Quick Reference Notes

Week 9: Introduction to `for` Loops

7th Grade Computer Science

1 Introduction

1.1 This Week's Big Question

How can we make a computer repeat tasks without writing the same code over and over? Imagine having to write "Happy Birthday" 100 times for decorations - there must be a better way! This week, we'll discover how loops turn repetitive tasks into elegant solutions.

Prerequisites

Before starting this week, you should be comfortable with:

- Creating and using variables (Week 3)
- Working with numbers and mathematical expressions (Week 5)
- Understanding how Python executes code line by line (Week 5)
- Using proper indentation (Week 7)

1.2 What You Already Know

You can store values in variables and perform calculations with them. You understand that Python reads your code from top to bottom, executing each line in order. You've learned to make decisions with if statements and create programs that respond differently based on conditions.

1.3 What You'll Be Able to Do

By the end of this week, you'll:

- Use for loops to repeat code a specific number of times
- Master the `range()` function to control loop behavior
- Create pattern printers and multiplication tables
- Trace through loops using memory diagrams
- Recognize when loops make code more efficient
- Debug loops by understanding exactly what happens in each iteration

2 VIDEO 1: Your First For Loop

2.1 From Repetition to Loops

Let's say you want to print "`Pakistan Zindabad!`" 5 times for Independence Day decorations. Without loops, you would write:

```python
print("Pakistan Zindabad!")
print("Pakistan Zindabad!")
print("Pakistan Zindabad!")
print("Pakistan Zindabad!")
print("Pakistan Zindabad!")
```

That's a lot of typing! What if you needed it 100 times?

There's a better way!

2.2 Let's See It In Action

Using a `for` loop - so much cleaner:

```python
for i in range(5):
    print("Pakistan Zindabad!")
```

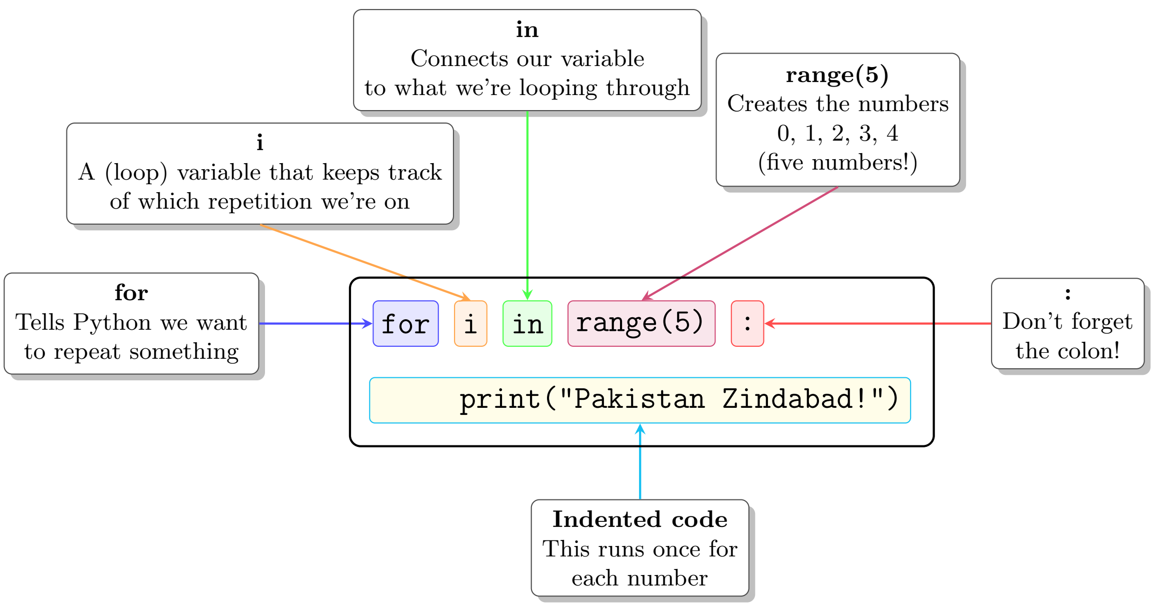
This prints exactly the same output, but with just 2 lines of loop code instead of 5 repetitive lines!

