Python for Loops

Introduction

Programs often need to repeat an operation.

You may need to:

- Print every item in a list
- Process every student in a class
- Calculate the total of numbers
- Check every character in a string
- Generate a sequence of values
- Search through data
- Process every resource on a website
- Repeat an operation a known number of times

Python's `for` loop is designed for this kind of repetition.

The core idea is:

> **Take one item from a sequence or iterable, run the code for that item, then move to the next item.**

Basic syntax:

```
for item in iterable:
    # code to execute
```

Example:

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

for name in names:
    print(name)
```

Output:

```
Ali
Sara
Ahmed
```

The loop automatically moves from one item to the next.

---

1. Why Do We Need for Loops?

Suppose you have five names:

```
names = ["Ali", "Sara", "Ahmed", "Hina", "Usman"]
```

Without a loop:

```
print(names[0])
print(names[1])
print(names[2])
print(names[3])
print(names[4])
```

This works, but it is repetitive and doesn't scale.

A `for` loop expresses the actual intention:

> For every name in the list, print the name.

```
for name in names:
    print(name)
```

Now if the list contains 5 items, 500 items, or 50,000 items, the same logic can be used.

---

2. The Mental Model

Think of a `for` loop as a worker processing a queue of items.

Suppose:

```
numbers = [10, 20, 30]
```

Python conceptually does:

```
Take 10 → run the loop body
Take 20 → run the loop body
Take 30 → run the loop body
No items left → stop
```

For:

```
for number in numbers:
    print(number)
```

the variable `number` refers to the current item.

The important idea is:

> **The loop variable changes automatically as Python moves through the iterable.**

Unlike a basic `while` loop, you usually don't need to manually write:

```
index += 1
```

---

3. Basic Syntax

```
for variable in iterable:
    statement
```

Example:

```
for number in [1, 2, 3]:
    print(number)
```

There are three important parts.

`for`

Starts the loop.

Loop variable

```
number
```

Stores the current item.

Iterable

```
[1, 2, 3]
```

Provides the items that the loop processes.

---

4. What Is an Iterable?

An **iterable** is something Python can go through one item at a time.

Common iterables include:

- Lists
- Tuples
- Strings
- Dictionaries
- Sets
- Ranges
- Many other Python objects

Examples:

```
for item in [1, 2, 3]:
    print(item)
```

```
for item in (1, 2, 3):
    print(item)
```

```
for character in "Python":
    print(character)
```

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

The `for` loop doesn't require the iterable to be a list.

---

5. Looping Through a List

This is one of the most common uses.

```
languages = ["Python", "JavaScript", "Dart"]

for language in languages:
    print(language)
```

Output:

```
Python
JavaScript
Dart
```

The loop executes once for every item.

---

6. Looping Through a Tuple

```
coordinates = (10, 20, 30)

for coordinate in coordinates:
    print(coordinate)
```

Output:

```
10
20
30
```

The same principle works because tuples are iterable.

---

7. Looping Through a String

Strings are also iterable.

```
word = "Python"

for character in word:
    print(character)
```

Output:

```
P
y
t
h
o
n
```

Python processes one character at a time.

This is useful for:

- character validation
- counting letters
- searching text
- checking patterns
- text processing

---

8. Looping Through a Set

```
skills = {"Python", "JavaScript", "React"}

for skill in skills:
    print(skill)
```

The loop processes each element.

Remember that sets do not guarantee the same ordering you would expect from a list.

Therefore, don't write logic that depends on a particular set iteration order unless you have deliberately handled ordering.

---

9. Looping Through a Dictionary

Dictionaries behave slightly differently.

Suppose:

```
student = {
    "name": "Hafsa",
    "age": 25,
    "skill": "Python"
}
```

If you write:

```
for item in student:
    print(item)
```

Python iterates over the **keys**.

Output:

```
name
age
skill
```

---

10. Dictionary Values

To iterate over values:

```
for value in student.values():
    print(value)
```

Output:

```
Hafsa
25
Python
```

---

11. Dictionary Keys and Values

Use `.items()`:

```
for key, value in student.items():
    print(key, ":", value)
```

Output:

```
name : Hafsa
age : 25
skill : Python
```

This pattern is extremely common when working with dictionaries and API data.

---

12. `range()`

One of the most important tools used with `for` loops is `range()`.

Example:

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

Output:

```
0
1
2
3
4
```

Notice that `5` is not included.

`range(5)` produces values starting from `0` up to, but not including, `5`.

---

13. Why Does range Start at 0?

Python uses zero-based indexing in many places.

So:

```
range(5)
```

represents:

```
0, 1, 2, 3, 4
```

There are five values.

Think of it as:

```
Start = 0
Stop before = 5
```

---

14. range(start, stop)

You can specify a starting point.

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

Output:

```
1
2
3
4
5
```

The first argument is the starting value.

The second argument is the stopping boundary, which is excluded.

So:

```
range(1, 6)
```

means:

```
1, 2, 3, 4, 5
```

---

15. range(start, stop, step)

You can also specify how much the value should change.

```
for number in range(0, 11, 2):
    print(number)
```

Output:

```
0
2
4
6
8
10
```

Here:

```
start = 0
stop = 11
step = 2
```

---

16. Counting Down With range()

Use a negative step.

