🐍 Complete Notes from Python Roadmap (Basics → Advanced)

1: Python Basics (Foundation)

What is Python?

Python is a **high-level programming language** used to tell computers what to do.

It is very easy to read, write, and understand — that's why beginners and experts both love it.

How is the Python?

Python is:

- **Simple** (almost like English)
- **Powerful** (can build very big systems)
- **Flexible** (can be used for many fields)
- **Portable** (runs on Windows, Mac, Linux)
- **Open-source** (free for everyone)

Example:

```
print("Hello, Python!")
```

This prints a message — very simple!

Why do we need Python?

Because Python helps us create:

- Websites
- Mobile / Desktop apps
- Artificial Intelligence & Machine Learning
- Data Science & Data Analytics
- Cyber Security tools
- Automation (saving time on repeated tasks)
- Game development
- Internet of Things (IoT)

In short: Python makes work **faster, easier, and smarter**.

****

1.1 Variables & Data Types

- ∉ **Numbers** → int, float, complex
- ∉ **Text** → str (string)
- ∉ **Boolean** → bool (True, False)
- ∉ **Empty** → NoneType (None)

👉 Example:

x = 10   # int

y = 3.14   # float

z = "Hello" # str

b = True   # bool

n = None   # NoneType

****

1.2 Operators

Operator means: A symbol or word used to perform an action on values. Example:

```
5 + 3  # Here '+' is an operator
```

Operator Precedence in pyt
PEMDAS
Parentheses ()

Exponents **
(2 ** 3 ** 2)
Right to left exponentiatic

Multiplication *, Division /,
Floor Division //,
Modulus %
Evaluated from left to rig

Addition +, Subtraction -
#Evaluated from left to righ

1. Arithmetic Operators

Used for **math calculations**.

Operator	Meaning	Example	Result
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division (float result)	5 / 2	2.5
//	Floor division (no decimals)	5 // 2	2
%	Modulus (remainder)	5 % 2	1
**	Exponent (power)	5 ** 2	25

==========

2. Comparison / Relational Operators

Used to **compare** values → return **True/False**.

Operator	Example	Meaning
==	a == b	Equal
!=	a != b	Not equal
>	a > b	Greater
<	a < b	Less
>=	a >= b	Greater or equal
<=	a <= b	Less or equal

==========

3. Logical Operators

Used in conditions to combine comparisons.

Operator	Meaning	Example	Output
and	both True	(5 > 2 and 5 < 10)	True
or	any one True	(5 > 10 or 5 > 2)	True
not	reverse result	not(5 > 2)	False

Memory trick: **and = strict**, **or = flexible**, **not = opposite**

==========

4. Assignment Operators

Used to **assign values** to variables, sometimes with math.

Operator	Example	Same as
=	a = 5	a = 5
+=	a += 3	a = a + 3
-=	a -= 3	a = a - 3
*=	a *= 3	a = a * 3
/=	a /= 3	a = a / 3
//=	a //= 3	a = a // 3
%=	a %= 3	a = a % 3
**=	a **= 3	a = a ** 3

==========

5. Bitwise Operators

Used in **binary operations** (0s and 1s).

Operator	Meaning
&	AND
`	`
^	XOR
~	NOT (flip bits)
<<	Left shift
>>	Right shift

Every number inside a computer is stored in binary form.

Example:

10 → 1010 (in binary)
4 → 0100 (in binary)

Bitwise operators compare **each bit** of the number.

& (Bitwise AND)

Both bits must be **1** → then result is 1, else 0

Example:

a = 10 → 1010
b = 4 → 0100

---

a & b → 0000 = 0

| (Bitwise OR)

If **any** bit is 1 → result is 1

Example:

1010
0100

1110 = 14

---

^ (Bitwise XOR)

XOR = **Exclusive OR**

Result is 1 **only when bits are different**

Example:

1010
0100

1110 = 14

(XOR gives same result as OR in this case)

~ (Bitwise NOT)

Reverses all bits (1 → 0, 0 → 1)

Also changes the sign because Python uses **Two's Complement** for negative numbers.

Example:

a = 10
~a = -11

(You will understand Two's complement later)

<< (Left Shift)

Moves bits towards **left** → adds zeros on right

Same as multiplying by 2 for every shift

Example:

10 = 1010
10 << 1 → 10100 = 20
10 << 2 → 101000 = 40

So:

a << 1 = a * 2

a << 2 = a * 4

>> (Right Shift)

Moves bits towards **right** → slices last bit

Same as dividing by 2 for every shift

Example:

10 = 1010
10 >> 1 → 0101 = 5
10 >> 2 → 0010 = 2

So:

a >> 1 = a / 2

a >> 2 = a / 4

🎯 Easy Memory Trick

Operator	Think as…
&	both yes
`	`
^	only one yes
~	flip values
<<	increase (×2)
>>	decrease (÷2)

==========

6. Membership Operators

Used to check if an item exists inside a sequence (string, list, etc.)

Operator	Example	Result
in	'a' in 'apple'	True
not in	5 not in [1,2,3]	True

==========

7. Identity Operators

Check if two variables **refer to the same memory**.

Operator	Example	Meaning
is	a is b	Same object
is not	a is not b	Different object

Example:

```
a = [1,2]
b = [1,2]
print(a == b)  # True (same values)
print(a is b)  # False (different memory)
```

****

1.3 Input/Output

```
name = input("Enter your name: ")
print("Hello,", name)
```

****

1.4 Python Comments

Comments are **notes inside code** that Python ignores while running the program.

They are only for:

✓ Understanding the code ✓ Writing reminders

✓ Explaining logic ✓ Making code readable

1.Single-Line Comment

Use # at the beginning of a line.

Example:

```
# This is a single-line comment
print("Hello")
```

Even if it is written at the end of a statement:

```
print("Hi")  # This prints Hi
```

==========

2.Multi-Line Comment

Use **triple quotes** (''' ... ''' or """ ... """)

Example:

```
'''
This is a
multi-line comment
'''
print("Python")
```

Or:

```
"""
These lines
```

are ignored by Python
"""

Why do we need comments?

Because they help:

- Explain complex code
- Make code easier for others to read
- Remind ourselves what code does

Memory Trick: **Comments talk to humans, not computers.**

Type	Symbol	Example
Single line	#	# Hello
Multi-line	''' ... '''	Block of comment

****

1.5 Typecasting

int("123") # str → int

str(123) # int → str

float("3.14") # str → float

Only **numeric strings** can convert to int/float. "abc"

****

2: Control Structures

1. Conditionals Statements

Conditional statements help a program **make decisions** based on conditions.

Think of it like real life:

If it rains → take an umbrella
Else if it's sunny → wear sunglasses
Else → stay norma

```
x = 10
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

👉 Mutually Exclusive → only **one block** runs.
👉 Multiple if → all conditions checked separately.

==========

2. Loops

While Loop

```
i = 0
while i < 3:
    print("hi")
    i += 1
```

For Loop

```
for i in range(3):
    print("hi)
```

Loop Control

- break → exit loop
- continue → skip current iteration
- else with loops → runs if loop finished without break

---

3. Nested Loops

Loop inside another loop → useful for tables, grids, etc.

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

---

4. Special Functions

∉ **enumerate(iterable)** → gives (index, value)
∉ **dict.items()** → gives (key, value)

Mini Examples

- **Prime Checker**

```
n = 7
for i in range(2, int(n**0.5)+1):
    if n % i == 0:
        print("Not Prime")
        break
else:
    print("Prime")
```

- **Guessing Game**

```
import random
r = random.randint(1, 20)
while True:
    n = int(input("Guess: "))
    if n == r:
        print("Win!")
        break
    else:
        print("Try again")
```

****

3: Data Structures

3.1 Python Strings: Quick Summary

Strings in Python are sequences of characters, written in quotes ('...' or "...").
They are **immutable** (can't be changed directly; any change creates a new string).

◇ 3.1.1 String Basics

- **ımmutable**: Once created, cannot be modified directly.
- **Indexing**: s[0], s[-1].

Slicing: s[start:end:step].
Example:

```
s = "hanifa"
s[1:5]     # "anif"
```

```
s[::-1]    # reverse → "afinah"
```

◇ 3.1.2 Case Conversion

Method	What it does
upper()	All uppercase
lower()	All lowercase
capitalize()	First letter uppercase, rest lowercase
title()	Capitalizes each word
swapcase()	Swaps case
casefold()	Aggressive lowercase (better for comparisons)

◇ 3.1.3 Case Checking

Method	What it checks
isupper()	All uppercase?
islower()	All lowercase?
istitle()	Title case?

◇ 3.1.4 Whitespace Removal

Method	What it does
strip()	Remove spaces (both sides)
lstrip()	Remove spaces from left
rstrip()	Remove spaces from right

◇ 3.1.5 Search Methods

Method	Use
find(sub)	First index (left), -1 if not found
rfind(sub)	Last index (right), -1 if not found
index(sub)	Like find, but raises error if not found
rindex(sub)	Like rfind, but raises error
count(sub)	Number of occurrences
startswith(sub)	True if string starts with sub
endswith(sub)	True if string ends with sub

◇ 3.1.6 Replace & Split

Method	Use
replace(old, new, count)	Replace substring
split(sep, maxsplit)	Split into list (default by spaces)
rsplit(sep, maxsplit)	Split from right
splitlines(keepends)	Split at line breaks

◇ 3.1.7 Join

Method	Use
"sep".join(iterable)	Join elements into one string with separator

Example:

"-".join(["a","b","c"]) # "a-b-c"

◇ 3.1.8 Checking Content (Character Type)

Method	What it checks
isalpha()	Only letters
isdigit()	Only digits
isalnum()	Letters or digits
isdecimal()	Only decimal digits
isnumeric()	Numeric (incl. fractions, Roman numerals)
isspace()	Only whitespace
isascii()	ASCII (0–127)

◇ 3.1.9 Other Checkers

Method	What it checks
isidentifier()	Valid Python variable name?
isprintable()	Printable characters?

◇ 3.1.

f-strings:

name = "Hanifa"; age = 20

print(f"My name is {name}, I am {age}")

.format():

"My name is {} and I am {}".format("Hanifa", 20)

Old style %:

"My name is %s and I am %d" % ("Hanifa", 20)

◇ 3.1.11 Encoding / Decoding

- encode("utf-8") → string → bytes.
- decode("utf-8") → bytes → string.
- Mostly used in files, networking, databases.

◇ 3.1.12 Miscellaneous

Method	Use
expandtabs(n)	Replace \t with spaces
partition(sep)	Split into 3 parts (first sep)
rpartition(sep)	Split into 3 parts (last sep)
zfill(width)	Pad with zeros on left
center(width, fill)	Center text
ljust(width, fill)	Left align
rjust(width, fill)	Right align
maketrans() + translate()	Fast multiple-char replace/delete
removeprefix(x)	Remove prefix if present
removesuffix(x)	Remove suffix if present

OBJ

- Strings are **immutable** → modifications always make a new string.
- Python gives **rich built-in methods** for searching, checking, formatting, replacing, splitting, and aligning text.

- Use the **right method for the right job**:
    - Checking → is*() Search → find/index/count Format → f-strings
    - Cleanup → strip, replace, translate Join/split → join, split Align → center/ljust/rjust/zfill

****

3.2 Python Lists: Quick Summary

◇ 3.2.1 What is a List?

- A **list** is a collection of items in Python, ordered and **mutable** (can be changed).
- Defined with square brackets [].
- fruits = ["apple", "banana", "cherry"]

☑ Stores **different data types** (numbers, strings, even other lists).

◇ 3.2.2 Creating Lists

