**PYTHON PROGRAMMING**

**UNIT-3**

**CONTROL FLOW, FUNCTIONS, STRINGS**

### Decision Making Statements

Decision-making statements are used to control the flow of execution in a program based on certain conditions. These conditions evaluate to either True or False, and the program executes the corresponding block of code.

### Types of Conditional Statements

Python provides different types of conditional statements:

1. if statement
2. if-else statement
3. elif statement (covered later)
4. Nested if-else statements
5. Ternary expressions

### If Statement

The if statement executes a block of code only when a specified condition is True.

#### Syntax:

if condition:

# Code block executed if the condition is True

#### Example:

num = 10

if num > 0:

print("The number is positive")



**Output:**

The number is positive

### If-Else Statement

The if-else statement provides an alternative block of code that runs when the condition is False.

#### Syntax:

if condition:

# Code executed if condition is True

else:

# Code executed if condition is False

#### Example:

num = -5

if num > 0:

print("Positive number")

else:

print("Negative number")



**Output:**

Negative number

### Boolean Expressions

Boolean expressions evaluate to either True or False. These are often used in conditional statements.

#### Example:

x = 5

y = 10

print(x > y) # False

print(x < y) # True



### Elif Statement

The elif (short for "else if") statement allows multiple conditions to be checked sequentially.

#### Syntax:

if condition1:

# Code if condition1 is True

elif condition2:

# Code if condition2 is True

else:

# Code if all conditions are False

#### Example:

marks = 85

if marks >= 90:

print("Grade: A")

elif marks >= 80:

print("Grade: B")

else:

print("Grade: C")



**Output:**

Grade: B

### Logical Operators in Conditions

Logical operators (and, or, not) are used to combine multiple conditions.

#### Example:

x = 15

y = 5

if x > 10 and y < 10:

print("Both conditions are True")



**Output:**

Both conditions are True

### Short-Hand Expressions

Python allows short-hand if statements when there is only one statement to execute.

#### Example:

x = 10

if x > 5: print("x is greater than 5")



**Output:**

x is greater than 5

### Ternary Expressions

A ternary expression (or conditional expression) allows a compact way to assign values based on a condition.

#### Syntax:

value = true\_value if condition else false\_value

#### Example:

age = 20

status = "Adult" if age >= 18 else "Minor"

print(status)



**Output:**

Adult



### Nested If-Else Statements

An if statement inside another if statement is called a nested if.

#### Example:

num = 15

if num > 0:

if num % 2 == 0:

print("Positive Even Number")

else:

print("Positive Odd Number")

else:

print("Negative Number")



**Output:**

Positive Odd Number

### Match Statement (Python 3.10+)

The match statement is used for pattern matching, similar to switch in other languages.

#### Syntax:

match variable:

case pattern1:

# Code block

case pattern2:

# Code block

case \_:

# Default case

#### Example:

def get\_day\_name(day):

match day:

case 1:

return "Monday"

case 2:

return "Tuesday"

case 3:

return "Wednesday"

case \_:

return "Invalid day"

print(get\_day\_name(2))



**Output:**

Tuesday

### Pass Statement

The pass statement is a placeholder when a block of code is required syntactically but should not execute anything.

#### Example:

x = 10

if x > 5:

pass # Placeholder for future code

print("End of program")



**Output:**

End of program



**Loops in Python**

### **Introduction to Loops**

Loops are used to execute a block of code multiple times. They help in reducing code redundancy and improving efficiency. Python provides three types of loops:

* **for loop**
* **while loop**
* **Loop Control Statements**

### **1. for Loop**

The for loop in Python is used to iterate over a sequence (list, tuple, dictionary, string, or range).

**Syntax:**

for variable in sequence:

# Code to execute



**Example:** Iterating over a list.

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print(fruit)



**Output:**

apple

banana

cherry



**Example:** Using range() in a for loop.

for i in range(5):

print(i)



**Output:**

0

1

2

3

4

### **2. while Loop**

The while loop executes as long as a condition remains true.