```
for number in range(5, 0, -1):
    print(number)
```

Output:

```
5
4
3
2
1
```

Notice:

```
range(5, 0, -1)
```

stops before `0`.

---

17. Understanding range()

The three common forms are:

```
range(stop)
```

```
range(start, stop)
```

```
range(start, stop, step)
```

Examples:

```
range(5)
```

```
0 1 2 3 4
```

```
range(2, 6)
```

```
2 3 4 5
```

```
range(2, 10, 2)
```

```
2 4 6 8
```

```
range(10, 0, -2)
```

```
10 8 6 4 2
```

---

18. A for Loop With a Counter

You can use `range()` when you need a known number of repetitions.

```
for i in range(5):
    print("Hello")
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

The variable `i` receives:

```
0
1
2
3
4
```

Even though you don't use it.

If the loop variable itself isn't needed, `_` is commonly used:

```
for _ in range(5):
    print("Hello")
```

This communicates:

> I need five repetitions, but I don't care about the current value.

---

19. for vs while

You have now learned both `while` and `for`.

The difference is important.

`for`

Usually answers:

> **For every item in this iterable, do something.**

Example:

```
for name in names:
    print(name)
```

`while`

Usually answers:

> **Keep doing something while this condition remains true.**

Example:

```
while attempts < 3:
    attempts += 1
```

A useful rule:

Situation	**Usually use**
Process every item in a list	`for`
Loop through a string	`for`
Repeat a known number of times	`for`
Iterate through a range	`for`
Keep asking until valid input	`while`
Repeat until an event happens	`while`

Repeat while a state is true	`while`

These are guidelines rather than strict rules.

---

20. Processing Every Item

Suppose you have:

```
prices = [100, 250, 500, 150]
```

You can calculate the total:

```
total = 0

for price in prices:
    total += price

print(total)
```

Output:

```
1000
```

This is a classic **accumulator pattern**.

---

21. The Accumulator Pattern

An accumulator stores a result that changes during the loop.

```
total = 0

for number in [10, 20, 30]:
    total += number
```

Execution:

```
total = 0
0 + 10 = 10
10 + 20 = 30
30 + 30 = 60
```

Final result:

```
60
```

This pattern is useful for:

- totals
- scores
- averages
- counts
- financial calculations
- statistics

---

22. Counting Items

Suppose you want to count numbers greater than 10.

```
numbers = [5, 15, 20, 7, 30]

count = 0

for number in numbers:
    if number > 10:
        count += 1

print(count)
```

Output:

```
3
```

The loop processes every number.

The `if` determines whether that number should be counted.

---

23. for Loop With if

Loops and conditions commonly work together.

```
numbers = [1, 2, 3, 4, 5, 6]

for number in numbers:
    if number % 2 == 0:
        print(number)
```

Output:

```
2
4
6
```

The loop handles repetition.

The condition handles decision-making.

---

24. Finding a Value

```
names = ["Ali", "Sara", "Ahmed", "Hina"]

for name in names:
    if name == "Ahmed":
        print("Found Ahmed")
```

Output:

```
Found Ahmed
```

Later, you'll learn more efficient and Pythonic ways to perform many search operations, but this is an important basic pattern.

---

25. `break`

`break` immediately stops the loop.

Example:

```
numbers = [10, 20, 30, 40, 50]

for number in numbers:
    if number == 30:
        break

    print(number)
```

Output:

```
10
20
```

When Python reaches:

```
number == 30
```

the `break` ends the entire loop.

---

26. Why Use break?

Suppose you're searching for something.

Once you've found it, there may be no reason to keep checking.

```
numbers = [10, 20, 30, 40, 50]

target = 30

for number in numbers:
    if number == target:
        print("Found")
        break
```

This avoids unnecessary iterations after the answer is found.

---

27. `continue`

`continue` skips the current iteration and moves to the next one.

Example:

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

    print(number)
```

Output:

```
1
2
4
5
```

When `number` is 3, Python skips the `print()` statement.

---

28. break vs continue

Remember:

```
break
↓
```

```
Stop the entire loop
```

```
continue
↓
Skip current iteration
↓
Continue with next iteration
```

Example:

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

    if number == 5:
        break

    print(number)
```

Output:

```
1
2
4
```

---

29. Nested for Loops

A `for` loop can contain another `for` loop.

```
for row in range(3):
    for column in range(3):
        print(row, column)
```

Output:

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

The inner loop completes all its iterations for every iteration of the outer loop.

---

30. Nested Loop Mental Model

Think:

```
Outer loop starts
    ↓
Inner loop runs completely
    ↓
Outer loop moves to next value
    ↓
Inner loop runs completely again
```

For:

```
for row in range(2):
    for column in range(3):
        print(row, column)
```

the execution is:

```
row = 0
    column = 0
    column = 1
    column = 2

row = 1
    column = 0
    column = 1
    column = 2
```

Nested loops are useful for:

- grids
- tables
- matrices
- game boards
- combinations
- patterns

---

31. Printing Patterns

Nested loops can generate patterns.

```
for row in range(4):
    for column in range(4):
        print("*", end=" ")

    print()
```

Output:

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

The outer loop controls rows.