2.3 Breaking It Down

Let's understand each part:

```python
for i in range(5):
    print("Pakistan Zindabad!")
```

Anatomy of a for Loop


in
Connects our variable to what we're looping through
range(5)
Creates the numbers 0, 1, 2, 3, 4 (five numbers!)
i
A (loop) variable that keeps track of which repetition we're on
for
Tells Python we want to repeat something
for
i
in
range(5)
:
print("Pakistan Zindabad!")
:
Don't forget the colon!
Indented code
This runs once for each number

2.4 Watching the Loop Variable

The variable `i` is called the loop variable. It changes each time through the loop:

```python
for i in range(5):
    print(f"Repetition {i}: Pakistan Zindabad!")

# Output:
# Repetition 0: Pakistan Zindabad!
# Repetition 1: Pakistan Zindabad!
# Repetition 2: Pakistan Zindabad!
# Repetition 3: Pakistan Zindabad!
# Repetition 4: Pakistan Zindabad!
```

Notice how `i` starts at 0 and goes up to 4 (not 5)!

Rule

Basic For Loop Pattern:

```python
for variable_name in range(number):
    # Code to repeat
    # Can be multiple lines
    # All must be indented
# This code runs after the loop
```

Remember:

- The loop variable can be any name (`i`, `j`, `count`, etc.)
- `range(n)` gives you n repetitions
  - Starts at `0` and goes up to `n-1`, incrementing by 1 each time
  - So `range(5)` gives 0, 1, 2, 3, 4
- Everything indented under the `for` statement runs each time

Quick Check

How many times will `"Hello!"` print?

```python
for x in range(3):
    print("Hello!")
    print("Goodbye!")
```

3 VIDEO 2: Understanding range()

3.1 The Magic of range()

The `range()` command *(also called a function)* is like a number generator for '`for`' loops. It creates a sequence of numbers for your loop to work through. Think of it as setting up a specific number of items to count through - like beads on a string or notches on a tally stick - you know exactly how many iterations you'll go through.

3.2 range() with One Number

When you give `range()` one number, it starts at 0 and stops before that number:

`range(5)` creates: `0, 1, 2, 3, 4`

```python
for num in range(5):
    print(num, end=" ")
# Output: 0 1 2 3 4
```

`range(3)` creates: `0, 1, 2`

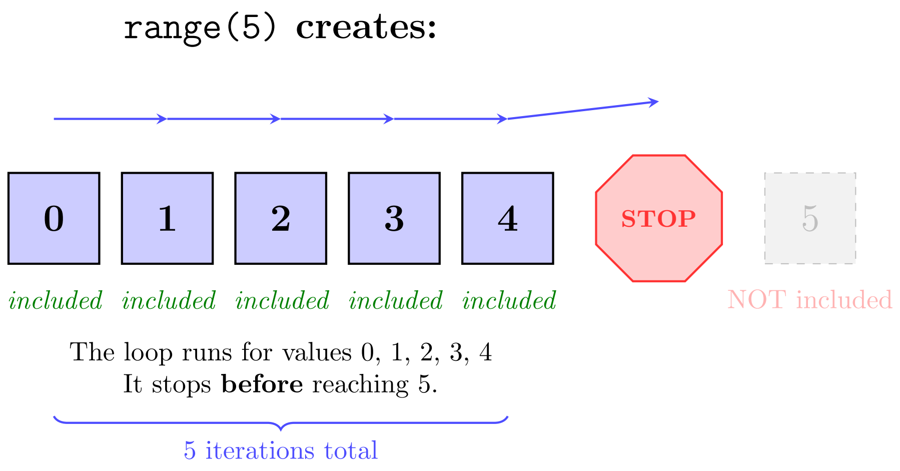
```python
for count in range(3):
    print(f"Count: {count}")
# Output:
# Count: 0
# Count: 1
# Count: 2
```

Tip

Note: the `end=" "` in the first example makes the output print on the same line, separated by spaces. If you don't use it, each number will print on a new line.

3.3 Why Start at 0?

Computer scientists count from 0! It's like ground floor being 0, first floor being 1. This matches how Python indexes strings and lists *(which you'll learn later)*.



Common Error

The range() Stop Sign:

`range(5)` does NOT include 5

Think of the number in `range()` as a "stop sign" - you stop **before** you reach it.

