Basics Of Python

- Python is a high-level, interpreted programming language created by Guido van Rossum in 1991.
- It is known for being easy to read, simple to write, and powerful.
- Its syntax is very close to English, which makes it beginner-friendly.

👉 Example:

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

Just one line prints a message—very simple compared to other languages like Java or C++.

Key Features of Python

Here are the most important features (the 20% that cover 80% of understanding):

1. **Easy to Learn and Readable**
- Syntax looks like English.
- Example: if age > 18: print("Adult")
2. **Interpreted Language**
- No need to compile; Python code runs line by line.
3. **Dynamically Typed**
- No need to declare variable types.
4. **High-Level Language**
- Programmer focuses on solving problems, not on memory management.
5. **Cross-Platform**
- Runs on Windows, Mac, Linux, and even mobile (with some frameworks).
6. **Object-Oriented & Functional**
- Supports classes/objects (OOP) and also functional style.
7. **Huge Standard Library & Community Support**
- Ready-made modules for math, file handling, web requests, etc.
8. **Extensible & Integrated**
- Can be combined with other languages like C, C++, Java.
9. **Open Source**
- Free to use, with a huge developer community.
10. **Versatile**
- One language for web, data science, ML, scripting, etc.

Interpreted Language

A language where code is executed line by line by an interpreter at runtime.

➞ Example: Python, JavaScript.

Compiled Language

A language where code is converted into machine code by a compiler before execution.

➞ Example: C, C++.

Interpreter

A program that reads and executes code line-by-line, without creating a separate executable.

Compiler

A program that converts the entire code into machine code at once, generating an executable file.

Python is both compiled and interpreted.
It first compiles the code into bytecode, and then the Python interpreter (PVM) interprets/executed that bytecode line-by-line.

Data Types:

Data types in Python define the **type of value** a variable can hold (numbers, text, list, etc.).
Python has the following main data types:

1. Numeric Types

- **int** → Whole numbers
- **float** → Decimal numbers
- **complex** → Numbers with real + imaginary part

a) int (Integer)

Whole numbers without decimals

```python
x = 10
y = -5
```

b) float

Numbers with decimal points

```python
a = 5.6
b = -3.14
```

c) complex

Numbers with imaginary part

```python
z = 2 + 5j
```

`j` represents √(-1)

2. String

- **str** → Text data (e.g., "Hello")

Used to store text

```python
name = "Raksha Shetty"
```

Strings are **immutable** (cannot modify characters directly).

3. Boolean

- **bool** → True or False

Stores only:

- **True**
- **False**

```python
is_valid = True
flag = False
```

Used mainly in conditions and comparisons.

4. Sequence Types

- **list** → Ordered, changeable
- **tuple** → Ordered, unchangeable
- **range** → Sequence of numbers (usually in loops)

a) list

- Ordered
- Changeable (mutable)
- Allows duplicates

```python
fruits = ["apple", "banana", "orange"]
```

b) tuple

- Ordered
- Unchangeable (immutable)
- Faster than list

```python
colors = ("red", "blue", "green")
```

c) range

Used to generate sequences of numbers

```python
r = range(1, 6)  # 1,2,3,4,5
```

5. Mapping Type

- **dict** → Key-value pairs

dict (Dictionary)

Stores data in key-value pairs

- Unordered
- Mutable

```python
student = {
    "name": "Raksha",
    "age": 22,
    "course": "AI & ML"
}
```

6. Set Types

- **set** → Unordered, unique items
- **frozenset** → Immutable set

a) set

- Unordered
- No duplicate values

```python
numbers = {1, 2, 3, 3}
# output → {1, 2, 3}
```

b) frozenset

- Same as set
- **Immutable**

```python
fs = frozenset([1, 2, 3])
```

How to check Datatype of value in Python:

```python
x = 10
print(type(x))   # <class 'int'>
```

---

1. List

"In Python, a **list** is a collection that can store multiple items in a single variable. It is one of the most commonly used data structures.

- A list is **ordered** (items have a defined order).
- It is **mutable**, meaning we can change, add, or remove elements after creating it.
- A list can store **different data types** together, like numbers, strings, or even other lists.

1. What a List Is

- **Definition:** A Python **list** is an **ordered**, **mutable** (changeable) collection that can hold **mixed data types**.
- Example:

```python
my_list = [10, "apple", 3.14, True]
```

- **Why important for Data Analysts:** You'll store rows, columns, cleaned datasets, temporary values, etc., in lists before putting them into Pandas or NumPy.

2. Core List Operations (The 20% You'll Use Most)

Here's the essential toolkit:

Action	Code	Notes
Create	`lst = [1, 2, 3]`	Most common way
Access item	`lst[0]`	Zero-based index
Access last item	`lst[-1]`	Negative indexing
Slice	`lst[1:3]`	Gets items from index 1 to 2
Add at end	`lst.append(4)`	1 item only
Add multiple	`lst.extend([5,6])`	Joins another list
Insert	`lst.insert(1, 99)`	At specific index
Remove by value	`lst.remove(2)`	Removes first match only
Remove by index	`lst.pop(1)`	Also returns the removed item
Change value	`lst[0] = 100`	Mutability in action
Length	`len(lst)`	Count elements
Check presence	`if 3 in lst:`	True/False

3. Variations of Lists You'll Encounter

These are still "lists" in concept but with **slightly different behaviors or uses**:

1. **List of Lists (2D List)**
   - Useful for storing table-like data before DataFrames.
   - Example:

```python
data = [
    ["Name", "Age"],
    ["Alice", 25],
    ["Bob", 30]
]
print(data[1][0])  # 'Alice'
```

2. **List Comprehension (Quick Creation)**
   - Compact way to create lists from loops:

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

   - Often used in data cleaning (e.g., lowercasing strings).

✅ When List Comprehension is Used?

