Python Loop Patterns and Problem Solving

Introduction

Loops become much more powerful when you learn how to use them to solve problems rather than simply repeat code.

A loop can help you:

- Generate patterns
- Process rows and columns
- Work with numbers
- Search through data
- Count and calculate values
- Build tables
- Compare multiple values
- Solve programming problems step by step

This guide focuses on **how to think with loops**.

The goal is not to memorize patterns. The goal is to understand the logic behind them so you can create your own solutions.

---

1. What Are Loop Patterns?

A loop pattern is an output structure created by controlling:

- How many times a loop runs
- What happens during each iteration
- How many items are printed in each row
- How the inner loop depends on the outer loop

For example:

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

This pattern contains 5 rows.

But there are not 15 separate `print()` statements.

Instead, loops generate the structure.

---

2. Why Learn Loop Patterns?

Loop patterns are useful because they teach you how to control repetition.

They help you understand:

- Nested loops
- Loop counters
- Relationships between rows and columns
- `range()`
- Conditions inside loops
- Incrementing and decrementing values
- Problem decomposition

Pattern problems are often used in beginner programming exercises and interviews because they reveal whether you understand loop logic.

---

3. The Most Important Mental Model

When working with a pattern, think in terms of:

```
Rows → Outer Loop
Columns → Inner Loop
```

For example:

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

There are:

- 4 rows
- 4 columns

So we can think:

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

The outer loop controls the rows.

The inner loop controls what appears inside each row.

---

4. Understanding `print()` and `end`

Normally:

```
print("*")
print("*")
print("*")
```

produces:

```
*
*
*
```

Each `print()` moves to a new line.

But:

```
print("*", end="")
```

keeps printing on the same line.

Example:

```
print("*", end="")
print("*", end="")
print("*", end="")
```

Output:

```
***
```

Then:

`print()`

moves to the next line.

This is essential for pattern problems.

---

5. Square Pattern

Let's create:

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

Code:

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

How it works

The outer loop runs 4 times.

For every row, the inner loop runs 4 times.

Therefore:

`4 rows × 4 columns`

produces 16 stars.

---

6. Rectangle Pattern

A rectangle can have different numbers of rows and columns.

Example:

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

There are:

- 3 rows
- 6 columns

Code:

```
for row in range(3):
    for column in range(6):
        print("*", end="")
    print()
```

General idea:

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

```
    print()
```

---

7. Increasing Triangle

Example:

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

Notice the number of stars:

```
Row 1 → 1
Row 2 → 2
Row 3 → 3
Row 4 → 4
Row 5 → 5
```

The number of columns depends on the current row.

Code:

```
for row in range(1, 6):
    for column in range(row):
        print("*", end="")
    print()
```

This is one of the most important pattern concepts:

> The inner loop can depend on the outer loop.

---

8. Decreasing Triangle

Example:

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

Here the number of stars decreases.

Code:

```
for row in range(5, 0, -1):
    for column in range(row):
        print("*", end="")
    print()
```

The outer loop produces:

```
5
4
3
2
1
```

The inner loop uses that value to determine how many stars to print.

---

9. Number Triangle

Patterns do not have to contain stars.

Example:

```
1
12
123
1234
12345
```

Code:

```
for row in range(1, 6):
    for number in range(1, row + 1):
        print(number, end="")
    print()
```

Output:

```
1
12
123
1234
12345
```

---

10. Repeating Number Triangle

Example:

```
1
22
333
4444
55555
```

Here each row uses the same number.

Code:

```
for row in range(1, 6):
    for column in range(row):
        print(row, end="")
    print()
```

The outer loop determines the number.

The inner loop determines how many times it appears.

---

11. Same Number in Every Row

Example:

```
11111
22222
33333
44444
55555
```

Code:

```
for row in range(1, 6):
    for column in range(5):
        print(row, end="")
    print()
```

Here:

- `row` controls the value
- `column` controls repetition

This distinction is important.

