Python

Python is a programming language which has a simple and easy syntax and is used to perform different repetitive tasks.

Python concepts:

1. Variables

Definition:

Variables are containers that store data values. Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Creating a Variable:

Python has no command for declaring a variable.A variable is created the moment you first assign a value to it.

Rules for Variables:

- Must start with a letter or underscore `_`
- Can contain letters, digits, and underscores
- Are case-sensitive (`Name` and `name` are different)

Example:

```
name = "Atiqa"      # string variable

age = 21            # integer variable

height = 5.4        # float variable

is_student = True   # boolean variable

print(name, age, height, is_student)
```

a. Data Types (of variables)

Data types define what kind of value a variable holds and what operations can be performed on it. Python has several built-in data types.

Main Data Types in Python:

1. **int** → Whole numbers (e.g., 5, -3, 100)
2. **float** → Decimal numbers (e.g., 3.14, 2.5)
3. **str (string)** → Sequence of characters (e.g., "Hello")
4. **bool** → Logical values (`True` or `False`)
5. **list** → Ordered, changeable collection (e.g., `[1, 2, 3]`)
6. **tuple** → Ordered, unchangeable collection (e.g., `(1, 2, 3)`)
7. **set** → Unordered, unique collection (e.g., `{1, 2, 3}`)
8. **dict (dictionary)** → Key-value pairs (e.g., `{"name": "Atiqa", "age": 21}`)

Examples:

```
x = 10              # int
y = 3.5             # float
name = "Python"     # str
flag = False        # bool
nums = [1, 2, 3]    # list
tup = (4, 5, 6)     # tuple
unique = {7, 8, 9}  # set
info = {"name": "Atiqa", "age": 21}  # dict

print(type(info))   # Output: <class 'dict'>
```

b.Type Casting

Type casting means converting one data type into another — for example, turning a string into an integer or a float into an integer. It's helpful when working with mixed data types or user input.

Type Casting Functions:

- `int()` → converts to integer

- `float()` → converts to float
- `str()` → converts to string
- `list()` → converts to list
- `tuple()` → converts to tuple
- `set()` → converts to set

# Example 1:

```
Float to Integer
x = 3.7
y = int(x)
print(y)   # Output: 3

# Example 2: String to Integer
num_str = "25"
num_int = int(num_str)
print(num_int + 5)   # Output: 30

# Example 3: List to Set
nums = [1, 2, 2, 3]
unique_nums = set(nums)
print(unique_nums)   # Output: {1, 2, 3}
```

2. Operations on Data Types

Numeric Operations:
You can perform mathematical operations like addition, subtraction, multiplication, and division on numeric types.

```
a = 10
b = 3
print(a + b)  # 13
print(a - b)  # 7
print(a * b)  # 30
print(a / b)  # 3.333...
```

String Operations:
You can concate, slice and repeat strings easily.

Example:

```
name = "Atiqa"
greet = "Hello " + name
print(greet)        # Hello Atiqa
print(name * 3)     # AtiqaAtiqaAtiqa
```

3.Strings in Python

A **string** in Python is a sequence of **characters** enclosed in single quotes (`'  '`), double quotes (`"  "`), or triple quotes (`'''  '''` or `"""  """`).
It is used to store and manipulate text — like names, sentences, or messages.

Example:

```
name = "Atiqa"
greeting = 'Hello World'
paragraph = """This is a multi-line string in Python."""
```

Strings are one of the most important data types because almost every program involves working with text — reading names, printing messages, processing data, etc.

1. Creating Strings

You can create strings in different ways:

```
a = 'Hello'
b = "Python"
c = '''This is
a multi-line
string.'''
```

2. Accessing Characters in a String

Each character in a string has an **index** (position number).
Indexing starts from **0** (zero).

```
name = "Python"
print(name[0])   # P
print(name[3])   # h
```

You can also use **negative indexing** to start from the end.

```
print(name[-1])  # n

print(name[-2])  # o
```

3. String Slicing

You can extract part of a string (called a *substring*) using **slicing**.

```
text = "MachineLearning"
print(text[0:7])   # Machine
print(text[7:])    # Learning
print(text[:7])    # Machine
```

You can even reverse a string using slicing:

```
print(text[::-1])  # gninraeLeniM
```

4. String Concatenation and Repetition

a. Concatenation (Joining Strings)

```
first = "Hello"
second = "Atiqa"
print(first + " " + second)
```

Output:

```
Hello Atiqa
```

b. Repetition

```
word = "Hi! "
print(word * 3)
```

Output:

```
Hi! Hi! Hi!
```

5. String Functions and Methods

Python provides many **built-in string methods** to work with text easily.