The inner loop controls columns.

---

32. Another Pattern

```
for row in range(1, 5):
    for column in range(row):
        print("*", end=" ")

    print()
```

Output:

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

This demonstrates how one loop can control the number of iterations of another.

---

33. `enumerate()`

A common beginner problem is wanting both:

- the item
- its position/index

Instead of manually managing an index:

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

for index in range(len(names)):
    print(index, names[index])
```

Python provides `enumerate()`:

```
for index, name in enumerate(names):
    print(index, name)
```

Output:

```
0 Ali
1 Sara
2 Ahmed
```

This is generally clearer.

---

34. Starting enumerate() From 1

By default, the index starts at 0.

You can change it:

```
for index, name in enumerate(names, start=1):
    print(index, name)
```

Output:

```
1 Ali
2 Sara
3 Ahmed
```

This is useful when displaying numbered lists to users.

---

35. Looping Through Two Lists With zip()

Suppose you have:

```
names = ["Ali", "Sara", "Ahmed"]
scores = [80, 92, 75]
```

You can combine them:

```
for name, score in zip(names, scores):
    print(name, score)
```

Output:

```
Ali 80
Sara 92
Ahmed 75
```

This is useful when two sequences contain related data.

---

36. `for` Loops With Dictionaries

Consider:

```
student = {
    "name": "Hafsa",
    "course": "Python",
    "level": "Beginner"
}
```

You can display everything:

```
for key, value in student.items():
    print(f"{key}: {value}")
```

Output:

```
name: Hafsa
course: Python
level: Beginner
```

This pattern becomes especially important when processing JSON and API responses.

---

37. Processing API-Like Data

Imagine an API returns:

```
resources = [
    {"title": "Python Variables", "category": "Python"},
    {"title": "Python Conditions", "category": "Python"},
    {"title": "Python Loops", "category": "Python"}
]
```

You can process each resource:

```
for resource in resources:
    print(resource["title"])
```

Output:

```
Python Variables
Python Conditions
Python Loops
```

This is similar to what real applications do when rendering or processing API data.

---

38. A Haas.dev Example

Imagine haas.dev has a collection of learning resources:

```
resources = [
    "Python Variables",
    "Python Conditions",
    "Python Loops",
    "Python Functions"
]
```

You could generate a simple resource list:

```
for resource in resources:
    print("Learning:", resource)
```

Output:

```
Learning: Python Variables
Learning: Python Conditions
Learning: Python Loops
Learning: Python Functions
```

In a real application, the loop might instead:

- validate resource data
- generate metadata
- process PDF information
- filter categories
- build search results
- render resource cards
- update a database

The syntax is simple, but the same repetition pattern appears inside much larger systems.

---

39. Searching With break

Suppose you want to find a resource:

```
resources = [
    "Python Variables",
    "Python Conditions",
    "Python Loops",
    "Python Functions"
]

target = "Python Loops"

for resource in resources:
    if resource == target:
        print("Resource found")
        break
```

Once the resource is found, the loop stops.

---

40. Filtering Data

A loop can select only the data you need.

```
numbers = [5, 12, 8, 20, 3, 15]

for number in numbers:
    if number >= 10:
        print(number)
```

Output:

```
12
20
15
```

The condition acts as a filter.

Later, Python's list comprehensions provide a shorter way to express many filtering operations.

---

41. Transforming Data

Loops can also transform each item.

```
numbers = [1, 2, 3, 4]

for number in numbers:
```

```
    squared = number ** 2
    print(squared)
```

Output:

```
1
4
9
16
```

The input remains unchanged, while a new value is calculated for each item.

---

42. Building a New List

You can store transformed values.

```
numbers = [1, 2, 3, 4]

squares = []

for number in numbers:
    squares.append(number ** 2)

print(squares)
```

Output:

```
[1, 4, 9, 16]
```

This is a foundational pattern.

Later, you'll learn:

```
squares = [number ** 2 for number in numbers]
```

This is called a **list comprehension**.

---

43. Looping Through Nested Lists

Consider:

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

You can use nested loops:

```
for row in matrix:
    for number in row:
        print(number)
```

Output:

```
1
2
3
4
5
6
7
8
9
```

The outer loop gets each row.

The inner loop gets each value inside that row.

---

44. Looping Through Characters

You can count a particular character.

```
word = "banana"
count = 0

for character in word:
    if character == "a":
        count += 1

print(count)
```

Output:

```
3
```

This introduces an important algorithmic pattern:

> Examine each element and update a result when a condition is satisfied.

---

45. Finding the Largest Number

You can find a maximum manually using a loop.

