# Programming Fundamental Lab Manual using Python

BY

**Dr. Muhammad Siddique**
**HOD AI, NFC IET Mulrtan**

# Programming Fundamentals Lab Manual - Python

Dr. Muhammad Siddique

October 2024

## Contents

# 1 Week 1: Introduction to Python

**Objective:** Set up Python environment and write your first Python program.

## Tasks:

1. **Task 1:** Install Python from the official website (www.python.org). Open IDLE or Jupyter Notebook, and ensure the setup is correct.
2. **Task 2:** Write a Python program that prints "Hello, World!" and explain each line of code.
3. **Task 3:** Use the Python shell to perform basic arithmetic operations: addition, subtraction, multiplication, and division.
4. **Task 4:** Write a program that calculates and prints the sum of two numbers entered by the user.
5. **Task 5:** Research and list five commonly used Python development tools other than IDLE (e.g., Jupyter, PyCharm).

## Details of Lab Experiment:

### Task 1: Install Python

Visit the official Python website (www.python.org) to download and install Python. Once installed, open IDLE or Jupyter Notebook to verify that Python is running correctly.

### Task 2: Write a Python program that prints "Hello, World!"

In this task, you will write your first Python program that prints "Hello, World!" on the screen.

```
# This is a simple Python program to print Hello, World!
print("Hello, World!")
```

Explanation:

- The `print()` function is used to display output on the screen.
- The string `"Hello, World!"` is passed to the `print()` function and displayed on the console.

### Task 3: Perform arithmetic operations using Python shell

In this task, use the Python shell to perform basic arithmetic operations:

```
# Perform basic arithmetic operations
5 + 3    # Addition
9 - 2    # Subtraction
4 * 7    # Multiplication
8 / 2    # Division
```

Explanation:

- The Python shell allows you to execute code interactively. You can perform arithmetic operations like addition, subtraction, multiplication, and division directly in the shell.

### Task 4: Write a program to calculate the sum of two numbers entered by the user

In this task, you'll write a Python program to accept two numbers from the user and display their sum.

```
# This program adds two numbers provided by the user

# Get input from the user
num1 = input("Enter the first number: ")
num2 = input("Enter the second number: ")

# Convert the inputs to integers
num1 = int(num1)
num2 = int(num2)

# Calculate the sum
sum = num1 + num2

# Display the result
print("The sum of", num1, "and", num2, "is", sum)
```

Explanation:

- `input()` is used to take input from the user. The inputs are converted to integers using `int()`.
- The sum of the two numbers is calculated and displayed using the `print()` function.

### Task 5: Research five commonly used Python development tools

Python is supported by many development environments. Research and list five commonly used Python development tools other than IDLE:

- Jupyter Notebook
- PyCharm
- Visual Studio Code
- Spyder
- Thonny

# 2 Week 2: Basics of Python Syntax

**Objective:** Understand Python's basic syntax, variables, operators, and data types.

## Tasks:

1. **Task 1:** Declare variables of types `int`, `float`, and `str`, and print their values.
2. **Task 2:** Perform arithmetic operations with two variables of type `int` and one of type `float`.
3. **Task 3:** Use comparison operators (`>, <, ==`) to compare two variables and print the results.
4. **Task 4:** Convert a string to an integer and perform arithmetic operations on the converted value.
5. **Task 5:** Write a program that swaps the values of two variables and prints the result.

## Details of Lab Experiment:

- **Task 1:** Declare variables and print their types:

```
x = 10        # Integer
y = 3.14      # Float
name = "Alice"  # String
print(type(x), type(y), type(name))

```

- **Task 2:** Perform arithmetic operations:

```
a = 5
b = 3
print(a + b)  # Addition
print(a * b)  # Multiplication

```

- **Task 3:** Compare two variables using comparison operators:

```
print(a > b)  # Greater than
print(a == b)  # Equal to

```

- **Task 4:** Convert a string to an integer and perform an arithmetic operation:

```
age = "25"  # String
age_int = int(age)  # Convert to integer
print(age_int + 5)  # Add 5 to age

```

- **Task 5:** Swap the values of two variables:

```
a, b = 10, 20
a, b = b, a
print("a =", a, "b =", b)  # Swapped values

```

# 3 Week 3: Variables and Data Types

**Objective:** Deeply understand variables, data types, and how Python manages data. Learn to declare variables, identify their data types, and explore type conversion techniques.

## Tasks:

1. **Task 1:** Learn the basic data types available in Python by declaring and initializing variables of type `int`, `float`, `str`, and `bool`. Print the type of each variable using the `type()` function.
2. **Task 2:** Perform arithmetic operations on variables of type `int` and `float`. Understand how Python handles operations between different numeric types.
3. **Task 3:** Investigate implicit and explicit type conversion. Use Python's built-in functions like `int()`, `float()`, and `str()` to convert one data type into another.
4. **Task 4:** Explore the concept of type checking and dynamic typing in Python by reassigning variables to different data types. Analyze the flexibility and dynamic nature of Python's type system.
5. **Task 5:** Write a program that accepts input from the user, checks the type of the input, and performs type conversion if necessary. Ensure that the program can handle invalid inputs gracefully using error handling.

## Details of Lab Experiment:

- **Task 1:** Declare variables of different types and print their types:

```
# Declare variables of different data types
x = 10              # Integer
y = 3.14            # Float
name = "Alice"      # String
is_student = True # Boolean

# Print the type of each variable
print("Type of x:", type(x))
print("Type of y:", type(y))
print("Type of name:", type(name))
print("Type of is_student:", type(is_student))

```

  In this task, students will learn how to declare variables of different data types and use the `type()` function to identify their types. This forms the foundation of understanding how data is managed in Python.