- When you want to **create a new list** from an existing iterable (like list, range, string).
- When you need to **apply a transformation** (e.g., square each number).
- When you want to **filter elements** while creating the list.
- When the logic is **simple enough to fit in one line** (for complex logic → normal loop is better).

3. Lists with Mixed Data Types

- Allowed in Python:

```python
mixed = [1, "apple", True]
```

- But in data analysis, usually keep lists with the **same type** for easier processing.

4. **Mixing Data Types in Calculations**

```python
lst = [1, "2", 3]
sum(lst)  # ❌ TypeError
```

5. **Copying 2D Lists**
   - `copy()` works for 1D lists but **not fully** for nested lists.
   - Use `import copy; copy.deepcopy(nested_list)`.

---

5. Lists in a Data Analyst's Daily Work

- **Before Pandas:**
  Reading CSV → store rows in a list of lists → later convert to DataFrame.
- **During Cleaning:**
  Apply transformations with list comprehensions.
- **Quick Stats:**
  Use `min()`, `max()`, `sum()`, `sorted()` on numeric lists.

---

2. Tuple

1. What is a Tuple?

- **Definition:** A **tuple** is an **ordered, immutable** collection of items.
- **Immutable** means **once created, you cannot change, add, or remove elements**.
- Example:

```python
my_tuple = (10, "apple", 3.14)
```

Why important for data analysts?

- They're great for storing **fixed, read-only data** like column names, date ranges, or configuration values that should not change during processing.

2. Core Tuple Operations (The 20% You'll Use Most)

Action	Code	Notes
Create	`t = (1, 2, 3)`	Standard tuple
Single element tuple	`t = (5,)`	Comma required, otherwise it's just an int
Access element	`t[0]`	Zero-based indexing

Negative index	`t[-1]`	Access from end
Slice	`t[1:3]`	Works like lists
Length	`len(t)`	Count elements
Check presence	`if 3 in t:`	True/False
Concatenate	`t1 + t2`	Creates a new tuple
Repeat	`t * 3`	Repeat values

3. Variations of Tuples You'll Encounter

These are not *different data types*, but different **use cases** and structures:

1. **Tuple of Tuples (2D Tuple)**
   - Used when data should stay fixed:

```python
data = (
    ("Name", "Age"),
    ("Alice", 25),
    ("Bob", 30)
)
```

2. **Tuple Packing & Unpacking**
   - **Packing**: Group multiple values into a tuple automatically.

```python
t = 1, 2, 3  # tuple without parentheses
```

   - **Unpacking**: Assign tuple values to variables in one step.

```
a, b, c = (1, 2, 3)
```

   - Useful in data analysis for returning multiple results from a function.

3. **Named Tuples** (from `collections`)
   - Like tuples but with named fields for better readability:

```python
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p.x, p.y)
```

5. Tuples in a Data Analyst's Daily Work

- **Column Headers**:
  Used as fixed references so they can't be accidentally modified.

```python
columns = ("Name", "Age", "Salary")
```

- **Return Multiple Values from a Function**:

```python
def min_max(values):
    return min(values), max(values)

low, high = min_max([3, 1, 4])
```

- **Coordinates / Key-Value Pairs in Dictionaries**:
  Tuples can be dictionary keys because they are immutable:

```python
location_data = {(10, 20): "Point A", (15, 25): "Point B"}
```

3. Set

1. What is a Set?

- **Definition:** A Python **set** is an **unordered, mutable** collection of **unique elements**.
- **Key points**:
  - No duplicates allowed.
  - No guaranteed order (items may appear in any sequence).
  - Elements must be **hashable** (immutable types like numbers, strings, tuples — but not lists or dictionaries).

Why important for data analysts?

- Quickly **remove duplicates** from a dataset.
- Perform **mathematical set operations** like union, intersection, and difference.

2. Core Set Operations (The 20% You'll Use Most)

Action	Code	Notes
Create	`s = {1, 2, 3}`	Curly braces
Empty set	`s = set()`	`{}` makes an empty dict, not set
Add item	`s.add(4)`	One item only
Add multiple	`s.update([5, 6])`	Adds from any iterable
Remove item	`s.remove(2)`	Errors if not found
Safe remove	`s.discard(2)`	No error if missing
Pop random item	`s.pop()`	Removes and returns an arbitrary element
Length	`len(s)`	Count elements
Check presence	`if 3 in s:`	True/False

3. Variations / Special Uses of Sets

1. **Frozen Set**
   - An **immutable** version of a set — useful when you want a set to be used as a dictionary key or stored inside another set.

```python
fs = frozenset([1, 2, 3])
```

2. **Mathematical Set Operations**
   - **Union** (combine unique elements):

```python
a | b
a.union(b)
```

   - **Intersection** (common elements):

```python
a & b
a.intersection(b)
```

   - **Difference** (items in a but not in b):

```python
a - b
a.difference(b)
```

   - **Symmetric Difference** (elements in either set but not both):

```python
a ^ b
```

3. **Set Comprehension**
   - Like list comprehensions but creates a set:

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

5. Sets in a Data Analyst's Daily Work

- **Removing duplicates** from a dataset quickly:

```python
ids = [101, 102, 101, 103]
unique_ids = set(ids)  # {101, 102, 103}
```

- **Finding common or missing values** between two lists:

```python
completed = {1, 2, 3}
required = {2, 3, 4}
missing = required - completed  # {4}
```

- **Fast membership tests** (much faster than lists for large data):

```python
if 5000 in big_set:
    ...
```

4. Dictionary

1. What is a Dictionary?

- **Definition:** A **dictionary** is an **unordered, mutable** collection of **key–value pairs**.
- **Keys**: Must be unique and immutable (numbers, strings, tuples).
- **Values**: Can be any data type, even lists or other dictionaries.
- Example:

```python
my_dict = {"Name": "Alice", "Age": 25, "Skills": ["Python", "SQL"]}
```

Why important for data analysts?

