THE DEVELOPER ARENA

PART 1: MONTH 1 – PYTHON BASICS AND DATA MANIPULATION

WEEK 0 1

Python is one of the most versatile, beginner-friendly programming languages in the world. In this module, I explored Python fundamentals through theory, hands-on practice, and small client-style projects to apply real-world logic. This report summarizes everything I've done — from writing basic programs to solving logic puzzles and building interactive games.

- **Variables**

Variables store data in memory during execution. Python is dynamically typed, so you don't need todeclare the type.

```
name = "Alice"  # string
age = 22        # integer
temp = 36.7     # float
```

- **Data Types**

Data Type	Description	Example
int	Whole numbers	x = 10
float	Decimal numbers	pi = 3.14
str	Text/characters	name = "Baby"
bool	Boolean values	is_adult = True
list	Ordered collections	colors = ["red", "blue"]
dict	Key-value pairs	person = {"name": "Zoe"}

- **Operators:**
- Arithmetic: + - * / % // **
- Comparison: == != > < >= <=
- Logical: and or not
- Assignment: = += -= *=

- **Input and Output:**

- name = input("Enter your name: ")
- print("Welcome,", name)

- **Conditional Statements**

  Used to perform different actions based on conditions using if, elif, and else.

```
marks = 85
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
else:
    print("Grade: C")
```

- **Loops**
- For Loop – known number of iterations

```
for i in range(1, 6):
    print("Number:", i)
```

- While Loop – runs until condition is false

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

- **Hands-On Programs Developed:**
  - **Simple Calculator (with basic operators)**

```
a = float(input("First number: "))
op = input("Operator (+, -, *, /): ")
b = float(input("Second number: "))
if op == '+':
        print("Result:", a + b)
elif op == '-':
         print("Result:", a - b)
elif op == '*':
         print("Result:", a * b)
elif op == '/':
          print("Result:", a / b)
else:
           print("Invalid operator")
```

Client Project

This mini-game combines logic, random module, loops, and conditions — and is highly engaging to demonstrate real-time interaction.

```
import random

secret = random.randint(1, 100)
attempts = 0
max_attempts = 7

print("🎯 Guess a number between 1 and 100")

while attempts < max_attempts:
        guess = int(input("Your guess: "))
         attempts += 1

        if guess == secret:
             print("🎉 Correct! You won.")
             break
        elif guess < secret:
             print("Too low.")
        else:
              print("Too high.")
else:
     print("❌ Out of tries! The number was", secret)
```

SUMMARY OF WEEK 01

Variables and Data Types:

Variables are used to store data values. Python supports various data types like integers, floats, strings, and booleans. Python is dynamically typed, meaning you don’t need to declare the type explicitly — it infers the type based on the value assigned.

Input and Output:

Input is taken from the user using input statements, which allows interactive programs. Output is displayed using print statements. These are essential for communicating with the user and debugging.

Operators:

Operators are sp:ecial symbols used to perform operations on variables and values.
Arithmetic operators are used for mathematical operations.
Comparison operators compare two values and return a boolean.
Logical operators (and, or, not) combine multiple conditions.
Assignment operators assign or update values in variables.

Conditional Statements

Conditional statements are used to control the flow of execution. They allow the program to make decisions based on specific conditions.
The basic structure includes if, elif, and else blocks to evaluate and respond to different logical branches.

Loops

Loops are used to execute a block of code multiple times. Python supports two types of loops:
For loops are used when the number of iterations is known or to iterate over a sequence.
While loops are used when the number of iterations depends on a condition being true.
Loops help automate repetitive tasks and are essential for efficient programming.

WEEK 0 2

- **Lists**

Ordered, mutable collections (can change).

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")     # Add item
fruits.remove("banana")     # Remove item
print(fruits[0])            # Access first item
```

Useful Methods:
append(), pop(), remove(), sort(), reverse(), len()

- **Tuples**

Ordered but immutable (can't change once created).

```
coords = (10, 20)
print(coords[0])  # Output: 10
```

Use when values shouldn't be modified.

- **Dictionaries**

Key-value pairs (like real-world data labels).

```
student = {"name": "Baby", "age": 21}
print(student["name"])
student["age"] = 22
```

Useful Methods:
get(), keys(), values(), items(), update()

- **Sets**

Unordered collections of unique values.

```
nums = {1, 2, 3, 2}
print(nums)  # Output: {1, 2, 3}
```

Set Operations:
union(), intersection(), difference()

- **Functions – The Brains of Reusable Code**

A block of code that does a task and can be called multiple times without rewriting the same logic.

- Syntax:

```
def function_name(parameters):
    # body
    return output
```

- Example:

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

print(greet("suhas"))  # Output: Hello, Suhas!
```

- **Lambda Functions – Shortcuts for Small Logic**

A one-liner anonymous function used for quick tasks like filtering, mapping, sorting.