Function	Description	Example	Output
`len()`	Returns length of string	`len("Python")`	`6`
`upper()`	Converts to uppercase	`"hello".upper()`	`HELLO`
`lower()`	Converts to lowercase	`"HELLO".lower()`	`hello`
`title()`	Capitalizes each word	`"python language".title()`	`Python Language`
`strip()`	Removes spaces from both sides	`" Hello ".strip()`	`Hello`
`replace(a, b)`	Replaces text	`"I like Java".replace("Java", "Python")`	`I like Python`
`split()`	Splits text into list	`"A,B,C".split(",")`	`['A', 'B', 'C']`
`find()`	Finds position of substring	`"Machine".find("a")`	`1`

6. String Formatting

You can insert variables directly into strings using **f-strings** or the `format()` method.

Using f-strings (modern way):

```
name = "Atiqa"
age = 21
print(f"My name is {name} and I am {age} years old.")
```

Using `format()` method:

```
print("My name is {} and I am {} years old.".format("Atiqa", 21))
```

Output:

```
My name is Atiqa and I am 21 years old.
```

7. Checking Membership

You can check whether a substring exists in another string using `in` or `not in`.

```
text = "Python is amazing"
print("Python" in text)       # True
print("Java" not in text)     # True
```

8. Iterating Through a String

Strings can be looped through using a `for` loop (since they’re sequences).

```
for char in "Data":
    print(char)
```

Output:

```
D
a
t
a
```

Example:

Here’s a simple example combining the above concepts:

```
name = input("Enter your name: ")
print(f"Hello, {name}!")

# Reverse the string
print("Reversed name:", name[::-1])

# Check length
print("Your name has", len(name), "characters.")

# Uppercase and replace
updated = name.upper().replace("A", "@")
print("Updated version:", updated)
```

Output:

```
Enter your name: Atiqa
Hello, Atiqa!
Reversed name: aqitA
```

```
Your name has 5 characters.
Updated version: @TIQ@
```

4. Data Structures in Python (Lists, Tuples, Sets, Dictionaries)

Data structures are special containers in Python used to store, organize, and manage data efficiently.
Each type — **List**, **Tuple**, **Set**, and **Dictionary** — has its own unique properties and use cases.

A. Lists

A **List** is an **ordered, mutable (changeable)** collection that allows duplicate elements.
It's one of the most commonly used structures for storing a group of related items.

Example:

```
fruits = ["apple", "banana", "mango", "banana"]

print(fruits[1])    # Output: banana
```

Common List Functions:

- `append(item)` → adds item to the end of list
- `insert(index, item)` → adds item at a specific position
- `remove(item)` → removes first matching item
- `pop(index)` → removes element by index
- `sort()` → sorts list in ascending order
- `reverse()` → reverses order of elements
- `len(list)` → returns number of elements

Example:

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

numbers.append(5)
numbers.sort()

numbers.reverse()  # Output: [5,4,3,2,1]



print(numbers)   # Output: [1, 2, 3, 4, 5]
```

B. Tuples

A **Tuple** is an **ordered, immutable (unchangeable)** collection.
Once created, you cannot modify or remove its elements. Tuples are used when you want data to remain constant.

Example:

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

print(colors[0])   # Output: red
```

Common Tuple Functions:

- `count(value)` → returns how many times a value appears
- `index(value)` → returns the index of first occurrence
- `len(tuple)` → returns number of items

```
nums = (1, 2, 2, 3)
```

- print(nums.count(2))   # Output: 2
- print(nums.index(2))   # Output: 1
- print(len(nums))   # Output: 4

C. Sets

A **Set** is an **unordered collection** of **unique elements**.
It automatically removes duplicates and is useful for mathematical operations like union or intersection.

Example:

```
unique_nums = {1, 2, 2, 3}

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

Common Set Functions:

- `add(item)` → adds an element
- `remove(item)` → removes an element (gives error if not found)
- `discard(item)` → removes element safely (no error)
- `union(other_set)` → combines two sets
- `intersection(other_set)` → common elements between sets
- `difference(other_set)` → elements only in one set

Example:

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a.union(b))         # {1, 2, 3, 4, 5}
print(a.intersection(b))  # {3}
```

D. Dictionaries

A **Dictionary** stores data in **key–value pairs**.
Each key must be unique, and you can quickly retrieve data by key rather than index.