```
empty = []                  # Empty list
numbers = [1, 2, 3, 4]          # List of integers
mixed = [1, "hello", 3.5, True]  # Mixed data
nested = [[1, 2], [3, 4]]       # Nested list
```

◇ 3.2.3 Indexing & Slicing

- **Indexing** → access items by position.
- **Slicing** → get parts of a list.

```
a = [10, 20, 30, 40, 50]
print(a[0])     # 10 (first item)
print(a[-1])    # 50 (last item)
print(a[1:4])   # [20, 30, 40] (slice)
```

◇ 3.2.4 Important List Methods

1. Adding Items

Method	What it does	Example	Output

append(x)	Add one item to the end	lst=[1,2] lst.append(3)	[1, 2, 3]
insert(i, x)	Add item at a specific index	lst=[1,2] lst.insert(1, 99)	[1, 99, 2]
extend(iterable)	Add multiple items from another iterable	lst=[1,2]; lst.extend([3,4])	[1, 2, 3, 4]

==========

2. Removing Items

Method	What it does	Example	Output
remove(x)	Removes the **first occurrence** of x	lst=[1,2,1] lst.remove(1)	[2, 1]
pop([i])	Removes item at index i (default = last)	lst=[1,2,3] lst.pop(0)	[2, 3]
clear()	Removes all items	lst=[1,2,3] lst.clear()	[]

==========

3. Searching & Counting

Method	What it does	Example	Output
index(x)	Find index of **first occurrence** of x	[10,20,30].index(20)	1
count(x)	Count how many times x appears	[1,1,2].count(1)	2

==========

4

Method	What it does	Example	Output
sort()	Sort ascending (modifies list)	lst=[3,1,2] lst.sort()	[1, 2, 3]
sort(reverse=True)	Sort descending	lst=[3,1,2] lst.sort(reverse=True)	[3, 2, 1]
reverse()	Reverse order (not sorting)	lst=[1,2,3] lst.reverse()	[3, 2, 1]

==========

5. Copying

Method	What it does	Example	Output
copy()	Makes a **shallow copy**	a=[1,2] b=a.copy() b[0]=99	a → [1,2], b → [99,2]

◇ 3.2.5 Built-in Functions with Lists

Function	Use
len(lst)	Number of items
sum(lst)	Sum of numbers
min(lst) / max(lst)	Smallest / largest
sorted(lst)	Returns new sorted list
any(lst)	True if any element is True
all(lst)	True if all elements are True

◇ 3.2.6 List Comprehension

👉 Short way to create lists.

```
squares = [x**2 for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]
```

With condition:

```
evens = [x for x in range(10) if x % 2 == 0]
```

◇ 3.2.7 Iterating Lists

```
# Using for loop
for item in fruits:
    print(item)


# Using while loop
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1


# Using enumerate
for i, item in enumerate(fruits):
    print(i, item)
```

◇ 3.2.8 Copying Lists

- **Assignment** → both point to same list.
- **Shallow copy (copy())** → new list, but nested lists still linked.
- **Deep copy (deepcopy)** → full independent copy (used in complex/nested data).

In Python, **copying a list** is tricky because lists are **mutable** (changeable). How you copy affects whether changes in one list affect the other.

1. Assignment (=)

Both variables point to the **same list in memory**.

```
a = [1, 2, 3]

b = a

b[0] = 99

print(a)  # [99, 2, 3] → a also changed!
```

⚠ Not a real copy, just another name for the same list.

==========

2. Shallow Copy (copy() or list())

Makes a **new list**, but nested lists (if any) are still linked.

```
a = [1, 2, [3, 4]]

b = a.copy()   # or b = list(a)

b[0] = 99

b[2][0] = 100

print(a)  # [1, 2, [100, 4]] → nested list changed!
```

☑ Safe if list has only simple values (numbers, strings).
⚠ Nested lists still share references.

==========

3. Deep Copy (copy.deepcopy())

Creates a **completely independent copy**, even for nested lists.

```
import copy

a = [1, 2, [3, 4]]

b = copy.deepcopy(a)
```

```
b[2][0] = 100
print(a)  # [1, 2, [3, 4]] → original safe!
print(b)  # [1, 2, [100, 4]]
```

☑ Best when working with **complex/nested data structures**.

When to Use Which?

- **Assignment (=)** → when you don't need an independent copy (just another reference).
- **Shallow Copy** → for simple lists without nesting.
- **Deep Copy** → for nested/complex lists to avoid accidental changes.

◇ 3.2.9 Nested Lists (2D lists)

```
matrix = [[1, 2], [3, 4], [5, 6]]
print(matrix[1][0])   # 3
```

☑ Useful for **tables, grids, Excel-like data**.

◇ 3.2.10 Membership & Comparison

```
# Membership
print("apple" in fruits)     # True
print("mango" not in fruits) # True

# Comparison
print([1,2,3] == [1,2,3])  # True
print([1,2] < [1,3])       # True (compares element by element)
```

🎯 Why Lists?

- Store multiple values in **one variable**.
- Easy to **modify, search, and sort**.
- Used everywhere: **shopping carts, student records, datasets, game boards, to-do lists, machine learning data**.

☑ **Conclusion**: Lists are the most flexible and commonly used Python data structure. They are powerful because they're ordered, mutable, and support tons of methods for real-world use cases.

****

3.3 Python Tuples: Quick Summary

◇ 3.3.1 What is a Tuple?

A tuple is an ordered, immutable collection in Python used to store multiple items in a single variable.
You create it using parentheses () instead of square brackets [].

my_tuple = (1, "apple", 3.5)

☑ Ordered – Items keep their position.

🚫 Immutable – You can't add, remove, or modify elements once created.

💡 Can store different data types.

◇ 3.3.2 Creating Tuples

Example	Description
t1 = (1, 2, 3)	Normal tuple
t2 = ("apple",)	Single-element tuple (must have a comma)
t3 = tuple([1, 2, 3])	Using the tuple() function
nested = ((1, 2), (3, 4))	Nested tuple

◇ 3.3.3 Accessing Tuple Elements

Tuples support indexing and slicing just like lists.

```
t = (10, 20, 30, 40, 50)
print(t[0])     # 10 (first element)
print(t[-1])    # 50 (last element)
print(t[1:4])   # (20, 30, 40)
```

◇ 3.3.4 Tuple Methods (Only 2)

Method	Description	Example	Output
count(x)	Counts occurrences of x	(1,2,2,3).count(2)	2
index(x)	Returns index of first x	(10,20,30).index(20)	1

◇ 3.3.5 Tuple Packing and Unpacking

You can pack multiple values into a tuple or unpack them into separate variables.

```
# Packing
student = ("Hanifa", "Python", 2025)
# Unpacking
name, course, year = student
print(name)   # Hanifa
```

◇ 3.3.6 Nested Tuples

Tuples can contain other tuples — like a 2D list, often used for structured data.

```
matrix = ((1, 2, 3), (4, 5, 6))
print(matrix[1][2])  # 6
```

◇ 3.3.7 Tuples vs Lists

Feature	**Tuple**	**List**
Syntax	( )	[ ]
Mutability	Immutable	Mutable
Speed	Faster	Slower
Memory	Less	More
Use Case	Fixed data	Changing data

◇ 3.3.8 Conversions

You can easily switch between tuple, list, set, and dictionary using built-ins.

