

Recap Handout – ACCA5036 - SPC

Table of Contents

Contents

1. Week 1: Introduction to Software Life Cycle and Programming Basics Using Python	2
2. Week 2: Basic Syntax: I/O Functions and Operators	6
3. Week 3: Control Structures: if, else, elif	9
4. Week 4: Looping: for and while	11
5. Week 5 & 6: Data Structures: Lists, Dictionaries, Tuples, and Sets	13
6. Week 7: User-Defined Functions	16
7. Week 8: Error Handling and Debugging	18
Debugging Techniques in Thonny Editor	19

1. Week 1: Introduction to Software Life Cycle and Programming Basics Using Python

In Week 1, we explore the basics of programming and the software development life cycle (SDLC).

Key Concepts:

- **Software Development Life Cycle (SDLC):**
Phases include planning, analysis, design, implementation, testing, deployment, and maintenance.
- **Programming Basics in Python:**
 - Variables
 - Basic data types (integers, floats, strings)
 - Simple operations

Example: Hello World & Simple Calculation

```
# Hello World Example
print("Hello, World!")
```

```
# Simple calculation example
a = 10
b = 5
sum = a + b
print("Sum:", sum)
```

Real-Time Example:

Imagine developing a simple budgeting app. In the planning phase, you decide the app should add and subtract expenses. The above snippet mimics this basic arithmetic calculation.

Here's a **simple budgeting app** in Python that uses only input() and print() statements (no GUI, no external libraries—just pure console-based interaction).

Features:

- Set your total budget
- Add multiple expenses with descriptions
- View total spent and remaining balance

```
print("=== Welcome to Simple Budgeting App ===")

# Step 1: Set your total budget
budget = float(input("Enter your total budget for the month: ₹ "))

# Variables to track expenses
expenses = []
total_spent = 0

# Step 2: Add expenses
while True:
    print("\nChoose an option:")
    print("1. Add an expense")
    print("2. View all expenses")
    print("3. View summary")
    print("4. Exit")

    choice = input("Enter your choice (1-4): ")

    if choice == "1":
        description = input("Enter expense description: ")
        amount = float(input("Enter amount spent (₹): "))
        expenses.append((description, amount))
        total_spent += amount
        print(f"Added: {description} - ₹{amount}")
    elif choice == "2":
        print("\n📖 Your Expenses:")
        for i, (desc, amt) in enumerate(expenses, start=1):
            print(f"{i}. {desc} - ₹{amt}")
    elif choice == "3":
        remaining = budget - total_spent
        print("\nBudget Summary:")
        print(f"Total Budget: ₹{budget}")
        print(f"Total Spent: ₹{total_spent}")
        print(f"Remaining Balance: ₹{remaining}")
        if remaining < 0:
            print("You are over budget!")
    elif choice == "4":
        print("\nExiting the app. Stay on budget!")
        break
    else:
        print("Invalid choice. Please enter 1-4.")
```

- What is the expected output when you run the below snippets?

```
x = input("Enter a number: ")
print(x * 2)
```

Why?

input() returns a **string**, so "5" * 2 repeats the string — not numerical multiplication.

To fix: int(x) * 2

```
print("The sum is", 5 + 3)
print("The sum is" + 5 + 3)
```

Why?

When using +, everything must be of the same type. You can't add a string and an integer.

```
a = input("Enter a number: ")
b = input("Enter another number: ")
print("Sum is:", a + b)
```

Why?

Both inputs are strings, so it's string concatenation.

To fix: int(a) + int(b)

```
print("""Hello World""")
```

Why?

Python allows nested quotes. Triple double quotes allow for internal quotes without escape sequences.

```
expr1 = input("Enter an expression: ")
print("Output:", expr1)

expr2 = eval(input("Enter an expression: "))
print("Output:", expr2)
```

Input 5 + 3 for the both and observe the output.