- **Task 2:** Perform arithmetic operations:

```
# Arithmetic operations
a = 15
b = 7.5

# Perform addition, subtraction, multiplication, and division
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)

```

This task explores how Python handles arithmetic operations between integers and floats. It demonstrates Python's ability to handle mixed types seamlessly and the result of operations on numeric data.

- **Task 3:** Perform type conversion:

```
# Implicit type conversion (auto-conversion)
result = a + b  # Integer + Float
print("Result (implicit conversion):", result, "Type:", type(result))

# Explicit type conversion
num_str = "50"
converted_num = int(num_str)  # Convert string to integer
print("Converted number:", converted_num, "Type:", type(converted_num
    ))

# Convert float to string
float_num = 23.75
converted_str = str(float_num)
print("Converted string:", converted_str, "Type:", type(converted_str
    ))

```

In this task, students explore both implicit (automatic) and explicit (manual) type conversions in Python. They will use Python's built-in conversion functions like `int()`, `float()`, and `str()` to convert between different types.

- **Task 4:** Dynamic typing and type checking:

```
# Dynamic typing in Python
var = 100       # Initially an integer
print("Type of var:", type(var))

var = "Now I'm a string"  # Now assigned as a string
print("Type of var after reassignment:", type(var))

# Reassigning different types to the same variable
var = 3.14       # Reassigned to a float
print("Type of var after float reassignment:", type(var))

```

This task demonstrates Python's dynamic typing system, where a variable can change its type during runtime by assigning different values. Students will observe how Python dynamically handles types without explicit declarations.

- **Task 5:** User input and type checking with error handling:

```
# Get input from the user
user_input = input("Enter a number: ")

# Try to convert the input to an integer
try:
    number = int(user_input)
    print("You entered the number:", number)
except ValueError:
    print("Invalid input! Please enter a valid number.")

```

In this task, students write a program that takes user input and attempts to convert it to an integer. They will also handle potential errors (such as entering non-numeric data) using a `try-except` block, ensuring the program runs smoothly even with incorrect inputs.

## Learning Outcomes:

- Understand how variables are used to store data and how Python manages data types dynamically.
- Gain experience in identifying and converting between various data types.
- Explore Python's flexibility with dynamic typing and how to manage different data types in real-time programming.
- Develop skills in handling user input and converting data types to avoid common errors.

# 4 Week 4: Basic Input and Output

**Objective:** Understand how to handle input from users and display output in Python. Learn how to use Python's built-in functions for input and output operations, format strings, and manage user interactions effectively.

## Tasks:

1. **Task 1:** Use the `input()` function to accept user input and store it in variables. Understand how input is treated as a string and how to convert it to other data types as needed.
2. **Task 2:** Practice displaying output using the `print()` function, explore various formatting techniques for output, including string concatenation and formatted strings (`f-strings`).
3. **Task 3:** Write a program that accepts two numbers from the user, performs arithmetic operations, and displays the result in a well-formatted manner.
4. **Task 4:** Implement a program that asks the user for their name and age, and then displays a personalized greeting, calculating how old they will be in the next 10 years.
5. **Task 5:** Experiment with multi-line output and how to properly format long outputs or complex messages, using escape sequences like `\n` for line breaks.

## Details of Lab Experiment:

- **Task 1:** Accept user input and handle type conversion:

```
# Accepting user input
name = input("Enter your name: ")
age = input("Enter your age: ")

# Convert the age to an integer
age = int(age)

# Displaying the input values
print("Your name is:", name)
print("Your age is:", age)

```

In this task, students will learn how to accept input from the user using the `input()` function and convert string inputs to appropriate data types. This builds a foundation for user interaction in Python programs.

- **Task 2:** Display output using the `print()` function:

```
# Displaying formatted output
x = 5
y = 7
print("The sum of", x, "and", y, "is", x + y)

# Using f-strings for formatting
print(f"The sum of {x} and {y} is {x + y}")

```

This task explores various techniques for displaying output in Python. Students will practice using both string concatenation and `f-strings`, which offer a more readable and efficient way to format output.

- **Task 3:** Perform arithmetic operations with user input:

```
# Accept two numbers from the user
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

# Perform arithmetic operations and display the results
print(f"Sum: {num1 + num2}")
print(f"Difference: {num1 - num2}")
print(f"Product: {num1 * num2}")
print(f"Quotient: {num1 / num2}")

```

In this task, students write a program that accepts two numbers from the user, performs basic arithmetic operations, and displays the results in a neatly formatted output using `f-strings`. This reinforces handling input and performing operations dynamically.

- **Task 4:** Personalized greeting and age calculation:

```
# Get the user's name and age
name = input("What is your name? ")
age = int(input("How old are you? "))

# Calculate age in 10 years
future_age = age + 10

# Display a personalized message
print(f"Hello, {name}! You are currently {age} years old.")
print(f"In 10 years, you will be {future_age} years old.")

```

This task is a simple exercise in combining input, output, and basic calculations. It also helps students understand how to interact with users through a personalized experience by dynamically using their input in the output message.

- **Task 5:** Multi-line output and formatting:

```
# Display a multi-line message
print("Welcome to Python Programming!\n"
      "This is a multi-line message.\n"
      "Each line is separated by a newline character.")

