Decide and Repeat: Controlling Program Flow

Applied Python Programming with AI and Raspberry Pi Interfaces

Instructor: Dr. Vikas Thammanna Gowda Semester: ABCD 20YX

Module Overview: *This module introduces students to essential control flow mechanisms in Python, enabling programs to make decisions and repeat actions based on conditions. It begins with **conditional branching**, covering `if`, `if-else`, `if-elif`, and nested `if` structures, supported by real-world analogies and examples. The module then explores **built-in functions** and **importing libraries**, with a focus on the `random` module for practical applications. Finally, it examines **infinite loops** using `while True`, highlighting their use cases, best practices, and safe termination with `break`. Through hands-on examples, learners develop the ability to write responsive, reusable, and efficient code that reacts to dynamic input and events. By the end, students can apply structured decision-making and looping techniques to solve a wide range of programming problems.*

Learning Objectives:

- Understand and implement conditional statements (`if`, `if-else`, `if-elif`, nested `if`).
- Apply built-in Python functions effectively for input handling, type conversion, and computation.
- Import and use standard libraries, with emphasis on the `random` module.
- Design and implement infinite loops using `while True` with controlled exit conditions.
- Apply best practices for structuring control flow in real-world programming scenarios.

Contents

Chapter 1

Conditional Branching

In programming, we often need to make decisions: "If it's raining, take an umbrella; otherwise, enjoy the sunshine." This basic example shows a condition (raining or not) that determines which action to take. In Python, `if-else` statements let your program choose between different actions based on whether a condition is true or false. This ability to control the flow of your code makes your programs dynamic and responsive. Without conditional branching, programs would simply run straight through every line the same way each time, regardless of any changing circumstances or input.

Conditional statements in Python come in a few flavors. The main types of conditional constructs are:

- **if statement** – a single condition and block.
- **if-else statement** – one condition, two alternative blocks (one if true, one if false).
- **if-elif statement** (else-if ladder) – multiple conditions checked in sequence.
- **Nested if-else** – an if or if-else structure inside another if or else block.

In the following sections, we will explore each of these in detail with syntax and examples.

1.1 if Statement

Syntax

```
if ( condition ):
    <statement block>
```

The Anatomy:

- **if** – keyword that begins the conditional statement.
- **condition** – an expression that evaluates to True or False (boolean context).
- **colon** (`:`) – marks the start of the indented block following the condition (mandatory in Python syntax).
- **statement block** – one or more indented lines that execute only if the condition is True.

An `if` statement is used to run a block of code conditionally. In other words, if the given condition is `True`, then the statements inside the `if` block will be executed. If the condition is `False`, the inner block is skipped entirely and the program continues after the `if` statement. **Note:** Unlike some languages, Python does not require the condition to be enclosed in parentheses; we include parentheses here only for clarity. It's the indentation (and the colon) that actually defines the block.

1.1.1 Illustration

Example 1

```
temperature = 85

if (temperature > 80):
    print("It's hot outside!")
```

Output

```
It's hot outside!
```

- Here, `temperature > 80` is the condition being tested.
- The condition translates to `85 > 80`, which is `True`.
- Because the condition is True, the indented `print` line runs, producing the output.
- If the condition were False (for example, if `temperature = 75`, making `75 > 80` false), then the print statement would be skipped and nothing would be output.

1.2 if-else Statement

Syntax

```
if (condition):
    <statement block 1>
else:
    <statement block 2>
```

The Anatomy:

- **if** – keyword that introduces the condition to test.
- **condition** – an expression that evaluates to True or False.
- **colon** (`:`) – required after the condition, marking the start of the if-block.
- **statement block 1** – one or more indented lines that run if the condition is True.
- **else** – keyword that introduces the alternative branch.
- **colon** (`:`) – follows the `else` and marks the start of the else-block.

- **statement block 2** – one or more indented lines that run if the condition is False.

