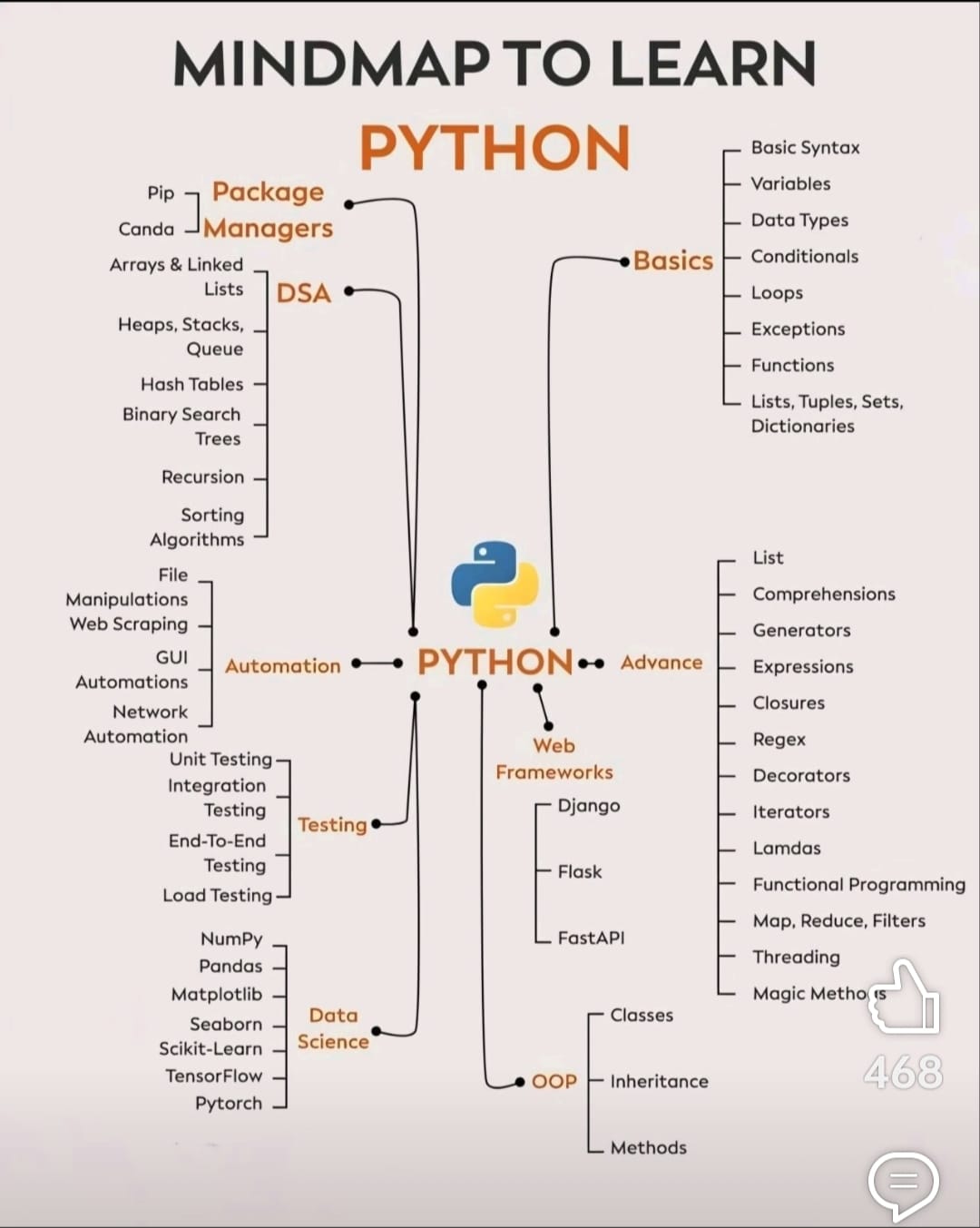
****

**#Python**.

* Python is a high level interpreted programming language.
* High level means any language which is in human readable form.
* Interpreted means execution of the program happen line by line.
* programming language means a medium of communicate or interact with system.

