**Python**

Table of Contents

[1. Reserved words or Keywords: 2](#_Toc202265111)

[2. Variable: 3](#_Toc202265112)

[3. Data Types: 3](#_Toc202265113)

[4. Operations: 4](#_Toc202265114)

[1. String Operations 4](#_Toc202265115)

[2. List Operations 6](#_Toc202265116)

[3. Set Operations 7](#_Toc202265117)

[4. Tuple Operations 8](#_Toc202265118)

[5. Dictionary Operations 9](#_Toc202265119)

[6. File Operations 9](#_Toc202265120)

[5. Operators: 10](#_Toc202265121)

[1. Arithmetic Operator 10](#_Toc202265122)

[2. Assignment Operator 11](#_Toc202265123)

[3. Comparison Operator 12](#_Toc202265124)

[4. Logical Operator 12](#_Toc202265125)

[5. Identity Operator 13](#_Toc202265126)

[6. Membership Operator 13](#_Toc202265127)

[7. Bitwise Operator 13](#_Toc202265128)

[6. Flow Control: 14](#_Toc202265129)

[1. Conditional Statement 14](#_Toc202265130)

[2. Looping Statement 14](#_Toc202265131)

[3. Loop Control Statement 15](#_Toc202265132)

[4. Exception Handling Statement 15](#_Toc202265133)

[5. Match Statement 16](#_Toc202265134)

[6. Assert Statement 16](#_Toc202265135)

[7. Collection Data Types 16](#_Toc202265136)

[8. Functions 17](#_Toc202265137)

[9. Object Oriented Programming: 19](#_Toc202265138)

**Python Notes**

- Popular Programming Language - Guido van Rossum in 1991 at CWI in the Netherlands

- Python is an interpreted, high-level, general-purpose programming language.

- It is designed to be easy to read and write, with a focus on code readability and simplicity.

- Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

# 1. Reserved words or Keywords:

- Keywords have special meanings to the compiler. There are 37 keywords.

- They cannot be used as identifiers (variable names, function names, etc.)

- Keywords are case-sensitive, meaning that they must be written in lowercase.

**Value Keywords** - True, False, None

**Operator Keywords** - and, or, not, in, is

**Control Flow Keywords** - if, elif, else, match, case

**Iteration Keywords**  - for, while, break, continue, pass

**Structure Keywords**  - def, class, with, lambda

**Returning Keywords**  - return, yield

**Import Keywords** - import, from, as

**Exception-Handling Keywords** - try, except, raise, finally, assert

**Asynchronous Keywords** - async, await

**Variable Handling Keywords** - del, global, nonlocal

**None - Represents no Value**

**Eg:**

def func(): - can't print the call

print("hello") - can't print the print

x = func() # hello print(x) # None

# 2. Variable:

- Containers for storing data values.

- **Literals** - Raw data assigned to a Variable or Constant

- Python is a dynamically-typed language, and there is no built-in mechanism to enforce constant values.

- Python is case sensitive. Eg: y and Y are Different

- Variable naming rules (e.g., no starting with numbers, no special characters except \_).

**Eg:** name = "Shiffana" – name is a Variable and Shiffana is a Literal

# 3. Data Types:

- Data types are classifications of data that determine the kind of value a variable can hold and the operations that can be performed on it.

- Python is a dynamically typed language, meaning you don't need to declare the data type of a variable explicitly.

- In Python, there are several built-in data types that you can use to store different kinds of data.

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Type** | **Description** | **Example** |
| **Numeric Types** | Int | Whole numbers, positive or negative | x = 10 |
| Float | Numbers with decimal points | pi = 3.14 |
| Complex | Numbers with real and imaginary parts | z = 2 + 3j |
| **Text Types** | Str | Sequence of characters (Unicode) | name = "Alice" |
| **Boolean Types** | Bool | Logical values: True or False | is\_active = True |
| **Sequence Types** | List | Mutable ordered sequence | fruits = ["apple", "banana"] |
| Tuple | Immutable ordered sequence | point = (10, 20) |
| Range | Immutable range of numbers | numbers = range(5) |
| **Mapping Types** | Dict | Key-value pair collection | u={"name":"Bob","age":30} |
| **Set Types** | Set | Unordered collection of unique elements | s = {1, 2, 3} |
| Frozenset | Unordered unique collection | fs = frozenset([1, 2, 3]) |
| **Binary Types** | Bytes | Immutable byte sequence | b = b"hello" |
| Bytearray | Mutable byte sequence | ba = bytearray(b"data") |
| Memoryview | View over byte data | mv = memoryview(b"abc") |
| **None Type** | NoneType | Represents absence of value | result = None ; type(None) |

Data Type: Type Casting: - Changing the type of data

print(type(x)) x = int("3")

# 4. Operations:

- Operations are actions performed on data types using operators or methods.

