Python Intensive Day 2: Control Structures and Functions

Objective:

By the end of Day 2, students will have a deep understanding of Python's control flow mechanisms (conditional statements and loops) and will be able to write reusable functions with various parameters. They will complete multiple exercises and mini-projects to practice these concepts.

1.1 Recap of Day 1 (10 minutes)

Goal:

Briefly review the core concepts learned in Day 1 to ensure all students are on the same page before proceeding with new material.

Topics to Review:

- Variables, data types (string, integer, float, boolean).
- Basic operators (arithmetic, comparison, logical).
- Input/output (`input()`, `print()`).
- Basic programs: Calculator and temperature converter.

1.2 Introduction to Conditional Statements (60 minutes)

Why This is Important:

Conditional statements (`if`, `else`, `elif`) allow programs to make decisions and execute different blocks of code depending on the conditions provided. This enables the creation of dynamic, interactive programs.

`if-else` Statements:

- **Syntax**:

```
if condition:
    # Code to run if the condition is True
else:
    # Code to run if the condition is False
```

Key Points: - Conditions are typically **comparison expressions** (`x == 5`, `x > 3`) or **boolean values** (`is_valid == True`). - Indentation is crucial in Python. Each block of code must be indented correctly to show its logical grouping.

Example:

```
age = int(input("Enter your age: "))

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

elif (else if) Statements:

- **Syntax**:

```
    if condition1:
        # Code to run if condition1 is True
    elif condition2:
        # Code to run if condition2 is True
    else:
        # Code to run if both conditions are False
```

- **Usage**: The `elif` statement allows you to check multiple conditions.

Example:

```
grade = int(input("Enter your exam grade: "))

if grade >= 90:
    print("You got an A!")
elif grade >= 80:
    print("You got a B!")
elif grade >= 70:
    print("You got a C!")
elif grade >= 60:
    print("You got a D.")
else:
    print("You got an F.")
```

Project 1: Grade Calculator (Mini-Project, 15 minutes)

Goal:

Write a program that takes a numerical grade as input and prints a letter grade (A, B, C, D, F) based on the following scale: - 90+ = A - 80-89 = B - 70-79 = C - 60-69 = D - Below 60 = F

Steps:

1. Ask the user to enter their grade.

2. Use `if`, `elif`, and `else` to determine the corresponding letter grade.
3. Print the grade.

Example Code:

```
grade = int(input("Enter your grade: "))

if grade >= 90:
    print("A")
elif grade >= 80:
    print("B")
elif grade >= 70:
    print("C")
elif grade >= 60:
    print("D")
else:
    print("F")
```

Challenge:

- Modify the program to allow multiple grades to be entered. Use a loop to keep asking for grades until the user inputs a negative number, then stop.

1.3 Loops (90 minutes)

Why This is Important:

Loops allow you to repeat actions multiple times, which is crucial for tasks like processing lists, automating tasks, and handling repeated user inputs.

---

for Loops:

- **Purpose**: Used to iterate over a sequence (e.g., list, string, range of numbers).
- **Syntax**:

```
    for variable in sequence:
        # Code to run for each item in sequence
```

Key Points: - Loops are very useful for performing repetitive tasks or iterating over data structures like lists and ranges. - The **`range()`** function is commonly used to generate sequences of numbers.

Example:

```
for i in range(5):  # Iterates 5 times
    print(f"Iteration {i}")
```

Hands-On Exercise: - Write a program that prints the first 10 even numbers using a `for` loop and `range()`.

Code Example:

```
for i in range(0, 20, 2):  # Starts at 0, increments by 2, ends at 19
    print(i)
```

---

while Loops:

- **Purpose**: Repeats a block of code while a specified condition is `True`.
- **Syntax**:

```
    while condition:
        # Code to run while the condition is True
```

Key Points: - Be careful with `while` loops to avoid infinite loops by ensuring the condition will eventually become `False`. - `while` loops are useful when you don't know in advance how many times the loop should run (e.g., waiting for user input).

Example:

```
count = 1
while count <= 5:
    print(f"Count is {count}")
    count += 1
```

Project 2: Number Guessing Game (Mini-Project, 30 minutes)

Goal:

Write a program that randomly selects a number between 1 and 100, and then prompts the user to guess the number. The program should provide feedback ("Too high", "Too low") and keep asking for guesses until the correct number is found.

Steps:

1. Use the `random` module to generate a random number.
2. Use a `while` loop to repeatedly ask the user for a guess.
3. Compare the guess to the target number and provide feedback.
4. End the loop when the user guesses correctly.

Example Code:

```
import random

target = random.randint(1, 100)
guess = None

while guess != target:
    guess = int(input("Guess a number between 1 and 100: "))
```

```
    if guess < target:
        print("Too low!")
    elif guess > target:
        print("Too high!")
    else:
        print("Correct!")
```

Challenge:

- Keep track of how many attempts the user made and print the number of attempts at the end.
- Add an option for the user to play again.

1.4 Nested Loops (45 minutes)

Why This is Important:

Nested loops allow you to perform repeated actions inside other loops. This is useful for working with multi-dimensional data (like tables or grids) or running complex iterative tasks.

Example of Nested Loops:

- **Syntax**:

```
    for i in range(3):
        for j in range(2):
            print(f"i = {i}, j = {j}")
```

Example 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
```

Practice Exercise:

- Write a program that prints a simple multiplication table from 1 to 10 using nested loops.

Code Example:

```
for i in range(1, 11):
    for j in range(1, 11):
        print(f"{i * j:4}", end=" ")
    print()  # Newline after each row
```

1.5 Functions (90 minutes)

Why This is Important:

Functions are reusable blocks of code that help organize and simplify programs. They allow you to break down complex tasks into smaller, manageable pieces.

Defining Functions:

- **Syntax**:

```
    def function_name(parameters):
        # Code block
        return result
```

- **Parameters**: Variables passed to the function as input.
- **Return Statement**: Functions can return values using the `return` keyword.

Example:

```
def greet(name):
    return f"Hello, {name}!"

message = greet("Alice")
print(message)
```

Key Points: - Functions help avoid repetition by allowing you to define code once and reuse it. - Parameters allow functions to work with different data inputs.

Project 3: Area Calculator (Mini-Project, 20 minutes)

Goal:

Write a function that calculates the area of a rectangle. The function should take two parameters: `width` and `height`. It should return the calculated area.

Steps:

1. Define a function called `calculate_area` that takes `width` and `height` as parameters.
2. The function should return the result of multiplying the width by the height.
3. Use the function to calculate and print the area of multiple rectangles with different dimensions.

Example Code:

```
def calculate_area(width, height):
    return width * height
```

```
# Test the function with different inputs
area1 = calculate_area(5, 10)
area2 = calculate_area(3, 7)

print(f"Area of rectangle 1: {area1}")
print(f"Area of rectangle 2: {area2}")
```

Challenge:

- Modify the program to calculate the area of other shapes, such as circles or triangles, by creating additional functions like `calculate_circle_area(radius)` and `calculate_triangle_area(base, height)`.

1.6 Function Parameters and Return Values (60 minutes)

Why This is Important:

Understanding how to pass different types of arguments to functions (positional, keyword, and default arguments) allows for more flexibility in function design.

Positional Arguments:

- **Explanation**: Arguments passed to the function based on the order in which they are defined.
- **Example**:

```
def multiply(a, b):
    return a * b

result = multiply(3, 5)  # Positional arguments
print(result)
```

Keyword Arguments:

- **Explanation**: Arguments passed to the function with the name of the parameter. This allows you to change the order of arguments.
- **Example**:

```
result = multiply(b=5, a=3)  # Keyword arguments
print(result)
```

Default Arguments:

- **Explanation**: You can set default values for function parameters. If no argument is passed for that parameter, the default value is used.
- **Syntax**:

```
def greet(name, message="Hello"):
    return f"{message}, {name}!"

print(greet("Alice"))        # Uses default message
print(greet("Bob", "Hi"))    # Custom message
```

Project 4: Tip Calculator (Mini-Project, 30 minutes)

Goal:

Write a function that calculates the total bill amount, including a tip. The function should take the bill amount and the tip percentage as arguments. If no tip percentage is provided, it should default to 15%.

Steps:

1. Define a function called `calculate_total_bill` that takes `bill_amount` and `tip_percentage` as parameters (with `tip_percentage` defaulting to 15).
2. The function should calculate the tip and add it to the bill amount.
3. Call the function with different inputs to test it.