```
t = (1, 2, 3)
lst = list(t)
st = set(t)
tpl = tuple(lst)
```

◇ 3.3.9 Tuples in Real World

Use Case	Example
Coordinates	location = (33.7, 72.8)
Database Rows	Each record stored as a tuple
Function Return Values	return (name, marks, grade)
Dictionary Keys	dict[((33,72))] = "Taxila"

◇ 3.3.10. Functions that Work with Tuples

Function	Description	Example	Output
len(t)	Length of tuple	len((1,2,3))	3
min(t)	Minimum value	min((2,8,1))	1
max(t)	Maximum value	max((2,8,1))	8
sum(t)	Sum (numeric only)	sum((1,2,3))	6
sorted(t)	Returns sorted list	sorted((3,1,2))	[1,2,3]

◇ 3.3.11. Why Use Tuples?

☑ **When data should not change** (like constants, coordinates).
☑ **Faster performance** and **less memory** than lists.
☑ **Can be used as dictionary keys** (unlike lists).
☑ **Great for returning multiple values** from functions.

⌨ Final Summary

Feature	Description
Type	Ordered, immutable sequence

Syntax	( ) or tuple()
Elements	Can include any type (even lists/dicts)
Methods	count(), index()
Conversions	List ↔ Tuple ↔ Set
Real Use	Fixed data, coordinates, database records, function returns

---

🌀 Python Tuple Conversions – Complete & Simple Guide

◇ 1. Convert Tuple → List

Why:

- To **modify** data (add, remove, change) — because tuples are immutable.
- Lists are editable, so we convert tuples to lists when updates are needed.

Example:

```
t = (1, 2, 3, 4)
lst = list(t)
print(lst)
```

Output:

```
[1, 2, 3, 4]
```

Now you can modify it:

```
lst.append(5)
print(lst)
```

◇ 2. Convert List → Tuple

Why:

- When you want to **protect data** from being changed accidentally.
- Used when data is fixed — e.g., coordinates, settings, or constants.

Example:

```
lst = ['apple', 'banana', 'cherry']

t = tuple(lst)

print(t)
```

Output:

```
('apple', 'banana', 'cherry')
```

◇ 3. Convert Tuple → Set

Why:

- To **remove duplicate elements**.
- When you need to perform **set operations** (like union, intersection, etc.).

Example:

```
t = (1, 2, 2, 3, 3, 4)

s = set(t)

print(s)
```

Output:

```
{1, 2, 3, 4}
```

⚠ **Note:** The order is not preserved, because sets are *unordered*.

◇ 4. Convert Set → Tuple

Why:

- To make a set **ordered** or **immutable**.
- Useful when you want to store a set as a key in a dictionary (since sets themselves can't be keys).

Example:

```
s = {'x', 'y', 'z'}

t = tuple(s)

print(t)
```

Output:

```
('x', 'y', 'z')
```

◇ 5. Convert Tuple → Dictionary

Why:

- When you have *paired data* (key, value) stored as tuples and want to create a dictionary.

Example:

```
t = (('name', 'Hanifa'), ('age', 20), ('course', 'Python'))

d = dict(t)

print(d)
```

Output:

```
{'name': 'Hanifa', 'age': 20, 'course': 'Python'}
```

⚠ **Each inner tuple must have exactly two elements** (key, value)

◇ 6. Convert Dictionary → Tuple

Why:

- When you need an *immutable copy* of a dictionary's data.
- Useful when passing data safely between functions

Example:

```
d = {'name': 'Hanifa', 'age': 20}

t = tuple(d.items())

print(t)
```

Output:

```
(('name', 'Hanifa'), ('age', 20))
```

◇ 7. Convert Tuple → String

Why:

- To join tuple elements into a readable or storable text format.

Example:

```
t = ('Python', 'is', 'fun')
```

```
s = " ".join(t)

print(s)
```

Output:

```
Python is fun
```

◇ 8. Convert String → Tuple

Why:

- To separate each character as an element in a tuple.

Example:

```
s = "AI"

t = tuple(s)
```

◇ 9. Convert Tuple → JSON (for storage or APIs)

Why:

- When sending data through APIs or storing structured data.
- JSON supports lists, not tuples — so tuples become lists automatically.

Example:

```
import json

t = ('Hanifa', 20, 'Python')

json_data = json.dumps(t)

print(json_data)
```

Output:

```
["Hanifa", 20, "Python"]
```

Conversion Summary Table

From → To	Code	Purpose	From → To	Code	Purpose

Tuple → List	list(t)	To modify data	Tuple → Dict	dict(t)	To create key-value pairs
List → Tuple	tuple(lst)	To make data fixed	Dict → Tuple	tuple(d.items())	To freeze dictionary data
Tuple → Set	set(t)	To remove duplicates	Tuple → String	" ".join(t)	To store or display as text
Set → Tuple	tuple(s)	To make data immutable	String → Tuple	tuple(s)	To get each char separately

🎯 Real-World Example

Imagine an **online student management system**:

Tuple for fixed student info

student = ("Nouman", "BS Software Engineering", "Taxila")

Convert to list to update values

student_list = list(student)

student_list[2] = "Lahore"

Convert back to tuple for final record

student = tuple(student_list)

print(student)

Output:

('Hanifa', 'BS Software Engineering', 'Lahore')

🌀 Conversions: List ↔ Dictionary ↔ Set

◇ 1. List → Set

Why:

- To **remove duplicate elements**.
- To perform **set operations** (like union, intersection, etc.).

Example:

fruits = ['apple', 'banana', 'apple', 'cherry']

fruit_set = set(fruits)

print(fruit_set)

Output:

{'apple', 'banana', 'cherry'}

☑ **Use case:** When you only care about *unique* items.

OBJ

Why:

- To **access elements by index**.
- To sort or modify the items.

Example:

fruit_set = {'apple', 'banana', 'cherry'}

fruit_list = list(fruit_set)

print(fruit_list)

Output:

['banana', 'apple', 'cherry']

⚠ **Note:** Order is not guaranteed since sets are unordered.

◇ 3. List → Dictionary

Why:

- To pair **related data** (keys and values) from two lists.
- To create key-value mappings quickly.

Example From two lists

keys = ['name', 'age', 'city']

values = ['Hanifa', 20, 'Taxila']

info = dict(zip(keys, values))

print(info)

Output:

{'name': 'Hanifa', 'age': 20, 'city': 'Taxila'}

☑ **zip()** pairs each element from keys and values together.

Example 2: From a list of pairs (nested list)

pairs = [['name', 'Hanifa'], ['age', 20], ['city', 'Taxila']]

info = dict(pairs)

print(info)

Output:

{'name': 'Hanifa', 'age': 20, 'city': 'Taxila'}

⚠ Each inner list must have **exactly two elements** → key and value.

◇ 4. Dictionary → List

Why:

- To **extract** only keys, values, or both from a dictionary.

Example:

info = {'name': 'Hanifa', 'age': 20, 'city': 'Taxila'}

Keys only

keys_list = list(info.keys())

print(keys_list)

Values only

values_list = list(info.values())

print(values_list)

Items (as key-value pairs)

items_list = list(info.items())

print(items_list)

Output:

['name', 'age', 'city']

['Hanifa', 20, 'Taxila']

[('name', 'Hanifa'), ('age', 20), ('city', 'Taxila')]

◇ 5. Dictionary → Set

Why:

- To get **unique keys** (by default).
- Useful in comparing key names between dictionaries.

Example:

info = {'name': 'Hanifa', 'age': 20, 'city': 'Taxila'}

key_set = set(info)

print(key_set)

Output:

{'city', 'age', 'name'}

☑ Only keys are converted to a set by default.
If you want both keys and values:

key_value_set = set(info.items())

print(key_value_set)

Output:

{('city', 'Taxila'), ('age', 20), ('name', 'Hanifa')}

◇ 6. Set → Dictionary

⚠ **Not direct!**

- You can only convert if the set contains *pairs* (tuples/lists with 2 items each).

Example:

data = {('name', 'Hanifa'), ('age', 20), ('city', 'Taxila')}

info = dict(data)

print(info)

Output:

{'name': 'Hanifa', 'age': 20, 'city': 'Taxila'}

💮 Conversion Summary Table

From → To	Code	Notes / Purpose	From → To	Code	Notes / Purpose

List → Set	set(lst)	Removes duplicates	Dict → List	list(d.keys()) / list(d.items())	Extract data from dictionary
Set → List	list(s)	Makes it indexable or sortable	Dict → Set	set(d)	Converts only keys
List → Dict	dict(zip(keys, values))	Combine two lists into key-value pairs	Set → Dict	dict(s)	Works only if set contains pairs

🎯 Real-World Analogy

Example: Online shopping system

```
# Lists of customers and their purchases

customers = ['Hanifa', 'Ali', 'Sara']

purchases = [3, 5, 2]

# Convert to dictionary for record

shopping_data = dict(zip(customers, purchases))

print(shopping_data)

# {'Hanifa': 3, 'Ali': 5, 'Sara': 2}

# Convert to set to get unique customers

unique_customers = set(customers)

print(unique_customers)
```

Relating each Python **data structure** to **how ChatGPT (or real apps like YouTube, Chrome, and Google)** work is one of the *best ways* to deeply understand both programming and real systems.

🧱 Python Data Structures in Real Life (and in ChatGPT!)

Data Structure	In Python (Definition)	In ChatGPT / Real Apps (How it works)	Why It's Useful
1. String (str)	Stores text data (immutable).	When you type a message, your text is handled as a **string** — like "Hello ChatGPT!". I read and process it as text.	Helps apps process and display words, names, commands, etc.
2. List ([])	Ordered, mutable collection.	I store your **conversation history** as a **list** of messages: ["Hi", "Hello!", "Explain loops"].	Apps like WhatsApp or Gmail use lists to store messages, emails, or search results in order.
3. Tuple (())	Ordered, immutable collection.	My **configuration settings** (like model type, version) are stored as **tuples**, so they don't accidentally change: ("GPT-5", "Chat Model").	Used in apps for **fixed data**, like GPS coordinates, product info, etc.

4. Set ({})	Unordered, unique collection.	I use sets to keep track of **unique topics** we discussed: {"AI", "Python", "Tuples"}.	In apps, sets help remove duplicates (e.g., unique search results in Google).
5. Frozen Set (frozenset())	Immutable set (cannot be changed).	Like I explained earlier — my **rules & permissions** are frozen. Example: frozenset({"read_input", "respond"}).	Used for **security and stability**, like fixed permissions in browsers or AI models.
6. Dictionary ({key: value})	Stores data as key-value pairs.	I store info like: {"name": "Hanifa", "topic": "Python"} — that helps me respond personally!	Apps use it for **user profiles**, **settings**, **JSON APIs**, and more.
7. List of Dictionaries	Combination of multiple user data entries.	ChatGPT's system stores multiple users as a list of dictionaries: [{"user": "Hanifa"}, {"user": "Ali"}].	Social media and databases use it for **user records** or **product catalogs**.
8. Dictionary of Dictionaries	Nested data, hierarchical.	My internal memory could look like: {"user": {"name": "Hanifa", "level": "beginner"}}.	Websites use it for **complex data**, e.g., Google stores your profile + preferences together.

🌍 Summary: How Apps Reflect Data Structures

App / Platform	Uses of Data Structures
YouTube	list → watch history, dict → video data, set → unique tags, tuple → video resolution.
Chrome Browser	dict → user preferences, set → blocked sites, frozenset → fixed security rules.
ChatGPT (Me!)	list → conversation history, dict → user context, frozenset → safety rules.
Google Search	set → remove duplicate results, list → search ranking order, dict → metadata.
E-commerce (Amazon)	list of dicts → product listings, set → unique categories, tuple → fixed product IDs.

****

3.4 Python Sets: Quick Summary

◇ What is a Set?

A **set** is an **unordered**, **mutable**, and **unindexed** collection of **unique elements**.
It automatically removes duplicates and allows you to perform powerful **mathematical operations** like union, intersection, and difference.