- They allow you to manipulate, compare, and transform data.

- Tuples and dictionaries support indexing and key-based access- Tuples and dictionaries support indexing and key-based access

## 1. String Operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| ' ' / " " | Define a string | 'Hello' , "World" |
| len() | Length of the string | len("Python") → 6 |
| in , not in | Check if a substring exists | "a" in "cat" → True |
| Indexing | Access character by position | "Hello"[1] → 'e' |
| Slicing | Access part of the string | "Hello"[1:4] → 'ell' |
| upper() | Convert to uppercase | "hi".upper() → "HI" |
| lower() | Convert to lowercase | "HI".lower() → "hi" |
| capitalize() | Capitalize first character | "python".capitalize() → "Python" |
| strip() | Remove whitespace from both ends | " hi ".strip() → "hi" |
| replace() | Replace parts of a string | "hi".replace("h", "H") → "Hi" |
| split() | Split into list by delimiter | "a,b".split(",") → ['a', 'b'] |
| format() | Format strings with placeholders {} | "Hello {}".format("World") |
| find() | Find first occurrence of substring | "apple".find("p") → 1 |
| rfind() | Find last occurrence of substring | "banana".rfind("a") → 5 |
| index() | Like find() but raises error if not found | "abc".index("b") → 1 |
| startswith() | Check if string starts with value | "apple".startswith("a") → True |
| endswith() | Check if string ends with value | "apple".endswith("e") → True |
| isalpha() | Check if all characters are letters | "abc".isalpha() → True |
| isdigit() | Check if all characters are digits | "123".isdigit() → True |
| isalnum() | Check if all characters are alphanumeric | "abc123".isalnum() → True |
| isupper() | Check if all characters are uppercase | "HELLO".isupper() → True |
| islower() | Check if all characters are lowercase | "hello".islower() → True |
| join() | Join iterable into string | ",".join(["a", "b"]) → "a,b" |
| zfill() | Pad string on the left with zeros | "7".zfill(3) → "007" |
| Escape Sequences | Special characters: newline, tab, quotes, etc. | \n , \t , \' , \\ , \" |

## 2. List Operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| len() | Get number of items in the list | len([1, 2, 3]) → 3 |
| in , not in | Check if item exists in the list | "apple" in fruits → True |
| Indexing | Access item by index | fruits[0] → "apple" |
| Slicing | Get sublist by range | fruits[1:3] → ['banana', 'mango'] |
| count() | Count occurrences of an item | fruits.count("apple") → 2 |
| index() | Find index of the first matching item | fruits.index("banana") → 1 |
| Change Item | Modify item by index | fruits[1] = "grape" |
| insert() | Insert item at a specific index | fruits.insert(2, "melon") |
| append() | Add item to the end of the list | fruits.append("kiwi") |
| extend() | Add multiple items from another list | fruits.extend(["peach", "plum"]) |
| remove() | Remove first occurrence of an item | fruits.remove("apple") |
| pop() | Remove item at index or last if no index provided | fruits.pop(1) , fruits.pop() |
| del | Delete item by index or entire list | del fruits[1] , del fruits |
| clear() | Remove all items from the list | fruits.clear() |
| sort() | Sort list in ascending order | fruits.sort() |
| sort(reverse=True) | Sort list in descending order | fruits.sort(reverse=True) |
| reverse() | Reverse the list order | fruits.reverse() |
| copy() | Create a shallow copy of the list | new\_list = fruits.copy() |
| List Comprehension | Create list using loop and condition | [x for x in range(5) if x % 2 == 0] |