```

This task introduces students to the concept of multi-line output using escape sequences like `\n`. It helps them understand how to format more complex outputs, such as long messages or formatted text blocks.

## Learning Outcomes:

- Develop skills in interacting with users through input and output in Python.
- Learn to convert user input into appropriate data types for further processing.
- Master formatting techniques for displaying output, including the use of f-strings and escape sequences.
- Gain experience in writing programs that involve basic arithmetic operations and personalized user interactions.
- Understand how to format and handle multi-line outputs in Python.

# 5 Week 5: Conditional Statements - Introduction

**Objective:** Understand the fundamentals of conditional statements in Python. Learn how to control the flow of a program using `if`, `elif`, and `else` statements. Explore the significance of Boolean expressions and how they influence decision-making in programming.

## Tasks:

1. **Task 1:** Write simple `if` statements to evaluate conditions and execute code blocks based on those conditions.
2. **Task 2:** Introduce `else` statements to handle cases where conditions are not met, ensuring programs can respond appropriately to various inputs.
3. **Task 3:** Implement `elif` statements to manage multiple conditions efficiently, allowing for a clear and concise decision-making process.
4. **Task 4:** Create a program that checks user input against specified criteria and displays relevant messages based on the input values.
5. **Task 5:** Explore nested conditional statements to evaluate complex conditions and enhance the logic of programs.

## Details of Lab Experiment:

- **Task 1:** Simple `if` statement:

```
# Basic if statement
number = int(input("Enter a number: "))
if number > 0:
    print("The number is positive.")

```

In this task, students will learn how to write a basic `if` statement to evaluate a condition (whether the number is positive) and execute a block of code accordingly.

- **Task 2:** Using `else`:

```
# if-else statement
number = int(input("Enter a number: "))
if number > 0:
    print("The number is positive.")
else:
    print("The number is not positive.")

```

This task introduces the `else` statement, allowing students to define an alternative action when the condition in the `if` statement is not met.

- **Task 3:** Implementing `elif`:

```
# if-elif-else statement
number = int(input("Enter a number: "))
if number > 0:
    print("The number is positive.")
elif number < 0:
    print("The number is negative.")
else:
    print("The number is zero.")

```

In this task, students will learn how to handle multiple conditions using `elif` statements. This approach allows them to efficiently manage different scenarios within a single conditional structure.

- **Task 4:** User input evaluation:

```
# Evaluating user input
age = int(input("Enter your age: "))
if age < 13:
    print("You are a child.")
elif age < 18:
    print("You are a teenager.")
else:
    print("You are an adult.")

```

This task involves writing a program that checks the user's age and categorizes them into child, teenager, or adult. It emphasizes the practical application of conditional statements in real-life scenarios.

- **Task 5:** Nested conditional statements:

```
# Nested if statement
number = int(input("Enter a number: "))
if number != 0:
    if number > 0:
        print("The number is positive.")
    else:
        print("The number is negative.")
else:
    print("The number is zero.")

```

In this task, students will explore nested conditional statements, where they can evaluate additional conditions within an existing condition. This helps them understand how to create complex decision-making processes in their programs.

## Learning Outcomes:

- Gain a solid understanding of conditional statements and their syntax in Python.
- Learn to implement simple and complex conditions using `if`, `elif`, and `else` statements.
- Develop skills in evaluating user input against various conditions and displaying appropriate responses.
- Master the use of nested conditional statements to create intricate decision-making logic within programs.
- Enhance problem-solving skills by applying conditional logic to real-world programming scenarios.

# 6 Week 6: Nested Conditionals and Elif

**Objective:** Explore the concept of nested conditionals and the use of `elif` statements to manage complex decision-making scenarios in Python. Understand how to structure conditions to handle multiple layers of logic and improve the clarity and efficiency of code.

## Tasks:

1. **Task 1:** Review and reinforce the syntax of `if`, `elif`, and `else` statements.
2. **Task 2:** Implement nested conditionals to evaluate multiple conditions, demonstrating how to build more complex logic within programs.
3. **Task 3:** Develop a program that utilizes both `elif` and nested conditionals to categorize user input effectively.
4. **Task 4:** Create a decision tree using nested conditionals, allowing the program to traverse different paths based on user input.
5. **Task 5:** Experiment with real-world scenarios where nested conditionals are beneficial, such as grading systems or game mechanics.

## Details of Lab Experiment:

- **Task 1:** Syntax review of conditional statements:

```
# Basic structure of if, elif, and else
x = int(input("Enter a number: "))
if x > 0:
    print("Positive number")
elif x < 0:
    print("Negative number")
else:
    print("Zero")

```

Students will review the syntax of conditional statements and understand the flow of execution based on different conditions.

- **Task 2:** Implementing nested conditionals:

```
# Nested conditionals example
age = int(input("Enter your age: "))
if age >= 18:
    if age < 65:
        print("You are an adult.")
    else:
        print("You are a senior.")
else:
    print("You are a minor.")

```

In this task, students will learn to implement nested conditionals, allowing for more detailed categorization based on user input.

- **Task 3:** Utilizing `elif` with nested conditionals:

```
# Using elif with nested conditionals
score = int(input("Enter your score: "))
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
else:
    if score >= 70:
        print("Grade: C")
    elif score >= 60:
        print("Grade: D")
    else:
        print("Grade: F")

```

This task emphasizes using both `elif` and nested conditionals to categorize scores into grades, highlighting the advantages of structured conditional logic.