Syntax:

```
lambda args: expression
```

Example:

```
square = lambda x: x**2
print(square(5))  # Output: 25
```

Works great with map(), filter(), sorted(), etc.

With filter():

```
nums = [1, 2, 3, 4, 5]
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)  # Output: [2, 4]
```

With map():

```
nums = [1, 2, 3]
squares = list(map(lambda x: x*x, nums))
print(squares)  # Output: [1, 4, 9]
```

- **Recursion – When a Function Calls Itself**

A function that calls itself to break a problem into smaller versions of itself.

Example: Factorial

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

When to Use:
Tree traversal
Factorials
Fibonacci
Repeating nested logic

- **List Comprehension – Pythonic Loops in One Line**

A compact way to create lists from existing lists using for and optional if.

Syntax:
[expression for item in iterable if condition]

Use case	Best Tool
Repeating logic with variation	Function
One-liner filtering or math	Lambda
Breaking into smaller problems	Recursion
Quick data transformation	List comprehension

- **Hands-On Exercises (Transformation Functions)**

A. **Sum of Squares**

```
def sum_of_squares(nums):
   return sum([x**2 for x in nums])
nums = [1, 2, 3, 4]
print("Sum of squares:", sum_of_squares(nums))  # Output: 30
```

B. Filter Even Numbers

```
def filter_even(nums):
   return [x for x in numbers if x % 2 == 0]
nums = [1, 2, 3, 4, 5, 6]
print("Even numbers:", filter_even(nums))  # Output: [2, 4, 6]
```

C. Use Lambda with Filter

```
nums = [10, 15, 20, 25]
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)  # Output: [10, 20]
```

D. Capitalize List of Words

```
words = ["python", "is", "fun"]
capitalized = [w.capitalize() for w in words]
print(capitalized)  # ['Python', 'Is', 'Fun']
```

Client Script: Remove Duplicates & Clean List

```
def clean_data(data_list):
   # Remove empty strings
     data_list = [item.strip() for item in data_list if item.strip() != ""]

   # Remove duplicates using set and preserve order
     clean = []
     seen = set()
     for item in data_list:
          if item not in seen:
               clean.append(item)
               seen.add(item)

          return clean

# Sample messy data
raw_data = ["apple", "banana", "apple", "  ", "banana", "mango", ""]

# Clean it!
cleaned = clean_data(raw_data)
print("Cleaned Data:", cleaned)
```

SUMMARY OF WEEK 02

Lists

Lists are ordered, mutable collections used to store multiple items in a single variable. They allow indexing, slicing, and dynamic modification of values. Lists are useful when working with sequences of related data that might need to be updated or iterated over.

Tuples

Tuples are similar to lists but are immutable, meaning their contents cannot be changed after creation. They are often used for fixed sets of values or to return multiple values from a function. Tuples also support indexing and can hold different data types.

Dictionaries

Dictionaries are unordered, mutable collections of key-value pairs. They allow fast data lookup based on unique keys instead of numerical indexes. Dictionaries are ideal for storing structured data such as records, configurations, or mappings.

Sets

Sets are unordered collections that store only unique elements. They are used to eliminate duplicates and perform operations like union, intersection, and difference. Sets are useful when working with membership checks and mathematical set operations.

Functions

Functions are reusable blocks of code designed to perform specific tasks. They improve modularity, readability, and maintainability of code. Functions can take input parameters, perform operations, and return output. They help reduce repetition and make code more organized.

Lambda Functions

Lambda functions are anonymous, one-line functions often used for simple operations. They are useful when a full function definition is unnecessary, especially in combination with other functions like map, filter, and sorted.

Recursion

Recursion is a technique where a function calls itself to solve smaller subproblems of the original problem. It is useful for tasks that can be defined in terms of themselves, such as computing factorials or traversing nested structures. Every recursive function must have a base case to avoid infinite loops.

List Comprehension

List comprehension is a concise way to create lists using a single line of code. It combines loops and conditionals into a compact expression and is used for filtering, transforming, or generating new lists from existing iterables. It improves readability and performance in data manipulation tasks.

WEEK 0 3

- **NumPy – Numerical Python**

A high-performance library used for numerical calculations, multi-dimensional arrays, and linear algebra. It is faster than regular Python lists.

