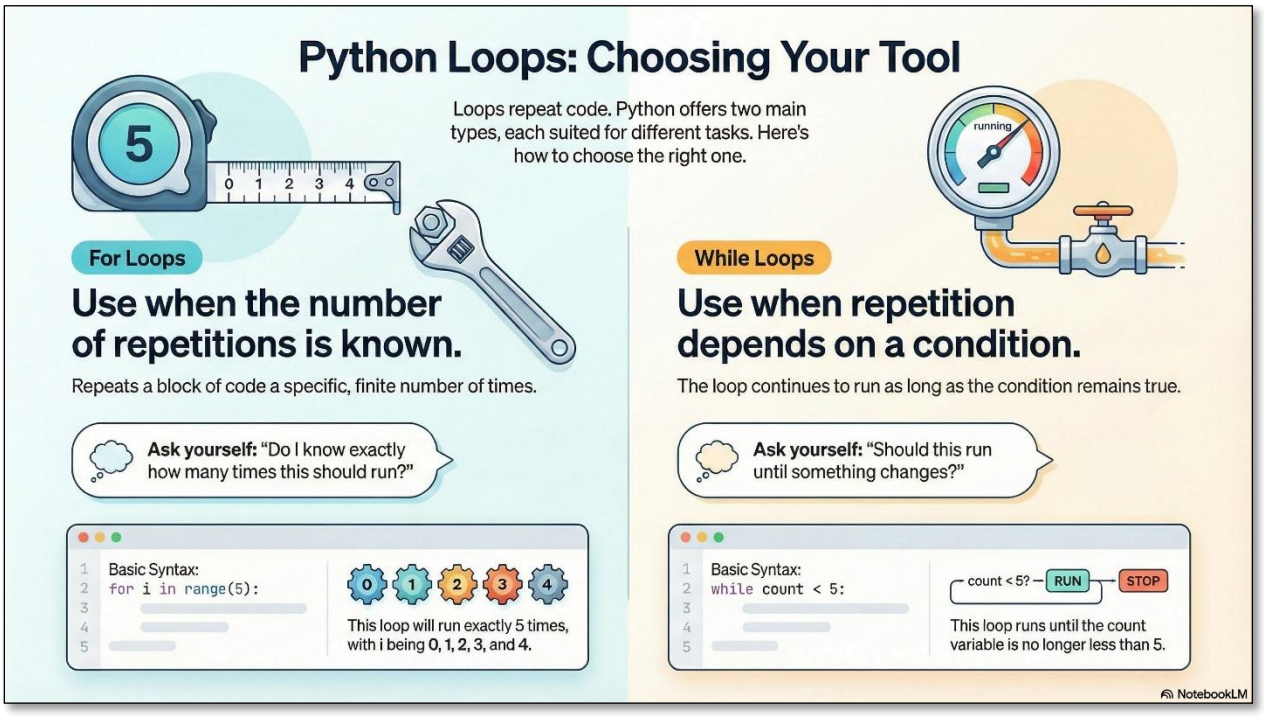
Chapter 4: Python Loops

Contents



Red light: No AI

Time required: 180 minutes

How Does Typing in a Code Tutorial Help with Learning?

Typing in a code tutorial can significantly enhance learning in several ways:

- **Active Engagement**: Typing the code yourself forces you to actively engage with the material, rather than passively reading or watching. This active participation helps reinforce the concepts being taught.
- **Muscle Memory**: Repeatedly typing code helps build muscle memory, making it easier to recall syntax and structure when you write code independently.
- **Error Handling**: When you type code, you're likely to make mistakes. Debugging these errors helps you understand common pitfalls and how to resolve them, which is a crucial skill for any programmer.

- **Understanding**: Typing out code allows you to see how different parts of the code interact. This deeper understanding can help you apply similar concepts to different problems.
- **Retention**: Studies have shown that actively doing something helps with retention. By typing out the code, you're more likely to remember the concepts and techniques.