- Perfect for storing **structured data** (like JSON from APIs, row/column mappings, configuration settings).
- You'll use them often when cleaning, transforming, or mapping data.

2. Core Dictionary Operations (The 20% You'll Use Most)

Action	Code	Notes
Create	`d = {"a": 1, "b": 2}`	Curly braces
Empty dictionary	`d = {}` or `d = dict()`	
Access value	`d["a"]`	Error if key missing
Safe access	`d.get("a")`	Returns `None` if missing
Add/Update	`d["c"] = 3`	Creates or updates
Delete by key	`del d["a"]`	Error if key missing
Safe delete	`d.pop("a", None)`	Returns value or default
Check key exists	`"a" in d`	True/False
Length	`len(d)`	Count of keys
Keys list	`d.keys()`	View of keys
Values list	`d.values()`	View of values
Items (pairs)	`d.items()`	Key–value tuples

3. Variations / Special Dictionary Types

1. **Nested Dictionary**
   - Dictionary inside a dictionary:

```python
student = {
    "name": "Alice",
    "marks": {"math": 90, "science": 85}
}
print(student["marks"]["math"])
```

2. **Dictionary Comprehension**
   - Compact way to create dictionaries:

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

Operators:

1. Arithmetic Operators – used for basic math

- `+` (Addition) → `5 + 3 = 8`
- `-` (Subtraction) → `5 - 3 = 2`
- `*` (Multiplication) → `5 * 3 = 15`
- `/` (Division) → `5 / 2 = 2.5`
- `//` (Floor Division) → `5 // 2 = 2` (gives integer result)
- `%` (Modulus) → `5 % 2 = 1` (remainder)
- `**` (Exponent) → `2 ** 3 = 8`

2. Comparison Operators – used to compare values (results are True/False)

- `==` (Equal to) → `5 == 5 → True`
- `!=` (Not equal to) → `5 != 3 → True`
- `>` (Greater than) → `5 > 3 → True`
- `<` (Less than) → `5 < 3 → False`
- `>=` (Greater than or equal to) → `5 >= 5 → True`
- `<=` (Less than or equal to) → `3 <= 5 → True`

3. Logical Operators – used to combine conditions

- `and` → Returns True if **both** conditions are true
  Example: `(5 > 3 and 10 > 5) → True`
- `or` → Returns True if **any one** condition is true
  Example: `(5 > 3 or 10 < 5) → True`
- `not` → Reverses the result
  Example: `not(5 > 3) → False`

4. Assignment Operators – used to assign values to variables

- `=` (Assign) → `x = 10`
- `+=` (Add and assign) → `x += 5` (same as `x = x + 5`)
- `-=` (Subtract and assign) → `x -= 3`
- `*=` (Multiply and assign) → `x *= 2`
- `/=` (Divide and assign) → `x /= 2`
- `//=` (Floor divide and assign)
- `%=` (Modulus and assign)
- `**=` (Power and assign)

String Operations

a) Accessing Parts of Strings

- Strings are sequences of characters; you can get parts using **indexing** and **slicing**.

```python
text = "Data Analyst"
print(text[0])     # 'D' - first character (index starts at 0)
print(text[-1])    # 't' - last character
print(text[5:12])  # 'Analyst' - substring from index 5 up to 11
```

b) Changing Case

- Convert text to lowercase, uppercase, or title case.

```python
print(text.lower())    # 'data analyst'
print(text.upper())    # 'DATA ANALYST'
print(text.title())    # 'Data Analyst'
```

c) Stripping Spaces

- Remove unwanted spaces from start/end.

```python
text = "  hello world  "
print(text.strip())    # 'hello world' - removes leading and trailing spaces
print(text.lstrip())   # removes only left spaces
print(text.rstrip())   # removes only right spaces
```

- .strip removes the spaces from beginning and end of the text only not from middle.
- .lstrip removes space from left side of the string
- .rstrip removes space from right side of the string

d) Splitting and Joining

- Split a string into parts (list) and join list back to string.

```python
sentence = "Data analyst working with Python"
words = sentence.split()          # splits on spaces by default
print(words)                      # ['Data', 'analyst', 'working', 'with', 'Python']

joined = "-".join(words)
print(joined)                   # 'Data-analyst-working-with-Python'
```

e) Finding and Replacing Text

- Find position of a substring, and replace parts of a string.

```python
text = "Data analysis is fun"
print(text.find("analysis"))     # 5 (start index of 'analysis')
print(text.replace("fun", "awesome"))  # 'Data analysis is awesome'
```

f) Checking Conditions

- Check if string contains a substring, or starts/ends with something.

```python
print("Data" in text)             # True
print(text.startswith("Data"))    # True
print(text.endswith("fun"))       # True
```

3. String Formatting

When you want to **create readable outputs** or insert variable values into strings, you use formatting.

a) Using `f-strings` (Python 3.6+)

```python
name = "Raksha"
score = 85
print(f"{name} scored {score} marks.")  # Raksha scored 85 marks.
```

- Easiest and most recommended way.

---

1. **What are Booleans?**

- A **Boolean** (bool) represents **two possible values**:
- True
- False
- They are used in **logical operations**, **conditions**, and **comparisons**

```
is_active = True
is_admin = False
print(type(is_active))   # <class 'bool'>
```

Conditional Statements

- Conditional statements help your program **make decisions** by checking if certain conditions are True or False.

1. `if` statement

Executes a block of code only if the condition is `True`.

```python
age = 20
if age >= 18:
    print("You are eligible to vote")
```

Output: You are eligible to vote

---

2. `if-else` statement

Provides two paths:

- One block if the condition is `True`
- Another block if it is `False`

```python
age = 16
if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible to vote")
```

Output: Not eligible to vote

---

3. `if-elif-else` chain

Checks multiple conditions in sequence.