- **Task 4:** Creating a decision tree:

```
# Simple decision tree example
weather = input("Is it raining? (yes/no): ")
if weather == "yes":
    temperature = int(input("Is it cold? (yes/no): "))
    if temperature == "yes":
        print("Wear a coat and take an umbrella.")
    else:
        print("Take an umbrella.")
else:
    print("Enjoy the nice weather!")

```

Students will create a decision tree based on user input, helping them visualize how nested conditionals can be used to navigate through various choices.

- **Task 5:** Real-world scenarios with nested conditionals:

```
# Grading system example
marks = int(input("Enter your marks: "))
if marks >= 90:
    print("Excellent!")
elif marks >= 75:
    print("Good job!")
    if marks >= 80:
        print("You are in the top 10% of your class.")
elif marks >= 50:
    print("You passed.")
else:
    print("Better luck next time.")

```

This task encourages students to think critically about how nested conditionals can enhance the decision-making process in real-life applications, such as grading systems or user feedback mechanisms.

## Learning Outcomes:

- Gain a deeper understanding of how to use `elif` and nested conditionals in Python programming.
- Develop the ability to write code that handles multiple conditions effectively, improving code readability and efficiency.

- Learn to construct logical decision trees that can guide program flow based on user input.
- Enhance problem-solving skills by applying nested conditionals to realistic scenarios, fostering creativity in coding practices.
- Prepare students to handle more complex programming challenges in future lab sessions and projects.

# 7 Week 7: Loops - For Loop

**Objective:** Understand the concept and usage of the `for` loop in Python. Learn how to iterate over sequences such as lists, tuples, and strings, enabling repetitive tasks to be executed efficiently and concisely.

## Tasks:

1. **Task 1:** Review the syntax of the `for` loop and its basic functionality.
2. **Task 2:** Implement a simple program using a `for` loop to iterate through a range of numbers.
3. **Task 3:** Create a program that uses a `for` loop to iterate over a list and perform operations on each element.
4. **Task 4:** Explore the use of nested `for` loops for working with multi-dimensional data structures.
5. **Task 5:** Develop a program that utilizes the `for` loop in a real-world application, such as processing user input or generating patterns.

## Details of Lab Experiment:

- **Task 1:** Syntax review of the `for` loop:

```
# Basic structure of a for loop
for i in range(5):
    print(i)

```

  In this task, students will review the syntax of the `for` loop and understand how it iterates through a sequence of numbers.

- **Task 2:** Implementing a simple `for` loop:

```
# Example of a for loop to iterate through a range
for number in range(1, 11):
    print("Square of", number, "is", number ** 2)

```

  Students will write a program that uses a `for` loop to calculate and display the squares of numbers from 1 to 10.

- **Task 3:** Iterating over a list:

```
# Using a for loop to iterate over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

```

  This task emphasizes the use of a `for` loop to iterate over a list of items, demonstrating how to access and manipulate each element.

- **Task 4:** Nested `for` loops:

```
# Example of a nested for loop
for i in range(3):
    for j in range(2):
        print(f"i = {i}, j = {j}")

```

In this task, students will learn how to use nested `for` loops to work with multi-dimensional data, allowing them to handle more complex data structures effectively.

- **Task 5:** Real-world application using a `for` loop:

```
# Using a for loop to generate a pattern
for i in range(1, 6):
    print("*" * i)

```

Students will create a program that uses a `for` loop to generate a simple pattern, reinforcing their understanding of iteration through practical examples.

## Learning Outcomes:

- Develop a solid understanding of the `for` loop syntax and its application in Python programming.
- Gain proficiency in iterating over different data structures, including ranges, lists, and tuples.
- Learn to implement nested `for` loops to handle complex data scenarios, enhancing problem-solving skills.
- Create programs that utilize loops effectively for various tasks, such as data processing and pattern generation.
- Prepare students for advanced topics in programming that require efficient iteration and control flow structures.

# 8 Week 8: Loops - While Loop

**Objective:** Learn the concept and usage of the `while` loop in Python. Understand how to implement loops that continue until a specified condition is met, allowing for greater flexibility in control flow compared to the `for` loop.

## Tasks:

1. **Task 1:** Review the syntax of the `while` loop and understand its functionality.
2. **Task 2:** Implement a simple program using a `while` loop to print numbers until a certain condition is met.
3. **Task 3:** Create a program that uses a `while` loop to accumulate the sum of user inputs until the user decides to stop.
4. **Task 4:** Explore the concept of infinite loops and how to avoid them by using proper condition checks.
5. **Task 5:** Develop a program that utilizes the `while` loop in a real-world scenario, such as a simple menu-driven application.

## Details of Lab Experiment:

- **Task 1:** Syntax review of the `while` loop:

```
# Basic structure of a while loop
while condition:
    # code to execute

```

  In this task, students will review the syntax of the `while` loop and understand how it executes code as long as the condition evaluates to true.

- **Task 2:** Implementing a simple `while` loop:

```
# Example of a while loop
count = 0
while count < 5:
    print("Count is:", count)
    count += 1

```

  Students will write a program that uses a `while` loop to print the value of a counter variable until it reaches a specified limit.

- **Task 3:** Accumulating user inputs:

```
# Using a while loop to accumulate user inputs
total = 0
while True:
    number = input("Enter a number (or type 'exit' to stop): ")
    if number.lower() == 'exit':
        break
    total += int(number)
print("Total sum is:", total)

```

  This task emphasizes the use of a `while` loop for continuously accepting user input until a specific command is issued, illustrating real-world application.

- **Task 4:** Exploring infinite loops:

```
# Example of an infinite loop
while True:
    print("This loop will run forever. Use Ctrl+C to stop.")