Why?

expr1- input() just returns a **string**, and you're printing that string. It does **not evaluate** the math expression.

expr2- Python actually **executes** the expression you typed, thanks to eval().

```
eval(input("Enter a print command: "))
```

Try the User Input(s) below one by one and observe the output;

```
print("Hello!")
print(5 + 10)
print("A" * 3)
print(input("Nested input: "))
```

```
print("=== Expression Tester: With vs Without eval() ===")

user_input = input("Enter a Python expression (e.g., 2 + 3, 'A' * 5, [1, 2]*2): ")

# Without eval: Just shows the string
print("\n Without eval():")
print("Output:", user_input)

# With eval: Actually executes the expression
print("\n With eval():")
try:
    result = eval(user_input)
    print("Output:", result)
except Exception as e:
    print("Error while evaluating:", e)
```

2. Week 2: Basic Syntax: I/O Functions and Operators

In Week 2, you learn how to handle input/output and the usage of various operators.

Key Concepts:

- **Input and Output Functions:**
 - `input()` for taking user input
 - `print()` for displaying output
- **Operators:**
 - Arithmetic: `+`, `-`, `*`, `/`, `%`
 - Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
 - Logical: `and`, `or`, `not`

Example: Simple Calculator

```
# Taking user input
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

# Performing basic arithmetic operations
print("Addition:", num1 + num2)
print("Subtraction:", num1 - num2)
print("Multiplication:", num1 * num2)
print("Division:", num1 / num2 if num2 != 0 else "Undefined")
```

Arithmetic Operators:

```
print("5" + "5")
print("5" * 3)
```

Why?

String `+` concatenates, and `*` repeats — no math happening here.

```
print(5 / 2)
print(5 // 2)
```

Why?

`/` gives float division. `//` is integer (floor) division.

```
print(-7 % 3)
```

Why?

Python ensures the result of `%` is always **non-negative** if the divisor is positive.

Comparison Operators:

```
print("5" == 5)
```

Why?

Different types: string vs int — not equal.

```
print(1 < 2 < 3)
print(1 < 2 > 3)
```

Why?

Python supports chained comparisons: $1 < 2$ and $2 < 3$

```
print(0.1 + 0.2 == 0.3)
```

Why?

Floating point precision error! Use `round()` or `math.isclose()`.

```
print(round(0.1 + 0.2, 10) == round(0.3, 10))
```

```
import math
print(math.isclose(0.1 + 0.2, 0.3))
```

Logical Operators:

```
print(0 or 5)
print(1 and 3)
```

Why?

Logical operators return the last evaluated value, not just True/False.

```
print(not 0)    # True
print(not 10)   # False
```

Why?

not checks truthiness: 0 is False, any non-zero number is True.

```
def check():
    print("called")
    return True

# All combinations of True/False with 'and' and 'or'
print("False and check():", False and check()) # check() not called
print("True and check():", True and check())    # check() called

print("False or check():", False or check())    # check() called
print("True or check():", True or check())      # check() not called
```

Why?

Python short-circuits and and or — stops evaluating as soon as result is known.

- and: If the first operand is False, Python doesn't evaluate the second (short-circuits).
- or: If the first operand is True, Python doesn't evaluate the second.

Real-Time Example:

A real-time scenario could be an ATM machine interface where the user inputs the amount to withdraw or deposit, and the system performs calculations to update the balance.

How It Works:

- **PIN Validation:**
The user is prompted to input the PIN. If it matches the stored PIN ("1234"), the program proceeds.
- **Transaction Options:**
The user selects either deposit or withdraw by entering 1 or 2.