- The first `True` condition executes, and the rest are ignored.
- If none are `True`, the `else` block runs.

```python
marks = 75
if marks >= 90:
    print("Grade A")
elif marks >= 70:
    print("Grade B")
elif marks >= 50:
    print("Grade C")
else:
    print("Fail")
```

Output: Grade B

---

4. Nested if

An `if` statement inside another `if`.
Useful when decisions depend on multiple conditions.

```python
age = 25
citizen = True

if age >= 18:
    if citizen:
        print("Eligible to vote")
    else:
        print("Not a citizen, cannot vote")
else:
    print("Underage")
```

Output: Eligible to vote

---

5. Ternary operator (shorthand if-else)

For simple one-line conditions.

```python
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)
```

Output: Adult

Loops

- Loops topic used to repeat tasks without writing the same code again and again.

Two types of Loops:

1. `for` loop

Used when you know how many times you want to repeat something (like iterating over a list, dictionary, or range of numbers).

Example 1 – Looping over a range:

python Copy Edit

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

Output:

```
0
1
2
3
4
```

👉 Note: `range(5)` goes from **0 to 4** (last number excluded).

Example 2 – Looping over a list:

python

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

Example 3 – Looping with index (`enumerate`):

python

```python
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

---

2. `while` loop

Used when you **don't know in advance** how many times to repeat. It keeps running as long as the condition is `True`.

Example 1 – Basic while loop:

```python
count = 0
while count < 3:
    print("Hello")
    count += 1
```

Output:

```
Hello
Hello
Hello
```

Example 2 – Infinite loop (⚠️ careful):

```python
while True:
    print("This will never stop!")  # unless we use break
```

---

4. Nested Loops (loop inside loop)

Useful when working with tables, matrices, or multiple levels of data.

```python
for i in range(3):
    for j in range(2):
        print(i, j)
```

Output:

```
0 0
0 1
1 0
1 1
2 0
2 1
```

Loop control statements

Loop control statements in Python are special keywords that change the normal flow of a loop.

- Normally, a loop runs from the first iteration to the last.
- With control statements, you can stop the loop early, skip certain iterations, or do nothing temporarily.

---

1. `break` → Exit the loop immediately

- When Python sees `break`, it **stops the loop completely**, no matter if iterations are left.

✔ Example:

```python
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output:

```
0
1
2
```

- **As soon as i == 3 loop ends.**

---

2. continue → Skip current iteration

- When Python sees continue, it **skips the rest of that iteration** and moves to the next one.

✔ Example:

```python
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Output:

```
0
1
3
4
```

👉 `2` is skipped.

---

3. pass → Do nothing (placeholder)

- pass literally means “do nothing”.
- Useful when you haven’t written code yet but need a syntactically valid loop.

✔ Example:

```python
for i in range(5):
    if i == 2:
        pass   # placeholder
    else:
        print(i)
```

Output:

```
0
1
3
4
```

👉 When `i == 2`, Python just ignores it and **does nothing**.

---

4. `else` with loops (unique in Python)

- Python allows `else` with loops.
- The `else` block executes **only if the loop finishes normally** (not interrupted by `break`).

✔ Example:

```python
for i in range(5):
    print(i)
else:
    print("Loop finished without break")
```

➡️ Output:

```kotlin
0
1
2
3
4
Loop finished without break
```

✔ Example with `break`:

```python
for i in range(5):
    if i == 2:
        break
    print(i)
else:
    print("Loop finished without break")
```

➡️ Output:

```
0
1
```

👉 The `else` didn't run because the loop was **stopped by break**.

Functions

A function is a block of code that performs a specific task.
You write a function once → you can use it many times.

Why do we use functions?

- To avoid repeating code
- To break complex programs into smaller parts
- To make code clean and understandable
- To reuse code anytime

How to Create Function:

Python uses **def** keyword:

```python
def function_name():
    # code
```

And call it like this:

```python
function_name()
```

Example 1: Basic Function

```
def greet():
    print("Hello, Welcome!")

greet()
```

Output:

```
Hello, Welcome!
```

TYPES OF FUNCTIONS IN PYTHON

1. Built-in Functions

These are already provided by Python.

Examples:

```python
print()
len()
type()
input()
max()
min()
sum()
```

2. User-defined Function

Created by you.

```python
def welcome():
    print("Welcome to Python!")

welcome()
```

3 Function with Parameters

Parameters = values you pass to the function.

```python
def add(a, b):
    print(a + b)

add(5, 3)
```

Output:

```
8
```

4 Function with Return Value

`return` sends a value back.

```python
def multiply(x, y):
    return x * y

result = multiply(4, 5)
print(result)
```

Output:

```
20
```

You can store the result in a variable.

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Arbitrary Positional Arguments (*args)
5. Arbitrary Keyword Arguments (**kwargs)

And the **order of definition** is very important:

```go
def func(positional, /, positional_or_keyword, *args, keyword_only, **kwargs)
```

Let's go in **detail with variations**:

1. Positional Arguments

- 👉 Positional arguments are the arguments that are passed to a function in the same order as the parameters are defined.
- Python assigns the values to the parameters based on their position (left to right) in the function call.

Example:

```python
def student_info(name, age, city):
    print(f"Name: {name}, Age: {age}, City: {city}")

# Correct order
student_info("Raksha", 22, "Udupi")

# Wrong order (misleading output)
student_info("Udupi", "Raksha", 22)
# Name: Udupi, Age: Raksha, City: 22
```

⚠️ Must match **number & order** exactly.

---

2. Keyword Arguments

- 👉 Keyword arguments are the arguments that are passed to a function by explicitly specifying the parameter name along with its value in the function call.
- Because of this, the order of arguments does not matter Python matches them using the parameter names.