## 3. Set Operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| add() | Adds an element to the set | s.add(5) |
| update() | Adds multiple elements | s.update([1, 2, 3]) |
| remove() | Removes element; error if not found | s.remove(2) |
| discard() | Removes element if present; no error if absent | s.discard(10) |
| pop() | Removes a random element | s.pop() |
| clear() | Removes all elements | s.clear() |
| union() | Returns union of two sets | s1.union(s2) |
| intersection() | Returns common elements | s1.intersection(s2) |
| difference() | Returns items only in first set | s1.difference(s2) |
| symmetric\_difference() | Items in either, but not both sets | s1.symmetric\_difference(s2) |
| issubset() | Checks if set is subset | s1.issubset(s2) |
| issuperset() | Checks if set is superset | s1.issuperset(s2) |
| isdisjoint() | Checks if sets have no items in common | s1.isdisjoint(s2) |

## 4. Tuple Operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| Indexing | Access item by index | t[1] → 'banana' |
| Slicing | Access a range of items | t[1:3] |
| count() | Count occurrences of a value | t.count("apple") → 2 |
| index() | Find index of a value | t.index("banana") → 1 |
| in | Check if value exists | "apple" in t → True |
| len() | Length of tuple | len(t) → 3 |
| Immutability | Cannot be modified after creation | t[0] = "new" ❌ Error |
| Nesting | Tuples can contain lists, other tuples | t = (1, [2, 3], (4, 5)) |

## 5. Dictionary Operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| get() | Gets value by key (returns None if not found) | d.get("name") |
| keys() | Returns list-like view of all keys | d.keys() |
| values() | Returns list-like view of all values | d.values() |
| items() | Returns view of key-value pairs | d.items() |
| update() | Updates dictionary with another dict or key-value | d.update({"age": 30}) |
| pop() | Removes key and returns its value | d.pop("name") |
| popitem() | Removes and returns last key-value pair | d.popitem() |
| del | Deletes key or entire dict | del d["name"] , del d |
| clear() | Empties the dictionary | d.clear() |
| in | Check if key exists | "name" in d → True |
| Indexing | Access or assign value with key | d["name"] = "Ali" |

## 6. File Operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| open() | Opens a file ( 'r' , 'w' , 'a' , 'rb' , 'wb' , etc.) | f = open("file.txt", "r") |
| read() | Reads entire file content | f.read() |
| readline() | Reads one line | f.readline() |
| readlines() | Reads all lines into a list | f.readlines() |
| write() | Writes content (overwrites in 'w' mode) | f.write("Hello") |
| writelines() | Writes a list of strings to a file | f.writelines(["a\n", "b\n"]) |
| close() | Closes the file | f.close() |
| with open() | Context manager to open file (auto-close) | with open("file.txt") as f: |
| seek() | Move the file cursor to given position | f.seek(0) |
| tell() | Returns current file position | f.tell() |
| flush() | Force write buffer to disk | f.flush() |

# 5. Operators:

- Operators are symbols that perform operations on variables and values.

- They are used to manipulate data and variables in Python.

- Operators can be classified into several categories based on their functionality.

## 1. Arithmetic Operator

- Numeric values to perform common Mathematical Operations

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |
| + | Addition | Adds two values | 3 + 2 = 5 |
| - | Subtraction | Subtracts right operand from left | 5 - 2 = 3 |
| \* | Multiplication | Multiplies both values | 4 \* 3 = 12 |
| / | Division | Divides left by right (float result) | 10 / 4 = 2.5 |
| // | Floor Division | Divides and removes decimal (rounds down) | 10 // 4 = 2 |
| % | Modulo | Returns remainder | 10 % 3 = 1 |
| \*\* | Exponentiation | Power of (left raised to right) | 2 \*\* 3 = 8 |

## 2. Assignment Operator

- Assign values to Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |
| = | Assignment | Assigns value | x = 5 |
| += | Add AND assign | x = x + 3 | x += 3 |
| -= | Subtract AND assign | x = x - 2 | x -= 2 |
| \*= | Multiply AND assign | x = x \* 4 | x \*= 4 |
| /= | Divide AND assign | x = x / 2 | x /= 2 |
| //= | Floor divide assign | x = x // 2 | x //= 2 |
| %= | Modulo AND assign | x = x % 2 | x %= 2 |
| \*\*= | Power AND assign | x = x \*\* 2 | x \*\*= 2 |
| &= | Bitwise AND assign | x = x & 1 | x &= 1 |
| |= | Bitwise OR assign | x = x | 2 | x |= 2 |
| ^= | Bitwise XOR assign | x = x ^ 3 | x ^= 3 |
| >>= | Right shift assign | x = x >> 1 | x >>= 1 |
| <<= | Left shift assign | x = x << 1 | x <<= 1 |