Example:

```
student = {"name": "Atiqa", "age": 21, "grade": "A"}
print(student["name"])   # Output: Atiqa
```

Common Dictionary Functions:

- `keys()` → returns all keys
- `values()` → returns all values
- `items()` → returns key-value pairs

- `get(key)` → returns value safely
- `update(dict)` → updates with another dictionary
- `pop(key)` → removes item by key
- `clear()` → removes all items

Example:

```
student = {"name": "Atiqa", "age": 21}

student["grade"] = "A"

student.update({"city": "Lahore"})

print(student.keys())     # dict_keys(['name', 'age', 'grade', 'city'])
```

5. Conditional Statements (if, elif, else)

1. Definition

Conditional statements in Python allow the program to make decisions based on certain conditions.
They check whether a condition is **True** or **False**, and then execute specific blocks of code accordingly.
It helps the program behave intelligently — for example, showing different messages for different inputs or situations.

Structure:

The basic syntax looks like th

```
if condition:
   # code runs if condition is True
elif another_condition:
   # code runs if previous condition was False but this is True
else:
   # code runs if all above conditions are False
```

Example 1 – Grade Calculator:

```
marks = 78
```

```
if marks >= 90:
   print("Grade: A+")
elif marks >= 75:
   print("Grade: A")
elif marks >= 60:
   print("Grade: B")
else:
   print("Grade: C")
```

6. Loops in Python

Loops in Python are used to **repeat a block of code multiple times** until a certain condition is met.
They make programs efficient by avoiding the need to write the same code again and again.

Types of Loops in Python:

Python mainly has **two types of loops**:

1. **for loop** — used when you know *how many times* you want to repeat something.

2. **while loop** — used when you want to repeat until a *condition becomes false*.

1. `for` Loop

The `for` loop is commonly used for **iterating over a sequence** — like a list, string, or range of numbers.

Example 1: Using range()

```
for i in range(5):
    print("Hello Atiqa!")
```

Output:

```
Hello Atiqa!
Hello Atiqa!
Hello Atiqa!
Hello Atiqa!
Hello Atiqa!
```

Here, the loop runs **5 times**, starting from 0 to 4.

Example 2: Looping through a List

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

Output:

```
apple
banana
mango
```

Example 3: Looping through a String

```
for char in "Python":
    print(char)
```

Output:

```
P
y
t
h
o
N
```

2. `while` Loop

The `while` loop keeps running **as long as a condition is true.**

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

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
```

```
Count: 5
```

If you forget to update the variable (`i += 1`), the loop runs forever (infinite loop).

a. Loop Control Statements

Python gives you some keywords to control how loops behave.

1. break

Stop the loop immediately.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Output:

```
0
1
2
3
4
```

2. continue

Skips the current iteration and moves to the next.

```
for i in range(6):
    if i == 3:
        continue
    print(i)
```

Output:

```
0
1
2
4
5
```

Here, when `i == 3`, the loop skips printing that number.

3. pass

Used as a placeholder when you want to leave the loop empty for now.

```
for i in range(3):
    pass  # does nothing
```

This prevents syntax errors if you haven't written the body yet.

Nested Loops:

You can place one loop inside another — called a **nested loop**.

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

7.Functions in Python

A **function** is a reusable block of code that performs a specific task.
Functions make programs cleaner, reduce repetition, and make code easier to test and maintain.
You can call a function anytime by its name instead of writing the same logic again and again.

Creating a Function:

Syntax:

```
def function_name(parameters):

   # code block

   return result
```

`def` → keyword used to define a function

`function_name` → name of the function

`parameters` → data or inputs passed into the function

`return` → sends the result back to where the function was called

1.Function Without Parameters:

The function doesn’t take any arguments — it just runs when called.

Example:

```
def greet():
    print("Hello, welcome to Python programming!")
```

2.Function With Parameters:

The function takes two arguments and returns their result after operations

Example:.

```
def add_numbers(a, b):
    result = a + b
    return result


sum = add_numbers(5, 3)
print(f"The sum is: {sum}")
```

3.Function With Default Parameters

Sometimes you can give parameters a **default value**.
If no argument is passed, the default value will be used.

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


greet()          # uses default value
greet("Harry")   # overrides default
```

4. Lambda (Anonymous Function)

A **lambda function** is a small, one-line function without a name.
It’s mostly used for short tasks or inside other functions.