DRY

Don't **R**epeat **Y**ourself

Learning Outcomes

Students will be able to:

- Write for and while loops
- Choose the correct type of loop for different problems
- Solve problems using loops (sum, max, counting, etc.)
- Write nested loops (loops inside loops)

What are Loops

Loops (also called **iterations**) are used to **repeat code** multiple times.

Computers handle repetitive tasks well—loops allow us to take advantage of that.

Types of Loops in Python

Each of these loop examples print 0-4.

1. for Loop

Used when the number of repetitions is **known or finite**.

```
for i in range(5):
    print(i)  # Prints 0 to 4
```

2. while Loop

Used when the repetition depends on a **condition** being true.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Control Flow

Control flow = how Python decides what to run next. It includes:

- Loops (repetition)
- Functions (modularity)
- Conditionals (decision logic)

Updating Variables in Python

An **update** changes the current value of a variable using an operation like addition or subtraction.

```
x = x + 1
```

This means:

- Get the current value of **x**
- Add 1
- Store the result back into **x**

Important: Initialize First!

You must **assign an initial value** before updating.

```
x = 0        # initialize
x = x + 1    # update (increment)
```

If you skip the first line:

```
x = x + 1    # Error: x is not defined
```

Terminology

Operation	Name
`x = x + 1`	Increment

`x = x - 1`	Decrement

You can also use **shortcuts**:

```
x += 1   # same as x = x + 1
x -= 1   # same as x = x - 1
```

For Loops

A **for** loop repeats a block of code **a specific number of times**.

The following will print **Hello** ten times:

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

How it Works

1. range(10) produces the numbers 0 to 9 (10 values).
2. i is a loop variable—it gets each number from the range one at a time.
3. The body of the loop is indented.
4. After all items in the range are used, the loop ends.

General Structure

```
for variable in range(number):
    # code block (indented)
    # runs 'number' times
```

Notes

- **for** must be lowercase
- A **colon (:)** is required at the end of the for line
- All indented lines **belong to the loop body**

If You Don't Use the Variable

Use **_** as a placeholder if the variable isn't needed.

The flow chart below shows the program flow.

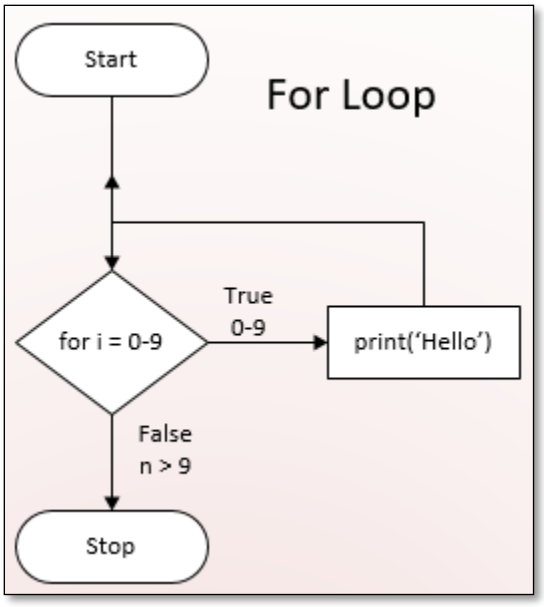


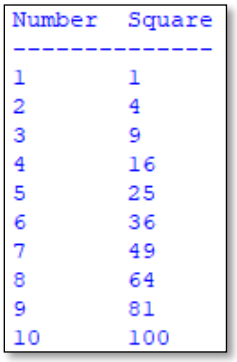
Figure 1 For loop

Here is an example.

Tutorial 4.1: Numbers and Squares

```
"""
    Name: numbers_and_squares.py
    Author:
    Created:
    Purpose:  Use a loop to print the squares of the numbers 1 to 10
"""

# Constant for number of dashes printed
NUMBER_OF_DASHES = 14

# Print the heading
print("Number\tSquare")

# Print dashes using string multiplication
print(NUMBER_OF_DASHES * "-")

# Print the numbers 1-10 and their squares
for number in range(1, 11):

    # Calculate the square of the number
    square = number ** 2

    # Print result using \t to tab
    print(f"{number}\t{square}")
```

Example run:



```
Number  Square
--------------
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
```

The range() Function

Python's **range()** function creates a sequence of numbers. It is commonly used with for loops to repeat actions a specific number of times.