- `range(10)` → `0,1,2,3,4,5,6,7,8,9` *(10 not included)*
- `range(3)` → `0,1,2` *(no 3)*
- `range(100)` → `0 through 99` *(100 not included)*

3.4 Practical Examples

```python
# Print numbers 0 to 9
print("Single digits:")
for digit in range(10):
    print(digit)
print()  # New line

# Create a simple counter
print("\nRocket launch countdown prep:")
for second in range(5):
    print(f"T-minus {5-second} seconds")
print("Ready for launch!")
```

Tip

Choosing Good Loop Variable Names:

- `i`, `j`, `k` - Traditional for simple counting
- `num`, `count` - When working with numbers
- `day`, `month` - When the number represents something specific
- Be descriptive when it helps understanding!

Quick Check

What numbers will this print?

```python
for n in range(4):
    print(n * 2)
```

Trace through each iteration in your head first!

4 VIDEO 3: Loop Patterns

4.1 Common Loop Patterns

Loops are perfect for creating patterns, calculations, and repetitive tasks. Let's explore three essential patterns every programmer uses.

4.2 Pattern 1: Building Strings

```python
#-------------------------------
# Create a row of stars
stars = ""
for i in range(7):
    stars = stars + "*"
print(stars)  # Output: *******
#-------------------------------
# Create a border
border = "+"
for i in range(10):
    border = border + "-"
border = border + "+"
print(border)  # Output: +----------+
#-------------------------------
```

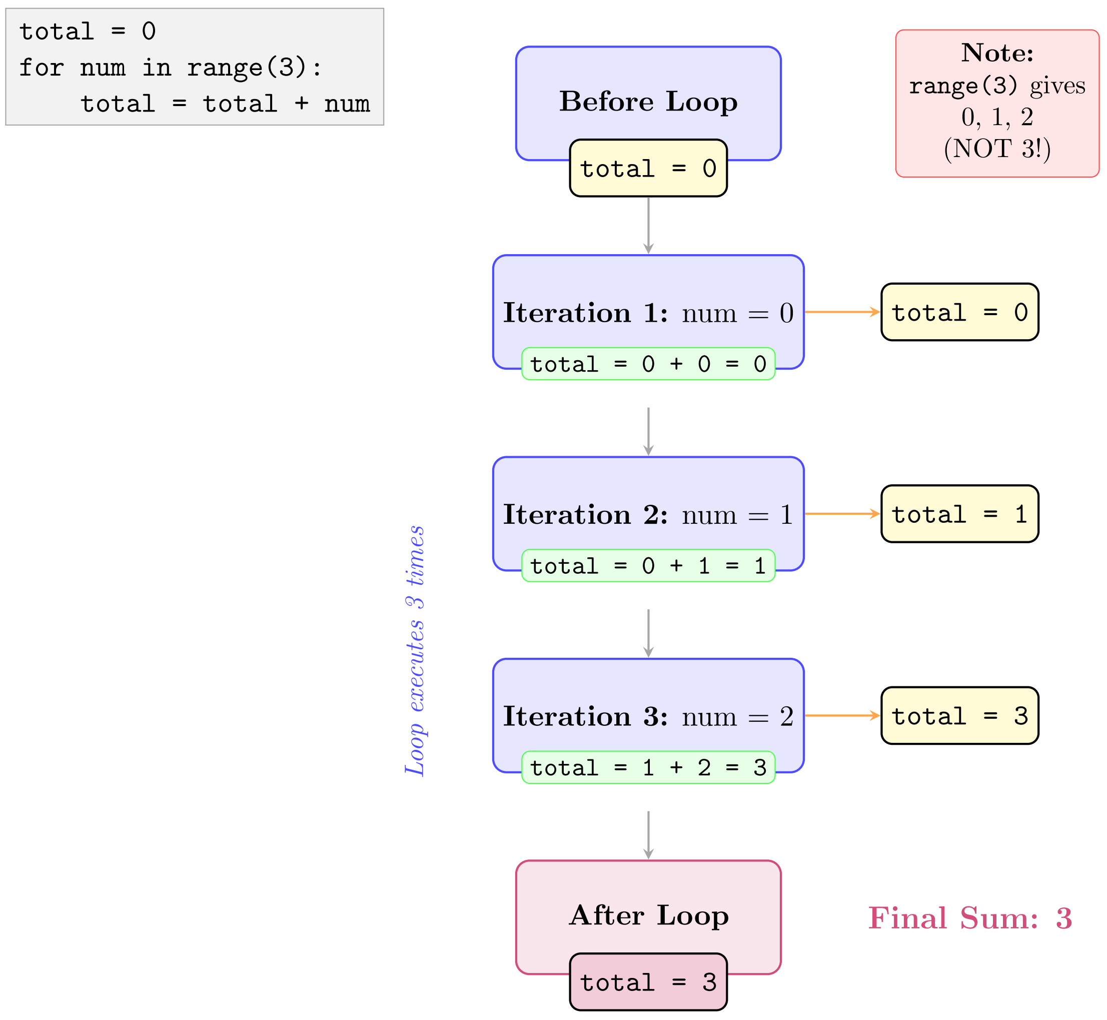
4.3 Pattern 2: Accumulating Values

```python
#--------------------------------
# Sum numbers from 0 to 5
total = 0
for num in range(6):
    total = total + num
    print(f"Added {num}, total is now {total}")
print(f"Final sum: {total}")
#--------------------------------
# Count by 10s
count = 0
for i in range(5):
    count = count + 10
    print(count, end=" ")
# Output: 10 20 30 40 50
#--------------------------------
```