## 3. Comparison Operator

- Compare two values

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal to | x == y |
| != | Not equal to | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

## 4. Logical Operator

- Combine Conditional Statements

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | True if both conditions are True | x > 2 and x < 10 |
| or | True if one condition is True | x < 5 or x > 20 |
| not | Reverses the condition | not(x == 5) |

## 5. Identity Operator

- Compare the objects

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | True if both variables point to same object | x is y |
| is not | True if they don't point to same object | x is not y |

## 6. Membership Operator

- Test is a sequence is present

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | True if value is in sequence | "a" in "apple" |
| not in | True if value not in sequence | "z" not in "dog" |

## 7. Bitwise Operator

- Compare Binary Numbers

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example (in binary)** |
| & | AND | 5 & 3 → 101 & 011 = 001 |
| | | OR | 5 | 3 → 101 | 011 = 111 |
| ^ | XOR (exclusive OR) | 5 ^ 3 → 101 ^ 011 = 110 |
| ~ | NOT (inverts bits) | ~5 → -(5+1) = -6 |
| << | Left shift | 2 << 1 → 4 (10 → 100) |
| >> | Right shift | 4 >> 1 → 2 (100 → 10) |

# 6. Flow Control:

- Flow control statements determine the order in which statements are executed in a program.

- They allow you to control the flow of execution based on conditions, loops, and exceptions.

- Flow control statements are essential for making decisions, repeating actions, and handling errors in your code.

## 1. Conditional Statement

- It lets the program decide which block of code to execute based on the condition. Decision making.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Definition** | **Syntax** | **Example** |
| if | Runs a block of code if the condition is True | if condition: | if x > 10:\n print("Big") |
| elif | Runs if the if is False and this condition is True | elif condition: | elif x == 10:\n print("Equal") |
| else | Runs if all above conditions are False | else: | else:\n print("Small") |
| Ternary if | One-line conditional expression | x if condition else y | result = "Yes" if x > 0 else "No" |

## 2. Looping Statement

- Used to repeate a block of code multiple times.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Definition** | **Syntax** | **Example** |
| for | Iterates over items in a sequence | for var in sequence: | for i in range(3):\n print(i) |
| while | Repeats code while the condition is True | while condition: | while x < 5:\n x += 1 |

## 3. Loop Control Statement

- Used to alter the flow of loops: exit early, skip iterations or do nothing.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Definition** | **Syntax** | **Example** |
| break | Exits the loop immediately | break | if x == 3:\n break |
| continue | Skips the rest of the loop iteration | continue | if x % 2 == 0:\n continue |
| pass | Does nothing (placeholder) | pass | if True:\n pass |
| else on loop | Executes if the loop was not exited using break | for/while ... else: | for i in range(3): ... else: print("Done") |

## 4. Exception Handling Statement

- Used to handle errors or unexpected conditions gracefully without crashing the program

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept** | **Explanation** | **Use Case** | **Example** |
| **try-except** | Catch and handle errors gracefully | Prevent crash when input is wrong | try:  x = int(input("Enter a number: "))  except ValueError:  print("Invalid input") |
| **Multiple except blocks** | Catch specific error types | Handle errors differently | try:  x = 10 / 0  except ZeroDivisionError:  print("Cannot divide by zero")  except ValueError:  print("Invalid value") |
| **else block** | Runs only if no exception occurred | Perform actions when no errors | try:  x = int(input())  except ValueError:  print("Invalid")  else:  print("Success", x) |
| **finally block** | Always runs, error or not | Clean up (e.g., close files) | try:  f = open("file.txt")  except FileNotFoundError:  print("Not found")  finally:  print("Done") |
| **raise** | Manually throw an error | Validate data, custom checks | def set\_age(age):  if age < 0:  raise ValueError("Age can't be negative") |
| **Custom Exception** | Create specific errors using custom classes | Business logic validation | class MyError(Exception):  pass  raise MyError("Something went wrong") |

## 5. Match Statement

- Pattern matching allows matching values and executing corresponding code blocks, similar to switch in other languages.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Definition** | **Syntax** | **Example** |
| match | Start of the pattern matching structure | match variable: | match status: |
| case | Handles each pattern | case value: | case 200:\n print("OK") |
| case \_ | Catch-all case (like default in switch) | case \_: | case \_:\n print("Unknown") |

## 6. Assert Statement

- Used to debugging. It tests if a condition is True; if not, it raises an AssertionError.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Definition** | **Syntax** | **Example** |
| assert | Tests if a condition is True | assert condition | assert x > 0, "x must be positive" |

# 7. Collection Data Types

- Collection data types are used to store multiple items in a single variable.