Basic Usage

```
range(stop)
```

- Starts at **0** by default.
- Ends **one less** than the stop value.
- Example:
  range(6) → [0, 1, 2, 3, 4, 5]

With a Start and Stop

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

- Starts at the **start** value.
- Ends **one less** than the **stop** value.
- Example:
  range(1, 5) → [1, 2, 3, 4]

With a Step

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

- Adds a **step** between values.
- Can count forward or backward.

- Example:
  range(1, 10, 2) → [1, 3, 5, 7, 9]
  range(5, 1, -1) → [5, 4, 3, 2]

Common Examples

Statement	Values Generated
`range(10)`	`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
`range(1, 10)`	`1, 2, 3, 4, 5, 6, 7, 8, 9`
`range(3, 7)`	`3, 4, 5, 6`
`range(2, 15, 3)`	`2, 5, 8, 11, 14`
`range(9, 2, -1)`	`9, 8, 7, 6, 5, 4, 3`

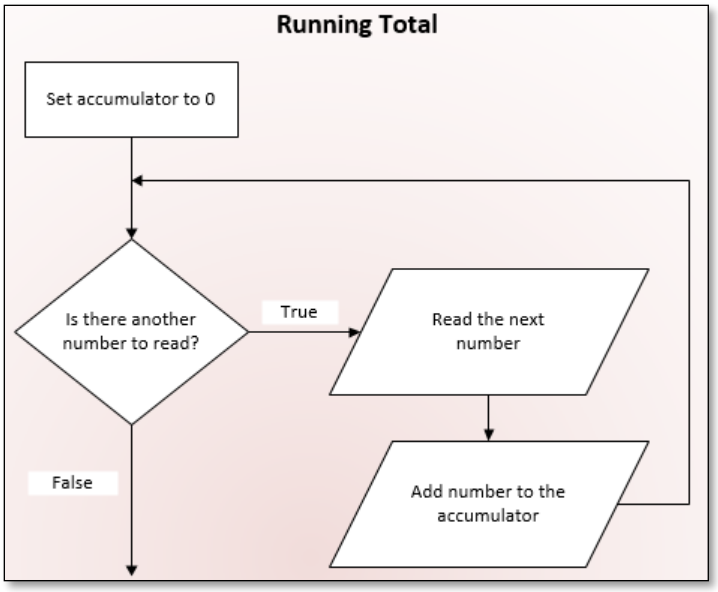
Countdown Example Using sleep()

```
from time import sleep

for i in range(5, 0, -1):
    print(i)
    sleep(1)

print("Blast off!!")
```

Sum and Loop Example (Running Total)

```
sum = 0
for counter in range(5):
    sum += counter
    print(f"counter = {counter} sum = {sum}")
print("=" * 22)
```

- Loops through [0, 1, 2, 3, 4]
- Adds and prints each number with running total



Tutorial 4.2: Running Total

The following program demonstrates how to keep a running total of numbers entered by the user. Save the program as **running_total.py**.

```python
"""
    Name: running_total.py
    Author:
    Created:
    Sum a series of numbers entered by the user
"""

# Constant for the maximum number of user entries
MAX = 5

# Initialize an accumulator variable
running_total = 0

# Explain to the user what the program does
print(f"Let's calculate the sum of {MAX} whole numbers you will enter.")

# Get the numbers and accumulate them into a total
for x in range(MAX):
    # Get the number from the user
    number = int(input("Enter a number: "))

    # Add number input to the total
    running_total = running_total + number
    print(f"Running total: {running_total}")

# Display the running total of the numbers
print(f"The total is {running_total}.")
```

Example run:

```
Let's calculate the sum of 5 whole numbers you will enter.
Enter a number: 10
Running total: 10
Enter a number: 5
Running total: 15
Enter a number: 69
Running total: 84
Enter a number: 2
Running total: 86
Enter a number: 1
Running total: 87
The total is 87.
```

Using Range with Strings

```
breed = "Affenpinscher"
for index in range(1, len(breed), 2):
    print(f"Letter at index {index} {breed[index]}")
```

- Indexes: [1, 3, 5, 7, 9, 11]
- Outputs every second letter, starting from index 1

Key Points

- range() is zero-based and **excludes** the stop value.
- You can control the starting point and the step.
- Useful for controlled iteration over numbers or string indices.