An `if-else` statement is used to run one of two blocks of code: one block executes if the given condition is `True`, and the other block executes if the condition is `False`. Exactly one of the two blocks will run when the statement is encountered. In summary, if the condition is True, the `else` block is skipped; if the condition is False, the `if` block is skipped and the `else` block runs.

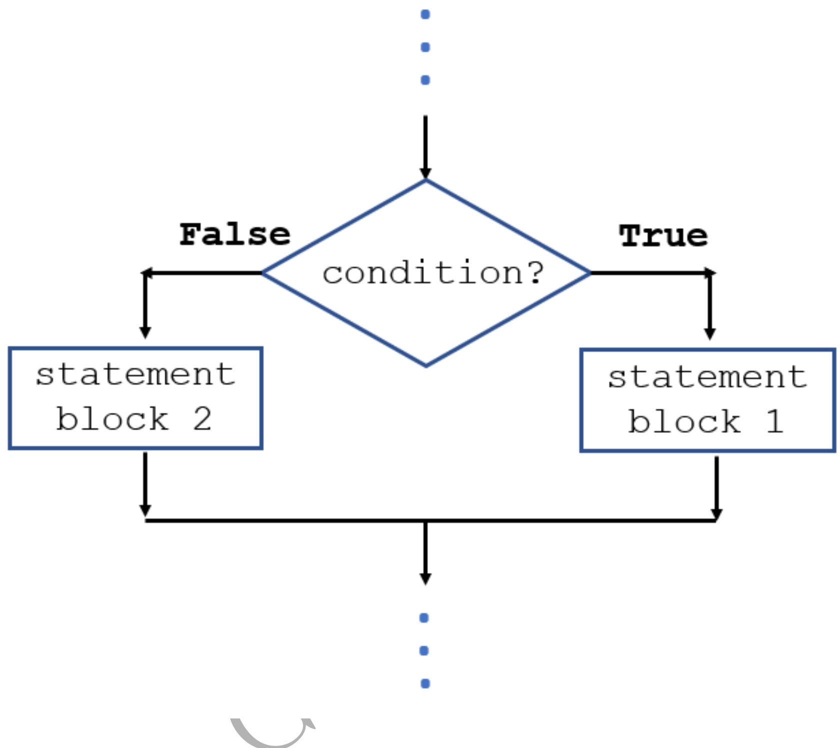

Figure 1.1: Flow chart for if else

1.2.1 Illustration

Example 2

```
temperature = 75

if ( temperature > 80):
    print("It's hot outside !")
else:
    print("The weather is comfortable.")
```

Output

```
The weather is comfortable.
```

- Here, `temperature > 80` is the condition being evaluated.
- The condition translates to `75 > 80`, which is `False`.

- Because the condition is False, the `else` branch executes and prints the message under `else`.