1. **NumPy Arrays**

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr)  # Output: [1 2 3 4]
```

2. Array Properties

```
print(arr.shape)     # (4,)
print(arr.dtype)     # int64
print(arr.ndim)
```

3. Array Creation

```
np.zeros((2, 3))         # 2x3 array filled with 0s
np.ones((3, 3))          # 3x3 array of 1s
np.arange(0, 10, 2)      # array([0, 2, 4, 6, 8])
np.linspace(0, 1, 5)     # 5 evenly spaced values from 0 to 1
```

4. Array Operations

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b)  # element-wise addition → [5 7 9]
print(a * b)  # element-wise multiplication → [4 10 18]
print(a.mean())  # average of array
```

5 Broadcasting :

```
a = np.array([1, 2, 3])
b = 2
print(a * b)  # → [2 4 6]
```

NumPy can automatically expand dimensions for compatible operations – this is called broadcasting.

Pandas :

A powerful library for working with tabular data, like CSVs, Excel files, or SQL queries. You use DataFrames (tables) and Series (columns).

1. Series:

```
import pandas as pd
s = pd.Series([10, 20, 30])
print(s)
```

2**. DataFrame:**

```
data = {
   "Name": ["Alice", "Bob", "Charlie"],
   "Age": [25, 30, 22]
}
df = pd.DataFrame(data)
print(df)
```

3. Read and Write CSV:

```
p
df = pd.read_csv("data.csv")
df.to_csv("output.csv", index=False)
```

4. Indexing & Slicing:

```
df['Age']          # single column (Series)
df[['Name', 'Age']] # multiple columns
df.iloc[0]         # row by index
df.loc[0, 'Age']    # specific cell
```

5. Filtering Rows:

```
df[df['Age'] > 25]
```

6. **Handling Missing Values:**

```
df.dropna()            # remove rows with missing
df.fillna(0)           # fill missing with 0
df['Age'].mean()       # calculate average
```

7. Grouping and Aggregation:

```
df.groupby('Gender')['Age'].mean()
```

HANDS-ON

NumPy: Basic Math

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print("Mean:", arr.mean())
print("Sum:", arr.sum())
print("Squared:", arr ** 2)
```

Pandas: Basic Data Manipulation

```
import pandas as pd
data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'Alice'],
   'Age': [25, None, 30, 25],
   'Gender': ['F', 'M', 'M', 'F']
}
df = pd.DataFrame(data)

# Drop duplicates
df = df.drop_duplicates()

# Fill missing age
df['Age'] = df['Age'].fillna(df['Age'].mean())

# Group by gender
print(df.groupby('Gender')['Age'].mean())
print("Average Sales by Gender:\n", avg_sales)

# Save cleaned file
df.to_csv("cleaned_sales_data.csv", index=False
```

CLIENT PROJECT – DATA CLEANING + AVERAGE CALCULATION

Name	Age		Sales
Alice	25	F	500
Bob	NaN	M	700
Charlie	30	M	650
Alice	25	F	500

GOAL:

- Remove duplicates
- Handle missing values
- Calculate average sales by gender

CLEANING SCRIPT:

```
import pandas as pd

# Load dataset
df = pd.read_csv("sales_data.csv")

# Drop duplicate rows
df = df.drop_duplicates()

# Fill missing Age with average age
df['Age'] = df['Age'].fillna(df['Age'].mean())

# Group by Gender and calculate average sales
avg_sales = df.groupby('Gender')['Sales'].mean()

)
```

OUTPUT:

```
Average Sales by Gender:
Gender
F    500.0
M    675.0
```

SUMMARY OF WEEK 03

- **NumPy (Numerical Python)**

NumPy is a core library for numerical computations in Python. It provides support for multi-dimensional arrays (called ndarrays) and a wide range of mathematical operations on these arrays.

- **NumPy Arrays**

NumPy arrays are more efficient than Python lists for large numerical data. They support vectorized operations, which allow element-wise calculations without explicit loops.

- **Array Operations**

NumPy supports operations like addition, subtraction, multiplication, division, and more on entire arrays. It also includes mathematical functions like mean, standard deviation, square root,

- **Broadcasting**

Broadcasting is a powerful feature in NumPy that allows operations between arrays of different shapes by automatically expanding one array to match the shape of the other. It helps perform arithmetic operations without writing explicit loops.

Pandas is a high-level data analysis library built on top of NumPy. It provides powerful data structures — Series and DataFrame — for handling structured data.

A DataFrame is a two-dimensional labeled data structure with rows and columns, similar to an Excel sheet or SQL table. It allows easy indexing, filtering, grouping, and reshaping of data. Pandas provides intuitive ways to access data using .loc[] for label-based indexing and .iloc[] for position-based indexing. It allows selecting rows, columns, or individual cells easily.
Pandas offers built-in methods to handle missing values, such as dropna() to remove rows/columns with missing values and fillna() to replace them with default or computed values.

The groupby() function in Pandas allows for easy grouping of data based on one or more keys and then applying aggregation functions like sum(), mean(), count(), etc., to gain insights from grouped data.

WEEK 0 4

Data Visualization with Matplotlib and Seaborn

Matplotlib

Matplotlib is a 2D plotting library in Python that allows you to create static, animated, and interactive visualizations. It is especially good for basic plots like line graphs, bar charts, histograms, and scatter plots.

- It is simple and flexible.
- It gives full control over every part of a figure (axes, labels, legends, colors, etc.).
- It is compatible with NumPy and Pandas.

Basic Structure of a Plot

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [5, 7, 4]
plt.plot(x, y)
plt.title("My Plot Title")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.show()
```