4.4 Pattern 3: Repeating with Variations

```python
#--------------------------------
# Print a times table
print("5 times table:")
for i in range(1, 11):      # You will learn about this in a future lecture
    result = 5 * i
    print(f"5 x {i} = {result}")
#--------------------------------
# Create numbered messages
for day in range(7):
    print(f"Day {day + 1}: Check your homework")
#--------------------------------
```

Memory Trace: `range(3)` Example



5 VIDEO 4: Memory Diagrams and Loop Tracing

5.1 Seeing Inside the Loop

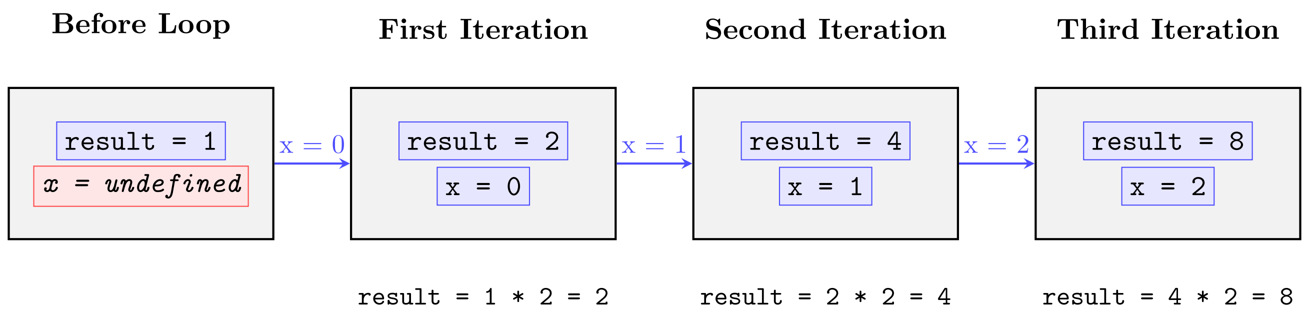
Understanding what happens inside a loop, step by step, is crucial for debugging. We'll use memory diagrams to visualize exactly how variables change during each iteration.

5.2 Tracing a Simple Loop

Let's trace through this multiplication loop:

```python
result = 1
for x in range(3):
    result = result * 2
```

Memory Trace: Doubling Pattern



```
result = 1
for x in range(3):
    result = result * 2
```

Final value: result = 8

5.3 Step-by-Step Trace

Here's what happens:

1. **Before loop**: `result` = 1, loop hasn't started
2. **First iteration**: `x` = 0, `result` = 1 × 2 = 2
3. **Second iteration**: `x` = 1, `result` = 2 × 2 = 4
4. **Third iteration**: `x` = 2, `result` = 4 × 2 = 8
5. **After loop**: Loop ends, `result` = 8

5.4 A More Complex Example: Running Total

```python
# Calculate sum of first 4 numbers
total = 0
for num in range(4):
    old_total = total  # Remember for display
    total = total + num
    print(f"Iteration {num+1}: {old_total} + {num} = {total}")
print(f"Sum of 0+1+2+3 = {total}")
```

Memory Trace Table:
When there are more variables to keep track of, a table trace could be a better alternative to the above flow based trace diagrams *(drawn above)*. We maintain a table showing how variables change each iteration:

Iteration	num	old_total	total
1	0	0	0
2	1	0	1
3	2	1	3
4	3	3	6

5.5 Debugging with Print Statements

When loops don't work as expected, add extra print statements to see what's happening. For examle:
Problem: Why doesn't this countdown reach 0?

```python
countdown = 10
for i in range(3):
    countdown = countdown - 2
print(f"Final countdown: {countdown}")
# Output: Final countdown: 4
# Expected: Final countdown: 0
```

To understand what's going wrong, add extra debug print statements, one at the start and one at the end of the loop:

```python
countdown = 10
for i in range(3):
    print(f"DEBUG: Loop {i}, countdown before: {countdown}")
    countdown = countdown - 2
    print(f"DEBUG: Loop {i}, countdown after: {countdown}")
print(f"Final countdown: {countdown}")

# Output:
# DEBUG: Loop 0, countdown before: 10
# DEBUG: Loop 0, countdown after: 8
# DEBUG: Loop 1, countdown before: 8
# DEBUG: Loop 1, countdown after: 6
# DEBUG: Loop 2, countdown before: 6
# DEBUG: Loop 2, countdown after: 4
# Final countdown: 4
```

Now we can see the loop only runs 3 times (i = 0, 1, 2), subtracting 2 each time, giving us 10 - 6 = 4. To reach 0, so we would need `range(5)`.

Tip

Loop Debugging Strategy:

1. Add print at the start of loop body
2. Print the loop variable and any changing values
3. Use descriptive labels like "DEBUG:" or "TRACE:"
4. Check if values match your expectations
5. Remove debug prints when the issues are fixed!

Quick Check

Trace this loop by hand. What's the final value of `power`?

```python
power = 1
for x in range(4):
    power = power * 3
```

Show each iteration: `x` value and `power` value.

6 VIDEO 5: Summary

6.1 The Power of Loops

This week, we've unlocked one of programming's most powerful tools. With `for` loops, you can now make computers do repetitive work instantly - whether it's printing patterns, calculating sums, or processing data.

6.2 Key Concepts We Covered

Concept	Key Points	Example
Basic **for** loop	Repeats code **n** times	`for i in range(5):`
range() function	Creates number sequences	`range(10)` → 0 to 9
Loop patterns	Building, accumulating, processing	String building, sums
Memory diagrams	Visualize variable changes	Track iterations
Loop debugging	Print to see what's happening	Add DEBUG prints

6.3 Common Gotchas to Remember

- `range(5)` goes 0,1,2,3,4 (not 1,2,3,4,5)
- Loop variables start at 0 by default
- Indentation matters - unindented code is outside the loop
- Variables changed inside loops keep their values after

Landmark Moment

You've mastered the fundamental concept of iteration! `for` loops transform tedious repetition into elegant code. This is a landmark moment - you can now write programs that work with large amounts of data, create complex patterns, and automate repetitive tasks. Next week, you'll learn even more powerful loop techniques!

7 Quick Reference

Quick Reference

Week 9 For Loop Patterns:

```python
# Basic for loop
for i in range(n):       # Runs n times (0 to n-1)
    # Code to repeat

# range() variations (next week preview!)
range(5)             # 0, 1, 2, 3, 4
range(1, 6)          # 1, 2, 3, 4, 5
range(0, 10, 2)      # 0, 2, 4, 6, 8

# Debugging loops
for i in range(n):
    print(f"DEBUG: i={i}, other_var={other_var}")
    # Your code here
```

8 Reflection

Think about your learning journey this week:

1. When you first saw a for loop, what made sense immediately? What confused you? How did you work through that confusion?
2. The concept of range(5) not including 5 trips up many students. How do you remember this? What mental model helps you?
3. Think about a time this week when your loop didn't do what you expected. What was wrong? How did you figure it out?
4. Memory diagrams help us trace loops. Do you find it easier to trace in your head or on paper? Why?
5. Look at the three loop patterns (building, accumulating, processing). Which one feels most natural to you? Which one do you want more practice with?
6. If you had to explain to a friend why programmers use loops instead of copying code, what would you say?
7. Compare how you feel about loops now versus at the start of the week. What clicked for you? What still feels challenging?

Next week, we'll expand your loop powers with advanced range() options and nested loops to create amazing patterns!