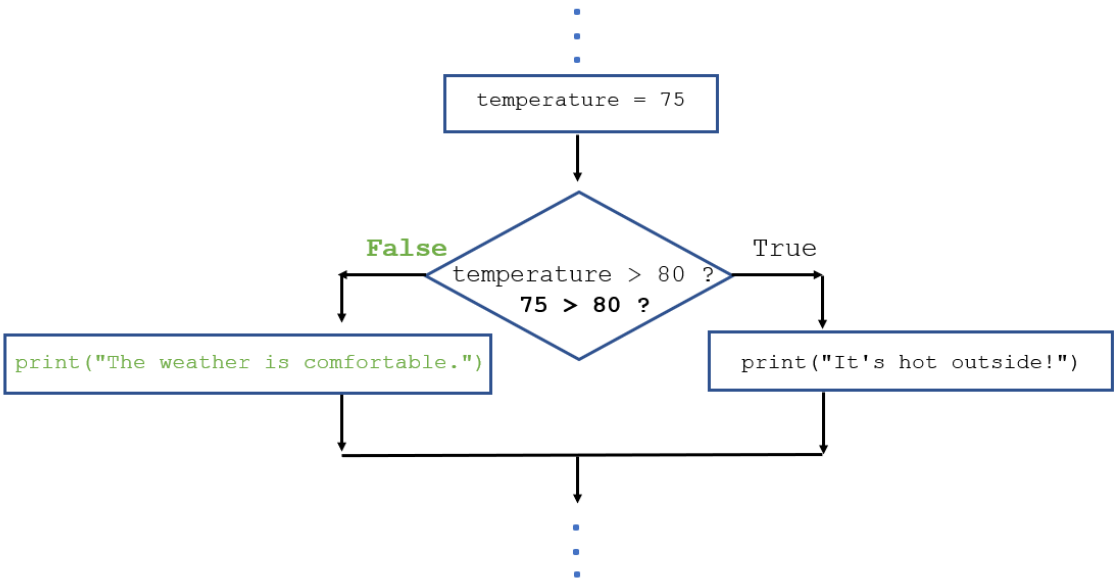

Figure 1.2: Flow chart for Example 2

Example 3

```
temperature = 85

if ( temperature > 80):
    print("It's hot outside !")
else:
    print("The weather is comfortable.")
```

Output

```
It's hot outside!
```

- Here, the condition `temperature > 80` translates to `85 > 80`, which is `True`.
- Since the condition is True, the `if`-block's print statement executes.
- The `else` block is skipped in this case.

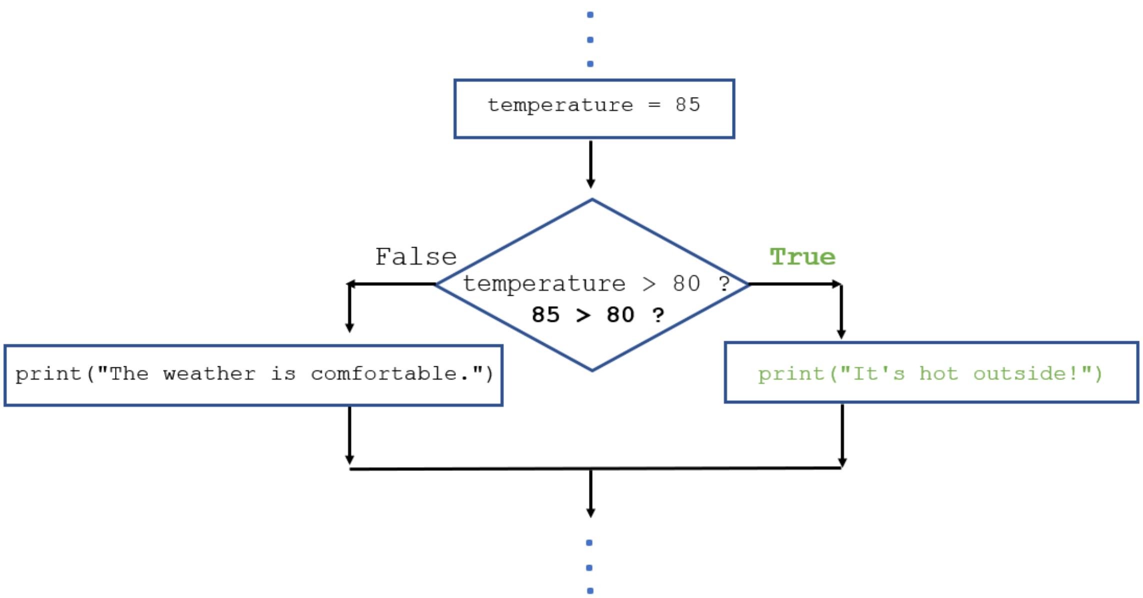

Figure 1.3: Flow chart for Example 3

1.3 if-elif Statement

Syntax:

Syntax

```
if ( condition 1 ):
    <statement block 1>
elif ( condition 2 ):
    <statement block 2>
elif ( condition 3 ):
    <statement block 3>
...
else:
    <statement block for "none of the above">
```

The `if-elif-else` structure is used when you have more than two possible paths to choose from. It checks multiple conditions one by one:

- Python evaluates each condition in order from top to bottom.
- As soon as one condition is found to be `True`, its associated block executes, and the rest of the conditions are not checked (the chain of checks stops there).
- If none of the conditions is True, then the block under the final `else` (if an `else` is provided) will execute.
- You can include any number of `elif` ("else if") clauses between the initial `if` and the final `else`. The `else` clause is optional, but if present, there can be only one and it must come last.

1.3.1 Illustration

Example 4

```
temperature = 50

if ( temperature > 80 ):
    print("It's hot!")
elif ( temperature >= 60 ):
    print("It's nice and warm.")
elif ( temperature >= 40 ):
    print("It's a bit chilly.")
else:
    print("Brrr... it's cold!")
```

Output

```
It's a bit chilly.
```

- First, the condition `temperature > 80` is checked. Here that means `50 > 80`, which is `False`, so the program moves on.
- Next, the condition `temperature >= 60` is evaluated. `50 >= 60` is also `False`, so it continues to the next condition.
- Then, the condition `temperature >= 40` is checked. `50 >= 40` is `True`.
- Because this third condition is True, the corresponding `print` statement runs: `"It's a bit chilly."` is printed.
- Once a True condition is found and its block executes, the remaining conditions (in this case, the `else` that follows) are skipped.
- In this example, the `else` was not reached because one of the `elif` conditions succeeded. If the temperature had been lower (say 30, making all the above conditions False), then the final `else` block would have executed, printing `"Brrr... it's cold!"`.

1.4 Nested if-else

Sometimes you may want to check a secondary condition only if a primary condition is True (or only if it is False). In Python, you can put an `if` (or even an entire `if-else` structure) inside another `if` or `else` block to handle more complex logic. This is known as nesting of control structures. Nested `if` statements allow you to further refine decisions after one decision has been made.

1.4.1 Illustration

Example 5

```
score = 85

if ( score >= 60 ):
    if ( score >= 90 ):
        print("Grade: A")
    else:
        print("Grade: B, C, or D depending on exact score.")
else:
    print("Student failed.")
```

Output

```
Grade: B, C, or D depending on exact score.
```

- The outer `if` checks if the student passed the exam (`score >= 60`).
- The condition `85 >= 60` is `True`, so the program enters the outer if-block.
- Inside the outer block, there is another `if` that checks for an A grade (`score >= 90`).
- This inner condition `85 >= 90` is `False`, so the inner `else` block executes.
- The inner `else` prints `"Grade: B, C, or D depending on exact score."`
- Since the outer `if` condition was True, the outer `else` (the "Student failed." message) is skipped entirely.
- Notice that the indentation determines which `else` corresponds to which `if`. In this example, the second `else` is indented to align with the inner `if`, so it pairs with that inner `if`. The outer `if` has its own `else` aligned with it.
- This nested logic first tests whether the student passed at all, and only if so, then it checks what grade range the score falls into. (If `score` had been 92, the inner `if` would have printed `"Grade: A"`. If `score` were 50, the outer `else` would have printed `"Student failed."`)

Note: The above nested structure could alternatively be written using an `elif` in a single `if-elif-else` chain (for example, checking `score >= 90` as an `elif` on the outer `if`). Both approaches produce the same result here. In practice, you should choose the approach that makes the program logic clearer.

1.5 Key Points for Conditional Statements in Python

- **Indentation matters**: Python uses indentation (whitespace at the beginning of lines) to group statements. All statements inside an `if`, `elif`, or `else` block must be indented to the same level. Typically we use 4 spaces for each indentation level. Incorrect indentation (or forgetting to indent at all) will cause a syntax error, because Python won't know which statements belong to the `if` block.

