Python Control Statements

1. Introduction

Control statements determine the **order in which instructions execute** in a program. Without them, a Python program runs strictly top to bottom, one statement after another. Control statements let a program make decisions, repeat work, and skip or exit parts of that work.

Python's control statements fall into three categories:

Category	Purpose	Statements
Conditional (decision-making)	Choose which block to execute	if, if-else, if-elif-else, nested if
Looping (iterative)	Repeat a block of code	for, while
Jump (transfer)	Alter the normal flow inside a loop	break, continue, pass

Each statement is explained below with a short theory note, followed by two worked examples. Every example states the **Input** used and the exact **Output** produced when the code is run.

2. Conditional (Decision-Making) Statements

2.1 if Statement

The if statement evaluates a Boolean condition. If the condition is True, the indented block runs; if it is False, the block is simply skipped and execution moves to the next statement. There is no alternative branch.

Example 1 — Voting eligibility

```
age = 20

if age >= 18:

  print("You are eligible to vote")
```

Input: age = 20 **Output:**

You are eligible to vote

2.2 if-else Statement

if-else provides exactly two mutually exclusive branches. The if block runs when the condition is True; otherwise the else block runs. Exactly one of the two blocks always executes.

Example 1 — Even or odd

```
num = 7
if num % 2 == 0:
  print("Even number")
else:
  print("Odd number")
```

Input: num = 7 **Output:**

```
Odd number
```

Example 2 — Pass or fail

```
marks = 35
if marks >= 40:
  print("Pass")
else:
  print("Fail")
```

Input: marks = 35 **Output:**

```
Fail
```

2.3 if-elif-else Ladder

Theory: Used when more than two mutually exclusive conditions must be checked. Python tests each condition from top to bottom and executes the **first** block whose condition is True. If none match, the final else block runs. Only one branch ever executes.

Example 1 — Grade calculator

```
marks = 82
if marks >= 90:
  print("Grade A")
elif marks >= 75:
  print("Grade B")
elif marks >= 60:
  print("Grade C")
else:
  print("Grade D")
```

Input: marks = 82 **Output:**

```
Grade B
```

Example 2 — Sign of a number

```
num = -5
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

Input: num = -5 **Output:**

Negative number

2.4 Nested if Statement

if statement placed inside another if (or else) block. The inner condition is only evaluated when the outer condition has already been satisfied. It is useful when a decision depends on more than one factor at different levels.

Example 1 — Voting eligibility with citizenship check

```
age = 17
citizen = False

if age >= 18:
   if citizen:
      print("Eligible to vote")
   else:
      print("Not a citizen, cannot vote")
else:
   print("Underage, cannot vote")
```

Input: age = 20, citizen = True **Output:**

Eligible to vote

Example 2 — Largest of three numbers

```
a, b, c = 10, 25, 15


if a >= b:
  if a >= c:
    print("Largest:", a)
  else:
    print("Largest:", c)
else:
  if b >= c:
    print("Largest:", b)
  else:
    print("Largest:", c)
```

Input: a = 10, b = 25, c = 15 **Output:**

Largest: 25

3. Looping (Iterative) Statements

3.1 for Loop

The for loop iterates over a sequence. Sequence includes a range, list, tuple, or string. It is the natural choice when the **number of iterations is known** or defined by a collection.

```
for variable in sequence:
  # Statements
```

The range() function generates a sequence of numbers.

Syntax of Range : range(start, stop, step)

```
for i in range(5):
  print(i)
for i in range(2, 8):
  print(i)


for i in range(2, 11, 2):
```

```
    print(i)
```

Sum of a list

```
numbers = [10, 20, 30]
total = 0
for n in numbers:
    total += n
print("Sum:", total)
```

Input: numbers = [10, 20, 30] **Output:**

Sum: 60

Example : Reverse Order

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

Example **:**

```
name = "Python"
for ch in name:
    print(ch)
```

Example:

```
fruits = ["Apple", "Banana", "Mango"]
for fruit in fruits:
    print(fruit)
```

3.2 while Loop

The while loop repeats a block **as long as its condition stays True**. The condition is checked before every iteration, so the loop body may run zero times. It is the natural choice when the **number of iterations is not known in advance**. The loop variable must be updated inside the body to eventually make the condition False; otherwise, the loop never ends.

Example 1 — Countdown

```
count = 5
while count > 0:
    print(count)
    count -= 1
```

Input: count = 5 **Output:**

5

4

3

2

1

Example 2 — Accumulate until a limit is reached

```
n = 1
total = 0
while total < 20:
    total += n
    n += 1
print("Total:", total)
```

Input: n = 1, total = 0 (stop once total reaches 20 or more) **Output:**

Total: 21

(Trace: 1 → 3 → 6 → 10 → 15 → 21; the loop stops as soon as total is no longer less than 20, so the final value slightly overshoots 20.)

4. Jump (Transfer) Statements

4.1 break

break immediately terminates the **nearest enclosing loop** (for or while) and passes control to the statement right after the loop — regardless of whether the loop's own condition is still True.

Example 1 — Stop at the first negative number

```
numbers = [4, 7, -2, 9]
for n in numbers:
    if n < 0:
        break
    print(n)
```

Input: numbers = [4, 7, -2, 9] **Output:**

4

7

Example 2 — Search for a target value