Assignment 4.1: For Loops with the range() Function

Learning Objectives

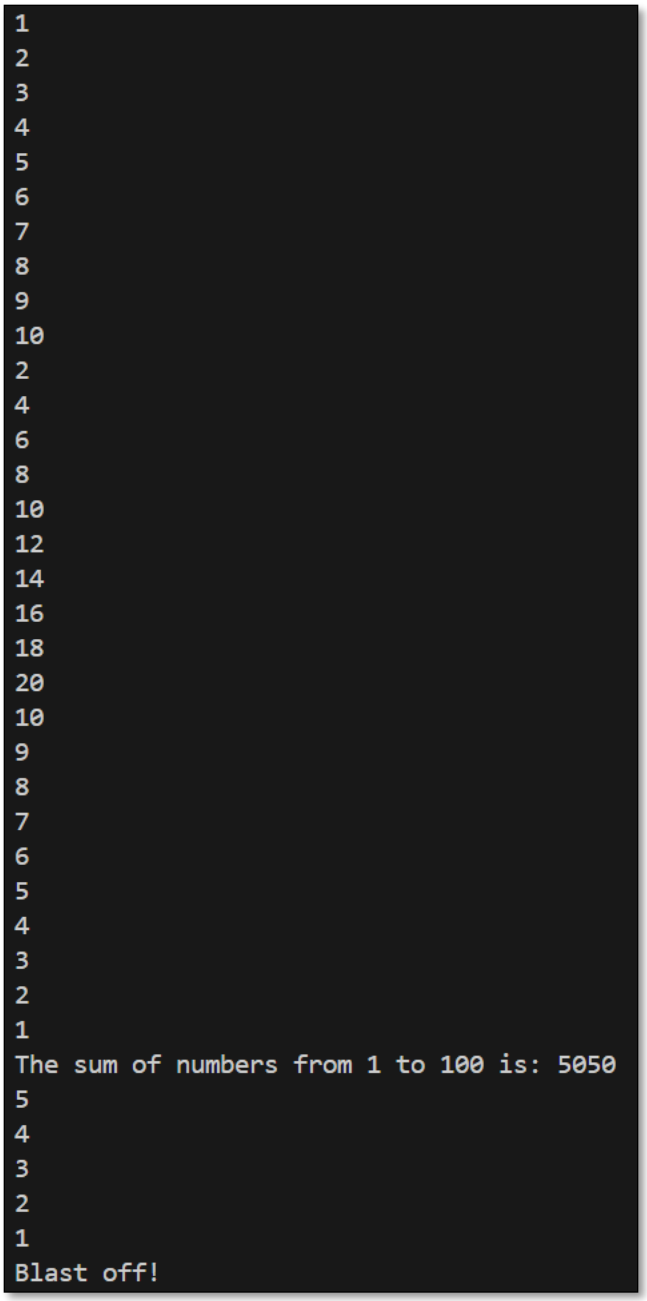
- Use range() with one, two, or three arguments
- Write for loops to count up and down
- Use loops to work with numbers and strings
- Understand loop indexing and output formatting

For Loops

Create a Python program named **for_loops.py**

- Write a loop that prints numbers from 1 to 10 (inclusive).
- Write a loop that prints even numbers from 2 to 20 using a step value.
- Write a loop that prints numbers in reverse from 10 down to 1.
- Write a loop that adds all numbers from 1 to 100 and prints the total sum.
- Use the sleep() function to create a countdown from 5 to 1.
    - After each number, pause the program for 1 second.
    - At the end, print "Blast off!".

