

Python Core Concepts

Compiled by

Mohit Rathore

For practice and revision

January 2026

▼ Control and flow statements

▼ Conditional statements

▼ Syntax

```
if (condition):
```

```
    work
```

```
elif (condition):
```

```
    work
```

```
else:
```

```
    work
```

▼ Loops

▼ While

```
while condition:
```

```
    # code block
```

▼ for

```
for variable in sequence:
```

```
    # code block
```

▼ Loop control statements

Statement	Stops loop?	Skips iteration?	Does nothing?
<code>break</code>	✓ Yes	✗ No	✗ No
<code>continue</code>	✗ No	✓ Yes	✗ No
<code>pass</code>	✗ No	✗ No	✓ Yes

▼ Methods

▼ String

Method	Use
<code>upper()</code>	Converts to uppercase
<code>lower()</code>	Converts to lowercase
<code>title()</code>	Capitalizes first letter of each word
<code>strip()</code>	Removes spaces from both ends
<code>replace(a, b)</code>	Replaces <code>a</code> with <code>b</code>
<code>find()</code>	Returns index of character
<code>count()</code>	Counts occurrences
<code>split()</code>	Splits string into list
<code>capitalize.()</code>	Capitalizes first word only
<code>swapcase.()</code>	swaps upper to lower and vice versa
<code>join()</code>	Joins list of strings into one
<code>startswith()</code>	Checks if the string starts with passed substring
<code>endswith()</code>	Checks if the string ends with passed substring
<code>isalpha()</code>	Checks if all characters are alphabets
<code>isdigit()</code>	Checks if all characters are digits
<code>isalnum()</code>	Checks if it has both alphabets and digits

▼ List

Method	Use
<code>append()</code>	Adds one item
<code>extend()</code>	Adds multiple items
<code>insert()</code>	Adds at specific index
<code>remove()</code>	Removes by value
<code>pop()</code>	Removes by index
<code>sort()</code>	Sorts list
<code>reverse()</code>	Reverses list
<code>clear()</code>	Removes all items
<code>copy()</code>	Creates a copy
<code>index()</code>	Returns first index occurrence of passed value

Method	Use
<code>count()</code>	Counts the number of occurrence of passed value
<code>min()</code>	Finds minimum value
<code>max()</code>	Finds maximum value

▼ List comprehension

▼ Syntax

```
[expression for item in iterable]
```

▼ Example

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

▼ Tuple

Method / Function	Type	Purpose / Use	Example	Output
<code>count(x)</code>	Tuple method	Counts how many times <code>x</code> appears in the tuple	<code>(1,2,2,3).count(2)</code>	<code>2</code>
<code>index(x)</code>	Tuple method	Returns index of first occurrence of <code>x</code>	<code>(10,20,30).index(20)</code>	<code>1</code>
<code>len(t)</code>	Built-in function	Returns number of elements	<code>len((1,2,3))</code>	<code>3</code>
<code>max(t)</code>	Built-in function	Returns largest element	<code>max((2,5,1))</code>	<code>5</code>
<code>min(t)</code>	Built-in function	Returns smallest element	<code>min((2,5,1))</code>	<code>1</code>
<code>sum(t)</code>	Built-in function	Returns sum of elements	<code>sum((1,2,3))</code>	<code>6</code>
<code>sorted(t)</code>	Built-in function	Returns sorted list	<code>sorted((3,1,2))</code>	<code>[1,2,3]</code>

Method / Function	Type	Purpose / Use	Example	Output
<code>tuple(iterable)</code>	Constructor	Converts iterable into tuple	<code>tuple([1,2,3])</code>	<code>(1,2,3)</code>

▼ Dictionary

Method	Use
<code>keys()</code>	Returns keys
<code>values()</code>	Returns values
<code>items()</code>	Key-value pairs
<code>get()</code>	Safe access
<code>update()</code>	Adds/updates
<code>pop()</code>	Removes key
<code>popitem()</code>	Removes and returns last inserted key-value pair

▼ Set

Method	Use
<code>add()</code>	Adds element
<code>remove()</code>	Removes element
<code>union()</code>	Combines sets
<code>intersection()</code>	Common elements

▼ Functions

▼ Syntax

```
def function_name(parameters):
    # function body
    return value
```

▼ Functions with parameters

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

```
add(5, 3)
```

▼ Types of arguments

▼ Positional

```
def function_name(param1, param2):  
    # code  
function_name(arg1, arg2)
```

Here:

- `arg1` → goes to `param1`
- `arg2` → goes to `param2`

▼ Keyword

```
def info(name, age):  
    print(name, age)  
info(age=20, name="Rahul")
```

▼ Default

```
def function_name(param=value):  
    # code
```

▼ Decorator

```
def my_decorator(func):  
    def wrapper():  
        print("Before function")  
        func()  
        print("After function")  
    return wrapper  
  
@my_decorator  
def greet():  
    print("Hi!")
```

```
greet()
```

output:-

```
Before function
Hi!
After function
```

▼ Generator

A **generator** is a special kind of function that **returns values one at a time**, instead of all at once. It uses the keyword `yield` instead of `return`.

Unlike lists, generators **do not store all values in memory**.

▼ Example

```
def count_up(n):
    for i in range(n):
        yield i
```

Generator vs Normal Function

Feature	Normal Function	Generator
Keyword	<code>return</code>	<code>yield</code>
Memory	High	Low
Execution	Runs fully	Pauses & resumes
Output	One value	Multiple values

▼ Lambda Function

A **lambda function** is a **small, anonymous (nameless) function** written in **one single line**.