- **Deposit:**
The user inputs an amount to deposit. The program uses eval() to convert the input to a number, checks if the amount is positive, and then updates the balance by adding the amount.
- **Withdraw:**
Similarly, the user inputs an amount to withdraw. The program checks that the withdrawal amount is positive and that sufficient funds are available before subtracting the amount from the balance.
- **Error Handling:**
If any validation fails (incorrect PIN, invalid amount, insufficient funds, or wrong operation choice), the program prints an appropriate error message.

```
# Simple ATM Machine Interface
# Set the correct PIN and starting balance
correct_pin = "1234"
balance = 1000.0
# Ask the user to enter the PIN
user_pin = input("Enter your 4-digit PIN: ")
# Check if the PIN is correct
if user_pin == correct_pin:
    print("PIN accepted.")
    print("Current balance: $", balance)
    # Provide the transaction options
    print("\nSelect Transaction:")
    print("1. Deposit")
    print("2. Withdraw")
    operation = input("Enter your choice (1 or 2): ")
    # Deposit Operation
    if operation == "1":
        # Using eval to convert the input to a number (be cautious with eval in real applications)
        amount = eval(input("Enter amount to deposit: "))
        if amount > 0:
            balance = balance + amount # Arithmetic operation: addition
            print("Deposit successful.")
            print("New balance: $", balance)
        else:
            print("Invalid deposit amount.")
    # Withdraw Operation
    elif operation == "2":
        amount = eval(input("Enter amount to withdraw: "))
        if amount > 0:
            # Use relational operator to check for sufficient funds
            if amount <= balance:
                balance = balance - amount # Arithmetic operation: subtraction
                print("Withdrawal successful.")
                print("New balance: $", balance)
            else:
                print("Insufficient funds. Transaction cancelled.")
        else:
            print("Invalid withdrawal amount.")

    else:
        print("Invalid selection. Please restart the transaction.")
else:
    print("Incorrect PIN. Access denied.")
```

3. Week 3: Control Structures: if, else, elif

This week, you'll understand how to control the flow of your program using conditional statements.

Key Concepts:

- **if statement:** Executes code block if condition is true.
- **elif statement:** Checks multiple conditions.
- **else statement:** Executes code if previous conditions are false.

Example: Grading System

```
score = float(input("Enter your score: "))
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print("Your grade is:", grade)
```

Real-Time Example:

A grading system at a university where the final grade is determined by the student's score can be implemented using conditional statements.

```
x = 5
if x > 0:
    print("Positive")
    print("Still in if")
print("Outside if")
```

Why?

The last print() is outside the if — indentation is the only thing defining blocks in Python!

```
x = 10
if x < 10:
    print("Less than 10")
elif x < 20:
    print("Less than 20")
else:
    print("Something else")
```

Why?

x = 10 is **not** < 10, so it skips the first if — and elif x < 20 is True, so that one runs.

```
x = "False"
y = ""

if x:
    print("x: It's True")
else:
    print("x: It's False")

if y:
    print("y: It's True")
else:
    print("y: It's False")
```

Why?

Any **non-empty string** is truthy — even the string "False"!

```
x = 7
if x > 5:
    print("Greater than 5")
elif x > 6:
    print("Greater than 6")
```

Why?

Only the **first matching condition** runs — even if more are true. elif isn't a backup if; it's an exclusive chain.

```
x = 0

if x == True:
    print("x is True")
elif x == False:
    print("x is False")
else:
    print("x is... confusing")
```

Why?

0 == False is True. In Python, False is equal to 0, and True is 1.

```
x = 10
if x > 5: print("A"); print("B")
```

Why?

Only "A" is part of the if — the second print is **not indented**, so it runs no matter what.

4. Week 4: Looping: for and while

Looping is essential for iterating over sequences or executing a block repeatedly.

Key Concepts:

- **for loop:** Iterates over a sequence (like lists or ranges).
- **while loop:** Repeats as long as a condition is true.

Example: Using For and While Loops

```
# For loop example: Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print("Fruit:", fruit)

# While loop example: Counting from 1 to 5
count = 1
while count <= 5:
    print("Count:", count)
    count += 1
```

Real-Time Example:

A real-time scenario for a for loop could be printing out a list of users, while a while loop might be used to create a countdown timer in an event registration system.

```
nums = [1, 2, 3, 4]
for num in nums:
    if num % 2 == 0:
        nums.remove(num)
print(nums)
```

Why?

You're changing the list while iterating — for uses an internal index, so it skips items after a removal.

```
x = 0
while x < 5:
    print(x)
    x += 1
    if x == 3:
        x = -1
```

Why?