```

In this task, students will learn about infinite loops, discussing their implications and how to avoid them by setting proper exit conditions.

- **Task 5:** Menu-driven application:

```
# Simple menu-driven program using a while loop
while True:
    print("Menu:")
    print("1. Option 1")
    print("2. Option 2")
    print("3. Exit")
    choice = input("Select an option: ")
    if choice == '3':
        break
    print("You selected option", choice)

```

Students will create a simple menu-driven application using a `while` loop, reinforcing their understanding of conditional statements and loops working together.

## Learning Outcomes:

- Develop a solid understanding of the `while` loop syntax and its application in Python programming.
- Gain proficiency in implementing loops that operate under specific conditions, providing greater flexibility in control flow.
- Learn to handle user input dynamically using `while` loops, enhancing interactive program development skills.
- Understand the concept of infinite loops and the importance of exit conditions to prevent unintended program behavior.
- Prepare students for advanced programming concepts that require iterative processes and conditional logic.

# 9 Week 9: Functions - Introduction

**Objective:** Understand the concept of functions in Python, their syntax, and their importance in structuring code. Learn how to define and call functions, pass arguments, and return values to promote code reusability and organization.

## Tasks:

1. **Task 1:** Review the definition and purpose of functions in programming.
2. **Task 2:** Learn the syntax for defining a function in Python.
3. **Task 3:** Implement a simple function that takes parameters and returns a value.
4. **Task 4:** Create functions that perform specific tasks and integrate them into a larger program.
5. **Task 5:** Explore the concept of function scope and variable lifetime within functions.

## Details of Lab Experiment:

- **Task 1:** Understanding functions: Functions are a fundamental building block in Python programming, allowing for code organization, reuse, and modularity. Discuss the significance of functions in breaking down complex problems into manageable pieces.
- **Task 2:** Defining a function:

```
# Syntax for defining a function
def function_name(parameters):
    # code to execute

```

  Students will learn the syntax for defining a function, highlighting the importance of the `def` keyword and how to specify parameters.

- **Task 3:** Implementing a simple function:

```
# Example of a simple function
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print("The sum is:", result)

```

  In this task, students will implement a function that adds two numbers and returns the result, reinforcing their understanding of function parameters and return values.

- **Task 4:** Creating and integrating functions:

```
# Functions performing specific tasks
def greet_user(name):
    print("Hello,", name)

def calculate_area(length, width):
    return length * width

greet_user("Alice")
```

```
area = calculate_area(5, 10)
print("Area is:", area)

```

Students will create multiple functions to perform different tasks and integrate them into a larger program, enhancing their ability to organize code.

- **Task 5:** Exploring function scope:

```
# Example illustrating variable scope
def outer_function():
    outer_variable = "I'm outside!"

    def inner_function():
        inner_variable = "I'm inside!"
        print(outer_variable)  # Accessing outer variable
        return inner_variable

    inner_function()
    # print(inner_variable)  # This would raise an error

outer_function()

```

This task will help students understand the concept of variable scope and how variables defined within a function can affect accessibility, promoting a better grasp of function structure.

## Learning Outcomes:

- Gain a comprehensive understanding of functions and their role in organizing and structuring Python code.
- Develop proficiency in defining and calling functions with parameters and return values.
- Learn to implement multiple functions for specific tasks and integrate them into larger applications.
- Understand the concept of variable scope and how it influences code behavior within functions.
- Prepare students for more advanced topics in programming, including recursion and higher-order functions.

# 10 Week 10: Functions - Scope and Recursion

**Objective:** Explore the concepts of variable scope in functions, including local and global scope. Understand recursion as a method for solving problems where a function calls itself and identify the base case to prevent infinite loops.

## Tasks:

1. **Task 1:** Review the definitions of local and global variables and their scope.
2. **Task 2:** Implement examples demonstrating local and global variable scope in functions.
3. **Task 3:** Learn the concept of recursion and its structure.
4. **Task 4:** Write recursive functions to solve simple problems.
5. **Task 5:** Analyze base cases in recursive functions to prevent infinite recursion.

## Details of Lab Experiment:

- **Task 1:** Understanding variable scope: Variable scope determines the accessibility of variables within different parts of a program. Local variables are defined within a function and are only accessible within that function, while global variables are defined outside any function and can be accessed anywhere in the program. Discuss the significance of scope in programming for managing state and avoiding conflicts.
- **Task 2:** Demonstrating scope:

```
# Example of local and global scope
global_var = "I'm global!"

def scope_example():
    local_var = "I'm local!"
    print(global_var)  # Accessing global variable
    print(local_var)   # Accessing local variable

scope_example()
# print(local_var)  # This would raise an error

```

  Students will implement examples to illustrate the difference between local and global variables, reinforcing their understanding of scope.

- **Task 3:** Introduction to recursion: Recursion occurs when a function calls itself to solve a problem by breaking it down into smaller, manageable sub-problems. Discuss the structure of a recursive function, including the recursive case and the base case.
- **Task 4:** Writing recursive functions:

```
# Example of a recursive function (factorial)
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case

result = factorial(5)
print("Factorial of 5 is:", result)

```

Students will implement recursive functions, such as calculating the factorial of a number, to solidify their understanding of recursion.

- **Task 5:** Analyzing base cases:

```
# Example of recursive function with base case
def fibonacci(n):
    if n <= 0:
        return 0  # Base case
    elif n == 1:
        return 1  # Base case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case

result = fibonacci(6)
print("Fibonacci of 6 is:", result)

```

This task will help students understand the importance of defining base cases in recursive functions to ensure they terminate properly and do not cause infinite recursion.