```
numbers = [10, 50, 20, 80, 30]

largest = numbers[0]

for number in numbers:
    if number > largest:
        largest = number

print(largest)
```

Output:

```
80
```

Python also provides:

```
max(numbers)
```

but understanding the loop version teaches you how the algorithm works.

---

46. Finding the Smallest Number

```
numbers = [10, 50, 20, 80, 30]

smallest = numbers[0]

for number in numbers:
    if number < smallest:
        smallest = number

print(smallest)
```

Output:

```
10
```

This is another example of maintaining state while iterating.

---

47. Calculating an Average

```
scores = [80, 90, 70, 100]

total = 0

for score in scores:
```

```
    total += score

average = total / len(scores)

print(average)
```

Output:

```
85.0
```

The loop calculates the total.

`len()` tells us how many scores exist.

---

48. Looping Over User Input

A `for` loop can work with multiple predefined inputs.

For example:

```
names = []

for _ in range(3):
    name = input("Enter a name: ")
    names.append(name)

print(names)
```

The loop asks three times.

This is useful when the number of repetitions is known.

---

49. A for Loop Does Not Mean "Numbers"

A common beginner misconception is:

> `for` loops are only for numbers.

Not true.

You can loop through:

```
for name in names:
```

```
for character in word:
```

```
for item in dictionary:
```

```
for resource in resources:
```

```
for row in matrix:
```

The real idea is:

> **Iterate over an iterable.**

---

50. Loop Variable Naming

You can technically use any valid variable name:

```
for x in numbers:
```

But meaningful names are better:

```
for number in numbers:
```

For names:

```
for name in names:
```

For products:

```
for product in products:
```

For resources:

```
for resource in resources:
```

Good naming makes code easier to understand.

---

51. Don't Modify a Collection Carelessly

Be careful when changing a list while iterating through it.

For example, this can produce confusing behavior:

```
numbers = [1, 2, 3, 4, 5]

for number in numbers:
    if number % 2 == 0:
        numbers.remove(number)
```

The list is changing while the loop is moving through it.

A safer approach is to build a new list:

```
numbers = [1, 2, 3, 4, 5]

remaining = []
```

```
for number in numbers:
    if number % 2 != 0:
        remaining.append(number)

print(remaining)
```

Output:

```
[1, 3, 5]
```

Later, list comprehensions can make this even cleaner.

---

52. `else` With for Loops

Python also allows an `else` block with a `for` loop.

```
numbers = [1, 2, 3]

for number in numbers:
    print(number)
else:
    print("Loop finished")
```

Output:

```
1
2
3
Loop finished
```

The `else` runs when the loop finishes normally.

---

53. for...else With Searching

This becomes particularly useful with `break`.

```
numbers = [10, 20, 30, 40]

target = 50

for number in numbers:
    if number == target:
        print("Found")
        break
```

```
else:
    print("Not found")
```

Output:

```
Not found
```

If the target exists:

```
target = 30
```

the output becomes:

```
Found
```

The `else` runs only if the loop completes without hitting `break`.

---

54. `range()` Does Not Create a List

Consider:

```
numbers = range(1_000_000)
```

`range()` represents a sequence of numbers without creating a giant list containing every number in memory.

You can iterate over it:

```
for number in range(1_000_000):
    ...
```

This is one reason `range()` is useful for large iteration counts.

---

55. Step Size and Algorithms

The `step` argument can control how you traverse a sequence.

```
for number in range(0, 100, 10):
    print(number)
```

Output:

```
0
10
20
30
40
```

```
50
60
70
80
90
```

The loop doesn't have to visit every integer.

---

56. Reverse Iteration

You can count backward:

```
for number in range(10, 0, -1):
    print(number)
```

Or reverse an existing sequence:

```
numbers = [1, 2, 3, 4, 5]

for number in reversed(numbers):
    print(number)
```

Output:

```
5
4
3
2
1
```

---

57. `enumerate()` vs Manual Indexing

Less clear:

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

for i in range(len(names)):
    print(i, names[i])
```

Usually clearer:

```
for i, name in enumerate(names):
    print(i, name)
```

Why?

Because `enumerate()` directly communicates:

> I need both the position and the item.

---

58. `zip()` and Parallel Iteration

Suppose:

```
students = ["Ali", "Sara", "Ahmed"]
scores = [80, 95, 75]
```

Use:

```
for student, score in zip(students, scores):
    print(f"{student}: {score}")
```

Output:

```
Ali: 80
Sara: 95
Ahmed: 75
```

This is much cleaner than manually accessing indexes.

---

59. Common Mistakes

Mistake 1: Forgetting the colon

Incorrect:

```
for number in numbers
    print(number)
```

Correct:

```
for number in numbers:
    print(number)
```

Mistake 2: Incorrect indentation

Incorrect:

```
for number in numbers:
print(number)
```

Correct:

```
for number in numbers:
    print(number)
```

Mistake 3: Confusing `range()` boundaries

This:

```
range(1, 5)
```

produces:

```
1 2 3 4
```

not:

```
1 2 3 4 5
```

Mistake 4: Using the wrong step

This:

```
range(5, 0)
```

doesn't count down because the default step is `+1`.

Use:

```
range(5, 0, -1)
```

Mistake 5: Modifying a list while iterating

This can cause skipped elements or unexpected behavior.

Use a separate collection or a carefully designed approach.