```
target = 5
values = [2, 4, 5, 8]
for v in values:
  if v == target:
    print("Found:", v)
    break
```

Input: target = 5, values = [2, 4, 5, 8] **Output:**

Found: 5

4.2 continue

Continue skips the **rest of the current iteration** only and jumps straight to the next iteration of the loop. Unlike break, the loop itself is not terminated.

Example 1 — Skip even numbers

```
for i in range(1, 6):
  if i % 2 == 0:
    continue
  print(i)
```

Input: range(1, 6) **Output:**

1

3

5

Example 2 — Sum only the positive values

```
numbers = [5, -3, 8, -1, 2]
total = 0
for n in numbers:
  if n < 0:
    continue
  total += n
print("Sum of positives:", total)
```

Input: numbers = [5, -3, 8, -1, 2] **Output:**

Sum of positives: 15

4.3 pass

pass is a **null operation, i**t does nothing. It acts as a syntactic placeholder — Python requires a statement wherever a block is expected, and pass satisfies that requirement while doing nothing. It is commonly used as a temporary placeholder in a function, loop, or conditional branch that will be implemented later.

Example 1 — Placeholder in an if branch

```
num = 10
if num > 0:
    pass  # TODO: handle the positive case later
else:
    print("Non-positive number")
print("Program continues")
```

Input: num = 10 **Output:**

Program continues

Example 2 — Placeholder loop body

```
for i in range(3):
    pass
print("Loop finished without action, i =", i)
```

Input: range(3) **Output:**

Loop finished without action, i = 2

5. match-case Statement (Python 3.10+) — Optional / Advanced

match-case performs **structural pattern matching**, similar to switch-case in languages like C or Java. Python compares the subject value against each case pattern in order and runs the first block that matches; case _ acts as the default, catch-all pattern.

Example 1 — Day of the week

```
day = 3
match day:
    case 1:
        print("Monday")
    case 2:
        print("Tuesday")
```

```
    case 3:
        print("Wednesday")
    case _:
        print("Invalid day")
```

Input: day = 3 **Output:**

Wednesday

Example 2 — Command dispatch

```
command = "start"
match command:
    case "start":
        print("Starting process")
    case "stop":
        print("Stopping process")
    case _:
        print("Unknown command")
```

Input: command = "start" **Output:**

Starting process

assert, range(), and exit()

assert Statement

assert is a debugging aid. It checks that a condition is True at a specific point in the program; if the condition is False, Python immediately raises an AssertionError and stops execution — optionally with a custom message explaining what went wrong. It is used to verify assumptions the programmer believes must always hold, not to validate ordinary user input (which should be handled with regular if-checks or exceptions instead).

- Syntax: assert condition, "optional error message"
- If condition is True — nothing happens, execution continues normally
- If condition is False — raises AssertionError and stops the program
- Commonly stripped out in optimized (-O) production builds, so it should never replace real input validation

Example 1 — A condition that passes

```
x = 10
assert x > 0
print("x is positive")
```

Input: x = 10

```
x is positive
```

Example 2 — A condition that fails, with a custom message

```
age = -5
assert age >= 0, "Age cannot be negative"
print("Valid age")
```

Input: age = -5

```
Traceback (most recent call last):
  ...
AssertionError: Age cannot be negative
```

Note: in Example 2, the program stops at the assert line — “Valid age” is never printed, because the condition (age >= 0) evaluated to False.

exit() Function

exit() immediately terminates a running Python program by raising the SystemExit exception. Any code written after the exit() call is never executed. It optionally accepts an exit code or a message: an integer signals a status code to the operating system (0 conventionally means success, non-zero means an error occurred), while a string is printed to the screen before the program closes.

- exit() is intended mainly for use in the interactive interpreter
- In scripts and larger programs, sys.exit() (from the sys module) is the recommended alternative, since it does not depend on the site module being loaded
- Any statements after exit() in the same block are skipped entirely

Example 1 — Basic exit()

```
print("Before exit")
exit()
print("This will not run")
```

Input: (none)

```
Before exit
```

Note: the third line never executes — the program terminates as soon as exit() runs.

Example 2 — exit() with a status code inside a validation check

```
value = -1
if value < 0:
    print("Invalid value, exiting program")
    exit(1)
print("Continuing program")
```

Input: value = -1

```
Invalid value, exiting program
```

Note: exit(1) reports a non-zero status code to the operating system, signalling that the program stopped due to an error; “Continuing program” is never printed.

range() Function

range() generates an immutable sequence of numbers and is most commonly used to control the number of times a for loop runs. It does not create a list in memory all at once — values are produced one at a time, which makes it memory-efficient even for very large ranges.

- range(stop) — numbers from 0 up to (not including) stop
- range(start, stop) — numbers from start up to (not including) stop
- range(start, stop, step) — same, but advancing by step each time (step may be negative)
- The stop value is always excluded from the sequence

Example 1 — range() with a single argument

```
for i in range(5):
    print(i)
```

Input: range(5)

```
0
1
2
3
4
```

Example 2 — range() with start, stop, and step

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

Input: range(2, 10, 2)

```
2
4
6
8
```

Example 3 — range() counting backwards with a negative step

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

Input: range(10, 0, -2)

```
10
8
6
4
2
```