**Syntax:**

while condition:

# Code to execute



**Example:** Print numbers from 1 to 5 using a while loop.

i = 1

while i <= 5:

print(i)

i += 1



**Output:**

1

2

3

4

5



## **Loop Control Statements**

Loop control statements modify the execution flow of loops. Python provides three such statements:

### **1. break Statement**

The break statement is used to exit a loop prematurely when a certain condition is met.

**Example:**

for num in range(10):

if num == 5:

break

print(num)



**Output:**

0

1

2

3

4

### **2. continue Statement**

The continue statement skips the current iteration and moves to the next one.

**Example:**

for num in range(5):

if num == 2:

continue

print(num)



**Output:**

0

1

3

4

### **3. pass Statement**

The pass statement is used as a placeholder when a statement is syntactically required but no action is needed.

**Example:**

for num in range(5):

if num == 2:

pass # Placeholder for future code

print(num)



**Output:**

0

1

2

3

4



## **Advanced Loop Operations**

### **1. Nested Loops**

A nested loop is a loop inside another loop.

**Example:**

for i in range(3):

for j in range(2):

print(f"i: {i}, j: {j}")



**Output:**

i: 0, j: 0

i: 0, j: 1

i: 1, j: 0

i: 1, j: 1

i: 2, j: 0

i: 2, j: 1

### **2. Loops in Conditional Statements**

Loops can be used within conditional statements to execute repetitive tasks based on conditions.

**Example:**

x = 10

while x > 0:

if x % 2 == 0:

print(f"Even number: {x}")

x -= 1

### **3. else in Loops**

Python allows the else statement to be used with loops. The else block executes when the loop completes normally (not terminated by break).

**Example with for loop:**

for i in range(3):

print(i)

else:

print("Loop completed successfully")



**Output:**

0

1

2

Loop completed successfully



**Example with while loop:**

x = 3

while x > 0:

print(x)

x -= 1

else:

print("Loop finished")



**Output:**

3

2

1

Loop finished



## **Practice Questions**

1. Write a program to print all even numbers from 1 to 50 using a for loop.
2. Implement a program that asks for user input and prints numbers in reverse order using a while loop.
3. Write a Python script to print the multiplication table of a given number using a loop.
4. Create a program to count occurrences of a specific character in a string using a for loop.
5. Use a break statement to exit a loop when a user inputs stop.

**Introduction to Strings**

### **What is a String?**

A string is a sequence of characters enclosed in single ('), double ("), or triple (''' or """) quotes in Python. Strings are immutable, meaning they cannot be changed after creation.

**Example:**

string1 = 'Hello'

string2 = "World"

string3 = '''Python Strings'''

print(string1, string2, string3)



**Output:**

Hello World Python Strings



### **Characteristics of Strings**

* Strings can contain letters, numbers, and symbols.
* Strings are immutable (cannot be changed after assignment).
* Strings support indexing and slicing.

### **String Declaration and Initialization**

Strings can be declared using single, double, or triple quotes.

string1 = 'Hello'

string2 = "Python"

string3 = '''Multi-line

String''' # Triple quotes allow multi-line strings



### **String Literals**

String literals are sequences of characters enclosed within quotes.

string\_literal = "This is a string literal"



### **Concatenation and Repetition**

* **Concatenation:** Combining two strings using +.
* **Repetition:** Repeating a string multiple times using \*.

str1 = "Hello "

str2 = "World"

result = str1 + str2 # Concatenation

print(result)

repeat\_str = "Python " \* 3 # Repetition

print(repeat\_str)



**Output:**

Hello World

Python Python Python



### **String Representation in Memory**

Strings are stored as arrays of characters and indexed accordingly.

string = "Python"

print(id(string)) # Memory address of the string



### **Iterating Through a String**

string = "Hello"

for char in string:

print(char)



**Output:**

H

e

l

l

o



### **Checking String Length**

string = "Python"

print(len(string))



**Output:**

6



## **Basic String Operations**

### **String Indexing**

Strings can be accessed using indexing, starting from 0.