```python
def student_info(name, age, city):
    print(f"Name: {name}, Age: {age}, City: {city}")

student_info(age=22, name="Raksha", city="Udupi")
# Output: Name: Raksha, Age: 22, City: Udupi
```

⚠️ Once you start using keywords, positional arguments **must come first**.

✅ Allowed:

```python
student_info("Raksha", age=22, city="Udupi")
```

❌ Not allowed:

```python
student_info(name="Raksha", 22, "Udupi")  # ERROR
```

---

3. Default Arguments

- 👉 Default arguments are function parameters that are given a default value in the function definition.
- If the caller provides a value → that value is used.
- If the caller does not provide a value → the default value is used automatically.

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

greet("Raksha")          # message takes default → Hello, Raksha!
greet("Raksha", "Hi")    # message overridden → Hi, Raksha!
```

⚠️ **Important Rule**:

👉 All default arguments must come **after non-default arguments**.

❌ Wrong:

```python
def f(a=10, b):  # SyntaxError
    pass
```

✅ Correct:

```python
def f(b, a=10):
    pass
```

Default arguments = parameters with pre-defined values. They must come after all required positional arguments.

---

4. Arbitrary positional arguments (* args)

- 👉 Arbitrary positional arguments allow a function to accept any number of positional arguments (0 or more) without knowing beforehand how many will be passed.
- They are defined using a single asterisk * followed by a variable name, usually args.
- Python collects all extra positional arguments into a tuple.

```python
def demo(a, b=10, *args, c, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)
    print("c:", c)
    print("kwargs:", kwargs)

demo(1, 2, 3, 4, c=99, x=100, y=200)
```

Output:

```yaml
a: 1
b: 2
args: (3, 4)
c: 99
kwargs: {'x': 100, 'y': 200}
```

- `1` → `a` (positional)
- `2` → `b` (default overridden)
- `(3,4)` → collected in `*args`
- `c=99` → keyword-only argument
- `x=100, y=200` → collected in `**kwargs`

5. Arbitrary Keyword Arguments (**kwargs)

- 👉 Arbitrary keyword arguments allow a function to accept any number of keyword arguments (0 or more) without knowing beforehand which keywords will be used.
- They are defined using double asterisks ** followed by a variable name, usually kwargs.
- Python collects all extra keyword arguments into a dictionary (dict), where keys are parameter names and values are the corresponding arguments.

```python
def show_details(**kwargs):
    print(kwargs)

show_details(name="Raksha", age=22, city="Udupi")
# {'name': 'Raksha', 'age': 22, 'city': 'Udupi'}
```

➡️ Can be combined with normal arguments:

```python
def student(name, **details):
    print("Name:", name)
    print("Other Details:", details)

student("Raksha", age=22, city="Udupi", course="CS")
# Name: Raksha
# Other Details: {'age': 22, 'city': 'Udupi', 'course': 'CS'}
```

🔹 **Combining All Types in One Function**

👉 **The correct order of arguments is:**

1. Positional arguments
2. Default arguments
3. *args
4. Keyword-only arguments
5. **kwargs

Example:

```python
def demo(a, b=10, *args, c, d=20, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)
    print("c:", c)
    print("d:", d)
    print("kwargs:", kwargs)

demo(1, 2, 3, 4, 5, c=30, d=40, x=100, y=200)
```

```
a: 1
b: 2
args: (3, 4, 5)
c: 30
d: 40
kwargs: {'x': 100, 'y': 200}
```

8. Lambda Function

A lambda function is a small, anonymous (nameless) function in Python.
Normally, when we create a function, we use def.

Example:

```python
def add(a, b):
    return a + b
```

But if we just need a short one-line function, writing def every time feels too long.
That's where lambda functions are useful.
Lambda functions are one-line functions without a name.

Syntax of Lambda Function:

```
lambda arguments: expression
```

- `lambda` → keyword
- `arguments` → input values (like parameters)
- `expression` → the logic (must be a single expression, not multiple lines)

🔹 Example 1: Add Two Numbers

```python
add = lambda a, b: a + b
print(add(5, 3))   # Output: 8
```

Here:

- `lambda a, b: a + b` → creates a function
- It takes two numbers and returns their sum
- We assign it to the variable `add`, so we can call it like a normal function.

🔹 Example 2: Square of a Number

```python
square = lambda x: x * x
print(square(4))   # Output: 16
```

🔹 Why Use Lambda Functions?

1. **Short and simple** → For quick operations, we don't need to define a full `def` function.
2. **Used with built-in functions** like:
   - `map()` → apply a function to every item in a list
   - `filter()` → filter items from a list
   - `sorted()` → sort using custom logic

Map Function with Lambda Function:

Map () is used to apply a function to every item in a sequence (like list, tuple, etc.) without using a loop.

It returns a map object, which can be converted into a list, tuple, etc.

Syntax:

```
map(function, iterable)
```

Simple Example without Lambda:

```
def square(n):
    return n * n

numbers = [1, 2, 3, 4]
result = map(square, numbers)

print(list(result))
```

Output:

```
csharp

[1, 4, 9, 16]
```

Lambda + Map Function:

Double each number in a list:

```python
numbers = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)   # [2, 4, 6, 8, 10]
```

Filter Function with Lambda Function:
filter() is used to filter items from a list (or any iterable) based on a condition.
It keeps only the elements for which the condition is True.

Syntax:

```
filter(function, iterable)
```

```python
numbers = [1, 2, 3, 4, 5, 6]

result = filter(lambda x: x % 2 == 0, numbers)

print(list(result))
```

Output:

```
[2, 4, 6]
```

Reduce with Lambda Function:
reduce() is used to apply a function to all elements of a sequence and reduce them to a single value.