Commonly Used Plots in Matplotlib

- **Line Plot**

Used to display trends or continuous data.

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 20, 15, 25]
plt.plot(x, y, marker='o', color='green')
plt.title("Line Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```

- **Bar Plot**

Used for categorical data comparison.

```
x = ['A', 'B', 'C']
y = [10, 20, 15]
plt.bar(x, y, color='blue')
plt.title("Bar Plot Example")
plt.xlabel("Category")
plt.ylabel("Values")
plt.show()
```

- **Histogram**

Used to show the distribution of numerical data

```
import numpy as np
data = np.random.normal(0, 1, 1000)
plt.hist(data, bins=30, color='purple', edgecolor='black')
plt.title("Histogram Example")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

- **Scatter Plot:**

Used to show the relationship between two continuous variables.

```
x = [1, 2, 3, 4, 5]
y = [5, 7, 4, 8, 6]
plt.scatter(x, y, color='red')
plt.title("Scatter Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Seaborn

Seaborn is a powerful data visualization library in Python built on top of Matplotlib. It provides a high-level interface for creating attractive and informative statistical graphics

Feature	Seaborn	Matplotlib
Ease of use	High-level & easy	Low-level & manual
Aesthetics	Beautiful by default	Needs manual styling
Statistical plotting	Built-in (e.g., correlation	Not direct
Integration	Works directly with Pandas DataFrames	Supports arrays/lists

- **Import Seaborn**

```
import seaborn as sns
import matplotlib.pyplot as plt
```

1. Scatter Plot
Shows relationship between two numeric variables.

```
tips = sns.load_dataset("tips")
sns.scatterplot(x='total_bill', y='tip', data=tips, hue='sex')
plt.title("Scatter Plot: Total Bill vs Tip")
plt.show()
```

2**. Histogram (distplot / displot)**
Shows distribution of a numeric variable.

```
sns.histplot(data=tips, x="total_bill", kde=True, color='skyblue')
plt.title("Histogram with KDE")
plt.show()
```

3. Box Plot
Shows distribution with median and outliers.

```
sns.boxplot(x='day', y='total_bill', data=tips, palette='Set2')
plt.title("Box Plot: Total Bill by Day")
plt.show()
```

4. Heatmap
Shows correlation or matrix data using color intensity.

```
iris = sns.load_dataset("iris")
sns.heatmap(iris.corr(numeric_only=True), annot=True,
cmap='coolwarm')
plt.title("Correlation Heatmap of Iris Dataset")
plt.show()
```

Dashboard using Seaborn:

```
df = sns.load_dataset("penguins")

# Scatter
sns.scatterplot(data=df, x="bill_length_mm", y="bill_depth_mm", hue="species")
plt.title("Bill Dimensions by Species")
plt.show()

# Histogram
sns.histplot(data=df, x="flipper_length_mm", kde=True)
plt.title("Flipper Length Distribution")
plt.show()

# Heatmap
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap='YlGnBu')
plt.title("Penguin Feature Correlation")
plt.show()
```

Week 4: Hands-On — Data Visualization with Matplotlib & Seaborn

```
# Importing Required Libraries
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Load built-in datasets
titanic = sns.load_dataset("titanic")
iris = sns.load_dataset("iris")

# ---------------------------
# 1. Matplotlib Visualizations
# ---------------------------

# Line Plot (Dummy example)
x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 30, 25]
plt.plot(x, y, marker='o', color='green')
plt.title("Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()

# Histogram: Age distribution in Titanic dataset
plt.hist(titanic['age'].dropna(), bins=20, color='orange', edgecolor='black')
plt.title("Age Distribution (Titanic)")
plt.xlabel("Age")
plt.ylabel("Frequency")
plt.show()

# Scatter Plot: Fare vs Age
plt.scatter(titanic['age'], titanic['fare'], alpha=0.5, color='blue')
plt.title("Fare vs Age (Titanic)")
plt.xlabel("Age")
plt.ylabel("Fare")
plt.grid(True)
plt.show()
```