## Learning Outcomes:

- Develop a thorough understanding of variable scope, including the differences between local and global variables.
- Gain hands-on experience implementing functions that demonstrate scope concepts.
- Learn the principles of recursion and how to write recursive functions effectively.
- Understand the significance of base cases in preventing infinite recursion.
- Prepare students for advanced topics in programming that involve recursion and complex problem-solving.

# 11 Week 11: Lists

**Objective:** Understand the concept of lists as a fundamental data structure in Python. Learn to create, manipulate, and perform various operations on lists, including indexing, slicing, and using built-in list methods.

## Tasks:

1. **Task 1:** Introduction to lists and their importance in Python.
2. **Task 2:** Create and access elements in a list.
3. **Task 3:** Perform list operations such as appending, inserting, and removing elements.
4. **Task 4:** Utilize list slicing and indexing to manipulate sublists.
5. **Task 5:** Explore built-in list methods and their applications.

## Details of Lab Experiment:

- **Task 1:** Understanding lists: Lists are versatile data structures in Python that can store multiple items in a single variable. They are ordered, mutable, and can hold heterogeneous data types. Discuss the significance of lists in managing collections of data efficiently.
- **Task 2:** Creating and accessing lists:

```
# Creating a list
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Accessing the first element

```

  Students will create lists and learn how to access individual elements using indexing.

- **Task 3:** List operations:

```
# List operations
fruits.append("orange")  # Adding an element
fruits.insert(1, "kiwi")  # Inserting an element at a specific
    position
fruits.remove("banana")  # Removing an element
print(fruits)

```

  This task will involve manipulating lists using various operations to understand how to modify lists dynamically.

- **Task 4:** Slicing and indexing:

```
# List slicing
sublist = fruits[1:3]  # Slicing to get elements from index 1 to 2
print(sublist)  # Output will be elements at index 1 and 2

```

  Students will practice slicing and indexing to extract sublists and explore their applications in data processing.

- **Task 5:** Built-in list methods:

```
# Using built-in list methods
print(len(fruits))  # Length of the list
fruits.sort()  # Sorting the list
print(fruits)

```

This task will familiarize students with various built-in methods available for lists, such as sorting, reversing, and finding the length, enhancing their ability to manage lists effectively.

## Learning Outcomes:

- Gain a comprehensive understanding of lists as a core data structure in Python.
- Develop skills to create and manipulate lists through various operations.
- Learn to utilize indexing and slicing techniques to access and modify list elements.
- Familiarize with built-in list methods to enhance data handling capabilities.
- Prepare students for advanced topics involving collections and data manipulation in Python programming.

# 12 Week 12: Tuples and Dictionaries

**Objective:** Explore tuples and dictionaries as essential data structures in Python. Learn to create, access, and manipulate these data types, understanding their unique properties and use cases.

## Tasks:

1. **Task 1:** Introduction to tuples and their characteristics.
2. **Task 2:** Create and access elements in a tuple.
3. **Task 3:** Understand the immutability of tuples and their applications.
4. **Task 4:** Introduction to dictionaries and their properties.
5. **Task 5:** Create and access key-value pairs in a dictionary.
6. **Task 6:** Perform dictionary operations such as adding, updating, and removing elements.

## Details of Lab Experiment:

- **Task 1:** Understanding tuples: Tuples are ordered, immutable collections of items in Python. They can store multiple items in a single variable and are defined using parentheses. Discuss the significance of tuples, particularly in situations where a constant set of values is required.
- **Task 2:** Creating and accessing tuples:

```
# Creating a tuple
colors = ("red", "green", "blue")
print(colors[1])  # Accessing the second element

```

  Students will create tuples and learn to access individual elements using indexing.

- **Task 3:** Tuple immutability:

```
# Attempting to modify a tuple
# colors[1] = "yellow"  # This will raise a TypeError

```

  This task will involve understanding the immutability of tuples and when to use them effectively in Python programming.

- **Task 4:** Introduction to dictionaries: Dictionaries are unordered collections of key-value pairs. They are defined using curly braces and are mutable, allowing for dynamic updates. Discuss the importance of dictionaries for data storage and retrieval.
- **Task 5:** Creating and accessing dictionaries:

```
# Creating a dictionary
student = {"name": "Alice", "age": 21, "major": "Computer Science"}
print(student["name"])  # Accessing value using key

```

  Students will create dictionaries and learn how to access values using their associated keys.

- **Task 6:** Dictionary operations:

```
# Adding and updating key-value pairs
student["graduation_year"] = 2025  # Adding a new key-value pair
student["age"] = 22  # Updating an existing value
print(student)

# Removing a key-value pair
del student["major"]  # Removing a key-value pair
print(student)

```

This task will involve practicing dictionary operations to understand how to manage key-value pairs effectively, enhancing data manipulation skills.

## Learning Outcomes:

- Gain a comprehensive understanding of tuples and their properties in Python.
- Develop skills to create and manipulate tuples and recognize their immutability.
- Understand dictionaries as a flexible data structure for storing key-value pairs.
- Learn to create and perform various operations on dictionaries, including adding, updating, and deleting entries.
- Prepare students for utilizing tuples and dictionaries in real-world applications and more complex data structures in Python programming.

# 13 Week 13: Working with Strings - Basic Operations

**Objective:** Understand string data types in Python and perform basic string operations. Learn how to manipulate, format, and process strings effectively.