- **Boolean conditions**: The conditions in `if/elif` statements must evaluate to either `True` or `False`. In Python, any expression can serve as a condition. Non-zero numbers and non-empty objects are treated as *True*, while zero, `None`, and empty sequences/collections are treated as *False*[1]. (The built-in function `bool()` can be used to test the truth value of a given expression.)
- **Comparison operators**: Use `==` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) to form conditions. For example, `x < 5` or `score >= 90`. *Remember not to confuse the assignment operator* `=` *with* `==` *for comparison.*
- **Logical operators**: You can combine multiple conditions using `and`, `or`, and use `not` to negate a condition. For example, `if (x > 0 and y > 0):` checks that *both* `x` and `y` are positive. These operators also short-circuit (e.g., in an `and` expression, if the first condition is False, the second is never evaluated).
- **Flow of execution**: When an `if` (or `if-elif-else`) statement is executed, the conditions are evaluated in order:
  1. Evaluate the `if` condition.
  2. If it is `True`, execute its block and skip all remaining `elif/else` parts.
  3. If it is `False`, move to the first `elif` (if present) and evaluate that condition. Continue down the chain of `elif` conditions until one is `True` or you run out of conditions.
  4. If an `elif` condition is found to be `True`, execute its block and skip the rest of the `elif/else` parts.
  5. If none of the `if` or `elif` conditions are True, then execute the `else` block (if an `else` is provided). If there is no `else` and all conditions are False, then nothing inside this `if`-statement executes, and the program continues after the entire if-structure.

[1] In Python, truthiness follows these general rules: constants defined to be false (like `False` and `None`), zero of any numeric type (e.g., `0`, `0.0`, `0j`), and empty sequences or collections (such as `''`, `[]`, {}, etc.) are all considered `False`. Nearly everything else is considered `True`.

Chapter 2

Built-in Functions and Libraries

2.1 Introduction

Definition: A function is a named, reusable block of code that performs a specific task. You *define* it once and *call* it wherever you need that task.

Real-world analogies.

- *Coffee machine:* Press a button (call), it runs an internal routine (body), and returns coffee (result).
- *Math formula:* Supply inputs (parameters), compute, and receive an answer (return value).
- *RPS (Rock–Paper–Scissors) round:* Given two choices, determine the outcome.

Why use functions?

1. **Modularity:** Break large programs into small, focused pieces (e.g., input handling, decision logic).
2. **Reusability:** Write once, use many times (e.g., a scoring function used across multiple game modes).
3. **Readability & Maintainability:** High-level code reads like a plan of action; changes are localized.
4. **Testability:** Small functions are easier to verify with unit tests.

2.2 Built-in Functions

Built-in functions are provided by Python and available without importing any module. They follow the pattern:

Syntax

```
output = function_name(arguments)
```

They often *return* a value to be used by your code (different from just printing to the screen).

Core Built-ins (with examples)

Example 1

```
# print: display text
print("Hello, world!")  # -> Hello, world!

# type: inspect the type of a value
print(type(42))         # -> <class 'int'>
print(type("hi"))       # -> <class 'str'>

# len: length of a container (string, list, etc.)
print(len("paper"))     # -> 5
print(len([10, 20, 30]))# -> 3

# int, float, str: type conversions (casting)
print(int("123"))       # -> 123
print(float("3.14"))    # -> 3.14
print(str(42))          # -> "42"

# abs: absolute value
print(abs(-7))          # -> 7

# round: round a number; optional ndigits
print(round(3.14159, 2))# -> 3.14
print(round(2.5))       # -> 2 or 3 (banker's rounding)

# min, max, sum: aggregate operations
print(min(4, 9, 2))     # -> 2
print(max([3, 8, 5]))   # -> 8
print(sum([1, 2, 3, 4]))# -> 10

# sorted: returns a new sorted list
print(sorted([3, 1, 2]))# -> [1, 2, 3]

# range: produces an arithmetic progression (often used in loops)
print(list(range(3)))   # -> [0, 1, 2]
print(list(range(2, 7, 2))) # -> [2, 4, 6]

# help: view documentation (docstring) for a function
# help(len)   # uncomment to see interactive help in a console
```