**#Features of python.**

* Simple and easy to learn.
* Cross platform language and platform independent language.
* Portable and Extensible language.
* Dynamically typed language.
* Free downloadable software.
* Huge Ecosystem of Libraries.
* Extensive Standard Library

#**Variables**.

* Variables are used to store data or information.
* Integer (x = 10)
* Float (x = 10.00)
* String (name = ‘shivam’)

#**Datatypes**.

* Datatypes is used to indicate or specified the type of data.
* Int
* Float
* Bool (Boolean (True/False))
* Str (String)

**#Operators**.

* operators are symbols or Symbolic representation to perform some task.

1. **Arithmetic operator: (+, -, \*, %, /, //, \*\*)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Example** | **Description** | **Calculation** | **Result** |
| + | a + b | Addition | 15 + 4 | **19** |
| - | a - b | Subtraction | 15 - 4 | **11** |
| \* | a \* b | Multiplication | 15 × 4 | **60** |
| / | a / b | Division (gives float) | 15 ÷ 4 | **3.75** |
| // | a // b | Floor Division (integer result) | 15 ÷ 4 = 3.75 → 3 | **3** |
| % | a % b | Modulus (remainder) | 15 ÷ 4 → remainder 3 | **3** |
| \*\* | a \*\* b | Exponentiation (power) | 15⁴ = 15×15×15×15 | **50625** |

1. **Assignment operators: (=, +=, -=, \*=, %=, /=, //=)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Meaning** | **Result** |
| = | a = 10 | Assign 10 to a | 10 |
| += | a += 5 | a = a + 5 | 15 |
| -= | a -= 3 | a = a - 3 | 12 |
| \*= | a \*= 2 | a = a × 2 | 24 |
| /= | a /= 4 | a = a ÷ 4 | 6.0 |
| %= | a %= 5 | a = remainder of a ÷ 5 | 1 |
| //= | a //= 3 | a = integer part of a ÷3 | 3 |
| \*\*= | a \*\*= 3 | a = a³ | 8 |

1. **Relational or Compersion: (==, !=, <, <=, >, >=)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Example** | **Description** | **Calculation / Comparison** | **Result** |
| == | a == b | Equal to | Is 15 equal to 4? | **False** |
| != | a != b | Not equal to | Is 15 not equal to 4? | **True** |
| > | a > b | Greater than | Is 15 greater than 4? | **True** |
| < | a < b | Less than | Is 15 less than 4? | **False** |
| >= | a >= b | Greater than or equal to | Is 15 greater than or equal to 4? | **True** |
| <= | a <= b | Less than or equal to | Is 15 less than or equal to 4? | **False** |

1. **Logical operators: (and, or, not)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Example** | **Description** | **Explanation** | **Result** |
| and | (a > b) and (b > c) | Returns True if **both** conditions are true | a > b → 10 > 5 → Trueb > c → 5 > 0 → TrueBoth True → True | **True** |
| or | (a > b) or (b < c) | Returns True if **at least one** condition is true | a > b → Trueb < c → FalseOne True → True | **True** |
| not | not(a > b) | Reverses the result | a > b → True → not(True) = False | **False** |

1. **Membership operator: (In, Not In)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Example** | **Description** | **Explanation** | **Result** |
| in | "apple" in fruits | Returns True if the value **exists** in the sequence | “apple” is in the fruits list | **True** |
| not in | "grapes" not in fruits | Returns True if the value **does not exist** in the sequence | “grapes” is not in the fruits list | **True** |
| in | 's' in name | Checks if character exists in string | 's' is in "shivam" | **True** |
| not in | 'z' not in name | Checks if character not in string | 'z' not in "shivam" | **True** |