## Tasks:

1. **Task 1:** Introduction to string data types.
2. **Task 2:** Creating and accessing strings.
3. **Task 3:** Performing basic string operations (concatenation, repetition).
4. **Task 4:** Exploring string methods for manipulation.
5. **Task 5:** String formatting techniques.
6. **Task 6:** Practical examples of string operations.

## Details of Lab Experiment:

- **Task 1:** Introduction to string data types: Strings are sequences of characters used to represent text. Discuss the significance of strings in programming and how they can be defined using single, double, or triple quotes in Python.
- **Task 2:** Creating and accessing strings:

```
# Creating strings
greeting = "Hello, World!"
print(greeting)  # Output: Hello, World!

```

  Students will create strings and learn how to access individual characters using indexing.

- **Task 3:** Basic string operations:

```
# Concatenation
full_greeting = greeting + " How are you?"
print(full_greeting)

# Repetition
repeated_greeting = greeting * 3
print(repeated_greeting)

```

  This task will involve performing concatenation and repetition to create new strings, enhancing understanding of string manipulation.

- **Task 4:** Exploring string methods:

```
# String methods
message = " python programming "
print(message.strip())  # Removing whitespace
print(message.lower())   # Converting to lowercase
print(message.upper())   # Converting to uppercase
print(message.replace("python", "Java"))  # Replacing substrings

```

  Students will explore built-in string methods to manipulate strings effectively, learning to modify, search, and format string data.

- **Task 5:** String formatting techniques:

```
# String formatting
name = "Alice"
age = 21
formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)

```

This task will introduce various string formatting techniques, including f-strings, to create dynamic and readable output.

- **Task 6:** Practical examples of string operations:

```
# Example of checking substring presence
if "Python" in message:
    print("Python is mentioned in the message.")

# Example of string length
length = len(greeting)
print(f"The length of the greeting is: {length}")

```

Students will apply their knowledge through practical examples, reinforcing their understanding of how strings are utilized in various contexts within Python programming.

## Learning Outcomes:

- Develop a comprehensive understanding of string data types and their significance in Python.
- Gain skills in creating and accessing strings, along with performing basic string operations.
- Learn to utilize built-in string methods for effective string manipulation and formatting.
- Understand various string formatting techniques for creating dynamic text output.
- Prepare students for more advanced string handling and manipulation tasks in future programming assignments.

# 14 Week 14: Working with Strings - Basic Operations

**Objective:** Enhance proficiency in string manipulation through basic operations. Understand string methods for processing text data.

## Tasks:

1. **Task 1:** Review of string data types.
2. **Task 2:** Understanding string indexing and slicing.
3. **Task 3:** Implementing string concatenation and repetition.
4. **Task 4:** Exploring common string methods.
5. **Task 5:** Using string formatting techniques.
6. **Task 6:** Applying strings in practical scenarios.

## Details of Lab Experiment:

- **Task 1:** Review of string data types: Start with a brief overview of string data types in Python. Discuss how strings are immutable sequences of characters and the different ways to define strings (single, double, and triple quotes).
- **Task 2:** Understanding string indexing and slicing:

```
# String indexing and slicing
text = "Hello, Python!"
first_char = text[0]  # Accessing the first character
substring = text[7:13]  # Slicing from index 7 to 12
print(first_char)  # Output: H
print(substring)   # Output: Python

```

  Students will learn to access specific characters and substrings within strings using indexing and slicing techniques.

- **Task 3:** Implementing string concatenation and repetition:

```
# Concatenation and repetition
part1 = "Hello"
part2 = "World"
greeting = part1 + ", " + part2 + "!"
repeated_greeting = part1 * 3  # Repeat the string
print(greeting)          # Output: Hello, World!
print(repeated_greeting) # Output: HelloHelloHello

```

  This task will emphasize combining strings and repeating them to create new text outputs.

- **Task 4:** Exploring common string methods:

```
# Common string methods
phrase = "  Welcome to Python programming!  "
cleaned_phrase = phrase.strip()  # Removing leading/trailing
    whitespace
uppercase_phrase = cleaned_phrase.upper()  # Converting to uppercase
print(cleaned_phrase)  # Output: Welcome to Python programming!
print(uppercase_phrase) # Output: WELCOME TO PYTHON PROGRAMMING!

```

Students will experiment with built-in string methods such as ‘.strip()‘, ‘.lower()‘, and ‘.upper()‘ to manipulate strings effectively.

- **Task 5:** Using string formatting techniques:

```
# String formatting
name = "Alice"
age = 30
formatted_string = f"{name} is {age} years old."
print(formatted_string)  # Output: Alice is 30 years old.

```

This task will introduce students to f-strings for dynamic string creation, allowing them to embed variables directly into strings.

- **Task 6:** Applying strings in practical scenarios:

```
# Practical application
user_input = input("Enter your favorite programming language: ")
if user_input.lower() == "python":
    print("Great choice! Python is a versatile language.")
else:
    print(f"{user_input} is interesting too!")

```

Students will apply their understanding of strings to handle user input, demonstrating how to utilize string methods and conditionals in real-world scenarios.

## Learning Outcomes:

- Master the basics of string manipulation, including indexing, slicing, concatenation, and repetition.
- Gain hands-on experience with built-in string methods for effective string processing.
- Understand and implement string formatting techniques for creating dynamic outputs.
- Apply string operations in practical programming scenarios, reinforcing knowledge through real-world applications.
- Prepare for advanced string handling and manipulation tasks in upcoming assignments and projects.

