** pandas/NumPy over row loops; use comprehensions and built-ins for small collections.

Quick quiz

6 questions · tap an answer to check it

1. What does `list(range(2, 10, 3))` give?

A [2, 5, 8, 11]

B [2, 5, 8]

C [3, 6, 9]

D [2, 10]

Answer: B. Start 2, step 3, stop 10 excluded.

2. What does `continue` do inside a loop?

- A Ends the loop
- B Skips the rest of this iteration and moves to the next
- C Restarts the program
- D Pauses the loop

Answer: B. `break` ends the loop; `continue` skips one iteration.

3. When does a loop's `else` block run?

- A When the loop has an error
- B When the loop finishes without hitting `break`
- C Before the loop
- D Every iteration

Answer: B. It is useful for "not found" logic after a search loop.

4. Why is this loop infinite? `n = 5 then while n > 0: print(n)`

- A `print` is wrong
- B `n` is never changed inside the loop
- C 5 is too large
- D `range` is missing

Answer: B. The condition never becomes False.

5. What is a better way to count rows with negative profit in pandas?

A A for loop over rows

B `(df["profit"] < 0).sum()`

C A while loop

D Nested loops

Answer: B. Vectorised operations are shorter and faster.

6. Which function walks two lists at the same time?

A `range`

B `zip`

C `sorted`

D `len`

Answer: B. `zip(a, b)` pairs items by position.

EXERCISE Loop problems

(1) Print the multiplication table of 7 up to 7×10 . (2) Given a list of order values, print the first one above 5,000 and its position, or "none". (3) Count vowels in a sentence. (4) Print the first 10 Fibonacci numbers. (5) FizzBuzz for 1..30.

Show solution (worked answer)

Python

```
for i in range(1, 11): print(f"7 x {i} = {7 * i}")
vals = [900, 4200, 6100, 800]
for i, v in enumerate(vals):
    if v > 5000: print(v, "at", i); break
else: print("none")
print(sum(ch in "aeiou" for ch in "data engineering is fun".lower()))
a, b = 0, 1
for _ in range(10): print(a, end=" "); a, b = b, a + b
print()
for n in range(1, 31): print("FizzBuzz" if n % 15 == 0 else "Fizz" if n % 3 == 0 else
"Buzz" if n % 5 == 0 else n, end=" ")
```

INTERVIEW CORNER Questions you may be asked

Q: FizzBuzz, reverse a string, print Fibonacci: why do interviewers ask these? They test loops, conditions, modulo, variable swapping and clear thinking, and whether you handle edge cases.

Q: for vs while? `for` iterates over a known sequence; `while` loops until a condition changes, for unknown counts such as retries or reading until a sentinel.

Q: How would you speed up a slow loop over a DataFrame? Vectorise (column operations, `np.where`, `groupby`), use built-ins or `apply` only when needed, or push the work into SQL.

Q: What does `for i in range(len(lst))` signal? Often a less Pythonic style; prefer iterating over items, `enumerate` or `zip` unless you need indexes to modify items.

Practise more online

Free, trusted places to test yourself on this topic. Links open the original site in a new tab.