✔ `reduce()` is NOT built-in

You must import it from **functools**:

```python
from functools import reduce
```

Syntax:

```
reduce(function, iterable)
```

The function must take **two arguments**.

```python
from functools import reduce

nums = [1, 2, 3, 4]

result = reduce(lambda x, y: x + y, nums)

print(result)
```

Output:

```
10
```

Error Handling / Exception Handling

- An **error** is a mistake in your program that stops it from working properly.

Two Main Types of Errors

1. **Syntax Error**

- Happens when you write code incorrectly.
- Python doesn't understand it.
- Example:

```
python

if True
    print("Hello")   # Missing colon
```

Error: `SyntaxError: expected ':'`

2. **Exception (Runtime Error)**

- Code is written correctly, but an error happens **while running** the program.
- Example:

```
python

number = int("abc")  # Can't convert text to number
```

Error: `ValueError: invalid literal for int() with base 10: 'abc'`

Why do we need Exception Handling?

- To prevent the program from crashing unexpectedly.
- To provide meaningful error messages instead of default Python errors.
- To continue program execution even after encountering an error.

Syntax of Exception Handling

```
try:
    # Code that may cause an exception
except ExceptionType:
    # Code to handle the exception
else:
    # Code that runs if no exception occurs (optional)
finally:
    # Code that always runs (optional, cleanup code)
```

Example 1: Basic Exception Handling

```
try:
    x = 10 / 0   # ZeroDivisionError
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

Output:

```
You cannot divide by zero!
```

Example 2: Multiple Exceptions

```
try:
    num = int("abc")   # ValueError
except ValueError:
    print("Invalid number format")
except ZeroDivisionError:
    print("Division by zero is not allowed")
```

Output:

```
Invalid number format
```

Example 3: Using else and finally

```
try:
    num = int("100")
    print(10 / num)
except ZeroDivisionError:
    print("Division by zero")
else:
    print("No error occurred")
finally:
    print("Execution complete")
```

Output:

```
0.1
No error occurred
Execution complete
```

Key Points

- try → Place risky code inside this block.
- except → Handles the error.
- else → Runs only if no error happens.
- finally → Runs no matter what (good for cleanup like closing files or DB connections).

File Paths

- A file path is like the "address" of a file or folder on your computer.
- When you want to open, read, write, or modify a file in Python, you must tell Python where that file is located. That's done using a file path.

Types of File Paths

1. Absolute Path

- Gives the complete location of a file from the root directory (starting point of your system).
- Works anywhere, but usually longer.

👉 Example (Windows):

```
python

"C:\\Users\\Raksha\\Documents\\data.txt"
```

👉 Example (Mac/Linux):

```
python

"/home/raksha/Documents/data.txt"
```

2. Relative Path

- Gives the location relative to the current working directory (CWD).
- Shorter and more convenient if files are in the same project folder.

```
"data.txt"              # file in the same folder
"files/data.txt"        # file inside 'files' folder
"../data.txt"           # file one step up in directory
```

Important Rules:

🔸 1. Backslashes in Windows Paths

In Python, `\` is an escape character (like `\n`, `\t`), so you must:

✔ Use double backslashes

```
python

path = "C:\\Users\\Raksha\\data.txt"
```

✔ OR use raw string (recommended)

```
python

path = r"C:\Users\Raksha\data.txt"
```

✔ OR use forward slashes (also works!)

```
python

path = "C:/Users/Raksha/data.txt"
```

How to get Current Working Directory:

```
import os
print(os.getcwd())
```

This shows where Python is currently working.

File Handling

Link - https://youtu.be/Sx1Hjr67xO0?si=cc8npR3S6svhLR69

- File handling in Python is a way to read from, write to, and manipulate files using Python programs.
- It allows your program to store data permanently in files (like .txt, .csv, .json, etc.) instead of keeping it only in memory, which is temporary.

File handling in Python allows you to read from and write to files. This is important when you want to store data permanently or work with large datasets.

Python provides built-in functions and methods to interact with files.

Why is File Handling Used? – Console Example

File handling is used because:

1. **Data Persistence:** Data in files remains even after the program ends. Example: saving user data, logs, or configurations.
2. **Large Data Storage:** Programs can't always store huge data in memory, so files are used for storage.
3. **Data Sharing:** Files allow sharing data between programs or with other users.
4. **Input/Output Operations:** Programs can read input from files or save output for future use.
5. **Automation and Reporting:** Files are used to generate reports, logs, or save results automatically.

Steps for File Handling in Python:

- Opening a file
- Reading from a file
- Writing to a file
- Closing the file

Open a File:

To perform any operation (read/write) on a file, you first need to open the file using Python's open() function.

Syntax: `file_object = open('filename', 'mode')`

- *'filename'*: Name of the file (can be relative or absolute path).
- *'mode'*: Mode in which the file is opened (read, write, append, etc.).

File Modes:

- 'r': Read (default mode). Opens the file for reading.
- 'w': Write. Opens the file for writing (if file doesn't exist, it creates one).
- 'a': Append. Opens the file for appending (if file doesn't exist, it creates one).
- 'rb'/'wb': Read/Write in binary mode.

Example: Opening a file for reading

```
file = open('example.txt', 'r')
```

Basic Operations we can do on Files:

1. **Open**
2. **Read**
3. **Write**
4. **Close**

- Closing a file is important because when we open, read, write to the file some amount of memory will be used to get free from that we have to close.

Different File Modes in Python:

🔹 1. r → Read Only

- Opens a file **for reading**
- **File must already exist**
- If file does NOT exist → **Error**

Example:

```python
f = open("data.txt", "r")
print(f.read())
f.close()
```

When to use?

- When you only want to **view** data.

🔹 2. w → Write Only

- Opens a file for **writing**
- **Creates** a new file if it doesn't exist
- **Deletes old content** if file exists → **DANGEROUS**

Example:

```python
f = open("data.txt", "w")
f.write("Hello Python!")
f.close()
```

When to use?

- When you want to **overwrite** a file.

🔹 3. a → Append

- Opens a file for **adding new data**
- Creates file if not exists
- **Does NOT delete old data**

Example:

```python
f = open("data.txt", "a")
f.write("\nNew line added")
f.close()
```

When to use?

- When you want to **add** data at the end.

🔹 4. x → Create

- Creates a **new file only**
- If file exists → **Error**

Example:

```python
f = open("newfile.txt", "x")
```

When to use?

- When you want to **force create** a new file.

🔹 5. r+ → Read + Write

- File must exist
- Can **read & write**
- Does NOT delete existing data
- Writing starts from **beginning** (may overwrite characters)

Example:

```python
f = open("data.txt", "r+")
f.write("Hi")
print(f.read())
f.close()
```

When to use?

- When you want to update a file **without deleting old content**.

🔹 6. w+ → Write + Read

- Creates file if not exist
- Deletes file content if exists
- Allows both read + write

Example:

```python
f = open("data.txt", "w+")
f.write("Python")
f.seek(0)
print(f.read())
f.close()
```

When to use?

- When you want to write first, then read.

🔹 7. a+ → Append + Read

- Creates file if not exist
- Does NOT delete old content
- Pointer is at **end** of file

Example:

```python
f = open("data.txt", "a+")
f.write("\nAdded line")
f.seek(0)
print(f.read())
f.close()
```

When to use?

- When you want to append AND read complete data.

Used for **images, videos, PDFs**, etc.

Mode	Meaning
rb	read binary
wb	write binary (clears file)
ab	append binary
rb+	read + write binary
wb+	write + read binary
ab+	append + read binary

Example:

```python
f = open("photo.jpg", "rb")
data = f.read()
f.close()
```

Read from a File

Once a file is open, you can read from it using the following methods:

- read(): Reads the entire content of the file.
- readline(): Reads one line from the file at a time.
- readlines(): Reads all lines into a list.

```
# Example: Reading the entire file
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

```
# Example: Reading one line at a time
file = open('example.txt', 'r')
line = file.readline()
print(line)
file.close()
```

Write to a File

To write to a file, you can use the write() or writelines() method:

- write(): Writes a string to the file.
- writelines(): Writes a list of strings.

```
# Example: Writing to a file (overwrites existing content)
file = open('example.txt', 'w')
file.write("Hello, world!")
file.close()

# Example: Appending to a file (add line to the end)
file = open('example.txt', 'a')
file.write("\nThis is an appended line.")
file.close()

# Close a file:
file.close()
```

---

1. Creating a File:

Method 1:

```
f1=open("file_1.txt","x")
```

- x is used to create a file
- But if file exists then I will use x then I will get file exists error.

Method 2:

```
f1 = open("file_1.txt","r")
```

- With the help of read also we can create a file
- But if file exists it will not give any error

---

2. Reading a File:

- 'r' mode by default it is the open to file

```
f1=open("file_1.txt","r")
data=f1.read()
print(data)
```

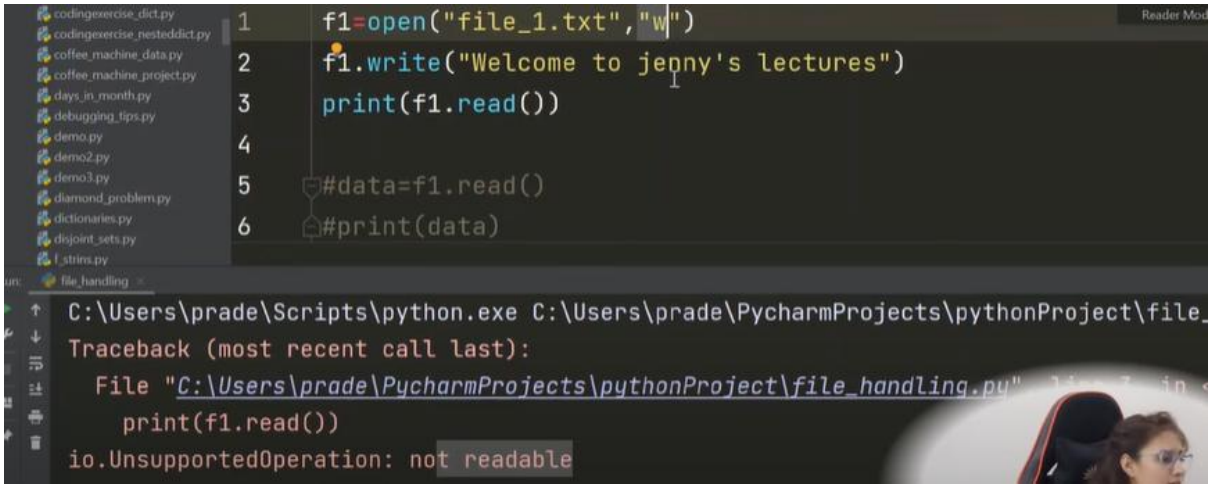
- If the file exists it will read the information
- But if it doesn't exists it will give an error

3. Writing a File:

```
f1=open("file_1.txt","w")
```

To Write Something:

```
f1=open("file_1.txt","w")
f1.write("Welcome to jenny's lectures")
```


codingexercise_dict.py
codingexercise_nesteddict.py
coffee_machine_data.py
coffee_machine_project.py
days_in_month.py
debugging_tips.py
demo.py
demo2.py
demo3.py
diamond_problem.py
dictionaries.py
disjoint_sets.py
f_strins.py
file_handling
1 f1=open("file_1.txt","w")
2 f1.write("Welcome to jenny's lectures")
3 print(f1.read())
4
5 #data=f1.read()
6 #print(data)
C:\Users\prade\Scripts\python.exe C:\Users\prade\PycharmProjects\pythonProject\file_
Traceback (most recent call last):
File "C:\Users\prade\PycharmProjects\pythonProject\file_handling.py"
print(f1.read())
io.UnsupportedOperation: not readable

Working With Modules in Python

A module in Python is simply a file that contains Python code—functions, variables, or classes—that you can reuse in your program.
Think of a module as a **toolbox**:
- You don't need to write everything from scratch.
- You can import ready tools and use them.

