



# Chapter 7

## Chapter 7: Loops in Python

### What Are Loops?

Loops are used to **repeat** a block of code **multiple times** efficiently. Instead of writing the same thing 100 times, you tell Python:

"Run this 100 times!" and it says "Sure!".

There are **2 main types** of loops in Python:

Loop Type	When to Use
<code>for</code> loop	When you know how many times to repeat something
<code>while</code> loop	When you want to repeat until a condition is false

### `for` Loop – The Repeater You Trust

```
for i in range(1, 101):  
    print(i)
```

- `range(1, 101)` → Starts at 1, goes up to but not including 101

- `i` → Loop variable (think of it as a counter)
  - Repeats **100 times** from 1 to 100
- 

### Print Name Multiple Times

```
for i in range(10):  
    print("Prathamesh")
```

- Starts at 0 by default if not specified
  - Repeats 10 times (0 to 9)
- 

### Range Syntax Breakdown

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

Example:

```
for i in range(0, 10, 2):  
    print(i)
```

**Output:** 0, 2, 4, 6, 8

- start = 0
  - stop = 10
  - step = +2 (skip 2 numbers)
- 

### **while** Loop – Repeat Until Condition Fails

```
i = 1  
while i < 51:  
    print(i)  
    i += 1
```

- Starts from 1
- Repeats while `i` is less than 51

- `i += 1` increases the value of `i` every loop

✓ Same as writing:

```
i = i + 1
```

## ! Caution: Infinite Loop Danger

```
i = 0
while i < 10:
    print(i)
    # If you forget i += 1 → it will run forever!
```

## Iterating Over Collections

Python makes it **super simple** to loop over lists, strings, tuples:

```
a = [23, 442, 4]
for b in a:
    print(b)

c = "Prathamesh"
for letter in c:
    print(letter) # One letter per line
```

## `for` with `else` → (Bonus Feature)

```
list = [1, 2, 3]
for item in list:
    print(item)
else:
    print("Done") # Executes *after* loop finishes successfully
```

If the loop **completes fully** without being broken, the `else` block runs.

## `break` and `continue` – Traffic Signals

## **break** – Emergency Exit

```
for i in range(100):
    if i == 34:
        break
    print(i)
```

**Breaks** the loop when `i == 34`. Nothing after that gets printed.

## **continue** – Skip That Iteration

```
for i in range(100):
    if i == 70:
        continue
    print(i)
```

**Skips** printing when `i == 70`. Everything else continues.

## **pass** – Do Nothing (Placeholder)

```
for i in range(10):
    pass # Used when you're not ready to write logic yet
```

Python expects *something* inside a loop or block. `pass` says:

| "I'm not doing anything here... yet."

## **Common Mistakes to Avoid**

Mistake	Problem
<code>while True:</code> without break	Creates infinite loop if not handled
<code>i =+ 1</code> instead of <code>i += 1</code>	Sets <code>i = +1</code> every time instead of incrementing
Using <code>=</code> instead of <code>==</code>	<code>=</code> is assignment, <code>==</code> is comparison
Forgetting <code>:</code> at end of loop	SyntaxError: expected ':'
<code>range(10)</code> confusion	Outputs 0 to 9, not 1 to 10

# Chapter 7: Advanced Loop Theory — Deep Dive Notes

These are the **not-so-obvious**, but **important and tricky** parts that beginner tutorials often gloss over.

## ◆ 1. `for...else` – Why Does This Even Exist?

### ? What's confusing?

You expect `else` to be used with `if`, not with `for`. So why does `for...else` even exist?

### 🔍 Deep Explanation:


- The `else` block **runs only when the loop didn't hit a `break`**.
- Python internally marks a flag if `break` was hit. If not, it goes to `else`.

### 🔍 Example: Searching

```
nums = [2, 4, 6, 8, 10]
for num in nums:
    if num == 5:
        print("Found 5!")
        break
else:
    print("5 not found.")
```

➡ Output: `"5 not found."`

✓ `else` only runs when `break` is *not* triggered.

 Think of `else` here as "didn't break out of the loop" — not "else condition".

## ◆ 2. `continue` vs Skipping with `if` – Subtle Behavior Difference

Both `continue` and using `if` to skip seem similar. But there's a subtle readability and structure difference.