Infinite Loop Alert! You're resetting `x = -1`, so the loop never breaks. Be cautious with logic that "rewinds."

```
for i in "123":
    i = int(i) + 1
print(i)
```

Why?

The loop variable gets reassigned, but only the **last value** survives — here, `i = int("3") + 1`.

```
for i in range(3):  
    print(i)  
else:  
    print("Done!")
```

Wait what?!

Yes, Python for/while can have an else, which runs **only if the loop wasn't broken with break**.

```
for i in []:  
    print("You won't see me")  
else:  
    print("But you'll see me!")
```

Why?

Since the loop didn't run at all, the else block still executes. Think of else as “no interruption occurred.”

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

Why?

(5) is just an int, not a tuple. Use (5,) if you want a tuple with one item.

```
for i in range(3):  
    for j in range(3):  
        print(i, j)  
        if j == 1:  
            break
```

Why?

break only exits the inner loop, not both. To break out of nested loops, use flags or return.

5. Week 5 & 6: Data Structures: Lists, Dictionaries, Tuples, and Sets

These weeks cover the different built-in data structures in Python, each suited for various tasks.

Key Concepts:

- **List:** Ordered, mutable sequence.
- **Dictionary:** Unordered, mutable collection of key-value pairs.
- **Tuple:** Ordered, immutable sequence.
- **Set:** Unordered collection of unique elements.

Example: Working with Data Structures

List example

```
numbers = [1, 2, 3, 4, 5]
numbers.append(6)
print("List:", numbers)
```

Dictionary example

```
student = {"name": "Alice", "age": 20, "major": "Computer Science"}
print("Student Name:", student["name"])
```

Tuple example

```
dimensions = (1920, 1080)
print("Dimensions:", dimensions)
```

Set example

```
unique_numbers = {1, 2, 2, 3, 4, 4, 5}
print("Unique Numbers:", unique_numbers)
```

Real-Time Example:

- **Lists:** Used to store a collection of items such as a playlist of songs.
- **Dictionaries:** Can represent a database record, like a student's information.
- **Tuples:** Ideal for fixed configurations such as RGB color values.
- **Sets:** Useful when you need to remove duplicate entries, like unique user IDs.

Below are several “tricky” Python code snippets that illustrate subtle behaviors for each data structure. Try running each snippet and consider the following questions:

- What is the output?
- Why might the behavior be unexpected?
- How could you modify the code if needed?

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        numbers.remove(num)
print("Modified list:", numbers)
```

- ☐ What is the final content of numbers?
- ☐ Why might some even numbers be skipped during removal?

Explanation:

- When iterating over numbers, removing an element shifts the positions of the subsequent items.
- For example, after removing 2, the list becomes [1, 3, 4, 5] and the iterator moves to the next index, causing the element 3 to be processed next.
- As a result, even numbers that follow immediately after a removed element might be skipped. In this case, both even numbers (2 and 4) are removed, leaving [1, 3, 5].

```
original = [[1, 2], [3, 4]]
shallow_copy = original[:] # Creates a shallow copy
shallow_copy[0][0] = 100
print("Original list after modification:", original)
```

- ☐ How does modifying an element in the shallow copy affect the original list?
- ☐ What would you do to avoid this behavior?

Explanation:

- The slicing ([:]) creates a shallow copy, which means that only the outer list is copied.
- The inner lists are still referenced by both original and shallow_copy.
- Therefore, modifying an element in one of the inner lists affects both lists.

```
keys = ["apple", "banana", "apple"]
fruit_dict = {k: len(k) for k in keys}
print("Dictionary with duplicate keys:", fruit_dict)
```

- ☐ What happens when duplicate keys appear in a dictionary comprehension?
- ☐ Which value is retained for the key "apple"?

Explanation:

- In a dictionary comprehension, if duplicate keys are encountered, the last occurrence overrides the previous one.
- The key "apple" appears twice; its value (length 5) from the second occurrence overwrites the first.