```
my_set = {1, 2, 3, 3, 4}

print(my_set)   # {1, 2, 3, 4}
```

◇ 3.4.1 Set Properties

Property	Meaning	Example
Unordered	No fixed position for elements	{3, 1, 2} → {1, 2, 3}
Unique	Duplicates are removed automatically	{1,1,2} → {1,2}
Mutable	You can add or remove elements	set.add() or set.remove()
Unindexed	Cannot access by index	set[0] (error)

◇ 3.4.2 Creating Sets

```
# Using curly braces
fruits = {"apple", "banana", "cherry"}
# Using set() constructor
nums = set([1, 2, 3])
# Empty set (must use set())
empty = set()    # not {}
```

◇ 3.4.3 Allowed and Disallowed Data Types

☑ Allowed: numbers, strings, tuples (immutable)
❌ Not allowed: lists, dictionaries, other sets (mutable)

```
valid_set = {1, 3.5, "hello", (1, 2)}
invalid_set = {1, [2, 3]}  # ❌ Error
```

◇ 3.4.4 Adding & Removing Elements

Method	Action	Example
add(x)	Adds one element	s.add(5)

update(iterable)	Adds multiple elements	s.update([6, 7])
remove(x)	Removes element, raises error if not found	s.remove(5)
discard(x)	Removes element, no error if not found	s.discard(5)
pop()	Removes a random element	s.pop()
clear()	Removes all elements	s.clear()

◇ 3.4.5 Set Operations (Mathematical)

Sets allow operations like in math — **union, intersection, difference**, and more:

Operation	Symbol	Description	Example
Union	`A	B`	Combines both sets (no duplicates)
Intersection	A & B	Common elements	{1,2} & {2,3} → {2}
Difference	A - B	Elements in A but not in B	{1,2,3} - {2} → {1,3}
Symmetric Difference	A ^ B	Elements not shared	{1,2,3} ^ {2,3,4} → {1,4}

◇ 3.4.6 Set Relationships

Method / Operator	Meaning	Example
A.issubset(B) / A <= B	All elements of A in B	{1,2}.issubset({1,2,3}) → True
A.issuperset(B) / A >= B	A contains all elements of B	{1,2,3}.issuperset({2}) → True
A.isdisjoint(B)	No common elements	{1,2}.isdisjoint({3,4}) → True

---

◇ 3.4.7 Set Comprehension

Used to build sets in one line (like list comprehension).
Duplicates are removed automatically.

```
squares = {x**2 for x in range(5)}

print(squares)  # {0, 1, 4, 9, 16}
```

◇ 3.4.8 Frozen Sets (Immutable Sets)

A **frozenset** is like a set, but you **cannot modify it** after creation.

```
f = frozenset([1, 2, 3])

# f.add(4) ❌ Error

print(f)  # frozenset({1, 2, 3})
```

Used for:

- ∉ Creating unchangeable constant sets
- ∉ Using as dictionary keys or inside other sets
- ∉ Fixed configuration values or roles

◇ 3.4.9 Real-World Examples

Use Case	Description
Duplicate removal	unique_words = set(word_list)
Membership test	Fast check if item exists (x in set)
User permissions	Compare roles with issubset / issuperset
Recommendation systems	Find overlap between interests (&, isdisjoint)
Frozen sets	Store unchangeable categories or constants

◇ 3.4.10 Important Notes

- ∉ Sets are **fast** for membership testing (O(1) on average).

∉ Cannot access by index — use for loops to iterate.

Converts easily between data structureslist(set(...)) # remove duplicates from list

set(tuple(...)) # convert tuple to set

✿ Quick Summary Table

Feature	Description	Example
Type	Unordered, Mutable, Unique	{"a", "b"}
Indexing	❌ Not supported	set[0] → Error
Duplicates	Automatically removed	{1,1,2} → {1,2}
Mutability	Can add/remove elements	add(), remove()
Conversion	list(), tuple(), dict() supported	list(myset)
Frozen Set	Immutable version	frozenset()
Operations	`	, &, -, ^`
Relationships	issubset(), issuperset(), isdisjoint()	Logical comparisons

****

3.5 Python Dictionaries: Quick🧾

What is a Dictionary?

A **dictionary** in Python is an *unordered, mutable* collection that stores data in **key-value pairs**.

- Keys are **unique** and **immutable** (strings, numbers, or tuples).
- Values can be **any data type** (lists, sets, or even other dictionaries).
- Dictionaries are written inside **curly braces** {}.

Example:

```
student = {
    "name": "Hanifa",
    "age": 20,
```

```
    "courses": ["Python", "AI"]
}
```

◇ Key Features of Dictionary

Feature	Description	Example
Unordered	Items have no fixed position (before Python 3.7).	{}
Mutable	You can add, change, or remove items.	student["age"] = 21
Key-Value Pairs	Each key maps to a value.	"name": "Hanifa"
Unique Keys	Duplicate keys overwrite earlier ones.	{"a":1, "a":2} → {"a":2}

◇ 3.5.1 Creating a Dictionary

```
# Empty dictionary
data = {}
# With values
student = {"name": "Ali", "age": 22}
# Using dict() constructor
person = dict(name="Sara", age=25)
```

◇ 3.5.2 Accessing & Modifying Values

Action	Code	Result
Access value	student["name"]	"Ali"
Safe access	student.get("grade", "Not found")	"Not found"
Add new pair	student["grade"] = "A"	Adds new key

Update value	student["age"] = 23	Changes value
Update multiple	student.update({"age": 24, "grade": "A+"})	Updates many

---

◇ 3.5.3

Method	Description	Example	Output
pop(key)	Removes a specific key and returns its value	d.pop("age")	returns removed value
popitem()	Removes last inserted key-value pair	d.popitem()	returns (key, value)
del	Deletes a specific key or entire dictionary	del d["age"]	key deleted
clear()	Removes all items	d.clear()	{}

◇ 3.5.4 Looping Through a Dictionary

Loop Type	Code	Description
Keys	for key in d:	Default loop (iterates over keys)
Keys explicitly	for k in d.keys():	Access only keys
Values	for v in d.values():	Access only values
Key + Value	for k, v in d.items():	Access both together

◇ 3.5.5

A dictionary can contain other dictionaries — useful for structured data.

```
students = {
    "student1": {"name": "Ali", "age": 20},
```

```
    "student2": {"name": "Sara", "age": 22}
}
print(students["student1"]["name"])  # Ali
```

◇ 3.5.6 Dictionary Methods

Method	Purpose	Example
get(key[, default])	Returns value or default	d.get("x", 0)
keys()	Returns all keys	d.keys()
values()	Returns all values	d.values()
items()	Returns key-value pairs	d.items()
update(dict)	Adds/updates from another dictionary	d.update(d2)
copy()	Creates a shallow copy	d.copy()
setdefault(k[, v])	Returns value or inserts default	d.setdefault("x", 0)
pop(), popitem(), clear()	Remove items	—

◇ 3.5.7 Built-in Functions with Dictionaries

Function	Description	Example
len(d)	Number of items	len(d)
sorted(d)	Sorted list of keys	sorted(d)
sum(d.values())	Sum of numeric values	sum(d.values())
min(d.keys()), max(d.values())	Finds smallest/largest	—

◇ 3.5.8 Dictionary Comprehension

Create or transform dictionaries in one line.

squares = {x: x**2 for x in range(1, 6)}

Output: {1:1, 2:4, 3:9, 4:16, 5:25}

◇ 3.5.9 Why Use Dictionaries?

Fast data lookup	Search by key instantly
Data mapping	Country → Capital, Word → Meaning
JSON / API data handling	Used in real-world apps like ChatGPT, YouTube API, etc.

✸ Key Takeaways

- ∉ Dictionaries store data in **key-value** pairs.
- ∉ They are **mutable** and **unordered** (but ordered since Python 3.7).
- ∉ Perfect for **fast lookups**, **JSON data**, and **structured storage**.
- ∉ Methods like get(), update(), and items() make them powerful and flexible.

****

4: Functions & Modules

4.1 Python Functions: Quick Summary

◇ 4.1.1 What is a Function?

A **function** is a reusable block of code that performs a specific task. It helps make programs **organized**, **modular**, and **efficient**.

Syntax:

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

◇ 4.1.2 Why Use Functions?

☑ Avoid repetition of code

☑ Make code modular (easy to understand & debug)

☑ Improve reusability and organization

☑ Separate logic into smaller, meaningful parts

◇ 4.1.3 Function Types

Type	Description	Example
Built-in	Predefined by Python	print(), len(), sum()
User-defined	Created by you	def greet(): ...

◇ 4.1.4 Parameters vs Arguments

Term	Meaning	Example
Parameter	Variable inside the function definition	def greet(name): → name is a parameter
Argument	Actual value passed when calling function	greet("Nouman") → "Nouman" is an argument

---

◇ 4.1.5 Return Statement

- Sends a value back to where the function was called.
- Ends the function execution.