string = "Python"

print(string[0]) # First character

print(string[-1]) # Last character



### **String Slicing**

string = "Python Programming"

print(string[0:6]) # Output: Python

print(string[:6]) # Output: Python

print(string[7:]) # Output: Programming



### **Character Checking in String (in, not in)**

string = "Python"

print('y' in string) # True

print('z' not in string) # True



### **String Conversion (str())**

num = 100

string = str(num)

print(string, type(string))



### **String Immutability**

Strings cannot be modified after creation.

string = "Hello"

# string[0] = 'h' # This will raise an error



### **String Formatting**

name = "Alice"

age = 25

print(f"My name is {name} and I am {age} years old.")



### **Escaping Special Characters (\n, \t, etc.)**

print("Hello\nWorld") # Newline

print("Hello\tWorld") # Tab space



## **Advanced String Operations**

### **Case Conversion Methods**

string = "Python Programming"

print(string.upper()) # UPPERCASE

print(string.lower()) # lowercase

print(string.title()) # Title Case

print(string.capitalize()) # Capitalize first letter



### **Alignment Methods**

string = "Python"

print(string.center(10, '-')) # --Python--

print(string.ljust(10, '\*')) # Python\*\*\*\*

print(string.rjust(10, '\*')) # \*\*\*\*Python



### **Split and Join Methods**

string = "Hello World Python"

words = string.split() # Splitting string into a list

print(words)

new\_string = "-".join(words) # Joining list into a string

print(new\_string)



### **Boolean String Methods**

string = "Hello123"

print(string.isalpha()) # False (contains digits)

print(string.isdigit()) # False (contains letters)

print("12345".isdigit()) # True



### **Find and Replace Methods**

string = "Hello World"

print(string.find("World")) # Returns starting index

print(string.replace("World", "Python"))



## **Practice Questions**

1. Write a Python program to count the number of vowels in a given string.
2. Create a program that checks if a given string is a palindrome.
3. Write a program to remove all spaces from a string.
4. Implement a program that finds and replaces a substring in a given string.
5. Develop a Python script that extracts digits from an alphanumeric string.

**Python Arrays**

### **Introduction to Arrays**

An array is a data structure that stores multiple values of the same data type in a single variable. In Python, arrays are not built-in like lists, but they can be implemented using the array module.

**Example:**

import array

arr = array.array('i', [1, 2, 3, 4, 5]) # 'i' represents integer type

print(arr)



**Output:**

array('i', [1, 2, 3, 4, 5])



### **Accessing Elements**

Elements in an array can be accessed using indexing.

print(arr[0]) # First element

print(arr[-1]) # Last element



**Output:**

1

5



### **Length of an Array**

The len() function returns the number of elements in an array.

print(len(arr))



**Output:**

5



### **Looping Through an Array**

for element in arr:

print(element)



### **Adding Elements to an Array**

* append(): Adds an element to the end.
* insert(): Adds an element at a specific position.

arr.append(6)

arr.insert(2, 99) # Insert 99 at index 2

print(arr)



### **Removing Elements from an Array**

* remove(): Removes the first occurrence of an element.
* pop(): Removes an element at a specified index.

arr.remove(99)

arr.pop(1) # Removes element at index 1

print(arr)



### **Slicing and Searching in an Array**

print(arr[1:4]) # Slicing from index 1 to 3

print(3 in arr) # Searching for an element



### **Array Methods**

| **Method** | **Description** |
| --- | --- |
| append(x) | Adds an element x to the end of the array |
| insert(i, x) | Inserts x at index i |
| remove(x) | Removes the first occurrence of x |
| pop(i) | Removes element at index i |
| index(x) | Returns index of x |
| reverse() | Reverses the array |
| count(x) | Returns number of occurrences of x |

## **Debugging in Python**

### **Introduction to Debugging**

Debugging is the process of finding and fixing errors in a program. Python provides several debugging techniques and tools.