- They allow you to group related data together and perform operations on the entire collection.

- Python has several built-in collection data types, each with its own characteristics and use cases.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Ordered** | **Mutable** | **Allows Duplicates** | **Unique Feature** | **Example** |
| **List** | Yes | Yes | Yes | Mutable list | ["a", "b", "c"] |
| **Tuple** | Yes | No | Yes | Immutable list | ("a", "b", "c") |
| **Set** | No | Yes | No | Only unique items | {"a", "b", "c"} |
| **Frozenset** | No | No | No | Immutable set | frozenset(["a", "b"]) |
| **Dict** | Yes | Yes | Keys only | Key-value pairs | {"name": "Alice", "age": 25} |

1. **List** – Use when you need a dynamic and ordered collection (e.g., shopping list).
2. **Tuple** – Use when the data should not change (e.g., GPS coordinates).
3. **Set** – Use when you want to store unique values (e.g., removing duplicates).
4. **Dictionary** – Use when you want to map keys to values (e.g., storing employee data).
5. **frozenset** – Use when you want a constant set (e.g., as a dictionary key or for hashable needs).

# 8. Functions

- Functions are reusable blocks of code that perform a specific task.

- They help organize code, avoid repetition, and make it easier to read and maintain.

- Functions can take inputs (parameters) and return outputs (return values).

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept** | **Explanation** | **Use Case** | **Example** |
| **Function Definition** | Use `def` to create a function | Reusable code blocks | def greet():  print("Hello") |
| **Function Call** | Executes function code | Trigger functionality | greet() |
| **Parameters** | Input variables in definition | Customize behavior | def greet(name):  print("Hello", name) |
| **Arguments** | Actual values passed | Provide input | greet("Shiffana") |
| **Return Statement** | Sends value to caller | Use result in code | def add(a, b):  return a + b |
| **Default Parameters** | Use default if no argument given | Optional arguments | def greet(name="Guest"):  print("Hello", name) |
| **Keyword Arguments** | Specify parameter names | Avoid ordering issues | def student(name, age):  print(name, age) student(age=20, name="Ali") |
| **\*args (Positional)** | Accept multiple positional args | Accept unlimited inputs | def show(\*args):  for i in args:  print(i) |
| **\*\*kwargs (Keyword)** | Accept multiple keyword args | Flexible named arguments | def show(\*\*kwargs):  for k, v in kwargs.items():  print(k, v) |
| **Lambda Function** | One-line anonymous function | Inline short logic | square = lambda x: x \*\* 2 print(square(4)) |
| **Higher-order Functions** | Takes or returns other functions | Flexible function passing | def apply(func, val):  return func(val) apply(lambda x: x \* 2, 5) |
| **Function Annotations** | Type hints for readability | IDE help, static checks | def add(a: int, b: int) -> int:  return a + b |
| **Docstring** | Function documentation string | Explain purpose of function | def greet():  """This says hello."""  print("Hi") print(greet.\_\_doc\_\_) |
| **Scope** | Local vs global variables | Prevent accidental overwrite | x = 10 def test():  x = 5  print(x) test() print(x) |
| **nonlocal** | Modify outer function’s variable | Update nested scope variable | def outer():  x = 'old'  def inner():  nonlocal x  x = 'new'  inner()  print(x) |
| **Nested Functions** | Define function inside another | Limit scope of helper functions | def outer():  def inner():  print("Hi")  inner() |
| **Closures** | Inner function remembers outer values | Preserve state | def outer():  msg = 'Hi'  def inner():  print(msg)  return inner |
| **Decorators** | Modify behavior of another function | Logging, auth, timing | def deco(func):  def wrap():  print("Before")  func()  return wrap |
| **Recursion** | Function calling itself | Factorial, Fibonacci, backtracking | def fact(n):  if n == 0:  return 1  return n \* fact(n - 1) |
| **\_\_name\_\_ == '\_\_main\_\_'** | Run code only if not imported | Script reuse and testing | def main():  print("Running directly") if \_\_name\_\_ == "\_\_main\_\_":  main() |
| **functools.partial** | Fix some arguments in advance | Preconfigure function behavior | from functools import partial double = partial(pow, 2) print(double(3)) |
| **functools.lru\_cache** | Memoize results to speed up function | Cache expensive calls | from functools import lru\_cache @lru\_cache(maxsize=None) def fib(n):  if n < 2:  return n  return fib(n-1) + fib(n-2) |
| **Function Introspection** | Access metadata like name/doc | Debugging and reflection | def f():  """Sample"""  pass print(f.\_\_name\_\_) print(f.\_\_doc\_\_) |
| **Async Functions** | Support async programming | Concurrent I/O, networking | import asyncio async def greet():  await asyncio.sleep(1)  print("Hello") |
| **Generator Functions** | Yield values one at a time | Handle large data streams | def count\_up():  yield 1  yield 2  yield 3 |
| **Callable Objects** | Objects behaving like functions | Advanced OOP use | class Greeter:  def \_\_call\_\_(self):  print("Hello") g = Greeter() g() |