Example Code:

```
def calculate_total_bill(bill_amount, tip_percentage=15):
    tip = bill_amount * tip_percentage / 100
    total = bill_amount + tip
    return total

# Test the function
total1 = calculate_total_bill(100)
total2 = calculate_total_bill(100, 20)

print(f"Total with default tip: ${total1}")
print(f"Total with 20% tip: ${total2}")
```

Challenge:

- Modify the program to handle splitting the bill between multiple people. Create a function that takes the number of people and divides the total bill by that number.

1.7 Anonymous (Lambda) Functions (30 minutes)

Why This is Important:

Lambda functions are a way to write small, anonymous functions that can be used when you need a short function and don't want to define a full function with a name. This is useful for quick, throwaway functions.

Lambda Functions:

- **Syntax**:

```
lambda arguments: expression
```

- **Example**:

```
multiply = lambda a, b: a * b
print(multiply(3, 5))  # Outputs 15
```

Project 5: Sorting with Lambda Functions (Mini-Project, 20 minutes)

Goal:

Use a lambda function to sort a list of tuples based on the second value in each tuple.

Steps:

1. Create a list of tuples where each tuple contains a name and an associated score.
2. Use the `sorted()` function with a lambda function to sort the tuples by score.

Example Code:

```
students = [("Alice", 85), ("Bob", 75), ("Charlie", 90)]

# Sort by the second element in each tuple (the score)
sorted_students = sorted(students, key=lambda x: x[1])

print(sorted_students)
```

Challenge:

- Modify the program to sort the list by name instead of score, using another lambda function.

1.8 Scope and Global Variables (30 minutes)

Why This is Important:

Understanding scope is crucial when working with functions. Variables can have **local** or **global** scope, and knowing how to manage these will help you write cleaner, error-free code.

Local vs Global Scope:

- **Local Variables**: Defined inside a function and can only be used within that function.

```
def my_function():
    x = 10  # Local variable
    print(x)

my_function()
print(x)  # Error, x is not defined outside the function
```

- **Global Variables**: Defined outside of any function and can be accessed anywhere in the program.

```
x = 10  # Global variable

def my_function():
    print(x)

my_function()  # Can access global variable
```

Project 6: Password Generator (Mini-Project, 40 minutes)

Goal:

Write a function that generates a random password of a specified length using lowercase letters, uppercase letters, digits, and symbols.

Steps:

1. Import the `random` and `string` modules.
2. Create a function that takes the password length as a parameter.
3. Use `random.choice()` to pick random characters from a pool of letters, digits, and symbols.
4. Return the generated password.

Example Code:

```python
import random
import string

def generate_password(length):
    characters = string.ascii_letters + string.digits + string.punctuation
    password = ''.join(random.choice(characters) for i in range(length))
    return password

# Generate a password of length 12
password = generate_password(12)
print(f"Generated password: {password}")
```

Challenge:

- Modify the function to allow the user to specify whether they want to include uppercase letters, digits, or symbols in the password.

1.9 Recap and Homework (15 minutes)

Recap of Key Concepts:

- **Control Flow**: Conditional statements (`if-else`, `elif`) and loops (`for`, `while`).
- **Functions**: How to define functions, pass parameters, and return values.
- **Function Scope**: Difference between local and global variables.
- **Lambda Functions**: Writing anonymous functions for quick tasks.

Homework/Exercises:

1. **Fibonacci Sequence Generator**: Write a function that generates the first `n` numbers in the Fibonacci sequence.

   Example:

   ```python
   def fibonacci(n):
       a, b = 0, 1
       for i in range(n):
           print(a)
           a, b = b, a + b

   fibonacci(10)
   ```

2. **Prime Number Checker**: Write a function that takes a number and checks if it is prime.

   Hint: A prime number is only divisible by 1 and itself.

Day 2 Summary:

By the end of **Day 2**, students will have: - **Mastered control flow** using `if-else`, loops, and nested loops. - Learned how to **define reusable functions** with parameters, return values, and default arguments. - Completed several hands-on projects, including: - A **grade calculator**. - A **number guessing game**. - A **password generator**. - And more!

This strong foundation prepares students for more advanced topics, such as object-oriented programming, which will be introduced in Day 3.