Example run:



Assignment 4.2: String Index Loop

Create a program that:

1. Asks the user to enter a word.
2. Uses a for loop and range() to:
    - len(word) will get the length of the word.

- Print every other character (index 0, 2, 4…)
- Print the index and the character

Example run:

```
Enter a word: Bill
Index 0 = B
Index 2 = l
```

Tutorial 4.3: And Now for Something Completely Different

There are a couple of ways to dress up your CLI (command line interface) programs..

You can use this web site to put your name in ascii art into your Python program.

https://www.patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20

1. Copy the ascii characters into a Python file.
2. Some of the fonts don't work with particular words. The example shown is with the Doom font.
3. Don't forget to add the r (raw string) where is it shown. The code will look a little funny because of VSCode's highlighting. It will print out just fine.

```
python = r"""
______       _   _
| ___ \     | | | |
| |_/ /   _| |_| |__   ___  _ __
|  __/ | | | __| '_ \ / _ \| '_ \
| |  | |_| | |_| | | | (_) | | | |
\_|   \__, |\__|_| |_|\___/|_| |_|
       __/ |
      |___/
"""
print(python)
```

Example run:

There is a Python library called **pyfiglet**.

```
# Run the following command at a command prompt.
pip install pyfiglet
# Run this to see all the fonts available.
pyfiglet -f
```

```
1   # Import figlet_format from pyfiglet for creating ASCII art
2   import pyfiglet
3
4   msg = "Bill"
5
6   result = pyfiglet.figlet_format(msg, font = "stick_letters")
7
8   print(result)
```

Example run:

While Loops

Why Use a while loop?

Use a while loop when:

- You **don't know ahead of time** how many times something should repeat.
- The loop depends on a **condition** staying true.

Syntax:

```
while condition:
    # indented code block
```

- The loop runs **as long as the condition is True**.
- When the condition becomes **False**, the loop stops.

Example: Counting from 1 to 5

```
num = 1
while num <= 5:
    print(num)
    num += 1
print("Done")
```

Loop Explanation:

- Starts at 1
- Prints numbers while num <= 5
- Stops after printing 5
- Prints "Done"

Example: Countdown with sleep()

```
from time import sleep

n = 5
while n > 0:
    print(n)
    n -= 1
    sleep(1)
print("Blastoff!")
```

This pauses the program 1 second between prints.

How a While Loop Works (Flow)

1. Check condition
2. If **True**, run the body
3. Update variable (called the **iteration variable**)
4. Go back to step 1

5. If **False**, exit the loop

Each run of the loop is called an **iteration**.

Avoid Infinite Loops

Make sure your condition eventually becomes False!

```
# This will loop forever!
while True:
    print("This never ends...")
```

Assignment 4.3: While Loops

Learning Objectives

- Write **while** loops to perform repetition
- Control loops with **Boolean** conditions
- Avoid infinite loops
- Use iteration variables and **sleep()** from **time**

Create a Python program named **while_loops.py**

Part 1: Count Up

Write a program that:

1. Starts at 1
2. Uses a while loop to count up to 10
3. Prints each number on a new line

Part 2: Input Validation

Write a program that:

1. Asks the user to enter a **positive number**
2. If the number is not positive, keep asking using a while loop
3. When a valid number is entered, print "Thank you!"

Part 3: Exit a Loop Based on User Input

1. Asks the user to type “exit” to end the loop

2. If they type anything else, repeat the question
3. When they type “exit”, print "Goodbye!"

Example run:

```
Counting from 1 to 10 using a wh
1
2
3
4
5
6
7
8
9
10
Enter a positive number: -1
That’s not positive. Try again.
Enter a positive number: 3
Thank you!
Type 'exit' to quit: ex
Type 'exit' to quit: exit
Goodbye!
```

Augmented Assignment Operators & Sentinel Loops

What is an Augmented Assignment Operator?