1. **Identity operators: (Is, Is not)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Example** | **Description** | **Explanation** | **Result** |
| is | a is c | Returns True if both variables **refer to the same object** in memory | c was assigned to a → both point to the same list | **True** |
| is | a is b | Checks if a and b are the **same object** | a and b have the same values but are stored separately | **False** |
| is not | a is not b | Returns True if both variables **refer to different objects** | a and b are different memory objects | **True** |

**#Special datatypes**

1. **#List**
2. List items are ordered collection.
3. List supports indexing both positive and negative.
4. List allows duplicate values.
5. List can be homogenous and heterogenous object.
6. List is mutable in nature.
7. List support slicing.
8. **#Tuple**
9. It is a collection which is ordered collection.
10. It supports both positive and negative index.
11. It can allow both homogenous and heterogenous object.
12. It is in-mutable in nature. We can’t update the values inside a tuple once it inserted.
13. It allows duplicate values or duplication.
14. It maintains insertion order.
15. It supports slicing.
16. **#Set**
17. Set are unordered collection.
18. Set are unindexed doesn’t support both positive and negative index.
19. Set can allow both homogenous and heterogenous object.
20. Set doesn’t allow duplicate values.
21. Set doesn’t maintain insertion order.
22. Set is mutable in nature.
23. Set doesn’t support slicing.
24. **#Dictionary**
25. It is an ordered collection.
26. It doesn’t support indexing like positive or negative.
27. It doesn’t support slicing.
28. Keys can’t be duplicate but values can.
29. Values can be duplicate or allows duplication.
30. Keys can be int, float, string.
31. It can be both homogeneous and heterogeneous.
32. It is a mutable collection.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Features | List | Tuple | Set | Dictionary |
| |  | | --- | |  | |  | |  | |  | |  | |  |   Mutability | Mutable | Immutable | Mutable | Mutable |
| Order | Ordered | Ordered (3.7+) | Unordered | Ordered (3.7+) |
| Duplicates | Allowed | Allowed | Not allowed | Keys unique |
| Syntax | [1, 2, 3] | (1, 2, 3) | {1, 2, 3} | {"a": 1, "b": 2} |
| Access | Index [0] | Index [0] | No index [] | Key [‘a’] |
| Use-Case | Dynamic sequences | Fixed data | Unique elements | Key-values mappings |

**#List comprehension:**

* List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax: *[****expression*** *for* ***item*** *in* ***iterable*** *if* ***condition****]*

Code:a = [2,3,4,5]

res = [val \*\* 2 for val in a]

print(res) # output [4, 9, 16, 25]

**List Built-in Methods.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Syntax** | **Example & Output** |
| 1. copy() | Creates a new list with the same elements as the original. | copy\_list = list\_name.copy() | original\_list = [1, 2, 3, [4, 5]] copied\_list = original\_list.copy() print(copied\_list) # Output: [1, 2, 3, [4, 5]] |
| 1. append() | Adds an element at the end of the list. | list\_name.append(element) | my\_list = [1, 2, 3] my\_list.append(4) print(my\_list) # Output: [1, 2, 3, 4] |
| 1. index() | Finds the index of a value in the list. | list\_name.index(value, start, end) | my\_list = [10, 20, 30, 40, 20] print(my\_list.index(20)) # Output: 1 |
| 1. extend() | Extracts the given values and adds them at the end of the list. | list\_name.extend(iterable) | my\_list = [1, 2, 3] my\_list.extend([4, 5]) print(my\_list) # Output: [1, 2, 3, 4, 5] |
| 1. clear() | Removes all elements from the list. | list\_name.clear() | fruits = ['apple', 'banana', 'cherry'] fruits.clear() print(fruits) # Output: [] |
| 1. pop() | Removes and returns the last element (or specified index). | list\_name.pop([index]) | fruits = ['apple', 'banana', 'cherry'] removed = fruits.pop(1) print(fruits) # Output: ['apple', 'cherry'] |
| 1. remove() | Removes the first matching value. | list\_name.remove(value) | numbers = [1, 2, 3, 2, 4] numbers.remove(2) print(numbers) # Output: [1, 3, 2, 4] |
| 1. count() | Returns the number of times a value appears in the list. | list\_name.count(value) | fruits = ['apple', 'banana', 'apple'] print(fruits.count('apple')) # Output: 2 |
| 1. reverse() | Reverses the list in place. | list\_name.reverse() | numbers = [1, 2, 3, 4, 5] numbers.reverse() print(numbers) # Output: [5, 4, 3, 2, 1] |
| 1. sort() | Sorts the list in ascending order. | list\_name.sort() | numbers = [4, 1, 3, 5, 2] numbers.sort() print(numbers) # Output: [1, 2, 3, 4, 5] |