🔹 2. Seaborn Visualizations

```
# Seaborn Scatterplot
sns.scatterplot(data=titanic, x="age", y="fare", hue="class")
plt.title("Scatter Plot: Age vs Fare by Class")
plt.show()

# Seaborn Histogram with KDE
sns.histplot(data=titanic, x="age", kde=True, bins=20, color="skyblue")
plt.title("Histogram with KDE: Age")
plt.show()

# Seaborn Boxplot
sns.boxplot(x="class", y="fare", data=titanic)
plt.title("Box Plot: Fare by Passenger Class")
plt.show()

# Seaborn Heatmap: Correlation matrix (Titanic)
titanic_corr = titanic.corr(numeric_only=True)
sns.heatmap(titanic_corr, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap (Titanic)")
plt.show()

# Seaborn Pairplot: Iris Dataset
sns.pairplot(iris, hue="species")
plt.suptitle("Pairplot: Iris Dataset", y=1.02)
plt.show()

# Seaborn Countplot: Passenger class count
sns.countplot(x="class", data=titanic, palette="pastel")
plt.title("Passenger Class Count (Titanic)")
plt.show()
```

Dashboard: Feature Relationships in the Penguins Dataset

Features :

Scatter plots – for relationships
Histograms – for distributions
Boxplots – for comparison across categories
Heatmap – for correlations

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Load the Penguins dataset
df = sns.load_dataset('penguins')

# 🎯 Set the visual theme
sns.set(style="whitegrid")

 1. Scatter Plot: Bill Length vs Depth (relationship)
plt.figure(figsize=(6, 4))
sns.scatterplot(data=df, x='bill_length_mm', y='bill_depth_mm', hue='species')
plt.title("Bill Length vs Bill Depth by Species")
plt.xlabel("Bill Length (mm)")
plt.ylabel("Bill Depth (mm)")
plt.tight_layout()
plt.show()

2. Histogram: Flipper Length Distribution
plt.figure(figsize=(6, 4))
sns.histplot(data=df, x='flipper_length_mm', bins=20, kde=True, color='teal')
plt.title("Distribution of Flipper Length")
plt.xlabel("Flipper Length (mm)")
plt.tight_layout()
plt.show()
```

3. Boxplot: Body Mass by Species

```
plt.figure(figsize=(6, 4))
sns.boxplot(x='species', y='body_mass_g', data=df, palette="pastel")
plt.title("Body Mass Comparison by Species")
plt.xlabel("Species")
plt.ylabel("Body Mass (g)")
plt.tight_layout()
plt.show()
```

4. Heatmap: Correlation Matrix

```
plt.figure(figsize=(6, 4))
corr = df.corr(numeric_only=True)
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.tight_layout()
plt.show()
```

5. Pairplot (optional - shows multiple relationships together)

```
sns.pairplot(df.dropna(), hue='species')
plt.suptitle("Pairwise Relationships in Penguin Features", y=1.02)
plt.show()
```

SUMMARY OF WEEK 04

Objective:

To create a dashboard for visualizing relationships between features in a dataset using scatter plots, histograms, boxplots, and a correlation heatmap. This helps in understanding the structure and distribution of data during Exploratory Data Analysis (EDA).

Dataset Used:

Palmer Penguins Dataset (built-in in Seaborn) — contains measurements such as bill length & depth, flipper length, body mass, and species.

Visualizations Created:

Scatter Plot: Showed the relationship between bill length and bill depth. Each species had a distinct pattern.

Histogram: Displayed the distribution of flipper length. Most penguins had lengths around 200 mm.

Boxplot: Compared body mass across different species. Adelie penguins tended to be lighter.

Heatmap: Showed correlations between numeric features — flipper length and body mass were highly correlated.

Pairplot: Gave an overall snapshot of multiple feature relationships and species groupings.

What I Learned:

How to use Seaborn for attractive, statistical data visualizations

Importance of using EDA techniques before model building

How to detect patterns, trends, and correlations using graphs

How visualizations help in making data-driven decisions

Conclusion:

This dashboard offered a clear visual summary of the dataset, highlighting important relationships and distributions. Such visual tools are essential for data analysts and data scientists to draw meaningful insights and prepare data for machine learning models.