# 9. Object Oriented Programming:

* OOP (Object-Oriented Programming) is a programming paradigm that emphasizes the use of objects and classes to represent and manipulate data.
* In Python, you can use OOP concepts to design and create classes, objects, and methods that encapsulate data and behavior into a single unit.

Here are some of the key OOP concepts in Python:

**Class:**

* A class is a blueprint or a template for creating objects. It defines the attributes (data) and methods (functions) that the objects of that class will have.

**Object:**

* An object is an instance of a class. It is created from a class and has its own unique data and behavior.
* Objects can be created, modified, and deleted dynamically during program execution.

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept** | **Explanation** | **Use Case** | **Example** |
| **Class and Object** | Class is a blueprint; Object is an instance | Model real-world entities | class Person:  pass p1 = Person() |

**Encapsulation:**

* Encapsulation is the process of hiding the implementation details of a class from the outside world and exposing only the necessary interface for interacting with the class.
* This helps to improve code modularity and reduce code complexity.

|  |  |  |  |
| --- | --- | --- | --- |
| **Encapsulation** | Hide internal data | Protect data with controlled access | class Bank:  def \_\_init\_\_(self):  self.\_\_balance = 0  def deposit(self, amt):  self.\_\_balance += amt  def get\_balance(self):  return self.\_\_balance |

**Inheritance:**

* Inheritance is the ability of a class to inherit the attributes and methods of another class.
* It allows you to reuse existing code and create new classes that extend the functionality of existing classes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Inheritance** | Child class inherits from parent | Reuse and extend functionality | class Animal:  def speak(self):  print("Sound") class Dog(Animal):  def speak(self):  print("Bark") |

**Polymorphism:**

* Polymorphism is the ability of different objects to respond to the same message (method call) in different ways.
* It allows you to write code that can work with different types of objects without knowing their specific types at compile-time.

|  |  |  |  |
| --- | --- | --- | --- |
| **Polymorphism** | Same method works differently in different classes | Flexible function behavior | class Cat:  def sound(self): print("Meow") class Dog:  def sound(self): print("Bark") def make\_sound(a): a.sound() |

**Abstraction:**

* Abstraction is the process of simplifying complex systems by breaking them down into smaller, more manageable parts.
* It allows you to focus on the essential features of an object while ignoring the irrelevant details.

|  |  |  |  |
| --- | --- | --- | --- |
| **Abstraction** | Hide details using abstract base class | Enforce structure in child classes | from abc import ABC, abstractmethod class Shape(ABC):  @abstractmethod  def area(self): pass |

In Python, you can use these OOP concepts to create reusable, modular, and maintainable code that is easier to understand and debug.

OOP can be a powerful tool for solving complex programming problems and building robust software systems.