**Tuple built-in methods**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| 1. count() | Returns the number of times a specified value occurs in a tuple | t.count(3) |
| 1. index() | Searches the tuple for a specified value and returns its position | t.index(5) |

**#Set comprehension:**

* **Set comprehension** is similar to list comprehension, but it creates a set instead of a list. Sets are **unordered collections of unique elements**, so any duplicates will be automatically removed.

**Syntax:** {expression for item in iterable if condition}

**Code:**

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

unique\_squares = {x\*\*2 for x in nums}

print(unique\_squares) # Output: {1, 4, 9, 16, 25}

**Set built-in methods:**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| 1. add() | Adds an element to the set. | s = {1, 2, 3}  s.add(4)  print(s) # Output: {1, 2, 3,4} |
| 1. clear() | Removes all elements from the set. | s = {1, 2, 3}  s.clear()  print(s) # Output: set() |
| 1. copy() | Returns a shallow copy of the set. | s = {1, 2, 3}  new\_s = s.copy()  print(new\_s) # Output: {1, 2, 3} |
| 1. difference() | Returns a set containing the difference between two or more sets. | a = {1, 2, 3}  b = {2, 3, 4}  print(a.difference(b)) # Output: {1} |
| 1. difference\_update() | Removes the items in this set that are also included in another specified set. | a = {1, 2, 3}  b = {2, 3}  a.difference\_update(b)  print(a) # Output: {1} |
| 1. discard() | Removes the specified item. | s = {1, 2, 3}  s.discard(2)  print(s) # Output: {1, 3} |
| 1. intersection() | Returns a set that is the intersection of two or more sets. | a = {1, 2, 3}  b = {2, 3, 4}  print(a.intersection(b)) # Output: {2, 3} |
| 1. intersection\_update() | Updates the set with the intersection of itself and another. | a = {1, 2, 3}  b = {2, 3, 4}  a.intersection\_update(b)  print(a) # Output: {2, 3} |
| 1. isdisjoint() | Returns True if two sets have no elements in common. | a = {1, 2}  b = {3, 4}  print(a.isdisjoint(b)) # Output: True |
| 1. issubset() | Returns True if all items in the set exist in the specified set. | a = {1, 2}  b = {1, 2, 3}  print(a.issubset(b)) # Output: True |
| 1. issuperset() | Returns True if all items in the specified set exist in the original set. | a = {1, 2, 3}  b = {1, 2}  print(a.issuperset(b)) # Output: True |
| 1. pop() | Removes and returns an arbitrary set element. Raises KeyError if the set is empty. | s = {1, 2, 3}  print(s.pop()) # Output: Random element |
| 1. remove() | Removes the specified element. Raises KeyError if not found. | s = {1, 2, 3}  s.remove(2)  print(s) # Output: {1, 3} |
| 1. symmetric\_difference() | Returns a set with elements in either the set or the specified set but not both. | a = {1, 2, 3}  b = {3, 4}  print(a.symmetric\_difference(b)) # Output: {1, 2, 4} |
| 1. symmetric\_difference\_update() | Updates the set with the symmetric difference of itself and another. | a = {1, 2, 3}  b = {3, 4}  a.symmetric\_difference\_update(b)  print(a) # Output: {1, 2, 4} |
| 1. union() | Returns a set containing the union of sets. | a = {1, 2}  b = {2, 3}  print(a.union(b)) # Output: {1, 2, 3} |
| 1. update() | Updates the set with the union of itself and others. | a = {1, 2}  b = {3, 4}  a.update(b)  print(a) # Output: {1, 2, 3, 4} |

**#Dict comprehension:**

* **Dictionary comprehension** is a concise way to create dictionaries. It allows you to construct dictionaries dynamically using a single line of code.