Mistake 6: Overusing indexes

If you only need the values:

```
for name in names:
```

is usually better than:

```
for i in range(len(names)):
```

---

60. Debugging for Loops

When a `for` loop doesn't behave as expected, ask:

1. What is the iterable?

```
for item in iterable:
```

2. What values does it contain?

3. How many iterations should happen?

4. What does the loop variable contain during each iteration?

5. Is there an `if` condition filtering some items?

6. Is `break` stopping the loop early?

7. Is `continue` skipping iterations?

8. Is the iterable being modified during the loop?

These questions usually reveal the problem.

---

61. Trace a for Loop Manually

Consider:

```
numbers = [2, 4, 6]

for number in numbers:
    print(number * 2)
```

Trace it:

Iteration	`number`	Expression	Output
1	2	`2 × 2`	4
2	4	`4 × 2`	8
3	6	`6 × 2`	12

Final output:

```
4
8
12
```

Tracing helps you understand exactly what the loop is doing.

---

62. for Loop Execution Flow

For:

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

think:

```
Get iterable
    ↓
Get next item
    ↓
Store it in item
    ↓
Run loop body
    ↓
Get next item
    ↓
Run loop body
    ↓
...
    ↓
No items left
    ↓
Exit loop
```

This is different from the condition-based mental model of a `while` loop.

---

63. while vs for: Deeper Mental Model

while

State-driven:

```
Is the condition still true?
        ↓
      YES
        ↓
     Work
        ↓
 Change state
```

```
        ↓
 Check again
```

for

Iterable-driven:

```
Are there more items?
       ↓
     YES
       ↓
Get next item
       ↓
    Do work
       ↓
Get next item
       ↓
...
       ↓
No items
       ↓
Stop
```

This distinction is one of the most important things to understand.

---

64. Real World Example: Processing Orders

Imagine an online store has orders:

```
orders = [
    {"id": 101, "total": 500},
    {"id": 102, "total": 1200},
    {"id": 103, "total": 750}
]
```

You can process each order:

```
for order in orders:
    print("Processing order:", order["id"])
```

Output:

```
Processing order: 101
Processing order: 102
Processing order: 103
```

A real application could use the same pattern to:

- calculate totals
- validate orders
- send notifications
- update inventory
- generate reports

---

65. Real World Example: Processing Resources

Suppose haas.dev has resource metadata:

```
resources = [
    {"title": "Python Variables", "category": "Python"},
    {"title": "Python Conditions", "category": "Python"},
    {"title": "JavaScript Functions", "category": "JavaScript"}
]
```

You can display Python resources:

```
for resource in resources:
    if resource["category"] == "Python":
        print(resource["title"])
```

Output:

```
Python Variables
Python Conditions
```

This combines:

- lists
- dictionaries
- loops
- conditions

These combinations are much closer to real programming than isolated syntax examples.

---

66. Real World Example: Student Scores

```
scores = [75, 82, 91, 68, 88]

total = 0
```

```
for score in scores:
    total += score

average = total / len(scores)

print("Average:", average)
```

Output:

```
Average: 80.8
```

You could extend this to classify scores:

```
for score in scores:
    if score >= 80:
        print(score, "Good")
    else:
        print(score, "Needs improvement")
```

---

67. Real World Example: Validation

Suppose you want to check whether every required field exists.

```
required_fields = ["name", "email", "password"]

user_data = {
    "name": "Hafsa",
    "email": "example@email.com",
    "password": "secret"
}

for field in required_fields:
    if field not in user_data:
        print("Missing:", field)
```

If nothing is printed, all required fields are present.

This type of iteration is common in form and API validation.

---

68. Real World Example: Permissions

Suppose a user has permissions:

```
permissions = ["read", "write"]

required = "delete"
```

```
for permission in permissions:
    if permission == required:
        print("Access granted")
        break
else:
    print("Access denied")
```

Output:

```
Access denied
```

The loop searches for the required permission.

---

69. Performance and Nested Loops

A single loop over `n` items usually performs approximately `n` iterations.

```
for item in items:
    ...
```

A nested loop can perform approximately:

```
n × m
```

iterations.

Example:

```
for i in range(100):
    for j in range(100):
        print(i, j)
```

That's 10,000 inner-loop executions.

This becomes important when you start studying algorithm complexity and Big O notation.

---

70. Avoid Unnecessary Work Inside Loops

Suppose you repeatedly calculate something that never changes.

Instead of:

```
for number in numbers:
    limit = len(numbers)
    print(number, limit)
```

calculate it once:

```
limit = len(numbers)

for number in numbers:
    print(number, limit)
```

The second version makes the intent clearer and avoids unnecessary repeated work.

As your programs become larger, this kind of thinking matters more.

---

71. Don't Use a Loop When Python Already Has the Operation

Sometimes beginners write:

```
numbers = [10, 20, 30]
total = 0

for number in numbers:
    total += number
```

This is excellent for learning.

But Python already provides:

```
sum(numbers)
```

Similarly:

```
max(numbers)
```

```
min(numbers)
```

```
len(numbers)
```

Learning the loop is still important because it teaches you the underlying logic.

But in production code, prefer clear built-in operations when they already express the desired operation.

---

72. `for` Loops and Functions

You can call functions inside a loop.

```
def greet(name):
    print("Hello", name)
```

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

for name in names:
    greet(name)
```

Output:

```
Hello Ali
Hello Sara
Hello Ahmed
```

This is an important step toward organizing larger programs.

---

73. `for` Loops and User Defined Functions

You can also process values returned by a function.