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept** | **Explanation** | **Use Case** | **Example** |
| **Class and Object** | Class is a blueprint; Object is an instance | Model real-world entities | class Person:  pass p1 = Person() |
| **\_\_init\_\_() Constructor** | Automatically called when an object is created | Initialize object state | class Person:  def \_\_init\_\_(self, name):  self.name = name p = Person("Ali") print(p.name) |
| **Instance vs Class Variables** | Instance is unique per object; class is shared | Store object-specific or common data | class Dog:  species = 'Canine'  def \_\_init\_\_(self, name):  self.name = name |
| **Instance, Class, Static Methods** | Different types of methods with self/cls/no arg | Control access to instance/class/utility logic | class Dog:  def bark(self):  print("Bark")  @classmethod  def info(cls):  print("Class method")  @staticmethod  def help():  print("Static method") |
| **Encapsulation** | Hide internal data | Protect data with controlled access | class Bank:  def \_\_init\_\_(self):  self.\_\_balance = 0  def deposit(self, amt):  self.\_\_balance += amt  def get\_balance(self):  return self.\_\_balance |
| **Inheritance** | Child class inherits from parent | Reuse and extend functionality | class Animal:  def speak(self):  print("Sound") class Dog(Animal):  def speak(self):  print("Bark") |
| **Polymorphism** | Same method works differently in different classes | Flexible function behavior | class Cat:  def sound(self): print("Meow") class Dog:  def sound(self): print("Bark") def make\_sound(a): a.sound() |
| **Abstraction** | Hide details using abstract base class | Enforce structure in child classes | from abc import ABC, abstractmethod class Shape(ABC):  @abstractmethod  def area(self): pass |
| **super()** | Calls method from parent class | Extend or override parent logic | class A:  def \_\_init\_\_(self): print("A") class B(A):  def \_\_init\_\_(self):  super().\_\_init\_\_()  print("B") |
| **Magic Methods** | Special methods with double underscores | Operator overloading, printing | class Book:  def \_\_init\_\_(self, title): self.title = title  def \_\_str\_\_(self): return self.title |
| **Private & protected** | Restrict variable access | Prevent misuse | class Person:  def \_\_init\_\_(self):  self.\_age = 30  self.\_\_salary = 5000 |
| **@property** | Access method like attribute | Read-only or controlled setters | class Circle:  def \_\_init\_\_(self, r): self.\_r = r  @property  def area(self): return 3.14 \* self.\_r\*\*2 |
| **Composition** | Class has another class | Better for has-a relationship | class Engine:  def start(self): print("Start") class Car:  def \_\_init\_\_(self): self.engine = Engine() |
| **Duck Typing** | Based on behavior not type | Use flexible objects | class Duck:  def quack(self): print("Quack") class Person:  def quack(self): print("Fake quack") |
| **Multiple Inheritance** | Class inherits from multiple classes | Combine behaviors | class A: pass class B: pass class C(A, B): pass |

**Collection Module:**

**Counter**

| Task / Feature | Example Code | Output / Description |
| --- | --- | --- |
| 1. Count elements | Counter("banana") | Counter({'a': 3, 'n': 2, 'b': 1}) – counts each character |
| 2. Get count of a value | c['a'] | 3 – returns how many times 'a' appears |
| 3. Safe access (missing key) | c['z'] | 0 – returns 0 if the item does not exist (no error) |
| 4. List all items | c.items() | dict\_items([('b', 1), ('a', 3), ('n', 2)]) |
| 5. List all keys only | c.keys() | dict\_keys(['b', 'a', 'n']) |
| 6. List all counts only | c.values() | dict\_values([1, 3, 2]) |
| 7. Most common N items | c.most\_common(2) | [('a', 3), ('n', 2)] – top 2 most frequent items |
| 8. All elements expanded | list(c.elements()) | ['b', 'a', 'a', 'a', 'n', 'n'] – repeats each item by its count |
| 9. Add two Counters | Counter("abc") + Counter("bcd") | Counter({'b': 2, 'c': 2, 'a': 1, 'd': 1}) |
| 10. Subtract Counters | Counter("abc") - Counter("bcd") | Counter({'a': 1}) – subtracts counts, removes negatives |
| 11. Minimum (Intersection) | Counter("abc") & Counter("bcd") | Counter({'b': 1, 'c': 1}) – keeps the lower count for common items |
| 12. Maximum (Union) | `Counter("abc") | Counter("bcd")` |
| 13. Update with more data | c.update("ana") | Adds counts from "ana" to the existing Counter |
| 14. Delete an item | del c['a'] | Removes the item 'a' entirely from the Counter |