**Syntax:** {key\_expression: value\_expression for item in iterable if condition}

nums = [1, 2, 3, 4]

squares = {x: x\*\*2 for x in nums}

print(squares) # Output: {1: 1, 2: 4, 3: 9, 4: 16}

**Dictionary Built-in Methods**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| dict.clear() | Removes all elements from the dictionary | my\_dict.clear() |
| dict.copy() | Returns a shallow copy of the dictionary | new\_dict = my\_dict.copy() |
| dict.fromkeys() | Creates a dictionary from keys with default value | dict.fromkeys(['a', 'b'], 0) |
| dict.get() | Returns the value for a key | my\_dict.get('key') |
| dict.items() | Returns a view of (key, value) pairs | for k, v in my\_dict.items(): |
| dict.keys() | Returns a view of all keys | my\_dict.keys() |
| dict.pop() | Removes and returns item with key | my\_dict.pop('key') |
| dict.popitem() | Removes and returns the last inserted item | my\_dict.popitem() |
| dict.setdefault() | Returns value if key exists; else inserts it | my\_dict.setdefault('key', 'default') |
| dict.update() | Updates dictionary with another dictionary | my\_dict.update({'new': 1}) |
| dict.values() | Returns a view of all values | my\_dict.values() |

**# What is the difference between is and == in Python?**

In Python:

* == checks if values of two variables are equal.
* is checks if two variables point to the same object in memory.

Example:

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

print(a == b) # True, because contents are same  
print(a is b) # False, because they are two different objects

* is is often used when comparing objects like None:

if my\_var is None:  
 print("Value is None")

Use == for equality of content and is when you care about identity, such as comparing singletons or cached objects.

**#Strings in Python:**

**Quoting Rules**

• Inside single quotes (' '), you cannot use another single quote directly unless escaped:  
 Example: text = 'It\'s a sunny day'  
  
• Inside double quotes (" "), you cannot use another double quote directly unless escaped.  
 Example: text = "She said, \"Hello\""

• Use triple quotes (''' or """) to:  
 - Represent multiline strings  
 - Include both single and double quotes  
 Example: message = '''He said, "It's a beautiful day!"'''

**String Properties:**

• Supports homogeneous and heterogeneous characters:  
 Example: s1 = "abcd" # homogeneous (only letters)  
 s2 = "abc123" # heterogeneous (letters + numbers)• Ordered collection – maintains the order of characters.  
• Allows duplicate characters:  
 Example: a = "aaabbb"• Supports positive and negative indexing: Example: a = "aaabbb"  
 print(a[0]) # Output: a (first character)  
 print(a[-1]) # Output: b (last character**)**• Supports slicing: Example: print(a[::]) # Output: aaabbb (full string)  
 print(a[::-1]) # Output: bbbaaa (reversed string)

**Fetching Characters from a String:**

**Using range() with indexing:**

s = "happy holiday"  
for i in range(len(s)):  
 print(s[i], end="")  
# Output: happy holiday

**Using direct iteration (for-each loop):**

s = "happy holiday"  
for char in s:  
 print(char, end="")  
# Output: happy holiday