```
d = dict.fromkeys(["x", "y", "z"], [])
d["x"].append(1)
print("Dictionary with shared mutable default:", d)
```

- ☐ Why does appending to d["x"] affect all keys?
- ☐ How would you initialize the dictionary so that each key gets its own list?

Explanation:

- The fromkeys method uses the same list object as the default value for all keys.
- When you append to the list for key "x", it affects the list for all keys since they all reference the same object.
- To give each key its own list, you can use a dictionary comprehension:

```
d = {key: [] for key in ["x", "y", "z"]}
```

```
t = ([1, 2], 3, 4)
t[0].append(99)
print("Tuple with a modified mutable element:", t)
```

- ☐ Although tuples are immutable, why can you still modify the list inside the tuple?
- ☐ What precautions should you take when including mutable elements in a tuple?

Explanation:

- Tuples are immutable, meaning you cannot reassign an element of the tuple.
- However, if an element within the tuple is a mutable object (like a list), that object can be modified.
- In this snippet, the list [1, 2] inside the tuple is modified by appending 99.

```
s = set([4, 2, 2, 3, 1, 4])
print("Set with duplicates removed:", s)
```

- ☐ How does Python treat duplicate values when constructing a set?
- ☐ Is the order of elements in the printed set guaranteed?

Explanation:

- Sets automatically eliminate duplicate elements.
- The order in which the elements are printed is arbitrary and not guaranteed, as sets are unordered collections.

```
s = {frozenset({1, 2}), frozenset({2, 1})}
print("Number of unique frozensets:", len(s))
```

- ☐ Even though the two frozensets are defined with different literal orders, why does the set consider them equal?
- ☐ What property of frozenset ensures this behavior?

Explanation:

- A frozenset is an immutable and hashable set.
- Two frozenset objects with the same elements are considered equal, regardless of the order in which the elements are provided.
- Therefore, even though they are defined with different literal orders, they are seen as the same object, and only one unique frozenset remains in the set.

6. Week 7: User-Defined Functions

In this week, you learn to write reusable blocks of code using functions.

Key Concepts:

- **Defining Functions:** Using the def keyword.
- **Parameters and Arguments:** Passing values to functions.
- **Return Statement:** Returning results from functions.

Example: Creating a Function to Calculate Factorial

```
def factorial(n):  
    """Return the factorial of n."""  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result  
  
# Test the function  
num = 5  
print(f"Factorial of {num} is:", factorial(num))
```

Real-Time Example:

Functions are used in web applications to handle repeated tasks, such as processing user input or performing complex calculations for data analysis.

```
def add_item(item, my_list=[]):  
    my_list.append(item)  
    return my_list  
  
# First call — expected to start with an empty list.  
print("First call:", add_item(1))  
# Second call — you might expect a new list, but the default list is reused.  
print("Second call:", add_item(2))
```

- ☐ The default list is created only once when the function is defined.
- ☐ Subsequent calls without an explicit list continue to modify the same list.

```
def add_item(item, my_list=None):  
    if my_list is None:  
        my_list = []  
    my_list.append(item)  
    return my_list  
  
# First call — expected to start with an empty list.  
print("First call:", add_item(1))  
# Second call — you might expect a new list, but the default list is reused.  
print("Second call:", add_item(2))
```

- use a default value of None and create a new list inside the function:

```
def make_multiplier(n):  
    def multiplier(x):  
        return x * n  
    return multiplier  
  
times3 = make_multiplier(3)  
print("3 multiplied by 10 is:", times3(10))
```

- ☐ The function `make_multiplier` returns a new function that “remembers” the value of `n`.
- ☐ This demonstrates a closure where the inner function retains access to the variable `n` even after the outer function has finished.