Instead of writing:

```
x = x + 2
```

You can write:

```
x += 2
```

Here’s a list of common augmented assignment operators:

Operator	Example	Equivalent To
+=	x += 10	x = x + 10
-=	y -= 3	y = y - 3
*=	z *= 4	z = z * 4

/=	a /= b	a = a / b
%=	c %= 3	c = c % 3
//=	x //= 3	x = x // 3
**=	y **= 2	y = y ** 2

Tutorial 4.4: Running Total with User Input

The code block below shows how to use a while loop to allow the user to enter numbers as long as they want, until they enter a zero. Once a zero is entered, the total is printed, and the program ends. This is called a sentinel value.

```
# Declare running total variable
running_total = 0

# Get input from the user before the loop
# This allows the input before the while loop test
num = int (input ( "Enter a number: "))

# Keep looping until the user enters 0
while num != 0:
    # Add up the running total
    running_total += num
    print(f"Running total: {running_total}")

    # Get another number from the user
    num = int (input ( "Enter a number: "))

# Display the running total
print(f"Total is {running_total}" )
```

Example run:

```
Enter a number: 5
Running total: 5
Enter a number: 10
Running total: 15
Enter a number: 0
Total is 15
```

Assignment 4.4: Augmented Operators

Learning Outcomes:

- Use +=, -= and other augmented operators
- Create while loops with iteration counters
- Design sentinel-controlled input loops

Create a Python program named **augmented_operators.py**

Part 1: Sum 5 Numbers

- Use a while loop and += to add 5 user-entered numbers and print the total.

Part 2: Multiplying Score

- Start with a variable at 1
- Use *= in a loop to multiply the user's number by itself 4 times (loop runs 4 times).
- Print the final result.

Part 3: Sentinel Input

- Keep asking the user for their favorite number until they enter 0.
- For every number they enter, print "Great choice!"

Example run:

```
Enter a number: 3
Enter a number: 5
Enter a number: 6
Enter a number: 3
Enter a number: 2
Total is 19
Enter a number to multiply: 25
Enter a number to multiply: 3
Enter a number to multiply: 1
Enter a number to multiply: 25
The multiplied result is 1875
Enter your favorite number (0 to quit): 42
Great choice!
Enter your favorite number (0 to quit): 0
Thanks for playing!
```

The break Statement

The break statement is used to **exit a loop early**, even if the loop condition is still True.

Why use break?

Sometimes, we don't know **exactly when** a loop should stop. The break statement allows your program to **stop the loop when a condition is met**.

For Loop with break

```
# Loop up to 10 times
for i in range(10):
    num = int(input("Enter a number: "))
    # Exit early if number is negative
    if num < 0:
        break
```

This loop allows up to 10 inputs, but will stop **early** if the user enters a negative number.

While Loop with break

```
i = 0

while i < 10:
    num = int(input("Enter a number: "))
    if num < 0:
        break
    i += 1
```

This version uses a while loop, but the effect is the same: it exits early if the input is negative.

Finishing Iterations with Continue

Purpose

Skip the **rest of the current loop iteration** and move immediately to the next one.

How it works

- When Python hits a continue, it **ends the current iteration early**.
- The loop jumps back to the start for the **next iteration**.

Example: Print Even if the number is even, otherwise print nothing from user input

```
while True:
    num_str = int(input("Enter a positive number (or 0 to end): "))
    if num_str == 0:
        break
    if num % 2 != 0:
        continue  # Skip odd numbers
    print(f"Even number: {num}")
print("Finished.")
```

Assignment 4.5: Temperature Converter with a Loop

Objective

Modify your temperature converter program to allow multiple temperature conversions without restarting the program. Use a while loop to repeat input and conversion until the user chooses to stop.

Requirements

1. Open your existing temperature converter program. Save it as **temperature_while.py**
2. **Add a while loop** to allow the user to enter temperatures repeatedly.
3. Use a **sentinel value** (e.g., -1000) for the user to indicate when they want to stop converting temperatures.
4. **Display the results** in the format:
   -41°F is -41°C
   Use the degree symbol (°). You can copy it from www.degreesymbol.net.
5. **Initialize** your temperature variable (e.g., fahrenheit = 0.0) before the loop starts to avoid a NameError.
6. Inside the loop:
   - Prompt the user to enter a temperature in Fahrenheit.
   - If the entered temperature is the sentinel value (e.g., -1000), exit the loop.
   - Otherwise, convert the temperature to Celsius and display the result.
   - Repeat.