**Built-in String Methods.**

|  |  |
| --- | --- |
| **Method** | **Description** |
| capitalize() | Converts first character to uppercase |
| casefold() | Converts string to lowercase (more aggressive than lower()) |
| center(width) | Returns a centered string in a field of given width |
| count(substring) | Returns the number of times a substring appears |
| encode() | Returns encoded version of string |
| endswith(suffix) | Checks if string ends with the specified suffix |
| expandtabs() | Replaces tab characters \t with spaces |
| find(sub) | Returns index of first occurrence of substring, or -1 if not found |
| format() | Formats string using placeholders {} |
| index(sub) | Returns index of first occurrence, raises error if not found |
| isalnum() | Returns True if all characters are alphanumeric |
| isalpha() | Returns True if all characters are alphabetic |
| isdigit() | Returns True if all characters are digits |
| islower() | Returns True if all characters are lowercase |
| isupper() | Returns True if all characters are uppercase |
| isspace() | Returns True if all characters are whitespace |
| istitle() | Returns True if string follows title case |
| join(iterable) | Joins elements of an iterable with the string as separator |
| lower() | Converts string to lowercase |
| upper() | Converts string to uppercase |
| lstrip() | Removes leading whitespace |
| rstrip() | Removes trailing whitespace |
| strip() | Removes both leading and trailing whitespace |
| partition(sep) | Splits the string into 3 parts: before, separator, after |
| replace(old, new) | Replaces occurrences of a substring with another |
| split(sep) | Splits string into list based on separator |
| rsplit(sep) | Splits string from the right |
| splitlines() | Splits string at line breaks |
| startswith(prefix) | Checks if string starts with specified prefix |
| swapcase() | Swaps case of characters |
| title() | Converts first character of each word to uppercase |
| zfill(width) | Pads string with zeros on the left to reach specified width |

**#Conditions.**

**Conditional Statements in Python:**

1. **if Statement**

* Executes a block of code if the condition is true.

Code:

age = 20  
if age >= 18:  
 print("You are an adult.")

1. **if-else Statement**

* Executes one block if the condition is true, another if it's false.

Code:

age = 16  
if age >= 18:  
 print("You are an adult.")  
else:  
 print("You are a minor.")

1. **if-elif-else Chain**

* Used when you have multiple conditions to check.

Code:

score = 75  
  
if score >= 90:  
 print("Grade: A")  
elif score >= 75:  
 print("Grade: B")  
elif score >= 60:  
 print("Grade: C")  
else:  
 print("Grade: F")

1. **Nested if Statements**

* You can place one if statement inside another.

Code:

x = 10  
y = 5  
  
if x > 0:  
 if y > 0:  
 print("Both numbers are positive.")

1. **Ternary (One-line if-else)**

* Compact way to write if-else statements.

Code:

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

6. Comparison & Logical Operators

x = 10  
y = 20  
  
if x > 5 and y > 15:  
 print("Both conditions are true.")

1. **Comparison & Logical Operators**

* You can use comparison (==, !=, >, <, >=, <=) and logical operators (and, or, not) in your conditions.

Code:

x = 10

y = 20

if x > 5 and y > 15:

print("Both conditions are true.")

**#Function**:

* A function is a block of code which is used to perform a specific task.
* A function can defined using the def keyword.
* A function executed only when it call or invoke.

**There are two types of functions.**

* User-defined function.
* built-in library function.
* **User-defined function**: these are the types of function which is defined by the user to any specific task.

# Function definition

def greet(name):

"""This function greets the user with their name."""

return f"Hello, {name}!"

# Function call

message = greet("Alice")

print(message) # Output: Hello, Alice!

* **Built-in function**: Built-in functions are pre-defined functions provided by the Python language that can be used to perform common tasks.

Some Benefits of Using Functions:

Increase Code Readability

Increase Code Reusability

Python has a rich set of built-in functions that are always available and do not require an import statement. These functions are part of the Python Standard Library.  
  