### Deep Logic:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

vs

```
for i in range(5):
    if i != 2:
        print(i)
```

While both seem to **skip i = 2**, the first clearly communicates:

🧠 "I'm intentionally skipping this case"

The second just hides the intention inside logic.

✔ Use `continue` when your default behavior is to do something, except for a few edge cases.

## ◆ 3. `range()` Doesn't Work Like List — It's a Lazy Generator

When you do `for i in range(1, 11)`, it's **not** storing all 10 numbers in memory.

### Under the Hood:

- `range()` creates a **range object**, not a list.
- This object **generates numbers one by one on demand**.
- That's why `range(1_000_000_000)` doesn't crash your system.

✔ Efficient in memory.

✗ But not indexable like a list unless you convert: `list(range(...))`

## ◆ 4. Nested Loops: Think Like a Matrix, Not a List

```
for i in range(3):  
    for j in range(2):  
        print(i, j)
```

Output:

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


### Think of it like:

Outer loop → Rows

Inner loop → Columns

Useful for:

- 2D arrays
- Patterns
- Simulations

 Inner loop completes fully every time outer loop runs once.

## ◆ 5. While vs For: Real Reason Behind Two Loops

When should you really use **while** ?

If your **loop count is unknown**, and you're waiting for something to happen (like input, error, or random event), go with **while**.

```
while True:  
    data = input("Enter: ")
```

```
if data == "quit":  
    break
```

`for` is **deterministic**, `while` is **event-driven**.

## ◆ 6. Modifying a List Inside a Loop = Pitfall 💣

```
nums = [1, 2, 3, 4]  
for i in nums:  
    if i % 2 == 0:  
        nums.remove(i) # BAD IDEA
```

👉 You're **modifying the list while looping over it**, and Python loses track of the index correctly. You'll skip items.

✅ Safer ways:

```
# Way 1: Loop over a copy  
for i in nums[:]:  
    ...  
  
# Way 2: List comprehension  
nums = [i for i in nums if condition]
```

## ◆ 7. Reversed Loops — Mind the Off-by-One

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

Looks like it prints 10 to 1? ✓ Yes. But...

🔍 **Why** `range(10, 0, -1)` ?

Because `range` **excludes the stop value**.

So if you want to include `1`, you must stop at `0`.

If you mistakenly write `range(10, 1, -1)`, you'll miss 1.



## ◆ 8. Loop Variable Scope (Closures Gotcha)

Advanced edge case:

```
funcs = []

for i in range(3):
    funcs.append(lambda: print(i))

for f in funcs:
    f() # Prints 2, 2, 2 — not 0,1,2!
```

Why? Because the lambdas **remember the same** `i`, and by the time they run, the loop has ended and `i = 2`.

✓ Fix with:

```
for i in range(3):
    funcs.append(lambda i=i: print(i))
```

## ◆ 9. Loops Can Be Used as Filters, Not Just Repeaters

You don't always loop to repeat; sometimes, you loop to **filter**:

```
names = ["prathamesh", "sanket", "vivek"]
short_names = []

for name in names:
    if len(name) < 8:
        short_names.append(name)

# Same logic with list comprehension:
short_names = [name for name in names if len(name) < 8]
```

## ◆ 10. Infinite Loops Are Useful (Sometimes)

Infinite loops like:

```
while True:  
    ...
```

Are used in:

- Games
- GUIs
- Servers
- Input validation loops

Just make sure you **have a `break` somewhere**, or you'll crash the terminal 🤯

### 🧠 Pro Tips:

- Always know **what your loop variable is doing each cycle**
- Use `print()` inside the loop to debug logic
- Dry run on paper for nested loops to avoid confusion
- **Use `enumerate()`** when looping over lists *and* tracking index

```
for idx, val in enumerate(["a", "b", "c"]):  
    print(idx, val)
```

## 📖 Chapter 7 – Loops (Practice Problems)

### 🔪 Problem 1: Multiplication Table using `for` loop

```
number = int(input("Enter the number: "))  
for i in range(1, 11):  
    print(f"{number} X {i} = {i*number}")
```

✓ Simple, clean. Loop from 1 to 10 and multiply.