```
def add(x, y):
    return x + y
```

If no return is given → the function returns **None** by default.

◇ 4.1.6 Types of Arguments

Type	Description	Example
Positional	Matched by order	add(2, 3)
Keyword	Matched by name	add(y=3, x=2)
Default	Predefined value	def greet(name="User"):
***Arbitrary (args)**	Accepts multiple positional arguments (tuple)	def add_all(*nums):
****Arbitrary Keyword (kwargs)**	Accepts multiple keyword arguments (dictionary)	def info(**details):

◇ 4.1.7 Scope of Variables

Scope	Description	Example

Local	Variable declared inside a function	Used only within that function
Global	Declared outside all functions	Accessible everywhere
Nonlocal	Used inside nested functions to modify outer variable	Used with nonlocal keyword

---

◇ 4.1.8 *args and **kwargs

Feature	Meaning	Output Type	Use
***args**	Multiple positional inputs	Tuple	For unknown number of items
****kwargs**	Multiple named inputs	Dictionary	For flexible data like user info

Example:

```
def info(*args, **kwargs):
    print(args)
    print(kwargs)
```

◇ 4.1.9 When to Use return, parameters, args, kwargs

Situation	Use
You want to reuse a result	return
Function needs input	Parameters
Inputs may vary	*args
Named inputs vary	**kwargs

💡 Example Summary

```
def student(name, *marks, **info):
    print("Name:", name)
    print("Marks:", marks)
    print("Details:", info)
    return sum(marks)/len(marks)
```

Output:

```
Name: Hanifa
Marks: (85, 90, 78)
Details: {'age': 20, 'course': 'Python'}
Average = 84.33
```

✹ In One Line Summary: Functions = Reusable blocks

that take **inputs (parameters)**, perform **actions (logic)**, and optionally **return outputs (results).**

4.2 Python Variable Scope & Lifetime

◇ 4.2.1 What is Scope?

Scope defines **where a variable can be accessed or modified** in your program.

It decides **which part of the code knows about which variables**.

Lifetime means **how long** that variable exists in memory.

◇ 4.2.2 The 4 Types of Scope (LEGB Rule)

Scope Type	Description	Where It Exists	Keyword	Lifetime	Example
L – Local	Variables created **inside a function**. Only accessible in that function.	Inside function	*(none)*	Exists until function finishes	def greet(): name = "Hanifa"
E – Enclosing	Variables in **outer (parent) function** used by inner (nested) function.	Outer function (if nested)	nonlocal	Until outer function ends	def outer(): def inner():
G – Global	Variables declared **outside any function**, accessible everywhere (if declared global inside function).	Whole program	global	Until program ends	name = "Khanam"
B – Built-in	Python's predefined names like print(), len(), sum().	Provided by Python	*(none)*	Always	print("Hello")

◇ 4.2.3 How Python Searches Variables

When you use a variable, Python looks in this exact order:

Local → Enclosing → Global → Built-in

If not found → NameError.

🧠 Lifetime of Variables

Type	Created When	Destroyed When
Local	Function starts	Function ends
Enclosing	Outer function runs	Outer function ends
Global	Program starts	Program ends
Built-in	Python interpreter starts	Interpreter closes

🪄 Example Summary

```
name = "Global Hanifa"  # Global

def outer():
    name = "Outer Khanam"  # Enclosing

    def inner():
        nonlocal name  # modifies enclosing variable
```

```
        name = "Inner Hanifa"
        print("Inner:", name)  # Inner Hanifa

    inner()
    print("Outer:", name)  # Inner Hanifa (changed by nonlocal)

outer()
print("Global:", name)  # Global Hanifa (unchanged)
```

💡 Quick Tips

☑ Use **local** variables for temporary data.

☑ Use **global** only when multiple functions must share data.

☑ Use **nonlocal** when inner function must modify outer variable.

☑ Built-ins are always available — don't override them (avoid naming variable len or sum).

🌈 One-Line Essence:

🧩 **Scope controls visibility** of variables. 🕰️ **Lifetime controls duration** of variables in memory.

****

4.3 Anonymous functions(lambda)

◇ 4.3.1 What is a Lambda Function?

A **lambda function** in Python is a **small, anonymous function** — meaning it doesn't need a name.

It's used when:

- You need a function **for a short time**.
- Defining a full function with def would be **too long or unnecessary**.

👉 The syntax:

lambda arguments: expression

- lambda → keyword to create the function
- arguments → like parameters
- expression → a single line of code that is automatically **returned**

💡 Example 1: Normal function vs Lambda

◇ Using def

```
def square(x):
    return x * x
print(square(5))
```

◇ Using lambda

```
square = lambda x: x * x
print(square(5))
```

☑ Both give the same result: 25

But the lambda version is shorter and doesn't need a separate return statement.

💡 Example 2: Multiple arguments

You can have more than one parameter too:

```
add = lambda a, b: a + b
print(add(3, 7))
```

☑ Output: 10

💡 Example 3: Inline use (without assigning to a variable)

Sometimes we use a lambda **directly inside another function**, for example:

```
print((lambda x, y: x * y)(4, 5))
```

☑ Output: 20

⚠ Important Notes

- Lambda functions can **only have one expression** — no loops, no multiple statements.
- They're great for **short tasks** but **not** for big logic.

🧠 Quick Summary

Normal Function	Lambda Function
Defined with def	Defined with lambda
Can have multiple lines	Single expression only
Has a name	Usually anonymous
Used many times	Used temporarily or inline

****

4.4 1️⃣ map() — Apply a Function to Every Item

◇ What it Does:

map() applies a given function to **each item** in an iterable (like a list, tuple, etc.) and returns a new map object (which you can convert to a list).

◇ Syntax:

map(function, iterable)

◇ Example:

numbers = [1, 2, 3, 4, 5]

# Double each number
doubled = map(lambda n: n * 2, numbers)
print(list(doubled))

Output:[2, 4, 6, 8, 10]

◇ Where & Why Use:

- When you want to **transform** all elements in a sequence.
- Easier and faster than writing loops.

💡 *Example Scenarios:*

- Converting list of strings to integers
- Converting temperatures (°C → °F)
- Applying discounts to prices

****

4.4 2️⃣ filter() — Filter Items Based on a Condition

◇ What it Does:

filter() keeps only those elements that satisfy a **given condition (True/False)**.

◇ Syntax:

filter(function, iterable)

◇ Example:

```
numbers = [10, 25, 30, 47, 50]

# Keep only even numbers
evens = filter(lambda n: n % 2 == 0, numbers)
print(list(evens))
```

Output:

```
[10, 30, 50]
```

◇ Where & Why Use:

- When you want to **select only specific** items from a list.
- More readable than using long for loops with if.

💡 *Example Scenarios:*

- Get only adult ages from a list of ages
- Filter valid email addresses
- Find students who passed (> 50 marks)

4.4 3️⃣ reduce() — Combine All Items into a Single Value

◇ What it Does:

reduce() repeatedly applies a function to **pair of items**, reducing the iterable to **a single result** (sum, product, etc.).

◇ Syntax:

```
from functools import reduce
reduce(function, iterable)
```

◇ Example:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

# Multiply all numbers
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

Output:

◇ Where & Why Use:

- When you want a **single cumulative value** (sum, product, max, etc.)
- Works well in analytics and mathematical operations.

💡 *Example Scenarios:*

- Find total sales amount
- Multiply all numbers in a list
- Combine strings into a sentence

****

4.4 4️⃣ zip() — Combine Multiple Iterables Together

◇ What it Does:

zip() combines two or more iterables **element by element** into tuples.

◇ Syntax:

```
zip(iterable1, iterable2, ...)
```

◇ Example:

```
names = ['Alice', 'Bob', 'Charlie']
marks = [85, 90, 78]

combined = zip(names, marks)
print(list(combined))
```

Output:

```
[('Alice', 85), ('Bob', 90), ('Charlie', 78)]
```

◇ Where & Why Use:

- When you want to **pair related data** (like names with marks)
- Makes it easy to loop through multiple lists at once.

💡 *Example Scenarios:*

- Combine student names and grades
- Match products with prices
- Combine keys and values to make a dictionary

****

❀ Summary Table

Function	Purpose	Returns	Typical Use Case
map()	Apply function to each item	Map object (convert to list)	Transform all items (e.g., square numbers)
filter()	Keep items that match a condition	Filter object	Select items meeting criteria (e.g., even numbers)
reduce()	Combine all items into one value	Single value	Total, product, concatenation
zip()	Combine multiple sequences element-wise	Zip object (of tuples)	Pair related data (e.g., name with mark)

❁ Example Mixing All 4 Together

```
from functools import reduce

names = ["Alice", "Bob", "Charlie", "David"]
marks = [85, 40, 95, 60]

# Combine using zip
students = zip(names, marks)

# Filter only those who passed
passed = filter(lambda s: s[1] >= 50, students)

# Get only marks
marks_only = map(lambda s: s[1], passed)

# Find total marks of passed students
total_marks = reduce(lambda a, b: a + b, marks_only)

print("Total marks of passed students:", total_marks)
```

Output:

```
Total marks of passed students: 240
```

Function	Purpose	Returns	Conversion Options	When to Convert	Example	Notes
lambda	Creates a small anonymous function (one-line only)	A callable (function)	❌ Not needed	Used inside map, filter, reduce	lambda x: x*x	No def or return
map()	Applies a function to **every element** of an iterable	Iterator (map object)	✅ list(), tuple(), set(), dict()*	When you want to see or reuse results	map(lambda x:x*2, [1,2,3])	*dict() only if (key, value) pairs
filter()	Keeps only elements that meet a **condition**	Iterator (filter object)	✅ list(), tuple(), set()	When you want to display or store results	filter(lambda x:x>0, nums)	Returns only values where

						condition → True
reduce()	Combines all elements into **a single value**	Single final value	❌ Not needed	Already gives one value	reduce(lambda x,y:x+y, nums)	Must import from functools
list()	Creates a list from iterable	List	—	To view ordered, mutable data	list(map(...))	Keeps duplicates, preserves order
tuple()	Creates an immutable sequence	Tuple	—	When you want results unchangeable	tuple(map(...))	Keeps order, immutable
set()	Creates a unique unordered collection	Set	—	When duplicates should be removed	set(map(...))	Removes duplicates, unordered
dict()	Creates a dictionary	Dictionary	—	When iterable has (key, value) pairs	dict(map(...))	Each element must be 2-element tuple

****

4.5 Python Modules — From Scratch to Mastery

◇ 4.5.1 What is a Module?

👉 In simple words:

A **module** in Python is a **file that contains Python code** — functions, variables, or classes — that you can **import and use** in another program.

So, you can think of a **module as a toolbox** 🧰.

Each tool (function or variable) inside the box does a specific job — and instead of writing those tools again and again, you just **import the toolbox** whenever you need it.

Example (Everyday Analogy)

Imagine you're building a house 🏠:

- You don't make new hammers and screwdrivers every time.
- You **bring your toolbox** (module) and use the tools you already have.

Same idea in Python:

Instead of rewriting the same code, you just **import it** from a module.

◇ 4.5.2 Why Do We Need Modules?

Reason	Explanation

Reusability	Write a function once and use it in multiple programs.
Organization	Keep your code clean and manageable by separating logic into files.
Collaboration	Multiple developers can work on different modules without interfering.
Saves Time	Many powerful modules are already built into Python (e.g., math, random, os).
Avoid Duplication	Instead of copying the same code everywhere, you just import it when needed.

◇ 4.5.3 When Should You Use a Module?

You should use (or create) a module when:

1️⃣ Your program is **getting long** — to divide it into smaller, meaningful parts.

2️⃣ You have functions or logic that you'll **reuse** in multiple programs.

3️⃣ You want to use **pre-written code** (e.g., math operations, random numbers, file handling).

4️⃣ You're working with **external libraries** (e.g., NumPy, Pandas, Django).

🧠 Example Thought Process

Situation:

You've written a program to calculate areas and perimeters of shapes.

Now you're making another program for geometry.

👉 Instead of copying those functions again —

you can **move them into a separate file (module)** and **import them** whenever needed.

◇ 4.5.4 Types of Modules in Python

There are **three main types** of modules:

Type	Description	Example
Built-in Modules	Already come with Python installation. You just import them.	math, random, datetime, os, sys
User-defined Modules	Modules you create yourself (your own Python files).	my_module.py
External Modules	Created by others — you install them using pip.	numpy, pandas, requests, flask

---

◇ 4.5.5 How to Import a Module

There are **several ways** to import modules depending on your needs:

◇ (a) Basic Import

Import the entire module.

```
import math

print(math.sqrt(25))   # 5.0
print(math.pi)         # 3.141592653589793
```

👉 Here:

- math is the module name.
- You access its contents using **dot notation** → math.function_name.

==========

◇ (b) Import a Specific Function or Variable

If you only need certain functions, import them directly.

```
from math import sqrt, pi

print(sqrt(16))  # 4.0
print(pi)        # 3.141592653589793
```

☑ Now you don’t need to write math. before each function.

==========

◇ (c) Import with an Alias (Nickname)

You can give a shorter name to a module for convenience.

```
import math as m
print(m.sqrt(49))
print(m.pi)
```

☑ Common practice in libraries like:

```
import numpy as np
import pandas as pd
```

==========

◇ (d) Import Everything from a Module (⚠ Not Recommended)

```
from math import *
print(sqrt(9))
print(pi)
```

☑ Works, but ⚠ can cause *name conflicts* if two modules have same function names.

◇ 4.5.6 Creating Your Own Module

Creating a module is **super easy** — you just make a Python file and reuse it!

💮 Step 1: Create a file my_module.py

```
def greet(name):
    return f"Hello, {name}! Welcome back."
def add(a, b):
    return a + b
```

💮 Step 2: Create another file main.py and import it

```
import my_module
print(my_module.greet("Khanam"))
print(my_module.add(5, 7))
```

☑ Output:

```
Hello, Khanam! Welcome back.
12
```

💡 Explanation:

- my_module.py → contains reusable functions (like your toolbox).
- main.py → your main program that uses those tools via import.

◇ 4.5.7 Where Does Python Look for Modules?

When you use import, Python looks for the module in this order:

1. The **current folder** (where your file is).
2. **Built-in modules** (like math, os)
3. **Installed packages** (in your Python environment).

You can check the search locations:

```
import sys
print(sys.path)
```

◇ 4.5.8 Advantages of Using Modules

Benefit	Explanation
🧩 **Code Reuse**	Use the same code across multiple programs.
🗂️ **Better Organization**	Divide large code into smaller logical files.
🔄 **Easy Maintenance**	Fix or update one module; all programs using it get updated.
⚙️ **Faster Development**	Use existing libraries instead of writing from scratch.

🧠 **Clarity**	Focus on what your code does, not how it does it internally.

🌸 Summary Table

Concept	Description	Example
Module	File that contains reusable Python code	math, os, random
Import	Bring code from another file/module	import math
Alias	Short name for module	import math as m
Specific Import	Import selected functions	from math import sqrt
User-defined Module	Your own module (e.g., my_module.py)	import my_module
External Module	Installed with pip	import numpy
Built-in Module	Comes with Python	import random

🧠 In Short:

A **module** is a **Python file** that helps you **reuse**, **organize**, and **extend** your code efficiently.

You use them when your code grows large or when you want to use pre-written tools (like math, random, etc.).

Module	Use / Domain
math	Mathematics functions (sqrt, sin, log, etc.)
random	Random numbers, shuffling, sampling
datetime	Dates and times (dates, timestamps, intervals)
os	Operating system interface (file system, paths)
sys	Python interpreter and environment info
collections	Specialized container datatypes (deque, Counter, defaultdict, etc.)
statistics	Statistical functions (mean, median, variance)
re	Regular expressions (pattern matching)
json	Working with JSON data (parse, dump)
time	Time-related functions (sleep, time, performance counters)
itertools	Tools for creating iterators and combinatoric sequences
functools	Higher-order functions and operations on callable objects (e.g. reduce)
os.path	Path operations (join, basename, dirname, etc.)
subprocess	Running external commands and processes
threading / multiprocessing	Concurrency and parallelism
socket	Network socket programming
logging	Logging framework
urllib	URL handling (fetching, parsing)
shutil	High-level file operations (copy, move, delete)
heapq	Heap queue algorithm (min-heap, etc.)

****

5: Python File Handling: Quick Summary

◇ 5.1 What is File Handling?

File handling allows Python programs to **store, read, write, and manage** data permanently on the disk — instead of keeping it temporarily in memory.

◇ 5.2 File Operations Overview

Operation	Description	Function
📄 **Open a file**	Access a file for reading/writing	open(filename, mode)
📖 **Read from file**	Get data from the file	read(), readline(), readlines()
✍️ **Write to file**	Save data into a file	write(), writelines()
➕ **Append data**	Add new data without deleting old content	Open file in "a" mode
❌ **Close file**	Free memory and resources	close() (auto with with)

◇ 5.3 File Modes

Mode	Meaning	Description
'r'	Read	Opens file for reading (must exist)
'w'	Write	Creates file or overwrites old data
'a'	Append	Adds new data at the end
'r+'	Read + Write	Opens file for reading and writing
'w+'	Write + Read	Overwrites old file and allows reading
'a+'	Append + Read	Adds data and can also read
'rb' / 'wb'	Binary Read/Write	For non-text files (images, models, etc.)

◇ 5.4 Reading Files

Method	Description	Example
read()	Reads entire file	file.read()
readline()	Reads one line	file.readline()
readlines()	Reads all lines into a list	file.readlines()

☑ **Best Practice:** Use for line in file: to read large files efficiently.

◇ 5.5 Writing Files

Method	Description	Example
write()	Writes a single string	file.write("Hello\n")
writelines()	Writes a list of strings	file.writelines(lines)
"a" mode	Appends new text to existing file	Keeps previous content

☑ Use with open("file.txt", "w") as file: to **auto-close** the file safely.

◇ 5.6 Binary Files

Mode	Purpose	Example
'rb'	Read binary file	open("photo.png", "rb")

'wb'	Write binary file	open("photo_copy.png", "wb")
'ab'	Append binary data	Used for media or models

☑ Used for images, videos, .exe, .pkl (AI models).

◇ 5.7 File Handling with JSON

JSON = **JavaScript Object Notation**, used to store **structured (nested) data** like dictionaries and lists.

◇ JSON Module Summary

Function	Description	How to Use	When / Where to Use
json.dump(data, file)	Write Python data to a JSON **file**	json.dump(dict, f, indent=4)	Save structured data permanently
json.dumps(data)	Convert Python object → JSON **string**	json_str = json.dumps(dict)	Send data via API or web
json.load(file)	Read JSON **file** → Python object	data = json.load(f)	Read saved configuration or dataset
json.loads(string)	Convert JSON **string** → Python object	data = json.loads(json_str)	Parse web API response

☑ **Best Practice:**

- Use indent=4 for readability.
- Use sort_keys=True for consistency.
- Use try...except (FileNotFoundError, JSONDecodeError) for safety.

🧩 1. Story Time

ımagine you're building a **student management app** that stores not only names and marks, but also details like which subjects each student has, and maybe their login info too. This data looks like this:

```
students = {
   "Areeba": {"age": 20, "courses": ["Python", "Math"], "marks": {"Python": 95, "Math": 88}},
   "Ali": {"age": 21, "courses": ["Python", "AI"], "marks": {"Python": 90, "AI": 85}}
}
```

If you tried to save this to a .txt or .csv file — it would get messy. That's where **JSON (JavaScript Object Notation)** comes in —it stores **nested** and **structured** data beautifully in key–value pairs like this 👇