```
def square(number):
    return number ** 2

numbers = [1, 2, 3, 4]

for number in numbers:
    result = square(number)
    print(result)
```

The function handles one task.

The loop handles repetition.

This separation makes programs easier to maintain.

---

74. `for` Loops and Strings

You can perform useful text operations.

Example: count vowels.

```
text = "programming"

vowels = "aeiou"
count = 0

for character in text:
    if character in vowels:
        count += 1
```

```
print(count)
```

Output:

```
3
```

This demonstrates:

- string iteration
- conditions
- membership testing
- counters

---

75. Example: Reverse Thinking

You can use a loop to construct a reversed string:

```
word = "Python"
reversed_word = ""

for character in word:
    reversed_word = character + reversed_word

print(reversed_word)
```

Output:

```
nohtyP
```

Python also provides easier approaches such as slicing:

```
word[::-1]
```

But understanding the loop helps you understand the underlying process.

---

76. Example: Detecting a Duplicate

Suppose:

```
numbers = [1, 2, 3, 2]
```

A basic approach:

```
seen = []

for number in numbers:
```

```
    if number in seen:
        print("Duplicate:", number)
        break

    seen.append(number)
```

Output:

```
Duplicate: 2
```

This introduces an important algorithmic idea:

> Maintain information about what has already been processed.

---

77. `pass` Inside a for Loop

Sometimes you need a loop structure but don't want to perform an action yet.

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

`pass` does nothing.

It is mainly useful as a placeholder while developing code.

Don't confuse it with:

```
break
```

or:

```
continue
```

---

78. break, continue, and pass

`break`

Stops the loop.

```
for number in numbers:
    if number == 5:
        break
```

`continue`

Skips the current iteration.

```
for number in numbers:
    if number == 5:

```

```
        continue
```

`pass`

Does nothing.

```
for number in numbers:
    pass
```

Remember:

```
break    → leave
continue → skip
pass     → do nothing
```

---

79. Common Beginner Confusion: Loop Variable

Consider:

```
numbers = [10, 20, 30]

for number in numbers:
    print(number)
```

`number` isn't a special Python keyword.

You could write:

```
for value in numbers:
    print(value)
```

or:

```
for item in numbers:
    print(item)
```

The variable simply receives the current item.

---

80. Common Beginner Confusion: range() vs List

These are different:

```
numbers = [1, 2, 3, 4, 5]
```

and:

```
numbers = range(1, 6)
```

The list explicitly stores the values.

The range represents a numeric sequence that can be iterated over.

Both can be used:

```
for number in numbers:
    print(number)
```

but they are different Python objects.

---

81. A Complete Example: Simple Grade Analyzer

```
scores = [75, 82, 91, 64, 88]

total = 0
highest = scores[0]
lowest = scores[0]
passed = 0

for score in scores:
    total += score

    if score > highest:
        highest = score

    if score < lowest:
        lowest = score

    if score >= 50:
        passed += 1

average = total / len(scores)

print("Average:", average)
print("Highest:", highest)
print("Lowest:", lowest)
print("Passed:", passed)
```

This one loop performs several related calculations.

It demonstrates how loops become useful when processing real datasets.

---

82. A Complete Example: Haas.dev Resource Analyzer

Imagine:

```
resources = [
    {"title": "Python Variables", "category": "Python"},
    {"title": "Python Conditions", "category": "Python"},
    {"title": "JavaScript Functions", "category": "JavaScript"},
    {"title": "React Native Navigation", "category": "Mobile"}
]
```

Count Python resources:

```
python_count = 0

for resource in resources:
    if resource["category"] == "Python":
        python_count += 1

print("Python resources:", python_count)
```

Output:

```
Python resources: 2
```

The same basic structure can be expanded into:

- category filtering
- search
- statistics
- dashboard data
- resource validation
- reporting

---

83. A Complete Mini Program: Shopping Cart

```
cart = [
    {"name": "Keyboard", "price": 2500},
    {"name": "Mouse", "price": 1200},
    {"name": "USB Cable", "price": 500}
]

total = 0

for item in cart:
```

```
    print(item["name"], "-", item["price"])
    total += item["price"]

print("Total:", total)
```

Output:

```
Keyboard - 2500
Mouse - 1200
USB Cable - 500
Total: 4200
```

This is a simple example of iterating through structured data.

---

84. Engineering Thinking

Don't think of a `for` loop simply as:

> "A way to repeat code."

Think:

> **"I have a collection or sequence of items, and I need to perform an operation for each item."**

This shift in thinking helps you recognize when a loop is appropriate.

For example:

Problem

You have 1,000 resources.

Requirement

Check every resource's category.

Mental model

```
Resources
    ↓
Take one resource
    ↓
Check category
    ↓
Perform required action
    ↓
Take next resource
```

```
    ↓
Repeat
```

That's a natural `for` loop problem.

---

85. The Core Pattern

Most basic `for` loops follow this structure:

```
for item in collection:
    process(item)
```

Examples:

```
for student in students:
    calculate_score(student)
```

```
for product in products:
    validate_product(product)
```

```
for resource in resources:
    generate_metadata(resource)
```

```
for character in text:
    check_character(character)
```

The syntax changes very little.

The work inside the loop changes according to the problem.

---

86. A More Advanced Mental Model

At a conceptual level, Python's `for` loop asks an iterable for its items one at a time.

You don't normally need to manage that mechanism manually.

For example:

```
for item in items:
    print(item)
```

can be understood as:

```
Get an iterator
     ↓
Ask for next item
     ↓
Process item
```

```
     ↓
Ask for next item
     ↓
...
     ↓
No more items
     ↓
Stop
```

Later, when you learn iterators and generators, this internal model will become much more important.

For now, remember:

> **A for loop consumes an iterable one item at a time.**

---

87. When Should You Use a for Loop?

Ask these questions:

Is there a collection of items?

Use `for`.

```
for item in items:
```

Do you need to process every character?

Use `for`.

```
for character in text:
```

Do you need a known number of repetitions?

Use `for` with `range()`.

```
for _ in range(10):
```

Do you need to continue until a condition changes?

Consider `while`.

```
while condition:
```

Does the user determine when the loop ends?

Usually consider `while`.

---

88. Mini Quiz

Question 1

What will this print?

```
for number in range(3):
    print(number)
```

A.

```
1
2
3
```

B.

```
0
1
2
```

C.

```
0
1
2
3
```

D. Nothing

Question 2

What does this loop do?

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

for name in names:
    print(name)
```

A. Prints only the first name

B. Prints every name

C. Prints the indexes

D. Creates a new list

Question 3

What does `break` do inside a loop?

A. Skips one iteration

B. Restarts the loop

C. Stops the entire loop

D. Does nothing

---

89. Mini Quiz Answers

Question 1

B

```
range(3)
```

produces:

```
0
1
2
```

The stop value is excluded.

Question 2

B

The loop processes every item in `names`.

Question 3

C

`break` immediately exits the loop.

---

90. 5 Minute Challenge

Write a program that prints:

```
Python
is
fun
to
learn
```

using:

```
words = ["Python", "is", "fun", "to", "learn"]
```

Requirements:

- Use a `for` loop.
- Do not manually print each word.
- Then modify the program to print each word with its number:

```
1. Python
2. is
3. fun
4. to
5. learn
```

Hint:

Use `enumerate()`.

---

91. Practice Exercises

Exercise 1: Numbers

Print numbers from 1 to 20 using `range()`.

Exercise 2: Even Numbers

Print even numbers from 2 to 50.

Exercise 3: Countdown

Print:

```
10
9
8
...
1
```

using `range()`.

Exercise 4: List Processing

Given:

```
names = ["Ali", "Sara", "Ahmed", "Hina"]
```

print every name.

Exercise 5: String Processing

Given:

```
word = "programming"
```

print every character.

Exercise 6: Sum

Calculate the total of:

```
numbers = [10, 20, 30, 40, 50]
```

Exercise 7: Count

Count how many numbers are greater than 50:

```
numbers = [20, 80, 45, 90, 30, 70]
```

Exercise 8: Search

Search for `"Python"`:

```
languages = ["Java", "Python", "C++", "JavaScript"]
```

Stop once you find it.

Exercise 9: enumerate()

Display:

```
1. Ali
2. Sara
3. Ahmed
```

using `enumerate()`.

Exercise 10: zip()

Given:

```
names = ["Ali", "Sara", "Ahmed"]
scores = [80, 90, 75]
```

print:

```
Ali: 80
Sara: 90
```

```
Ahmed: 75
```

---

92. Mini Project: Student Grade Analyzer

Build a program that analyzes student scores.

Start with:

```
scores = [78, 92, 65, 88, 45, 91]
```

Your program should:

1. Print every score.
2. Calculate the total.
3. Calculate the average.
4. Find the highest score.
5. Find the lowest score.
6. Count how many students passed.
7. Count how many students failed.

A passing score is:

```
50 or above
```

Expected information:

```
Scores: ...
Total: ...
Average: ...
Highest: ...
Lowest: ...
Passed: ...
Failed: ...
```

Try solving it using loops before using built-in functions such as:

```
sum()
max()
min()
```

The goal is to understand the algorithm first.

---

93. Mini Project Extension: Haas.dev Resource Analyzer

Create a list:

```
resources = [
    {"title": "Python Variables", "category": "Python"},
    {"title": "Python Conditions", "category": "Python"},
    {"title": "JavaScript Functions", "category": "JavaScript"},
    {"title": "React Native Navigation", "category": "Mobile"},
    {"title": "Python Loops", "category": "Python"}
]
```

Build a program that:

1. Prints every resource title.
2. Counts Python resources.
3. Prints only Python resources.
4. Counts resources in each category.
5. Searches for a specific resource.
6. Stops searching once the resource is found.

This exercise moves the concept from simple numbers into the kind of structured data you may encounter in real applications.

---

94. Interview Questions

Beginner

1. What is a `for` loop?

A `for` loop iterates over the items of an iterable and executes a block of code for each item.