Why Do We Use Modules?

1. Reuse code
2. Reduce complexity
3. Organize code
4. Use Python's built-in functionalities (math, random, datetime, statistics, etc.)

Types of Modules:

Python provides:

✔ **1. Built-in modules (already available)**

Examples:

- math
- random
- datetime
- statistics
- os
- sys

✔ **2. User-defined modules**

Modules created by you (your own .py files).

Import Module (3 Main Ways):

🔹 1. import module_name

This imports the entire module.

Example:

```python
import math
print(math.sqrt(25))
print(math.pi)
```

✔ You must use `module_name.function()`
❗ Simple and beginner-friendly.

🔹 2. from module_name import function_name

Imports only the selected function(s).

Example:

```python
from math import sqrt, pi
print(sqrt(25))
print(pi)
```

✔ No need to write `math.`
❗ But you can use only the imported functions.

🔹 3. from module_name import * (import all)

Imports **everything** from the module.

Example:

```python
from math import *
print(sqrt(16))
print(pi)
```

✔ Short and fast

❗ Not recommended in big programs (can cause confusion)

Common Built in Modules to Know:

🔹 1. math module (mathematical operations)

```python
import math

print(math.sqrt(36))
print(math.ceil(4.3))
print(math.floor(4.9))
print(math.pow(2, 3))
print(math.pi)
```

🔹 2. random module (generating random values)

```python
import random

print(random.randint(1, 10))     # random integer
print(random.random())           # 0 to 1
print(random.choice([10,20,30]))
```

🔹 3. datetime module (dates & time)

```python
from datetime import datetime

now = datetime.now()
print(now)
print(now.year)
print(now.strftime("%d-%m-%Y"))
```

🔹 4. statistics module (useful for Data Analysis)

```python
import statistics

data = [10, 20, 30, 40]
print(statistics.mean(data))
print(statistics.median(data))
```

🔹 5. os module (file & directory operations)

```python
import os

print(os.getcwd())              # current working directory
print(os.listdir())             # list files
```

User Defined Modules:

You can create your own module.

Step 1: Create a file `mymodule.py`

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

Step 2: Import it

```python
import mymodule
print(mymodule.greet("Raksha"))
```

Using Module Aliases

```
import numpy as np
import pandas as pd
import math as m

print(m.sqrt(49))
```

Important Built-In Exception Types

Below are the most common exceptions with easy examples.

1. NameError

When you use a variable that is **not defined**.

```python
print(x)
```

Output:

```pgsql
NameError: name 'x' is not defined
```

2. TypeError

When an operation is done on **wrong data types**.

```python
print("Hello" + 5)
```

Output:

```python
TypeError: can only concatenate str (not "int") to str
```

3. ValueError

Correct type but **wrong value**.

```python
int("abc")
```

Output:

```csharp
ValueError: invalid literal for int()
```

4. IndexError

Index is **out of range** in a list, tuple, string, etc.

```python
numbers = [1, 2, 3]
print(numbers[5])
```

Output:

```pgsql
IndexError: list index out of range
```

5. KeyError

Trying to access a dictionary key that **does not exist**.

```python
data = {"name": "Raksha"}
print(data["age"])
```

Output:

```vbnet
KeyError: 'age'
```

6. ZeroDivisionError

Dividing by zero.

```python
print(10 / 0)
```

Output:

```vbnet
ZeroDivisionError: division by zero
```

7. AttributeError

When you try to use a function/attribute that the object **does not have**.

```python
name = "Raksha"
print(name.append("S"))
```

Output:

```pgsql
AttributeError: 'str' object has no attribute 'append'
```

8. ModuleNotFoundError

When Python cannot find the module you are importing.

```python
import maths
```

Output:

```vbnet
ModuleNotFoundError: No module named 'maths'
```

9. FileNotFoundError

When you try to open a non-existing file.

```python
open("abc.txt", "r")
```

Output:

```vbnet
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

🔶 10. IndentationError

When the indentation (spaces) is wrong.

```python
def hello():
print("Hi")
```

Output:

```makefile
IndentationError: expected an indented block
```

🔶 11. ImportError

When a specific part of a module cannot be imported.

```python
from math import square
```

Output:

```pgsql
ImportError: cannot import name 'square'
```

🔶 12. RuntimeError

General error that occurs during run-time.

Example:

```python
raise RuntimeError("Something went wrong")
```

Comments

Comments are notes written in the code to make it easier to understand.
Python ignores comments completely
Used for explanation, documentation, and readability
Does NOT affect program output

🔵 Types of Comments

1️⃣ Single-line Comment

Use the **# symbol**.

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

You can also place it at the end:

```python
print("Hello")  # printing message
```

2️⃣ Multi-line Comment

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

```python
"""
This is a multi-line comment.
Used for longer explanations.
"""
print("Welcome")
```

- ✔ Good for documentation
- ✔ Python treats the content as a string literal, so it is ignored unless assigned to a variable

Indentation

Indentation = spaces at the beginning of a line.
Python uses indentation to define blocks of code
Unlike other languages (C, Java), Python does not use curly braces { }
Default indentation = 4 spaces
Mixing tabs and spaces is an error

Example of Correct Indentation

```
if True:
    print("Hello")      # indented
    print("Python")     # same block
```

Example of Incorrect Indentation:

```
if True:
print("Hello")
```

❌ Output:

```
makefile

IndentationError: expected an indented block
```

Indentation is required for:

- if, elif, else
- for loop
- while loop
- functions
- classes
- try...except
- with statement