Key clarifications

- **Printing vs. Returning:** `print()` displays to the screen; it does not return a useful value. Many other built-ins *return* values that your program can use.
- **`input()`:** Returns a `str`. Convert to `int` or `float` as needed:

Example 2

```
age = int(input("Enter age: "))  # may raise ValueError if input
    isn't numeric
```

2.3 Importing Libraries (Modules)

A **module** is a file that groups related functions, classes, and constants. Python's *standard library* includes many useful modules you can load with `import`.

2.3.1 Import Syntax

Example 3

```python
# 1) Import the whole module; use its namespace prefix:
import random
x = random.randint(1, 6)

# 2) Import specific names into the current namespace:
from random import randint, choice
y = randint(1, 6)
z = choice(["Rock", "Paper", "Scissors"])

# 3) Import with an alias (shorter prefix):
import random as rnd
w = rnd.randrange(3)
```

Best practices

- Place imports at the top of the file.
- Prefer explicit imports over wildcard (`from module import *`) to avoid name conflicts.
- Use aliases (`as`) for long module names when it improves readability.

2.4 The `random` Module

The `random` module provides functions for pseudo-random numbers and selections—handy for simulations, games (like RPS), and sampling.

2.4.1 Common `random` Functions

Function	Description
`random()`	Float in $[0.0, 1.0)$
`randint(a,b)`	Integer in $[a, b]$ (inclusive)
`randrange(stop)`	One of {`0,...,stop-1`}; also `randrange(start, stop, step)`
`choice(seq)`	Random element from a non-empty sequence
`shuffle(x)`	Shuffle list `x` in place
`sample(pop, k)`	New list of `k` unique elements from `pop`
`uniform(a,b)`	Float in $[a, b]$
`seed(n)`	Initialize generator for reproducible results

2.5 Common Pitfalls and Tips

- **Printing vs. Returning:** Functions that compute values should usually *return* them; use `print()` only for user-facing output.
- **Type conversion:** `input()` returns a `str`. Convert to `int`/`float` as needed and handle bad input (e.g., with `try`/`except`).
- **Off-by-one with ranges:** Remember `range(stop)` excludes `stop`. Similarly, `randrange(start, stop)` excludes `stop`.
- **Avoid wildcard imports:** Prefer `import random` or `from random import choice`.
- **Use help:** `help(random)` or `help(random.choice)` to see built-in documentation.

Chapter 3

Infinite Loops

3.1 Motivation: Why Intentional Infinite Loops?

An intentional infinite loop is a deliberate design technique for problems where:

Syntax

```
while True:
        # do something
```


1. **The number of iterations is unknown**: We cannot determine a precise loop bound before starting (e.g., “keep asking until the user gives valid data”).
2. **The stop condition arises inside the loop**: Whether to stop depends on the *result* of work done during the iteration (e.g., the success/failure result of a network call).
3. **The program is reactive or service-like**: Servers, GUIs, and games are often structured as loops that process events until a shutdown signal appears.
4. **Sensor readings**: In IoT, you may continuously monitor the sensors. (e.g., radiation sensor at a nuclear plant).
5. **Game and UI Loops**: Video games and user interfaces often run a main loop that continuously updates the game state, renders graphics, processes user input, and manages other game logic until the user exits the game or the program is terminated.
6. **Polling or monitoring**: Continuously check for new data, events, or changes in state (e.g., checking for new messages in a chat application).
7. **Event-driven systems**: Wait for events (like user input or network messages) and process them as they arrive, often using a loop that runs until a specific exit condition is met.
8. **Long-running background tasks**: Some applications need to run continuously in the background, performing periodic checks or updates (e.g., a monitoring service that checks system health).
9. **Real-time data processing**: Continuously process incoming data streams (e.g., reading sensor data, processing log files) until a stop condition is met