▼ Syntax

```
lambda arguments : expression condition
```

▼ Example

```
add = lambda a, b: a + b
print(add(3, 4)) # 7
```

▼ OOP's

▼ Creating a class

```
class Student:
    def study(self, hours):
        self.hours = hours
        print("Student is studying")
```

▼ Creating an object

```
s1 = Student()
s1.study(6)
```

▼ Constructor

▼ Syntax

```
class ClassName:
    def __init__(self, parameters):
        # initialization code
```

▼ Example

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

s1 = Student("Rahul", 20)
print(s1.name) # Rahul
print(s1.age) # 20
```

▼ Types

▼ Default

```
class Demo:
    def __init__(self):
        print("Constructor called")

d = Demo()
```

▼ Parameterized

```
class Car:
    def __init__(self, brand):
        self.brand = brand

c = Car("Tesla")
```

▼ With Default value

```
class ClassName:
    def __init__(self, param=value):
        self.param = param
```

▼ Polymorphism

In Object-Oriented Programming, it allows **the same function, method, or operator to behave differently** depending on the object or data it is acting upon.

```
class Animal:
    def sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def sound(self):
        print("Dog barks")

obj = Dog()
obj.sound()
```

▼ Encapsulation

Encapsulation is the process of **binding data (variables) and methods (functions) together into a single unit (class)** and **restricting direct access to data** to protect it from misuse.

▼ Public

```
class Student:
    def __init__(self, name):
        self.name = name # public

obj = Student("Alex")
print(obj.name)
```

▼ Protected

```
class Student:
    def __init__(self, marks):
        self._marks = marks # protected

class Result(Student):
    def show(self):
        print(self._marks)

obj = Result(90)
obj.show()
```

▼ Private

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance # private

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        self.__balance += amount

obj = BankAccount(1000)
```

```
print(obj.get_balance())
obj.deposit(500)
print(obj.get_balance())
```

▼ Inheritance

Inheritance is an OOP concept where a **child class** acquires the **properties (variables) and behaviors (methods)** of a **parent class**.

```
class Parent:
    def show(self):
        print("This is parent class")

class Child(Parent):
    def display(self):
        print("This is child class")

obj = Child()
obj.show()    # inherited method
obj.display() # child method
```

▼ Abstraction

Abstraction is an OOP concept that **hides internal implementation details** and shows **only essential features** of an object to the user.

Abstraction in Python

Python achieves abstraction using:

1. **Abstract Classes**
2. **Abstract Methods**

This is done using the `abc` module.

▼ Abstract Class

```
from abc import ABC, abstractmethod

class ClassName(ABC):
    pass
```

▼ Abstract Method

```
@abstractmethod
def method_name(self):
    pass
```

▼ Example

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass
```

Child class

```
class Rectangle(Shape):
    def area(self):
        return 10 * 5
```

Usage

```
obj = Rectangle()
print(obj.area())
```

✓ Works

✗ Shape()

→ Error (cannot instantiate abstract class)

▼ Exception Handling

▼ Basic Structure

```
try:
    # risky code (may cause error)
```

```
except:
    # runs if error occurs
```

▼ Example

```
try:
    a = int("abc")
    b = 10 / 0
except ValueError:
    print("Invalid value")
except ZeroDivisionError:
    print("Division error")
```

▼ Common exceptions

Exception	When it happens
<code>ZeroDivisionError</code>	Divide by zero
<code>ValueError</code>	Wrong value type
<code>TypeError</code>	Wrong data type
<code>IndexError</code>	List index out of range
<code>KeyError</code>	Dictionary key not found
<code>FileNotFoundError</code>	File missing

▼ Finally Block

`finally` **always runs**, error or no error.

Used for:

- Closing files
- Releasing resources

▼ Example

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Error")
```

```
finally:  
    print("This will always run")
```

▼ Raising Your Own Exception

```
age = -5  
if age < 0:  
    raise ValueError("Age cannot be negative")
```

▼ Custom Exception

```
class MyError(Exception):  
    pass  
  
raise MyError("This is my custom error")
```

▼ File Handling

▼ Opening file

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

▼ Syntax

```
open(filename, mode)
```

▼ Modes

Mode	Description	Creates File (if not exists)?	Overwrites Existing Data?
'r'	Read mode – Opens a file for reading only.	✗ No	✗ No
'w'	Write mode – Opens a file for writing. If the file exists, it is overwritten.	✓ Yes	✓ Yes
'a'	Append mode – Opens a file for writing, but data is added at the end.	✓ Yes	✗ No
'x'	Exclusive creation mode – Creates a file but fails if it already exists.	✓ Yes	✗ No

Mode	Description	Creates File (if not exists)?	Overwrites Existing Data?
'r+'	Read & Write mode – Opens a file for both reading and writing.	✗ No	✓ Yes
'w+'	Write & Read mode – Opens a file for both reading and writing, but it overwrites if the file exists.	✓ Yes	✓ Yes
'a+'	Append & Read mode – Opens a file for reading and appending, preserving existing data.	✓ Yes	✗ No
'x+'	Exclusive creation mode with read and write – Creates a new file but fails if it exists.	✓ Yes	✗ No
'rb'	Read binary mode – Used for non-text files (images, audio, etc.).	✗ No	✗ No
'wb'	Write binary mode – Writes binary data, overwriting if file exists.	✓ Yes	✓ Yes

▼ Reading a file

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

▼ With Statement

Automatically closes the file

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