```
{
  "Areeba": {"age": 20, "courses": ["Python", "Math"], "marks": {"Python": 95, "Math": 88}},
  "Ali": {"age": 21, "courses": ["Python", "AI"], "marks": {"Python": 90, "AI": 85}}
}
```

So now your data is clean, organized, and readable in **any programming language**!

◇ 5.7.1 What is JSON?

- JSON stands for **JavaScript Object Notation**.
- It's a **lightweight, text-based** data format.
- It represents data as **key–value pairs** (just like Python dictionaries).
- Commonly used for **data exchange** between programs, APIs, and databases.

==========

◇ 5.7.2 Python's json Module

Python's built-in **json module** allows you to:

- Convert Python objects → JSON format (called **serialization**)
- Convert JSON data → Python objects (called **deserialization**)
- Read JSON data from a file
- Write JSON data to a file

==========

◇ 5.7.3 Commonly Used JSON Functions

Function	Purpose	When to Use	Example
json.dump()	Write JSON data to a **file**	When saving data permanently	json.dump(data, file)
json.load()	Read JSON data from a **file**	When loading JSON from file	data = json.load(file)
json.dumps()	Convert Python object to **JSON string**	When sending JSON over network or printing	json.dumps(data)
json.loads()	Convert JSON string to **Python object**	When receiving JSON as a string	data = json.loads(json_string)

==========

◇ 5.7.4 Writing JSON Files

➤ Example — json.dump()

```
import json
student = {
    "name": "Areeba",
    "age": 20,
    "marks": {"Python": 95, "Math": 88}
}
with open("student.json", "w") as f:
    json.dump(student, f, indent=4)  # indent makes it readable
```

Output (student.json file):

```
{
    "name": "Areeba",
    "age": 20,
```

```
    "marks": {
        "Python": 95,
        "Math": 88
    }
}
```

==========

◇ 5.7.5 Reading JSON Files

➤ Example — json.load()

```
import json
with open("student.json", "r") as f:
    data = json.load(f)
print(data["name"])     # Output: Areeba
print(data["marks"]["Python"])  # Output: 95
```

==========

◇ 5.7.6 Working with JSON Strings (not files)

Sometimes you'll receive data from an API or web server as a **JSON string**, not a file.

➤ Convert Python object → JSON string

```
import json
student = {"name": "Ali", "age": 21}
json_string = json.dumps(student)
print(json_string)
```

Output:

```
{"name": "Ali", "age": 21}
```

➤ Convert JSON string → Python object

```
data = json.loads('{"name": "Ali", "age": 21}')
print(data["name"])   # Output: Ali
```

==========

◇ 5.7.7 Combining JSON with Lists

You can easily store a list of dictionaries too 👇

```
import json
students = [
```

```
    {"name": "Areeba", "age": 20, "marks": 95},
    {"name": "Ali", "age": 21, "marks": 90}
]
with open("students.json", "w") as f:
    json.dump(students, f, indent=4)
```

Reading back:

```
with open("students.json", "r") as f:
    data = json.load(f)

for student in data:
    print(student["name"], "→", student["marks"])
```

==========

◇ 5.7.8 Handling Non-Serializable Data

Not all Python objects can be converted to JSON (like sets or custom classes).

Example:

```
import json
data = {"students": {"Ali", "Areeba"}}  # set is not serializable
json.dumps(data)   # ❌ TypeError
```

☑ Solution:

Convert it to a list before dumping:

```
data = {"students": list({"Ali", "Areeba"})}
json.dumps(data)  # Works fine
```

==========

◇ 5.7.9 Summary Table — JSON Functions

Function	Type	Purpose	Example Usage
json.dump(obj, file)	Writing	Write JSON data to a file	json.dump(data, f, indent=4)
json.load(file)	Reading	Read JSON data from a file	data = json.load(f)
json.dumps(obj)	Writing (String)	Convert Python object to JSON string	json.dumps(data)
json.loads(string)	Reading (String)	Convert JSON string to Python object	json.loads(json_str)

****

◇ 5.8 File Handling with CSV

CSV = **Comma-Separated Values**, used for storing **tabular data** (rows and columns).

◇ CSV Module Summary

Function / Class	Description	How to Use	When / Where to Use
csv.writer(file)	Write data row-by-row	writer.writerow([...])	Save records like Excel
csv.reader(file)	Read data as list of rows	for row in reader:	Read datasets line by line
csv.DictWriter(file, fieldnames)	Write data from dictionaries	writer.writerows(list_of_dicts)	When working with dict-based data
csv.DictReader(file)	Read CSV as dictionaries	for row in reader:	When you want access by column name

☑ **Best Practice:**

- Always open files with newline="".
- Headers must match field names.
- Convert string values to int() or float() when needed.

==========

🕮 **Story Time**

Imagine you're managing a **student record system** for your class. You want to store their data in a table format like this:

Name	Age	Marks
Areeba	20	95
Ali	21	90
Hina	19	88

If you save this as a text file — it's messy.

But if you save it as a **CSV (Comma-Separated Values)** file, it looks neat:

Name,Age,Marks
Areeba,20,95
Ali,21,90
Hina,19,88
That's what the **csv module** in Python helps us do easily —**read, write, and manage tabular data** just like Excel!

==========

◇ 5.8.1 What is the csv Module?

csv is a **built-in Python module** that provides functions to:

- Read data from .csv files
- Write data into .csv files
- Handle rows and columns automatically

You don't need to install anything — it's included with Python!

==========

◇ 5.8.2 Commonly Used CSV Functions

Function / Class	Purpose	When to Use	Example
csv.reader()	Reads CSV file as rows (lists)	When file has simple data (no headers)	reader = csv.reader(file)
csv.DictReader()	Reads CSV file as dictionaries (column names as keys)	When file has headers	reader = csv.DictReader(file)
csv.writer()	Writes data row by row	When writing lists to CSV	writer = csv.writer(file)
csv.DictWriter()	Writes data using dictionaries	When writing with headers	writer = csv.DictWriter(file, fieldnames=[...])
writer.writerow()	Writes a single row	When writing one line at a time	writer.writerow(["Name", "Age", "Marks"])
writer.writerows()	Writes multiple rows	When writing many records at once	writer.writerows(data)
reader.line_num	Returns current line number	To track progress while reading	print(reader.line_num)
csv.Sniffer()	Detects CSV file structure automatically	To detect delimiter or dialect	dialect = csv.Sniffer().sniff(sample)
csv.register_dialect()	Defines custom CSV formatting rules	For custom delimiters like ;	csv.register_dialect("semi", delimiter=';')

==========

◇ 5.8.3 Reading CSV Files

➤ Using csv.reader()

```
import csv
with open("students.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age', 'Marks']
['Areeba', '20', '95']
['Ali', '21', '90']
['Hina', '19', '88']
```

➤ Using csv.DictReader()

Automatically uses column headers as dictionary keys 👇

```
import csv
with open("students.csv", "r") as f:
    reader = csv.DictReader(f)
```

```
    for row in reader:
        print(row["Name"], "→", row["Marks"])
```

Output:

```
Areeba → 95
Ali → 90
Hina → 88
```

◇ 5.8.4 Writing CSV Files

➤ Using csv.writer()

```
import csv
data = [
    ["Name", "Age", "Marks"],
    ["Areeba", 20, 95],
    ["Ali", 21, 90],
    ["Hina", 19, 88]
]
with open("students.csv", "w", newline='') as f:
    writer = csv.writer(f)
    writer.writerows(data)
```

☑ Creates a CSV file with all rows at once.

➤ Using csv.DictWriter()

Best when your data is stored in **dictionaries**.

```
import csv
students = [
    {"Name": "Areeba", "Age": 20, "Marks": 95},
    {"Name": "Ali", "Age": 21, "Marks": 90},
    {"Name": "Hina", "Age": 19, "Marks": 88}
]
with open("students.csv", "w", newline='') as f:
    fieldnames = ["Name", "Age", "Marks"]
    writer = csv.DictWriter(f, fieldnames=fieldnames)

    writer.writeheader()      # Writes header row
    writer.writerows(students)  # Writes all student records
```

◇ 5.8.5 Detecting CSV File Format

If you don’t know how a CSV file is formatted (maybe uses ; instead of ,),

you can detect it automatically 👇