**Commonly Used Built-in Functions:**

|  |  |
| --- | --- |
| **Function** | **Description** |
| print() | Outputs data to the console |
| len() | Returns the number of items in an object |
| type() | Returns the type of an object |
| input() | Takes input from the user |
| int(), float(), str() | Convert between data types |
| range() | Generates a sequence of numbers |
| sum() | Returns the sum of all items in an iterable |
| min(), max() | Returns the smallest/largest item in an iterable |
| sorted() | Returns a sorted list from the given iterable |
| list(), tuple(), dict(), set() | Create new list, tuple, dictionary, set respectively |
| abs() | Returns the absolute value of a number |
| round() | Rounds a number to the nearest integer or specified decimal places |
| enumerate() | Adds a counter to an iterable and returns it as an enumerate object |
| zip() | Combines two or more iterables into tuples |
| any(), all() | Returns True if any/all of the elements in an iterable are true |
| eval() | Executes a Python expression passed as a string |
| help() | Invokes the built-in help system |

**#Function aliasing in Python.**

In Python, we can give another name of the function. For the existing function, we can give another name, which is nothing but function aliasing.

Example:

Code:

def fun(name):

print(f"Hello {name}, welcome to GeeksForGeeks!!!")

cheer = fun #fun alias

print(f'The id of fun() : {id(fun)}')

print(f'The id of cheer() : {id(cheer)}')

fun('Geeks')

cheer('Geeks')

**#Format functions**.

Format function are generally used to substitute values.

syntax: ' '.format()

or

'{}{}'.format(value1,value2)

Code:

# Basic formatting

name = "Alice"

age = 30

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

# Output: My name is Alice and I am 30 years old.

# Positional arguments

print("I have {0} apples and {1} oranges.".format(5, 10))

# Output: I have 5 apples and 10 oranges.

# Named arguments

print("My favorite color is {color}.".format(color="blue"))

# Output: My favorite color is blue.

# Formatting numbers

pi = 3.14159265

print("Pi is approximately {:.2f}".format(pi))

# Output: Pi is approximately 3.14

**#Object oriented programming**

**class:**

* class is a blueprint of an object.
* class is a platform to store states and behaviour of an object.
* class is declared using class keyword.

syntax:- class class name:

implementation

ex: class car:

pass

* class is used to provide empty implementation to a block.

**object:**

* Anything which is present in real-world and also has physical existence is called object.
* In other words any real world entity is called as object.

ex: car, laptop, pen.

* In every object will have certain properties.
  + states
  + behaviour
    - states are used to store some data.
    - behaviour are used to perform some task.

Note: In python object name is considered as class-name.

* + States are considered as variables.
  + Behaviour are considered as function or methods.

Code

# Define a class

class Person:

# Constructor to initialize object attributes

def \_\_init\_\_(self, name, age):

self.name = name # Attribute (instance variable)

self.age = age

# Method to display details

def greet(self):

return f"Hello, my name is {self.name} and I am {self.age} years old."

# Creating objects (instances of the class)

person1 = Person("Alice", 30) # Object 1

person2 = Person("Bob", 25) # Object 2

# Accessing object attributes

print(person1.name) # Output: Alice

print(person2.age) # Output: 25

# Calling object methods

print(person1.greet()) # Output: Hello, my name is Alice and I am 30 years old.

print(person2.greet()) # Output: Hello, my name is Bob and I am 25 years old.

**#Self**

* It is a reference to the object created of our class.
* It is passed whenever we are calling any function.
* Only an object can access or helpful in connecting attributes and methods inside our class
* When we have a functions inside a class, the first parameter is always self.

**# Constructor or \_\_init\_\_().**

* Constructor is asset of statement or block of code which is used to initialization.
* Constructor is also called as a special function which gets executed at the time of object creation.

Code:

class Car:

# Constructor method (\_\_init\_\_ is called automatically when creating an object)

def \_\_init\_\_(self):

print("It is a car")

# Create an object 'c' of class Car

c = Car() # This automatically calls Car.\_\_init\_\_(c)

**#\_\_str\_\_()**

* It is a special function which returns the strings representation of an object.
* In python, when we print the reference Variable it implicitly calls \_\_str\_\_() function.
* To provide implementation when we print reference Variable, we should always override \_\_str\_\_() function.