Example 1

```
from gpiozero import LED
from time import sleep

led = LED(17)

try:
    while True:
        led.on()                # Turn LED on
        sleep(1)                # Wait 1 second
        led.off()               # Turn LED off
        sleep(1)                # Wait 1 second
except KeyboardInterrupt:
    print("\nExiting")
```

- `from gpiozero import LED`
  Imports the `LED` class from the `gpiozero` library.
- `from time import sleep`
  Imports the `sleep()` function from Python's built-in `time` module.
- `led = LED(17)`
  Initializes an LED object connected to GPIO pin 17.
- **`while True:`**
  This is an **infinite loop** that will keep running until the user stops the program. It ensures the LED turns on and off repeatedly.
- `led.on()`
  Turns the LED on (sends power to the pin).
- `sleep(1)`
  Pauses the program for 1 second while the LED remains on.
- `led.off()`
  Turns the LED off (stops sending power to the pin).
- `sleep(1)`
  Pauses the program for 1 second while the LED is off.
- `except KeyboardInterrupt:`
  Detects when the user presses `Ctrl+C` to stop the program.
- `print("\nExiting")`
  Prints a message and exits cleanly after the user interrupts the program.

3.2 Using `while True` with `break`

Placing the stop decision *exactly where you detect it* (via `break`) often yields clearer code than scattering state flags or duplicating checks in multiple places.

Syntax

```
while True:
    # 1) do work that may reveal a stop condition
    if stop_condition:
        break    # 2) exit immediately and cleanly
    # 3) otherwise, loop naturally continues
# execution resumes here (first line after the loop)
```

Key points

- `True` is always true, so the loop itself never becomes false.
- `break` exits the *nearest* loop instantly; control continues after the loop.

Example 1

```
while True:
    s = input("Enter a positive integer (or 'q' to quit): ")
    if s.lower() == "q":
        print("Goodbye!")
        break                      # exit on explicit user intent
    if s.isdigit() and int(s) > 0:
        n = int(s)
        print(f"Thanks. You entered {n}.")
        break                      # exit on success
    print("Invalid input. Please try again.")  # stay in the loop
```

- `while True:` Creates an **infinite loop** — it will run forever unless we explicitly tell it to stop using `break`.
- `s = input("Enter a positive integer (or ’q’ to quit’):")` `input()` waits for the user to type something. Whatever the user types is stored in the variable `s`. The message in quotes is the prompt shown to the user.
- `if s.lower() == "q":` `.lower()` converts what the user typed into lowercase, so `"Q"` and `"q"` are treated the same. If the result equals `"q"`, the program prints `"Goodbye!"` and executes `break`, which immediately ends the loop and stops the program.
- `if s.isdigit() and int(s) > 0:` `s.isdigit()` checks if the input contains only digits (no letters or symbols). `int(s) > 0` checks if that number is positive. If both conditions are true:
  - The string is converted to an integer and stored in `n`.
  - A thank-you message is printed.
  - The loop ends with `break`.
- `print("Invalid input.  Please try again.")` If neither of the above `if` statements was true, the program reaches this line. It tells the user the input was invalid and then loops back to ask again.

3.3 Design Checklist Before Using `while True`

1. **Is the stop rule only knowable inside the loop?** If yes, `while True` is a candidate.
2. **Where exactly should the loop end?** Put the `break` there; avoid distant flag variables when possible.
3. **Do you need multiple exit points?** If so, are they clearly labeled and justified (success vs. failure)?
4. **Could a simple condition-based `while` be clearer?** Prefer simplicity when possible.
5. **Will the loop accidentally spin?** If polling, add delay/backoff; if I/O-bound, prefer blocking calls.