Example:

```
square = lambda x: x * x

print(square(5))
```

8.Object-Oriented Programming (OOP) in Python:

Object-Oriented Programming (OOP) is a way of writing programs by organizing data and behavior into **objects**.

Instead of writing long code blocks, we create **classes** (blueprints) and **objects** (real examples) that represent real-world things.

Example:
A *Car* can be a **class**, and *Honda Civic* or *Toyota Corolla* can be **objects** of that class.

1.Class

A **class** is a blueprint that defines the structure and behavior of objects.

```
class Car:

    def __init__(self, brand, model):

        self.brand = brand

        self.model = model


    def show_details(self):

        print(f"Brand: {self.brand}, Model: {self.model}")
```

2. Object

An **object** is an instance of a class — it's like a real version of that blueprint.

```
car1 = Car("Toyota", "Corolla")

car2 = Car("Honda", "Civic")


car1.show_details()

car2.show_details()
```

`3.__init__()` Method

The `__init__()` method is a special function that automatically runs when an object is created.
It initializes the object's variables.

4. Self Keyword

The **self** keyword refers to the current object — it helps access variables and methods inside the class.

Four Main Pillars of OOP

There are four main concepts in OOP which are called four pillars of oop.These are as follows:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

1. Encapsulation

Encapsulation means **hiding the internal details** of an object and showing only what's necessary.
It protects data using private variables.

```
class Student:

    def __init__(self, name, age):

        self.__name = name  # private variable

        self.__age = age



    def display(self)

 print(f"Name: {self.__name}, Age: {self.__age}")
```

Output:

```
Name: Atiqa, Age: 21
```

Encapsulation makes data secure and prevents accidental changes.

2. Inheritance

Inheritance allows one class to **get features from another class**.
It helps reuse code.

```
class Animal:
    def speak(self):
        print("Animal speaks")


class Dog(Animal):
    def bark(self):
        print("Dog barks")
obj = Dog()
obj.speak()
obj.bark()
```

Output:

```
Animal speaks
Dog barks
```

3. Polymorphism

Polymorphism means “many forms.”
It allows methods to have the same name but behave differently depending on the object.

```
class Bird:
    def sound(self):
        print("Bird chirps")
class Cat:
    def sound(self):
        print("Cat meows")
for animal in (Bird(), Cat()):
    animal.sound()
```

Output:

```
Bird chirps
Cat meows
```

4. Abstraction

Abstraction hides **complex details** and shows only the **essential part** to the user.

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Circle(Shape):
    def area(self):
        print("Area of circle = πr²")
c = Circle()
c.area()
```

Output:

```
Area of circle = πr²
```

Abstraction helps simplify large programs.

Here is comparison of oop pillars:

Sr. no	**OOP Pillar**	**Definition**	**Main Purpose**	**Example Concept**	**Keyword / Module Used**	**Real-Life Example**
1	**Encapsulation**	Binding data and methods inside a class and restricting direct access.	To protect data and control access.	Private variables, getters/setters.	Using `__` (double underscore) for private attributes.	A mobile phone hides its inner circuits; you only use the screen.
2	**Inheritance**	One class inherits attributes and methods from another.	To promote code reuse and reduce redundancy.	Parent and child classes.	`class Child(Parent)`	A car class inherited from a vehicle class.
3	**Polymorphism**	One method name behaves differently for different objects.	To allow flexibility and code reusability.	Method overriding and overloading.	Same method name in different classes.	“Sound()” function works differently for dogs, cats, and birds.
4	**Abstraction**	Hiding complex details and showing only the necessary parts.	To make code simple and user-friendly.	Abstract classes and methods.	`abc` module, `@abstractmethod`.	Driving a car—you don’t see the engine, only the steering and pedals.

9.File Handling in Python

File handling in Python allows us to **create, read, write, and delete files**.
It helps store data permanently (unlike variables, which are temporary and lost when the program ends).
Using file handling, we can manage text files, logs, or even datasets.

1. Why Is File Handling Important??

It's used when we need to:

- Save program output for future use
- Read existing data from files
- Process text or CSV files for data analysis
- Store logs or configurations
  Example: Saving user details, writing reports, or reading datasets.

2. Opening a File

To handle files, Python uses the `open()` function:

`file = open("filename.txt", "mode")`**Common Modes:**

Mode	Meaning	Description
`"r"`	Read	Opens file for reading (error if not found)
`"w"`	Write	Creates a new file or overwrites existing
`"a"`	Append	Adds data to the end of the file
`"r+"`	Read & Write	Reads and writes in the same file

`"x"`	Create	Creates a new file (error if exists)

3. Reading Files

You can read data in three main ways:

```
file = open("data.txt", "r")


print(file.read())      # reads entire file

# or

print(file.readline())  # reads one line

# or

print(file.readlines()) # reads all lines as a list


file.close()
```

Example File (data.txt):

```
Learning file handling
```

Output:

```
Learning file handling
```

4. Writing to Files

You can create a new file or overwrite an existing one using `"w"` mode.

