🧠 Module 1: Python Basics — Summary Notes

🔹 1. Data Types

Type	Description	Example
int	Integer (whole number)	`x = 10`
float	Real/decimal number	`y = 3.14`
str	String (text/characters)	`name = "Ali"`
bool	Boolean (`True` / `False`)	`is_valid = True`

Typecasting (Conversion):

- `float(2)` → `2.0`
- `int(1.9)` → `1` (loses decimal part)
- `int("5")` → `5`
- `str(3.5)` → `"3.5"`
- `bool(1)` → `True`, `bool(0)` → `False`

Check type: `type(x)`

🔹 2. Expressions & Operators

Operands → numbers; **Operators** → symbols like `+`, `-`, `*`, `/`

Operator	Function	Example	Output
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`9 - 4`	`5`

`*`	Multiplication	`3 * 4`	`12`
`/`	Division (float)	`7 / 2`	`3.5`
`//`	Floor division (integer)	`7 // 2`	`3`
`%`	Modulus (remainder)	`7 % 2`	`1`

Order of Operations (PEMDAS):
Parentheses → Exponents → Multiplication/Division → Addition/Subtraction

Example:

```
result = (30 + 2) * 60  # Output: 1920
```

🔹 3. Variables

- Used to **store values**.

Assignment uses `=`

```
 x = 5
y = x + 3
```

-

Variables can be updated:

```
 x = 10
```

-
- Use **meaningful names**:
  `total_min`, `total_hour`

Example:

```
total_min = 142
total_hour = total_min / 60  # 2.366...
```

---

4. Strings

- A **sequence of characters** inside quotes `' '` or `" "`
- Example: `name = "Michael Jackson"`

➢ Indexing

- Index starts at **0**
  `name[0] → 'M'`
  `name[-1] → last character`

➢ Slicing

```
name[0:4]   # 'Mich'
name[::2]   # every 2nd char
```

➢ Common Operations

Function	Description	Example	Output
`len(s)`	Length of string	`len("Hello")`	`5`
`+`	Concatenation	`"Hi " + "Ali"`	`"Hi Ali"`
`*`	Repetition	`"Hi" * 3`	`"HiHiHi"`

---

5. Escape Sequences

Code	Meaning	Example Output
`\n`	New line	`Hello\nWorld` →

Hello

World

`\t`	Tab space	`Hello\tWorld` → `Hello World`
`\\`	Backslash	`print("\\")` → `\`
`r" "`	Raw string	`r"\n"` → prints `\n`

🔹 6. String Methods

Method	Description	Example	Output
`.upper()`	Converts to uppercase	`"hello".upper()`	`"HELLO"`
`.lower()`	Converts to lowercase	`"HELLO".lower()`	`"hello"`
`.replace(a,b)`	Replaces substring	`"Hi John".replace("John","Ali")`	`"Hi Ali"`
`.find(sub)`	Finds index of substring	`"Jack".find("a")`	`1`
`.split(delim)`	Splits string by delimiter	`"a,b,c".split(",")`	`['a','b','c']`
`.strip()`	Removes whitespace	`" Hi ".strip()`	`"Hi"`

🔹 7. Comments & Print

Comments: ignored by Python

```
# This is a comment
```

-
- **Print:** displays output

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

🧩 Module 2: Python Data Structures

🔷 1. Tuples

- **Definition:** Ordered sequence enclosed in `()`
  Example: `ratings = (10, 9, 6, 5, 10)`
- **Can contain:** int, float, str, or other tuples (nested)
- **Indexing:**
  - Starts at 0 → `ratings[0]` = `10`
  - Negative index → `ratings[-1]` = last element
- **Slicing:**
  - `ratings[0:3]` → first three elements
  - `ratings[3:]` → from index 3 to end
- **Concatenation:** `(1, 2) + (3, 4)` → `(1, 2, 3, 4)`
- **Length:** `len(ratings)` → number of elements
- **Immutability:** Cannot modify elements directly.
  - You must create a **new tuple** instead.
- **Sorting:** `sorted(ratings)` → returns a **list**, not tuple.
- **Nested tuples:** Access like `nt[2][1]` for inner elements.

📌 *Key Point:* Tuples are **immutable** → faster & memory efficient.

🔷 2. Lists

- **Definition:** Ordered and **mutable** sequence enclosed in `[]`
  Example: `L = ["Rock", 10, 1.2]`

- **Indexing & Slicing:** same as tuples

  - `L[0]`, `L[-1]`, `L[1:3]`

- **Concatenation:** `L1 + L2`

Mutation (changeable):

```
L[0] = "HardRock"
L.append("Jazz")       # add 1 item
L.extend(["Pop", 2020]) # add multiple
del L[1]               # delete element
```

-

Conversion:

```
"A B C".split()     # ['A','B','C']
"a,b,c".split(',')  # ['a','b','c']
```

-

Aliasing:

```
a = [1,2,3]; b = a
a[0] = 99  # affects both
```

-

Cloning (safe copy):

```
b = a[:]  # or list(a)
```

-
- **Nested lists:** access via double index `L[2][1]`

- **Methods:**
  `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `clear()`, `sort()`,

```
reverse()
```

📌 *Key Point:* Lists are **mutable**, allowing item addition, deletion, and updates.