Example run:

```
Enter a Fahrenheit temperature (9999 to quit): -40

-40.0°F is equal to -40.00 °C
-40.0°F is equal to 233.15 K

Enter a Fahrenheit temperature (9999 to quit): 32

32.0°F is equal to 0.00 °C
32.0°F is equal to 273.15 K
```

Notes

- Think of the while condition as "keep looping while the input is **not** the sentinel value."
- Initializing your variable before the loop allows the condition to be checked the first time.
- The loop repeats until the user signals they are done.

Debugging with Bisection

Why It Matters

As your programs grow, **bugs become harder to find**. Checking each line one-by-one is slow.
A smarter approach is to **divide and conquer**—called **debugging by bisection**.

What is Bisection Debugging?

Instead of checking all 100 lines of code:

1. **Go to the middle** of your program or process.
2. Insert a check, such as a print() statement or a test value.
3. Run the program

What the Check Tells You

- If the result **looks wrong**, the bug is likely **before** your check.
- If the result **looks correct**, the bug is likely **after** your check.

You have now cut your search area **in half**.

Repeat the process. After just a few steps, you'll narrow the issue down to 1–2 lines.

Key Idea

- You don't need to count lines or find the exact middle.
- Instead, choose a point where:
    - The logic shifts,
    - A variable should have a known value,
    - Or output should match expectations.

Example

Let's say a program calculates the total cost of an order but the final number is wrong. Add checks like:

```
print("Subtotal:", subtotal)
print("Tax:", tax)
print("Total:", total)
```

- If subtotal is wrong → problem is earlier.
- If subtotal is right, but total is wrong → problem is later.

Tip

Use print() statements, comments, or logging to track variables and flow.

Remove or comment them out once the bug is fixed.

Summary

- Bisection debugging saves time.
- Add checks at logical breakpoints.
- Narrow the problem by halves instead of searching line-by-line.

Assignment 4.6: Bisection Debugging Practice

Objective

Use **debugging by bisection** to locate and fix a logic error in a broken program.

Instructions

You are given a Python program **debug_average.py** that calculates the average test score for a student. The program runs but gives the **wrong result**. Your job is to **locate the bug**, explain where it is, and fix it.

1. Copy and paste the broken code below into a file called debug_average.py.
2. Run the program and observe the incorrect output.
3. Use **print statements** at logical points to trace the values.
4. Apply the **bisection debugging technique** to narrow down where the problem occurs.
5. Fix the bug.
6. Answer the questions in the **Reflection** section.

Broken Code (Don't fix yet — test it first)

Copy and paste this code into VSCode to run it. Name it **debug_average.py**

```
total = 0
count = 0

scores = [80, 90, 85, 100, 70]

for score in scores:
    total = score  # Should add score to total
    count += 1

average = total / count
print("Average score:", average)
```

Reflection Questions (Add your answers as comments at the bottom of your file)

1. What did you expect the output to be?
2. What was the actual output?
3. Where did you place your first print() check?
4. How did you narrow down the location of the bug?
5. What change did you make to fix the bug?

Deliverables

- Submit your corrected **debug_average.py** file.

- Include your reflection answers as **Python comments** at the end of the file.

Glossary

accumulator A variable used in a loop to add up or accumulate a result.

counter A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to "count" something.

decrement An update that decreases the value of a variable.

initialize An assignment that gives an initial value to a variable that will be updated.

increment An update that increases the value of a variable (often by one).

infinite loop A loop in which the terminating condition is never satisfied or for which there is no terminating condition.

iteration Repeated execution of a set of statements using either a function that calls itself or a loop.

Assignment Submission

1. Attach the pseudocode or create a TODO.
2. Attach all tutorials and assignments.
3. Attach screenshots showing the successful operation of each tutorial program.
4. Submit in Blackboard.