### **Debugging Tools and Techniques**

1. **Using print() statements**
   * Simple but effective for small programs.

x = 10

print(f"Value of x: {x}")



1. **Using Assertions (assert statement)**
   * Helps check conditions and raises an error if the condition fails.

assert x > 0, "x should be greater than 0"



1. **Using the pdb Module (Python Debugger)**
   * Allows step-by-step execution.

import pdb

pdb.set\_trace()



1. **Using Logging (logging module)**
   * More advanced than print statements.

import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message")



### **Common Debugging Strategies**

* **Reading error messages carefully**
* **Checking variable values at different points**
* **Using a debugger (pdb or an IDE debugger)**
* **Testing code in small increments**
* **Using try-except blocks to handle exceptions**

**Functions in Python**

### **Defining Functions**

A function is a block of reusable code that performs a specific task. Functions help in modular programming and code reusability.

**Syntax:**

def function\_name(parameters):

"""Function Docstring"""

# Function body

return value # Optional



**Example:**

def greet(name):

return f"Hello, {name}!"

print(greet("Alice"))



**Output:**

Hello, Alice!



### **Types of Functions**

1. **Built-in Functions** – Predefined functions like print(), len(), max(), etc.
2. **User-defined Functions** – Created by programmers.
3. **Lambda (Anonymous) Functions** – One-line functions defined using lambda.
4. **Recursive Functions** – Functions that call themselves.

### **Calling Functions**

Functions are called using their name followed by parentheses containing arguments (if any).

def add(a, b):

return a + b

result = add(3, 5)

print(result) # Output: 8



### **Return Statement**

The return statement sends a value back to the caller.

def square(n):

return n \* n

print(square(4)) # Output: 16



### **Pass by Reference vs Pass by Value**

* **Immutable objects (int, float, string, tuple)** → Passed by value (creates a copy).
* **Mutable objects (list, dict, set)** → Passed by reference (modifies original object).

**Example:**

def modify\_list(lst):

lst.append(100)

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Output: [1, 2, 3, 100] (Modified because lists are mutable)



### **Function Arguments**

Python allows different types of arguments:

1. **Positional Arguments** – Based on position.
2. **Keyword Arguments** – Using parameter names.
3. **Default Arguments** – Providing default values.
4. **Variable-length Arguments** – Using \*args (for tuples) and \*\*kwargs (for dictionaries).

**Example:**

def greet(name, message="Welcome!"):

print(f"{message}, {name}")

greet("Alice") # Uses default message



**Output:**

Welcome!, Alice



## **Recursive Functions**

A recursive function is a function that calls itself to solve smaller subproblems of a larger problem.

**Example: Factorial using Recursion**

def factorial(n):

if n == 1:

return 1

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120



### **Anonymous Functions (Lambda Functions)**

Lambda functions are small, unnamed functions defined using the lambda keyword.

**Syntax:**

lambda arguments: expression



**Example:**

square = lambda x: x \* x

print(square(5)) # Output: 25



### **Scope of Variables**

1. **Local Scope** – Variables defined inside a function.
2. **Global Scope** – Variables defined outside a function.
3. **Nonlocal Scope** – Variables in the enclosing function (used with nonlocal).

**Example:**

def outer():

x = 10

def inner():

nonlocal x

x += 5

inner()

print(x) # Output: 15



### **Function Annotations**

Annotations specify types for parameters and return values.

**Example:**

def add(a: int, b: int) -> int:

return a + b

print(add(3, 5)) # Output: 8



## **Practice Questions**

1. Write a function to check if a number is prime.
2. Create a recursive function to compute the nth Fibonacci number.
3. Implement a lambda function to calculate the cube of a number.
4. Write a function that takes variable-length arguments and returns their sum.
5. Implement a function with both default and keyword arguments.

**Recursive Functions in Python**

### **1. Anonymous Functions**

Anonymous functions, also known as **lambda functions**, are functions without a name. They are typically used for short, simple operations and are often passed as arguments to higher-order functions.