🔷 3. Dictionaries

Definition: Collection of **key–value pairs**, enclosed in `{}`

```
album = {"Thriller":1982, "Back in Black":1980}
```

-
- **Keys:** must be **unique** and **immutable** (like strings, numbers)
- **Accessing values:** `album["Thriller"] → 1982`

Add / Modify:

```
album["Graduation"] = 2007
album["Thriller"] = 1983  # update
```

-
- **Delete:** `del album["Thriller"]`
- **Check existence:** `"Thriller" in album`
- **View data:**
  - `album.keys()` → all keys
  - `album.values()` → all values
  - `album.items()` → list of (key, value) pairs

Copying:

```
new_album = album.copy()
```

-

Update multiple entries:

```
album.update({"Dangerous": 1991, "Bad": 1987})
```

-
- **Clear:** `album.clear()`

📌 *Key Point:* Dictionaries are **unordered**, **mutable**, and **indexed by keys** instead of numbers.

🔷 4. Sets

Definition: Unordered collection of **unique** elements, enclosed in `{}`

```
A = {"ACDC", "BackInBlack", "Thriller"}
```

-
- **Duplicates removed automatically**

Typecasting from list:

```
set([1,1,2,3])  # {1,2,3}
```

-

Add / Remove:

```
A.add("NSYNC")
A.remove("NSYNC")
A.discard("NSYNC")  # safer (no error if missing)
```

-
- **Check membership:** `"ACDC" in A`

Operations:

```
A & B    # Intersection
A | B    # Union
A - B    # Difference
A ^ B    # Symmetric difference
```

-

Subset / Superset:

```
A.issubset(B)
A.issuperset(B)
```

-
- **Copy:** `new_A = A.copy()`
- **Clear:** `A.clear()`

📌 *Key Point:* Sets are great for **mathematical operations** and **removing duplicates**.

⚡ Quick Summary Table

Structure	Ordered	Mutable	Allows Duplicates	Indexed By	Example
Tuple	✅	❌	✅	Index	`(1, 2, 3)`
List	✅	✅	✅	Index	`[1, 2, 3]`
Dictionary	❌	✅	❌ (keys)	Key	`{"a":1, "b":2}`
Set	❌	✅	❌	N/A	`{"a", "b", "c"}`

Module: 3

🧠 1. Conditions and Branching

Purpose: To make decisions in your code.

- **Comparison Operators:**
    - `==` → Equal to
    - `!=` → Not equal to
    - `>` → Greater than
    - `<` → Less than
    - `>=` → Greater than or equal to
    - `<=` → Less than or equal to

If Statement Example:

```
age = 19
if age >= 18:
    print("You can enter")
print("Move on")
```

-

If–Else Example:

```
age = 17
if age >= 18:
    print("You can enter")
else:
    print("Go to the other concert")
```

-

Elif Example:

```
age = 18
```

```python
if age > 18:
    print("Go to ACDC concert")
elif age == 18:
    print("Go to Pink Floyd concert")
else:
    print("Go to Meatloaf concert")
```

-
- **Logical Operators:**
  - `and` → both conditions must be true
  - `or` → at least one condition must be true
  - `not` → reverses True/False

```python
album_year = 1983
if album_year > 1980 and album_year < 1989:
    print("This album was made in the 80s")
```

-

🔁 2. Loops

Purpose: To repeat actions multiple times.

Range Function:

```python
 range(3)         # Output: 0, 1, 2
range(10, 15)    # Output: 10, 11, 12, 13, 14
```

-

For Loop Example:

```python
 squares = ["red", "yellow", "green"]
for color in squares:
    print(color)
```

-

Using `enumerate`:

```
squares = ["red", "yellow", "green"]
for i, color in enumerate(squares):
    print(i, color)
```

-

While Loop Example:

```
squares = ["orange", "orange", "purple"]
new_squares = []
i = 0
while squares[i] == "orange":
    new_squares.append(squares[i])
    i += 1
print(new_squares)
```

-

⚙️ 3. Functions

Purpose: To reuse code and make it organized.

Defining a Function:

```
def add_one(a):
    return a + 1
print(add_one(5))   # Output: 6
```

-

Built-in Functions:

```
len([1, 2, 3])          # 3
sum([10, 20, 30])       # 60
sorted([3, 1, 2])       # [1, 2, 3]
```

-

Function with Multiple Parameters:

```
 def mult(a, b):
    return a * b
print(mult(2, 3))   # Output: 6
```

-

Function with No Return (prints only):

```
 def say_hi():
    print("Hello")
say_hi()
```

-

Using `pass` for Empty Function:

```
 def no_work():
    pass
```

-
- **Variable Scope:**
    - Variables inside a function → *local scope*
    - Variables outside → *global scope*

```
date = 2017
def thriller():
    date = 1982
    return date
print(thriller())  # 1982 (local)
print(date)        # 2017 (global)
```

-

🧩 4. Objects and Classes

Purpose: To represent real-world things using data and functions (OOP).