---

12. Right Aligned Triangle

Consider:

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

Now we need two things:

1. Spaces
2. Stars

For each row:

`Spaces + Stars`

For 5 rows:

Row 1 → 4 spaces + 1 star
Row 2 → 3 spaces + 2 stars
Row 3 → 2 spaces + 3 stars
Row 4 → 1 space + 4 stars
Row 5 → 0 spaces + 5 stars

Code:

```
for row in range(1, 6):
    for space in range(5 - row):
        print(" ", end="")

    for star in range(row):
        print("*", end="")

    print()
```

The important observation is:

spaces decrease
stars increase

---

13. Inverted Right Aligned Triangle

Example:

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

Code:

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

    for space in range(5 - row):
        print(" ", end="")

    for star in range(row):
        print("*", end="")

    print()
```

Again, separate the problem into components:

```
Spaces
+
Stars
```

---

14. Pyramid Pattern

Example:

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

Each row contains:

```
Spaces + Stars
```

The number of stars increases by 2:

```
1
3
5
7
9
```

Code:

```
for row in range(1, 6):

    for space in range(5 - row):
        print(" ", end="")

    for star in range(2 * row - 1):
        print("*", end="")

    print()
```

The formula is:

```
stars = 2 × row - 1
```

This is an example of using mathematics to control a loop.

---

15. Inverted Pyramid

Example:

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

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

Code:

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

    for space in range(5 - row):
        print(" ", end="")

    for star in range(2 * row - 1):
        print("*", end="")

    print()
```

The number of stars follows:

```
9
7
5
3
1
```

---

16. Hollow Square

A pattern can also use conditions.

Example:

```
*****
```

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

We only print `*` when the position is:

- First row
- Last row
- First column
- Last column

Code:

```
size = 5

for row in range(size):
    for column in range(size):

        if (
            row == 0
            or row == size - 1
            or column == 0
            or column == size - 1
        ):
            print("*", end="")
        else:
            print(" ", end="")

    print()
```

This introduces a powerful idea:

> A loop can generate every position, while a condition decides what belongs at that position.

---

17. Hollow Triangle

Example:

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

The logic becomes:

```
size = 5

for row in range(1, size + 1):

    for column in range(row):

        if column == 0 or column == row - 1 or row == size:
            print("*", end="")
        else:
            print(" ", end="")

    print()
```

The final row is completely filled.

The other rows only contain stars at their boundaries.

---

18. Character Patterns

Patterns can use letters too.

Example:

```
A
AB
ABC
ABCD
ABCDE
```

Code:

```
for row in range(1, 6):
    for column in range(row):
        print(chr(65 + column), end="")
    print()
```

`chr()` converts a number into its corresponding character.

For example:

```
chr(65)
```

gives:

```
A
```

and:

```
chr(66)
```

gives:

```
B
```

---

19. Character Repetition

Example:

```
A
BB
CCC
DDDD
EEEEE
```

Code:

```
for row in range(5):
    for column in range(row + 1):
        print(chr(65 + row), end="")
    print()
```

Here the row determines the character.

---

20. Multiplication Table Using Loops

Loop problem solving is not limited to visual patterns.

For example, generate a multiplication table:

```
number = 5

for i in range(1, 11):
    print(number, "x", i, "=", number * i)
```

Output:

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
...
5 x 10 = 50
```

---

21. Multiple Tables With Nested Loops

To generate tables from 1 to 5:

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

    for i in range(1, 11):
        print(number * i, end=" ")

    print()
```

The outer loop chooses the table.

The inner loop calculates the multiples.

---

22. Finding Factors

Loops can solve mathematical problems too.

For example:

```
number = 12

for i in range(1, number + 1):
    if number % i == 0:
        print(i)
```

Output:

```
1
2
3
4
6
12
```

The condition:

```
number % i == 0
```

checks whether `i` divides the number evenly.

---

23. Checking Prime Numbers

A prime number has exactly two factors:

1
itself

A simple solution:

```
number = 17
is_prime = True

if number < 2:
    is_prime = False
else:
    for i in range(2, number):
        if number % i == 0:
            is_prime = False
            break

if is_prime:
    print("Prime")
else:
    print("Not prime")
```

Notice the problem-solving structure:

```
Assume
 Check
 Change result if necessary
 Stop when answer is known
 Print result
```

---

24. Sum of Numbers

Problem:

> Find the sum of numbers from 1 to 100.