```
def combine(*args, **kwargs):  
    print("Positional arguments:", args)  
    print("Keyword arguments:", kwargs)  
  
combine(1, 2, 3, a="apple", b="banana")
```

- ☐ The `*args` syntax collects all positional arguments into a tuple.
- ☐ The `**kwargs` syntax collects all keyword arguments into a dictionary.

7. Week 8: Error Handling and Debugging

This final week introduces error handling techniques to write robust code and tools/methods to debug errors.

Key Concepts:

- **Error Handling:**
 - Using try, except, else, and finally blocks
- **Debugging:**
 - Print statements, logging, and interactive debuggers
- **Common Exception Types:**
 - ValueError, TypeError, IndexError, etc.

Example: Handling Division Errors

```
def safe_divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
        return None
    else:
        print("Division successful.")
        return result
    finally:
        print("Execution of safe_divide complete.")

# Test cases
print("Result:", safe_divide(10, 2)) # Expected output: Division successful and result 5.0
print("Result:", safe_divide(10, 0)) # Expected to catch an error and print an error message
```

Key Points:

- try: Code that might raise an exception.
- except: Block to catch specific exceptions.
- else: Runs if no exception occurs.
- finally: Always executes, even if an exception was raised.

```
def tricky_return():
    try:
        print("In try block")
        return "from try"
    except Exception:
        print("In except block")
        return "from except"
    finally:
        print("In finally block")
        return "from finally" # This return overrides the try/except returns

result = tricky_return()
print("Result:", result)
```

Explanation:

- **Flow:** The try block executes and returns "from try". However, before the function completes, the finally block always runs.
- **Override:** Since the finally block also contains a return statement, its value ("from finally") overrides any previous return from try or except.
- **Trick:** This behavior can be unexpected if you assume the return in the try block would be final. In practice, avoid returning values from within finally to prevent such confusion.

```
def square_root(x):  
    if x < 0:  
        raise ValueError("Cannot compute square root of a negative number.")  
    return x ** 0.5  
  
try:  
    print("Square root:", square_root(9))  
    print("Square root:", square_root(-4))  
except ValueError as e:  
    print("Caught an error:", e)
```

Explanation:

- **Function Logic:**
The square_root function checks if x is negative. If it is, it raises a ValueError with an appropriate message; otherwise, it returns the square root of x.
- **Error Handling:**
In the try block, the function is called first with a positive value (9), which works correctly, and then with a negative value (-4), which triggers the ValueError. The except block catches the ValueError and prints the error message.

Real-Time Example:

Error handling is crucial when building systems like online payment gateways, where improper input or network issues can cause unexpected failures.

Debugging Techniques in Thonny Editor

Thonny is an excellent beginner-friendly Python IDE that includes several built-in debugging tools:

Debugging Tips in Thonny:

- **Breakpoints:** Click next to the line number in Thonny to set a breakpoint. The execution will pause at these points, allowing you to inspect variables.
- **Step Over / Step Into:** Use these buttons in Thonny to execute your code line by line or dive into function calls.
- **Variable Inspector:** Check the current values of variables in the "Variables" pane while debugging.
- **Exception Report:** When an error occurs, Thonny highlights the error line and provides a traceback that you can click to navigate to the error.

Example Debugging Scenario

Consider this snippet with a logical error that you might want to debug in Thonny:

```
def calculate_average(numbers):  
    total = 0  
    # Intentional error: Loop should iterate over all numbers.  
    for i in range(len(numbers) - 1):  
        total += numbers[i]  
    average = total / len(numbers)  
    return average  
  
# Test case: Observe unexpected average due to error in loop  
numbers = [10, 20, 30, 40]  
print("Average:", calculate_average(numbers))
```

Debugging Steps in Thonny:

1. **Set a Breakpoint:** Click next to the line with the for loop.
2. **Run in Debug Mode:** Use the debug button to start the code.
3. **Step Through:** Step through each iteration to observe that the loop omits the last element.
4. **Inspect Variables:** Look at the variable total and the current index i to understand the logic error.
5. **Fix the Error:** Modify the loop to iterate over the full range:

```
for i in range(len(numbers)):  
  
    total += numbers[i]
```

6. **Rerun the Code:** Ensure that the average is now correctly computed.