- **Everything in Python is an Object:**
  Example: integers, strings, lists, dictionaries, etc.

Using `type()`

```
type(3)          # <class 'int'>
type([1,2,3])    # <class 'list'>
```

-

Example of Methods (functions that belong to objects):

```
ratings = [10, 6, 7]
ratings.sort()   # Sorts the list itself
```

-

Defining a Class:

```
class Circle(object):
    def __init__(self, radius, color):
        self.radius = radius
        self.color = color
```

-

Creating Objects:

```
circle1 = Circle(4, "red")
circle2 = Circle(2, "green")
```

- `print(circle1.color)   # red`

🧩 Module 4: File Handling, NumPy, and Pandas

---

1. File Handling in Python

File handling allows you to **store and retrieve data** from files on your computer.
Python provides built-in functions to open, read, write, and close files.

Opening a File

```
file = open("filename.txt", "mode")
```

Mode	Description
`'r'`	Read (default) – error if file doesn't exist
`'w'`	Write – creates new file or overwrites existing
`'a'`	Append – adds data to the end of the file
`'r+'`	Read and Write
`'w+'`	Write and Read (overwrites existing data)

Reading Files

```
file.read()        # Reads the entire file
file.readline()    # Reads one line
file.readlines()   # Reads all lines into a list
```

Writing to Files

```
file.write("Hello World\n")
file.writelines(["Line1\n", "Line2\n"])
```

Closing a File

```
file.close()
```

Using `with` Statement (Recommended)

Automatically closes the file after use.

```
with open("data.txt", "r") as f:
```

```python
    content = f.read()
```

Example: Copying File Content

```python
with open("input.txt", "r") as f1, open("output.txt", "w") as f2:
    for line in f1:
        f2.write(line)
```

---

2. NumPy (Numerical Python)

NumPy is a library used for **numerical computations**, supporting arrays, matrices, and mathematical functions.

Importing

```python
import numpy as np
```

Creating Arrays

```python
a = np.array([1, 2, 3])          # 1D array
b = np.array([[1, 2], [3, 4]])   # 2D array
```

Array Attributes

Attribute	Description	Example
`a.ndim`	Number of dimensions	`1`
`a.shape`	(rows, columns)	`(2, 3)`
`a.size`	Total elements	`6`
`a.dtype`	Data type	`int32`, `float64`

Array Operations

```python
a + b     # Element-wise addition
a - b     # Element-wise subtraction
a * 2     # Scalar multiplication
a * b     # Element-wise multiplication
a @ b     # Matrix multiplication
```

Universal Functions

```
np.mean(a)
np.max(a)
np.min(a)
np.sqrt(a)
np.sin(a)
```

Array Generation

```
np.arange(0, 10, 2)      # [0, 2, 4, 6, 8]
np.linspace(0, 1, 5)    # [0. , 0.25, 0.5, 0.75, 1.]
```

Broadcasting

Automatic operation between arrays of different shapes (if compatible).
Example:

```
a = np.array([1,2,3])
b = 2
print(a + b)   # [3 4 5]
```

---

3. NumPy 2D Arrays (Matrices)

A 2D array is like a matrix with rows and columns.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
```

Accessing Elements

```
A[0][1]   # 2
A[1, :]   # second row
A[:, 0]   # first column
```

Matrix Operations

```
A + B      # Element-wise addition
A * 2      # Scalar multiplication
```

```
A * B      # Element-wise multiplication
A @ B      # Matrix multiplication
```

---

4. Pandas

Pandas is a powerful library for **data manipulation and analysis**, built on top of NumPy.

Importing

```
import pandas as pd
```

Creating DataFrame

```
data = {"Name": ["Ali", "Sara", "Omar"], "Marks": [85, 90, 78]}
df = pd.DataFrame(data)
```

Reading and Writing Files

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

df = pd.read_excel("data.xlsx")
df.to_excel("output.xlsx", index=False)
```

Viewing Data

```
df.head()       # First 5 rows
df.tail(3)      # Last 3 rows
df.info()       # Summary
df.describe()   # Statistics summary
```

Accessing Data

```
df["Name"]             # Access a column
df.loc[0]              # Access by label/index
df.iloc[1:3]           # Access by position
df[df["Marks"] > 80]   # Filtering
```

Useful Operations

```
df["Marks"].mean()
```

```
df["Marks"].max()
df["Marks"].unique()
```

Saving Filtered Data

```
high = df[df["Marks"] > 80]
high.to_csv("high_scorers.csv", index=False)
```

🧾 Summary Table

Concept	Library	Key Functions
File Handling	Built-in	`open()`, `.read()`, `.write()`, `.close()`
NumPy	`import numpy as np`	`np.array()`, `np.mean()`, `np.dot()`
Pandas	`import pandas as pd`	`pd.read_csv()`, `.loc[]`, `.to_csv()`