Code:

```
total = 0

for number in range(1, 101):
    total += number

print(total)
```

The important variable is:

```
total
```

It stores the result as the loop progresses.

This pattern is called an **accumulator**.

---

25. Counting Values

Problem:

> Count how many even numbers exist from 1 to 20.

```
count = 0

for number in range(1, 21):
```

```
    if number % 2 == 0:
        count += 1

print(count)
```

The loop checks every number.

The condition identifies even numbers.

The counter records how many were found.

---

26. Finding the Largest Number

Suppose:

```
numbers = [12, 5, 27, 8, 19]
```

We can solve the problem using a loop:

```
largest = numbers[0]

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

print(largest)
```

The logic is:

```
Start with a candidate
 Compare
```

 Replace when a better candidate appears

This is a reusable problem-solving pattern.

---

27. Searching for a Value

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

target = 30
found = False

for number in numbers:
    if number == target:
        found = True
        break

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

This demonstrates a very common loop strategy:

Search Find Stop

---

28. Nested Data Problem

Suppose we have:

```
students = [
```

```
    ["Ali", 80, 75, 90],
    ["Sara", 92, 88, 95],
    ["Ahmed", 70, 85, 78]
]
```

We can process every student's data:

```
for student in students:

    print("Student:", student[0])

    total = 0

    for marks in student[1:]:
        total += marks

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

The outer loop processes students.

The inner loop processes marks.

---

29. Problem Solving Framework

When you receive a loop problem, do not immediately start coding.

Use this process.

Step 1: Understand the Input

Ask:

What data am I given?

Example:

numbers
rows
columns
students
marks

Step 2: Identify the Desired Output

Ask:

What exactly should the program produce?

Step 3: Find the Repetition

Ask:

What needs to happen repeatedly?

Step 4: Decide the Loop

Ask:

Do I need for?
Do I need while?
Do I need nested loops?

Step 5: Identify Conditions

Ask:

```
Does every item get the same treatment?
Or do some items need special handling?
```

Step 6: Track the State

Ask:

```
Do I need a counter?
A total?
A maximum?
A minimum?
A flag?
```

Step 7: Test With Small Input

Before testing 1000 values, test:

```
1
2
3
```

Small examples make mistakes easier to see.

---

30. Convert a Problem Into Smaller Parts

Consider:

> Print a pyramid.

Do not think:

`How do I write a pyramid?`

Break it down:

```
For each row:
    print spaces
    print stars
    move to next line
```

Then determine the formulas:

```
spaces = size - row
stars = 2 × row - 1
```

Then code.

This is the difference between **memorizing code** and **solving problems**.

---

31. Dry Run Your Loop

Suppose:

```
total = 0

for number in range(1, 4):
    total += number
```

Create a table:

Iteration	number	total
Start	—	0
1	1	1
2	2	3
3	3	6

Final result:

```
6
```

Dry runs are one of the best ways to understand loops.

---

32. Trace Nested Loops

Consider:

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

Execution:

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

Notice:

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

row = 1
   column = 0
   column = 1

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

The inner loop completes all its iterations before the outer loop moves to the next iteration.

---

33. Common Loop Problem Patterns

Many programming problems can be recognized by their structure.

Pattern 1: Counting

```
count = 0
```

Use when you need to know:

How many?

Pattern 2: Accumulation

total = 0

Use when you need:

Sum
Total
Combined value

Pattern 3: Searching

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

Use when you need:

Does it exist?
Where is it?

Pattern 4: Maximum

```
largest = items[0]

for item in items:
    if item > largest:
        largest = item
```

Use when you need the largest value.

Pattern 5: Minimum

```
smallest = items[0]

for item in items:
    if item < smallest:
        smallest = item
```

Pattern 6: Flag

```
found = False
```

Use when the loop needs to remember whether something happened.

Pattern 7: Early Exit

```
break
```

Use when you already have the answer and no longer need to continue.

---

34. Common Mistakes

Mistake 1: Forgetting `print()`

Without:

`print()`

your rows may appear on one line.

Mistake 2: Using the Wrong Range

For:

`1 to 5`

you usually need:

`range(1, 6)`

because the ending value is excluded.

Mistake 3: Confusing Row and Column

Remember:

```
outer □ row
inner □ column
```

This is not an absolute rule for every problem, but it is the most useful mental model for pattern questions.

Mistake 4: Wrong Indentation

Incorrect:

```
for row in range(5):
for column in range(row):
   print("*")
```

Correct:

```
for row in range(5):
   for column in range(row):
      print("*")
```

Mistake 5: Changing the Wrong Variable

Example:

```
for row in range(5):
   for row in range(5):
      ...
```

Avoid reusing the same variable name for nested loops.

Prefer:

```
for row in range(5):
   for column in range(5):
      ...
```

---

35. How to Debug a Loop

If the output is wrong, do not randomly change code.

Ask:

1. How many times does the outer loop run?

2. How many times does the inner loop run?

3. What is the value of each variable?

4. When does the loop stop?

5. Should the current iteration continue?

6. Is the output printed at the correct location?

You can temporarily print variables:

```
for row in range(3):
    print("ROW:", row)

    for column in range(3):
        print("COLUMN:", column)
```

Debugging becomes much easier when you can see what the loop is actually doing.

---

36. Efficiency and Time Complexity

Nested loops often increase the amount of work.

Example:

```
for i in range(n):
    for j in range(n):
```

```
        print(i, j)
```

The inner loop runs n times for every outer iteration.

Total:

$n \times n = n^2$

So the time complexity is:

$O(n^2)$

Three nested loops can lead to:

$O(n^3)$

This does not mean nested loops are always bad.

They are often exactly what the problem requires.

The important skill is understanding the cost.

---

37. When Should You Use Nested Loops?

Nested loops are useful when one repeated operation exists inside another repeated structure.

Examples:

```
Rows and columns
Students and subjects
```

Products and categories
Matrix operations
Comparing pairs
2D grids
Tables
Game boards

Before using nested loops, ask whether the problem actually contains multiple levels of repetition.

---

38. A Practical Problem Solving Example

Problem

Print all pairs from:

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

Expected:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Solution:

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

```
for first in numbers:
    for second in numbers:
        print(first, second)
```

Think about it as:

```
Take first number
    Compare with every number

Move to second number
    Compare with every number

Move to third number
    Compare with every number
```

The code becomes easier once the problem is decomposed.

---

39. Haas.dev Example

Suppose haas.dev has resources grouped by category:

```
resources = [
    ["Python", "Variables", "Loops"],
    ["JavaScript", "Arrays", "Functions"],
    ["React Native", "Navigation", "API"]
]
```

We want to display every category and its resources.

```
for category in resources:
```

```
    print("Category:", category[0])

    for resource in category[1:]:
        print(" -", resource)
```

Output:

```
Category: Python
 - Variables
 - Loops

Category: JavaScript
 - Arrays
 - Functions

Category: React Native
 - Navigation
 - API
```

The outer loop handles categories.

The inner loop handles resources inside each category.

This is the same logic used when processing real nested data.

---

40. Mini Quiz

Question 1

What does the outer loop usually control in a pattern?

A. Individual characters
B. Rows
C. Conditions
D. Functions

Answer: B

Question 2

What does `print("*", end="")` do?

A. Prints a star and moves to the next line
B. Prints nothing
C. Prints a star without automatically moving to a new line
D. Deletes the previous star

Answer: C

Question 3

What is the time complexity of:

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

A. O(1)
B. O(n)

C. O(log n)
D. O(n^2)

Answer: D

---

41. 5 Minute Challenge

Without copying the solution, create:

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

Then modify your program to produce:

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

Finally create:

```
1
12
123
1234
12345
```

Your goal is to understand what changed in the loops rather than memorizing three programs.

---

42. Practice Exercises

Beginner

1. Print a 5 × 5 square of stars.
2. Print a 4 × 8 rectangle.
3. Print numbers from 1 to 10.
4. Print an increasing star triangle.
5. Print a decreasing star triangle.
6. Print numbers in a triangle.
7. Print a multiplication table.

Intermediate

8. Print a right aligned triangle.
9. Print a pyramid.
10. Print an inverted pyramid.
11. Print a hollow square.
12. Print a hollow triangle.
13. Print an alphabet triangle.
14. Find all factors of a number.
15. Check whether a number is prime.
16. Count even and odd numbers in a list.
17. Find the largest number without using `max()`.

Problem Solving

18. Find the smallest number without using `min()`.
19. Calculate the average of numbers.

20. Count how many numbers are greater than 50.
21. Search for a target value in a list.
22. Find duplicate values.
23. Generate all pairs from a list.
24. Process a nested list of students and marks.
25. Create a multiplication table grid using nested loops.

---

43. Mini Project: Student Result Analyzer

Create a program that processes:

```
students = [
    ["Ali", 80, 75, 90],
    ["Sara", 92, 88, 95],
    ["Ahmed", 70, 85, 78]
]
```

Your program should:

1. Display each student's name.
2. Calculate total marks.
3. Calculate average marks.
4. Determine whether the student passed.
5. Find the student with the highest total.

Example output:

```
Ali
Total: 245
Average: 81.67
Status: Pass
```

```
Sara
Total: 275
Average: 91.67
Status: Pass

Ahmed
Total: 233
Average: 77.67
Status: Pass
```

Suggested approach

Break the project into smaller problems:

```
Get student
 Get marks
 Calculate total
 Calculate average
 Check status
 Compare totals
```

Do not try to solve the entire project in one step.

---

44. Interview Questions

Beginner

1. What is a loop pattern?
2. Why are nested loops useful for patterns?
3. What is the difference between the outer and inner loop?
4. What does `end=""` do?

5. Why is `print()` often used after an inner loop?
6. How does `range()` affect pattern size?

Intermediate

7. How would you create an increasing triangle?
8. How would you create a right aligned triangle?
9. How would you create a hollow square?
10. How can conditions be used inside nested loops?
11. How do you calculate the time complexity of nested loops?
12. When should you use `break` inside a search?
13. What is an accumulator?
14. What is a flag variable?
15. How would you find the largest value using a loop?

Problem Solving

16. How would you approach a loop problem you have never seen before?
17. Why is dry running useful?
18. How can you break a complex loop problem into smaller parts?
19. How would you debug a nested loop?
20. When can nested loops become inefficient?

---

45. Loop Patterns Cheat Sheet

Goal	Typical approach
Repeat something	`for` / `while`

Rows	Outer loop
Columns	Inner loop
Same number per row	Use outer loop value
Increasing pattern	Inner loop depends on row
Decreasing pattern	Reverse range
Alignment	Spaces + content
Hollow pattern	Conditions
Count	`count += 1`
Sum	`total += value`
Maximum	Compare and replace
Minimum	Compare and replace
Search	Condition + optional `break`
Stop early	`break`

Skip item	`continue`
Remember state	Flag variable
Process nested data	Nested loops

---

46. The Bigger Lesson

Loop patterns are not really about stars.

They are about learning to recognize structure.

When you see:

```
Rows
Columns
Repetition
Conditions
Changing values
Nested data
```

you should start thinking:

```
What repeats?
What controls the repetition?
What changes each iteration?
What condition matters?
```

`When should the loop stop?`

Once you can answer these questions, many loop problems become much easier.

The goal is not:

> "I remember the code for this pattern."

The goal is:

> "I understand why this pattern needs these loops."

That mindset transfers directly to real programming problems.

Key Takeaways

- Loop patterns teach you how to control repetition.
- The outer loop commonly represents rows.
- The inner loop commonly represents columns or repeated work inside a row.
- `print(..., end="")` allows output on the same line.
- `print()` moves output to the next line.
- Pattern problems often combine loops with conditions.
- Spaces can be used to control alignment.
- Many real programming problems use the same structures as pattern exercises.
- Counting, accumulation, searching, maximum/minimum, and flags are reusable problem-solving patterns.
- Dry runs help you understand what a loop actually does.

- Nested loops can increase time complexity, often to $O(n^2)$.
- The best way to learn loops is to solve problems rather than memorize patterns.

Visit haas.dev for more resources and guides.

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

haas.dev