### 🔪 Problem 2: Greet names starting with 'S'

```
l = ["Harry", "Soham", "Sachin", "Rahul"]
```

```
for name in l:
```

```
    if name.startswith("S"): # Or name[0] == "S"
```

```
        print(f"Welcome {name}")
```

 You wrote `startswith("S" or "s")` — small bug.

 `"S" or "s"` always returns `"S"`

 Fix: `name.startswith("S")` or manually check: `if name[0] == "S"`

### Problem 3: Multiplication Table using `while` loop


```
number = int(input("Enter the number: "))
```

```
i = 1
```

```
while i <= 10:
```

```
    print(f"{number} X {i} = {i*number}")
```

```
    i += 1
```

 Use `while` when you want *more control* than a `for` loop gives.

### Problem 4: Prime Number Checker

```
number = int(input("Enter the number: "))
```

```
for i in range(2, number):
```

```
    if number % i == 0:
```

```
        print("This number is not a prime number")
```

```
        break
```

```
else:
```

```
    print("This number is a prime number")
```

 That `else:` attached to the `for` loop runs **only if the loop wasn't broken**.

Cool Python feature that many miss!

### Problem 5: Sum of first n natural numbers (using `while` )

```

number = int(input("Enter the number: "))
i = 0
sum = 0
while i <= number:
    sum += i
    i += 1
print(sum)

```

⚠ Don't forget you can also do it with formula:

```

sum = number * (number + 1) // 2

```

### Problem 6: Factorial using loop

```

number = int(input("Enter the number: "))
multiplication = 1
for i in range(1, number + 1):
    multiplication *= i
print(f"The factorial of {number} is {multiplication}")

```

💡 A must-know loop problem. Very useful in math, recursion, ML, and more.

### Problem 7: Centered Star Pyramid

For `n = 3`

```


*
***
*****

```

```

n = int(input("Enter Number: "))
for i in range(1, n + 1):
    print(" " * (n - i), end="")
    print("*" * (2 * i - 1), end="")
    print("")

```

 Concept:

- $n - i$  = spaces
- $2*i - 1$  = stars (makes it odd numbers like 1, 3, 5)

## Problem 8: Right-Angle Triangle Pattern

For  $n = 3$

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

```
n = int(input("Enter Number: "))
for i in range(1, n + 1):
    print("*" * i)
```

 Simpler than pyramid. No spaces needed.

## Problem 9: Hollow Rectangle Star Pattern

For  $n = 3$

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

```
n = int(input("Enter Number: "))
for i in range(1, n + 1):
    if i == 1 or i == n:
        print("*" * n)
    else:
        print("*" + " " * (n - 2) + "*")
```

 Used nested logic to skip middle stars and just print border.

## Problem 10: Reverse Multiplication Table

```
number = int(input("Enter the number: "))
for i in range(10, 0, -1):
    print(f"{number} X {i} = {i*number}")
```

⚠ You wrote:

```
for i in range(1, 11):
    print(f"{number} X {11-i} = {(11-i)*number}")
```

✓ Also works, just reverse logic manually.

## ✓ Chapter 7 Summary Table (All Concepts)

Concept	Description
<code>for</code> loop	Iterate a block of code for a fixed number of times
<code>while</code> loop	Repeat code while a condition is <code>True</code>
<code>range(start, stop)</code>	Used in <code>for</code> loop to define start and end (end exclusive)
<code>range(start, stop, step)</code>	Add steps (like +2, -1) to control loop flow
<code>break</code>	Exit the loop immediately
<code>continue</code>	Skip current iteration and go to next
<code>else</code> with <code>for/while</code>	Runs only if loop finishes without <code>break</code>
<code>pass</code>	Null statement – tells Python “do nothing here (yet)”
<code>for item in list:</code>	Loop through each element in a list, string, or tuple
<code>len()</code>	Gets total number of elements in iterable (used in loops often)
<code>str.startswith()</code>	Checks if a string starts with a particular substring
<code>end=""</code> in <code>print()</code>	Prevents newline; helps create patterns or aligned output
Nested Loops	Loops inside loops – used for complex patterns or grid-based logic