```
import csv
with open("data.csv", "r") as f:
    sample = f.read(1024)
    dialect = csv.Sniffer().sniff(sample)
    print("Delimiter used:", dialect.delimiter)
```

==========

◇ 5.8.6 Summary Table of CSV Functions

Function / Class	Type	Purpose	Example Usage
csv.reader(file)	Reading	Reads CSV rows as lists	for row in csv.reader(f): print(row)
csv.DictReader(file)	Reading	Reads CSV rows as dictionaries	for row in csv.DictReader(f): print(row['Name'])
csv.writer(file)	Writing	Writes rows as lists	writer.writerow(["Areeba", 20, 95])
csv.DictWriter(file, fieldnames)	Writing	Writes rows as dictionaries	writer.writerow({"Name":"Ali","Age":21,"Marks":90})
writer.writerow(row)	Writing	Writes one row	Single record
writer.writerows(rows)	Writing	Writes multiple rows	List of lists or dicts
csv.Sniffer().sniff(sample)	Detecting	Detects delimiter automatically	Useful for unknown CSVs
csv.register_dialect()	Formatting	Defines custom delimiters	For ; or \t separated files

==========

◇ 5.9 JSON vs CSV — Quick Comparison

Feature	JSON	CSV
Structure	Hierarchical / Nested	Tabular / Flat
Data Type	Dictionaries, Lists	Rows, Columns
Best For	APIs, configs, complex data	Datasets, reports, tables
File Size	Larger	Smaller
Read/Write Module	json	csv
Nested Data Support	✅ Yes	❌ No
Used In	Web, AI, APIs	Data Science, Excel
Human Readable	✅ Yes	✅ Yes
Works in Excel	❌	✅

==========

◇ 5.10 File Error Handling

Error Type	Cause	How to Handle
FileNotFoundError	File doesn't exist	except FileNotFoundError:
JSONDecodeError	Broken JSON data	except json.JSONDecodeError:
General I/O	Any input/output error	try...except: block

☑ Use try...except to prevent crashes and display friendly messages.

==========

🧩 11. Real-World Use Cases

Field	Example
🤖 **AI / ML**	Save model results in JSON, datasets in CSV
🌐 **Web Development**	Exchange API data in JSON, export reports in CSV
📊 **Data Science**	Store and analyze data in CSV
⚙️ **Automation**	Log data in text, summaries in JSON/CSV

==========

💬 12. Tech Analogy

File Type	Feels Like	Use Case
.txt	Notebook 📝	Simple notes or logs
.json	Tree 🌳	Structured, nested data (APIs, configs)
.csv	Spreadsheet 📊	Tabular datasets, reports
Binary	Flash drive 💾	Images, videos, model files

==========

◇ 5.11 JSON vs CSV vs TXT — Quick Comparison

Feature	TXT	CSV	JSON
Structure	Unstructured	Tabular (rows & columns)	Hierarchical (key–value)
Nested Data?	❌ No	❌ No	✅ Yes
Readable by Humans?	✅	✅	✅
Readable by Programs?	❌	✅	✅
Best For	Notes, logs	Data tables, spreadsheets	APIs, structured data
Python Module	open()	csv	json
Supports Dictionaries?	❌	⚠️ Limited	✅ Fully

✨ In short:

- Use **TXT** for plain text
- Use **CSV** for table
- Use **JSON** for structured or nested data

💡 Best Practices Summary

1. ☑ Always use with open() for safety.
2. ⚙ Choose the correct mode ('r', 'w', 'a', 'rb', 'wb').
3. 🧠 Use json for structured data; csv for tables.
4. 🧹 Handle missing/corrupt files with try...except.
5. 💾 Backup files before overwriting ('w' mode deletes old content).
6. 🌐 Validate JSON/CSV content before using in AI or web apps.

****

6: Error & Exception Handling in Python: Quick Summary

🐞 1️⃣ What Is an Error?

An **error** is a problem that stops your program from running correctly.

In Python, errors are mainly divided into two types:

Type	Description	Example
Syntax Error	You broke Python's grammar rules	print("Hello" → missing)
Logical Error	Code runs but gives wrong output	Wrong formula, e.g., avg = a + b / 2
Runtime Error (Exception)	Error occurs *while program is running*	10 / 0 → ZeroDivisionError

==========

🐞 2️⃣ What Is an Exception?

An **exception** is a *runtime error* that interrupts the flow of a program.

Examples: dividing by zero, using a variable that doesn't exist, reading a missing file, etc.

==========

🐞 3️⃣ Handling Exceptions with try and except

◇ Syntax

```
try:
    # Code that might cause an error
except ExceptionType:
    # Code that runs if that error occurs
```

◇ Example

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Please enter numbers only.")
```

==========

🐞 4️⃣ Using else with Try-Except

◇ Purpose:

Run a block only if no error occurred in the try section.

```
try:
    num = int(input("Enter number: "))
    print(10 / num)
```

```
except ZeroDivisionError:
    print("Division by zero not allowed!")
else:
    print("Division successful!")
```

☑ else executes only when the try block runs successfully (no error).

==========

🐞 5️⃣ The finally Block

◇ **Purpose:**

Code inside finally always runs — whether an error occurs or not.

Used for cleanup (like closing files or showing final messages).

```
try:
    file = open("data.txt", "r")
    print(file.read())
except FileNotFoundError:
    print("File not found!")
finally:
    print("File operation completed.")
```

☑ Even if there’s an error, finally runs at the end.

==========

🐞 6️⃣ Raising Your Own Exceptions (raise)

◇ **Purpose:**

Used to manually trigger an error when a condition fails.

```
num = int(input("Enter a positive number: "))
if num < 0:
    raise ValueError("Number cannot be negative.")
else:
    print("Number is valid.")
```

☑ Output:

ValueError: Number cannot be negative.

==========

🐞 7️⃣ Creating Custom (User-Defined) Exceptions

◇ **Purpose:**

To make your own meaningful errors that match your program’s logic.

```
class TooSmallError(Exception):
    def __init__(self, number):
        super().__init__(f"Please enter a number greater than 10. You entered {number}.")
try:
    number = int(input("Enter a number: "))
    if number < 10:
        raise TooSmallError(number)
except TooSmallError as e:
    print(e)
```

☑ Output:

Please enter a number greater than 10. You entered 5.

==========

🧽 8️⃣ When to Use Each Block

Block	Purpose	Runs When
try	Code that might fail	Always first
except	Handle errors	When an exception occurs
else	Run only if no error	When no exception occurs
finally	Cleanup or end message	Always (error or not)
raise	Manually trigger error	When you detect invalid condition

==========

🧽 9️⃣ Why Exception Handling Is Important

☑ Prevents your program from crashing

☑ Makes debugging easier

☑ Helps guide users with friendly messages

☑ Improves software reliability

☑ Enables controlled program flow

==========

🧠 1️⃣0️⃣ Common Built-in Exceptions in Python

Here's a table of frequently used **built-in exceptions**:

Exception Name	When It Occurs
ZeroDivisionError	Division by zero
ValueError	Invalid data type or value
TypeError	Incompatible data types (e.g., 2 + "hi")
NameError	Variable not defined
IndexError	Invalid list index
KeyError	Missing dictionary key
FileNotFoundError	File does not exist

IOError	General input/output error
AttributeError	Accessing an undefined attribute or method
ImportError	Module or object cannot be imported
RuntimeError	General runtime failure
MemoryError	Program runs out of memory
OverflowError	Arithmetic operation exceeds limit
AssertionError	Assertion statement failed
KeyboardInterrupt	Program interrupted by user (Ctrl+C)
EOFError	No input received (end of file reached)
PermissionError	Access denied to file or directory
RecursionError	Too many recursive calls (infinite recursion)

==========

❁ Summary: Custom Exception vs Inline Custom vs Inline True Custom Exception

🧩 1️⃣ Custom Exception — (The Professional Way)

- **Definition:**

A user-defined exception class that inherits from Python's built-in Exception. It represents a specific, meaningful error type in your program.

```
class LimitExceededError(Exception):
    """Raised when a limit is exceeded."""
    pass

raise LimitExceededError("Book issue limit crossed!")
```

- **Logic Behind It:**

Creates a new data type (class) that Python treats like an Exception

Can store custom attributes or methods

Lets you use except LimitExceededError: to catch only that error type

- **Benefits:**

☑ Semantic clarity (error name explains purpose)

☑ Reusable in large applications

☑ Can include documentation, data, and behavior

☑ Helps in debugging and modular exception handling

- **When to Use:**

For structured, real-world projects

When you need specific domain errors (like PaymentFailedError, DataNotFoundError)

🧩 2️⃣ Inline Custom Exception — (The Quick & Simple Way)

- **Definition:**

Using Python’s built-in Exception class directly with a custom message.

raise Exception("Custom Error: Invalid input detected!")

- **Logic Behind It:**

Doesn’t define a new exception type

Simply creates an instance of Exception with a custom message

- **Benefits:**

☑ Very simple and fast to use

☑ Ideal for temporary or small scripts

🏵 Technically lightweight (no class creation)

- **Limitations:**

❌ All such errors are of type Exception → no unique identity

❌ Can’t be caught separately (you must catch Exception)

❌ Not suitable for professional, reusable systems

- **When to Use:**

Quick testing, demos, or student-level projects

Simple validation where unique error types aren’t needed

🏵 3️⃣ Inline True Custom Exception — (The Dynamic Way)

- **Definition:**

Dynamically creates a new exception class at runtime using the type() function —then raises it immediately.

raise type("TemperatureTooHigh", (Exception,), {})("Temperature exceeds safe limit!")

- **Logic Behind It:**

type() in Python can create a class dynamically

Here it creates:

→ a new class named "TemperatureTooHigh"

→ that inherits from Exception

→ with no body ({})

Then it instantiates and raises that class immediately.

- **Benefits:**

☑ Dynamic creation of named exceptions (no prewritten class needed)

☑ Great for code generation, automation, or meta-programming

☑ Keeps code compact and flexible

- **Limitations:**

❌ Less readable and harder for beginners

❌ Not reusable or documented

❌ Difficult to debug in large projects

- **When to Use:**

In dynamic or generated code

In automation tools, AI systems, or scripts that create new logic on the fly

---

📊 Comparison Table

Feature	Custom Exception	Inline Custom Exception	Inline True Custom Exception
Creation	Defined using class	Raised using built-in Exception	Created dynamically with type()
Syntax	class MyError(Exception): ...	raise Exception("msg")	raise type("MyErr",(Exception,),{})("msg")
Exception Type	Unique class	Generic Exception	Unique runtime class
Readability	⭐ Clear	⭐ Simple	⚠️ Complex
Reusability	✅ High	❌ None	⚠️ Temporary
Use Case	Large, real apps	Small scripts/tests	Dynamic runtime code
Debugging Ease	🧩 Very easy	😐 Medium	😕 Difficult
Performance Impact	Slight (class defined once)	None	Slight (class built at runtime)

💬 Real-World Analogy

Scenario	Equivalent Exception Type
Installing a permanent fire alarm system	Custom Exception
Shouting “Fire!” manually when something’s wrong	Inline Custom Exception
Building a temporary, smart alarm on the fly for testing	Inline True Custom Exception

🧠 In Short (Essence)

You want to…	Use this
Create professional, reusable, domain-specific errors	🧱 Custom Exception
Just throw a quick message error	⚡ Inline Custom Exception
Dynamically generate and raise unique exceptions at runtime	🧩 Inline True Custom Exception

🔍 **Technical Insight**

Custom Exception = Static class creation (before runtime)

Inline Custom = Exception instance creation (no new type)

Inline True Custom = Dynamic class creation + instance raising (during runtime)

🌈 **In Summary: The Full Exception Handling Flow**

```
try:
    # risky code
except SpecificError:
```

```
    # handle specific error
except AnotherError:
    # handle another error
else:
    # runs only if try succeeds
finally:
    # always runs (cleanup)
```

****