#### **Syntax:**

lambda arguments: expression

#### **Example:**

# A lambda function to add two numbers

add = lambda x, y: x + y

print(add(5, 3)) # Output: 8

#### **Use Cases:**

* Used in functional programming.
* Helpful in short-lived operations.
* Often used with map(), filter(), and reduce().

### **2. Scope of Variables**

The scope of a variable determines where it can be accessed within a program. Python has three main scopes:

1. **Local Scope** – Variables declared inside a function.
2. **Global Scope** – Variables declared outside any function.
3. **Non-local Scope** – Variables used in nested functions that are neither local nor global.

#### **Example:**

x = 10 # Global variable

def outer\_function():

y = 5 # Local variable

def inner\_function():

nonlocal y # Refers to the y in outer\_function

y += 1

print("Inner y:", y)

inner\_function()

print("Outer y:", y)

outer\_function()

print("Global x:", x)

#### **Key Points:**

* **Local variables** exist only inside the function where they are created.
* **Global variables** can be accessed throughout the program.
* **Non-local variables** allow modification of a variable in an enclosing function.

### **3. Function Annotations**

Function annotations provide metadata about the function parameters and return value. They do not enforce types but serve as documentation.

#### **Syntax:**

def function\_name(param: type) -> return\_type:

pass

#### **Example:**

def greet(name: str) -> str:

return f"Hello, {name}!"

print(greet("Alice"))

#### **Key Points:**

* Annotations help in better readability and understanding.
* They can be accessed using \_\_annotations\_\_.
* They do not affect the actual function execution.

### **4. Recursive Functions**

A **recursive function** is a function that calls itself in order to solve a problem. It is useful for problems that can be broken down into smaller, similar problems.

#### **Base Case & Recursive Case:**

A recursive function must have:

1. **Base Case** – The termination condition.
2. **Recursive Case** – The function calling itself.

#### **Example:** Factorial Calculation

def factorial(n):

if n == 0 or n == 1: # Base case

return 1

else:

return n \* factorial(n - 1) # Recursive case

print(factorial(5)) # Output: 120

#### **Example:** Fibonacci Sequence

def fibonacci(n):

if n <= 0:

return "Invalid Input"

elif n == 1:

return 0

elif n == 2:

return 1

else:

return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(6)) # Output: 5

#### **Key Points:**

* Recursive functions **must have a base case** to prevent infinite recursion.
* Recursion can lead to stack overflow if not handled properly.
* Tail recursion optimization is not available in Python, so deep recursion should be used cautiously.

### **Practice Questions**

1. Write a **lambda function** to check if a number is even or odd.
2. Modify the **factorial function** to use iteration instead of recursion.
3. Explain the difference between **local and global scope** with an example.
4. Implement a recursive function to compute the sum of the first n natural numbers.
5. Use **function annotations** to define a function that calculates the area of a circle.

### **Summary of Topics**

#### **Conditional Statements**

* Decision-making in Python using if, if-else, elif, and boolean expressions.
* Logical operators, shorthand expressions, and ternary operations for concise conditions.
* Advanced control with nested if-else, match, and pass statements.

#### **Loops and Control Statements**

* Iterating with for and while loops.
* Controlling loops using break, continue, and pass.
* Advanced techniques like nested loops, conditional loops, and using else in loops.

#### **String Operations**

* Understanding strings, their characteristics, declaration, and initialization.
* Basic operations like indexing, slicing, length checking, and formatting.
* Advanced methods for case conversion, alignment, splitting, joining, searching, and replacing.

#### **Python Arrays**

* Working with arrays, accessing elements, slicing, searching, and looping through arrays.
* Adding and removing elements, along with common array methods.

#### **Debugging in Python**

* Introduction to debugging, essential tools, and strategies to identify and fix errors in Python programs.

#### **Functions and Recursive Functions**

* Defining and calling functions, return statements, and argument types.
* Differences between pass-by-value and pass-by-reference.
* Introduction to recursion, anonymous functions, function annotations, and variable scope.

**UNIT-3**