# 15 Week 15: File Handling - Basics

**Objective:** Develop foundational skills in file handling in Python. Learn how to read from and write to files, enabling data persistence and manipulation.

## Tasks:

1. **Task 1:** Understanding file types and their importance.
2. **Task 2:** Opening and closing files.
3. **Task 3:** Reading from files.
4. **Task 4:** Writing to files.
5. **Task 5:** File modes and context managers.
6. **Task 6:** Handling exceptions in file operations.

## Details of Lab Experiment:

- **Task 1:** Understanding file types and their importance: Begin with a discussion about various file types (e.g., text files, CSV, JSON) and their roles in data storage and exchange. Emphasize the significance of file handling in programming, particularly in data-driven applications.
- **Task 2:** Opening and closing files:

```
# Opening and closing a file
file = open('sample.txt', 'r')  # Open a file in read mode
content = file.read()  # Read the file's content
file.close()  # Close the file
print(content)  # Output: Content of sample.txt

```

  Students will learn the process of opening a file for reading and the importance of closing the file after operations to free up system resources.

- **Task 3:** Reading from files:

```
# Reading from a file
with open('sample.txt', 'r') as file:
    lines = file.readlines()  # Read all lines into a list
for line in lines:
    print(line.strip())  # Output each line without extra whitespace

```

  This task introduces the 'read()', 'readline()', and 'readlines()' methods, demonstrating how to retrieve content from files efficiently.

- **Task 4:** Writing to files:

```
# Writing to a file
with open('output.txt', 'w') as file:
    file.write("Hello, World!\n")  # Write a string to the file
    file.write("This is a file handling example.\n")

```

  Students will learn how to create new files or overwrite existing ones using the 'write()' method, emphasizing proper formatting for multi-line entries.

- **Task 5:** File modes and context managers: Discuss different file modes:

  - ‘’r’‘: Read
  - ‘’w’‘: Write
  - ‘’a’‘: Append
  - ‘’r+’‘: Read and write

  Explain how using context managers (‘with‘ statement) ensures that files are properly closed, even in case of exceptions.

- **Task 6:** Handling exceptions in file operations:

```
# Handling exceptions
try:
    with open('nonexistent.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")

```

  Students will learn to use try-except blocks to manage potential errors during file operations, ensuring that the program remains robust and user-friendly.

## Learning Outcomes:

- Acquire foundational knowledge of file handling, including the significance of files in programming.
- Develop skills to open, read from, and write to files in various modes.
- Understand the concept of context managers and their importance in file operations.
- Gain experience in handling exceptions related to file handling, fostering robust programming practices.
- Prepare for more advanced topics in file handling, including working with different file formats and data serialization techniques.

# 16 Week 16: File Handling - Exception Handling

**Objective:** Understand and implement exception handling techniques in file operations to ensure robust and error-resistant programs.

## Tasks:

1. **Task 1:** Review of file handling concepts.
2. **Task 2:** Introduction to exception handling.
3. **Task 3:** Common exceptions in file operations.
4. **Task 4:** Implementing try-except blocks.
5. **Task 5:** Finally clause and its use cases.
6. **Task 6:** Raising exceptions and creating custom exceptions.

## Details of Lab Experiment:

- **Task 1:** Review of file handling concepts: Begin the lab with a brief recap of the previous week's topics on file handling, emphasizing the importance of effective file operations in programming. Discuss scenarios where files may not be accessible or readable.
- **Task 2:** Introduction to exception handling: Explain the concept of exceptions in Python, focusing on how they help manage errors during program execution. Introduce the syntax of 'try' and 'except' blocks, illustrating their role in error handling.
- **Task 3:** Common exceptions in file operations: Discuss typical exceptions encountered during file handling, such as:
  - 'FileNotFoundError': Raised when trying to access a file that does not exist.
  - 'PermissionError': Raised when trying to open a file without the necessary permissions.
  - 'IOError': Raised for input/output related issues, such as a full disk.

  Provide examples of each exception and discuss strategies to prevent them.
- **Task 4:** Implementing try-except blocks:

```
# Using try-except blocks
try:
    with open('data.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("Error: The specified file was not found.")
except PermissionError:
    print("Error: You do not have permission to access this file.")

```

  Students will practice writing try-except blocks to handle different exceptions that may arise during file operations, reinforcing their understanding of error management.
- **Task 5:** Finally clause and its use cases:

```python
# Using finally clause
try:
    file = open('data.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("Error: The specified file was not found.")
finally:
    file.close()  # Ensures the file is closed regardless of errors

```

Explain how the 'finally' clause is executed whether an exception is raised or not, ensuring that necessary cleanup actions, such as closing files, are always performed.

- **Task 6:** Raising exceptions and creating custom exceptions:

```python
# Raising a custom exception
class CustomFileError(Exception):
    pass

try:
    raise CustomFileError("This is a custom file error message.")
except CustomFileError as e:
    print(e)

```

Discuss how to raise exceptions intentionally using the 'raise' keyword. Guide students in creating custom exception classes to handle specific error scenarios effectively.

## Learning Outcomes:

- Develop a thorough understanding of exception handling in Python, specifically related to file operations.
- Gain practical experience in implementing 'try' and 'except' blocks to manage errors gracefully.
- Learn to utilize the 'finally' clause to ensure that critical cleanup tasks are performed.
- Acquire the skills to raise built-in and custom exceptions, enhancing error reporting and handling strategies.
- Prepare for advanced error handling techniques and practices in larger applications, ensuring robust software development.