2. What can you iterate over in Python?

Lists, tuples, strings, dictionaries, sets, ranges, and many other iterable objects.

3. What does `range(5)` produce?

Values from `0` through `4`.

4. Does `range()` include its stop value?

No.

5. What does `break` do?

It immediately terminates the loop.

6. What does `continue` do?

It skips the rest of the current iteration and moves to the next iteration.

Intermediate

7. What is the difference between `for` and `while`?

A `for` loop generally iterates over an iterable, while a `while` loop continues based on a Boolean condition.

8. What does `enumerate()` do?

It allows you to iterate over items while also receiving their indexes.

9. What does `zip()` do?

It combines multiple iterables so their corresponding elements can be processed together.

10. What is a nested loop?

A loop placed inside another loop.

11. What is `for...else`?

A Python construct where the `else` block executes if the loop finishes normally without encountering `break`.

12. Why can modifying a list during iteration be dangerous?

Changing the collection while iterating can cause elements to be skipped or produce unexpected behavior.

---

95. Advanced Thinking: Iterables and Iterators

At the beginner level, you can simply remember:

```
for item in collection:
```

But internally, Python uses an **iterator** to retrieve items one by one.

An iterable can provide an iterator.

You can see this concept explicitly:

```
numbers = [10, 20, 30]

iterator = iter(numbers)

print(next(iterator))
print(next(iterator))
print(next(iterator))
```

Output:

```
10
20
30
```

A `for` loop handles this process for you.

You normally don't need to call `iter()` and `next()` yourself.

But understanding this concept prepares you for:

- iterators
- generators
- lazy evaluation
- large data processing
- advanced Python programming

---

96. Generators and for Loops

Later, you'll learn generators.

For example:

```
def numbers():
    yield 1
    yield 2
    yield 3
```

You can iterate over them:

```
for number in numbers():
    print(number)
```

Output:

```
1
2
3
```

This demonstrates that `for` loops are not limited to lists.

They can work with many kinds of iterable data sources.

---

97. Loop Design Checklist

Before writing a `for` loop, identify:

1. What am I iterating over?

```
for item in ______:
```

2. What does one item represent?

For example:

```
student
product
resource
character
score
```

3. What should happen for each item?

```
process(item)
```

4. Do I need the index?

If yes, consider:

```
enumerate()
```

5. Do I need two related sequences?

Consider:

```
zip()
```

6. Do I need to stop early?

Use:

```
break
```

7. Do I need to skip certain items?

Use:

```
continue
```

This makes loop design intentional rather than trial-and-error.

---

98. Common Patterns to Memorize

Process every item

```
for item in items:
    print(item)
```

Repeat a fixed number of times

```
for _ in range(5):
    print("Hello")
```

Process with index

```
for index, item in enumerate(items):
    print(index, item)
```

Process two sequences together

```
for a, b in zip(list_a, list_b):
    print(a, b)
```

Filter

```
for item in items:
    if condition(item):
        print(item)
```

Search

```
for item in items:
    if item == target:
        break
```

Accumulate

```
total = 0

for number in numbers:
    total += number
```

Transform

```
results = []

for item in items:
    results.append(transform(item))
```

Nested processing

```
for row in rows:
    for item in row:
        process(item)
```

---

99. Key Takeaways

1. A `for` loop is used to iterate over an iterable.

2. The loop processes one item at a time.
3. Lists, tuples, strings, sets, dictionaries, and ranges can all be iterated over.
4. The loop variable stores the current item.
5. `range()` is useful for numeric sequences and fixed repetition.
6. `range()` excludes its stop value.
7. `break` exits the loop immediately.
8. `continue` skips the current iteration.
9. `enumerate()` gives you both an index and an item.
10. `zip()` lets you iterate through related sequences together.
11. Nested loops are useful for grids and multidimensional data.
12. `for...else` can be useful for search operations.
13. Avoid modifying a collection carelessly while iterating over it.
14. Use meaningful loop variable names.
15. Use built-in functions when they already express the required operation clearly.
16. A `for` loop is generally about **iterating over data**, while a `while` loop is generally about **continuing while a condition is true**.
17. Understanding iteration prepares you for more advanced Python concepts such as iterators and generators.

Final Practice Task

Build a **Haas.dev Resource Report Generator**.

Use:

```
resources = [
    {"title": "Python Variables", "category": "Python"},
    {"title": "Python Conditions", "category": "Python"},
    {"title": "JavaScript Functions", "category": "JavaScript"},
    {"title": "React Native Navigation", "category": "Mobile"},
    {"title": "Python Loops", "category": "Python"}
]
```

Your program should:

1. Print every resource with a number.
2. Count the total number of resources.
3. Count Python resources.
4. Print only Python resources.

5. Search for `"Python Loops"`.
6. Stop searching once it is found.
7. Print a message if the resource doesn't exist.
8. Display the final number of Python resources.

Try to solve it using:

- `for`
- `if`
- `enumerate()`
- `break`
- `for...else`
- a counter

The goal is not just to learn the syntax of `for`. The goal is to become comfortable with the idea of **iterating through data, inspecting each item, and building a result from the items you process**.

Visit haas.dev for more resources and guides.

haas.dev

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