```
file = open("data.txt", "w")

file.write("Hello, this is Atiqa.\n")

file.write("I'm learning File Handling in Python.")
```

```python
file.close()
```

This will create a file named `data.txt` with the written text.

5. Appending to Files

Appending adds new content **without deleting old data**.

```python
file = open("data.txt", "a")
file.write("\nThis line was added later.")
file.close()
```

6. Using `with` Statement

Instead of manually opening and closing files, Python provides a safer way using `with`.
It automatically closes the file after execution (even if errors occur).

```python
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

7. Checking File Existence

Before performing operations, check if the file exists using the `os` module.

```python
import os
if os.path.exists("data.txt"):
    print("File exists!")
else:
    print("File not found!")
```

8. Deleting Files

You can delete files using the same `os` module.

```
import os

os.remove("data.txt").
```

Example :

Here’s a small full example combining everything:

```
# Writing to a file
with open("notes.txt", "w") as file:
    file.write("Python File Handling Example\n")
    file.write("This file is created by Atiqa.\n")
# Reading file content
with open("notes.txt", "r") as file:
    print("File content:")
print(file.read())


# Appending new content
with open("notes.txt", "a") as file:
    file.write("File updated successfully.\n")
```

10.Exception Handling in Python

Exception Handling in Python is a way to **handle runtime errors** — the kind that stop your program from running.
Instead of the program crashing, Python lets you **catch** these errors and **respond** to them gracefully.

Think of it like this:
If your code makes a mistake, instead of showing a red error message, you can tell Python what to do next (like “show a friendly message” or “try again”).

1. Why Do We Need Exception Handling??

Without exception handling:

- Program stops when an error occurs
- Users see confusing error messages
- Data might be lost

With exception handling:

- Code continues running smoothly
- You can show user-friendly messages
- It makes your program reliable and professional

2. The `try-except` Block

This is the most common way to handle errors.

```
try:

    x = int(input("Enter a number: "))

    result = 10 / x

    print("Result:", result)

except:

    print("Something went wrong!")
```

If you enter `0` or a wrong value (like a letter), instead of crashing, it prints:

```
Something went wrong!
```

3. Handling Specific Exceptions

You can handle **different types of errors** separately.
This helps you understand what exactly went wrong.

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ValueError:
    print("Please enter a valid number!")
except ZeroDivisionError:
    print("Division by zero is not allowed!")
except Exception as e:
    print("Unexpected error:", e)
```

Example Outputs:

```
Enter a number: a
Please enter a valid number!
Enter a number: 0
Division by zero is not allowed!
```

4. The `else` Block

If no error happens, the `else` block runs.

```
try:
    num = int(input("Enter a number: "))
    print("Square:", num * num)
except ValueError:
    print("Invalid input!")
else:
    print("No error occurred.")
```

5. The `finally` Block

The `finally` block always runs — no matter what.
It's used for cleanup (like closing a file or ending a connection).

```
try:

    file = open("data.txt", "r")

    content = file.read()

except FileNotFoundError:

    print("File not found!")

finally:

    print("File handling complete.")
```

Even if there's an error, "File handling complete." will still print.

6. The `raise` Keyword

You can **manually raise** an exception using `raise`.
It's useful when you want to stop execution on purpose if something goes wrong.

```
age = int(input("Enter your age: "))

if age < 18:

    raise ValueError("You must be 18 or older.")

else:

    print("Access granted.")
```

Output:

```
Enter your age: 15

ValueError: You must be 18 or older.
```

7. Nested Try-Except Example

You can also put one try-except block inside another.

```
try:
    num = int(input("Enter number: "))
    try:
        print(10 / num)
    except ZeroDivisionError:
        print("Inner block: Division by zero error.")
except ValueError:
    print("Outer block: Invalid input.")
```