Code:

class Car:

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

# Define the string representation of the object

def \_\_str\_\_(self):

return f"{self.brand} {self.model}"

# Create an object

car = Car("Toyota", "Corolla")

# Now print() and str() will use the \_\_str\_\_() method

print(car) # Output: Toyota Corolla

print(str(car)) # Output: Toyota Corolla

**# Encapsulation.**

* The process of binding a grouping variables and functions inside a class is called as encapsulation.
* We achieve encapsulation using private variables and public function (setter and getter)
* Setter method is always is used to set/initialize the data.
* Getter method is always is used to get/return the data.

Code:

class BankAccount:

def \_\_init\_\_(self, owner, balance):

self.owner = owner # Public attribute

self.\_\_balance = balance # Private attribute (notice the double underscore)

# Public method to access balance (getter)

def get\_balance(self):

return self.\_\_balance

# Public method to deposit money

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

print(f"Deposited {amount}. New balance: {self.\_\_balance}")

else:

print("Invalid deposit amount.")

# Public method to withdraw money

def withdraw(self, amount):

if 0 < amount <= self.\_\_balance:

self.\_\_balance -= amount

print(f"Withdrew {amount}. New balance: {self.\_\_balance}")

else:

print("Insufficient funds or invalid amount.")

# Example usage

account = BankAccount("Alice", 1000)

# Access public attribute

print(account.owner) # Alice

# Access private attribute directly (will raise an error)

# print(account.\_\_balance) # AttributeError

# Use public methods to access and modify private data

print(account.get\_balance()) # 1000

account.deposit(500) # Deposited 500. New balance: 1500

account.withdraw(200) # Withdrew 200. New balance: 1300

# Attempting invalid operations

account.withdraw(5000) # Insufficient funds or invalid amount.

**# Inheritance.**

* **Inheritance** is an **OOP (Object-Oriented Programming)** concept where a child class acquires the properties and behaviors of a parent class.
* It is referred to as an **IS-A relationship**.
* In Python, a child class can inherit **variables, methods, and constructors** from a parent class.

1. **Single Inheritance**

A single child class inherits from a single parent class. This is the simplest form of inheritance.

Cc

Pc

Code

# Parent class

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

print(f"{self.name} makes a generic sound.")

# Child class inheriting from Animal

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name) # Initialize parent class

self.breed = breed

def speak(self):

print(f"{self.name} (a {self.breed}) says Woof!")

# Create objects

animal = Animal("Generic Animal")

dog = Dog("Buddy", "Golden Retriever")

animal.speak() # Output: Generic Animal makes a generic sound.

dog.speak() # Output: Buddy (a Golden Retriever) says Woof!

1. **Hierarchical Inheritance**

In hierarchical inheritance, multiple child classes inherit from a single parent class.

Pc

Cc

Cc

Cc

Code

# Base class

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

print(f"{self.name} makes a sound.")

# Derived classes (hierarchical inheritance)

class Dog(Animal):

def speak(self):

print(f"{self.name} says Woof!")

class Cat(Animal):

def speak(self):

print(f"{self.name} says Meow!")

class Bird(Animal):

def speak(self):

print(f"{self.name} says Chirp!")

# Create objects

dog = Dog("Buddy")

cat = Cat("Whiskers")

bird = Bird("Tweety")

dog.speak() # Output: Buddy says Woof!

cat.speak() # Output: Whiskers says Meow!

bird.speak() # Output: Tweety says Chirp!

1. **Multilevel Inheritance**

In multilevel inheritance, a child class inherits from a parent class, which in turn inherits from another parent class.

GrandPc

Pc

Cc

Code

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def eat(self):

print(f"{self.name} is eating.")

class Mammal(Animal):

def \_\_init\_\_(self, name, has\_fur):

super().\_\_init\_\_(name) # Initialize parent (Animal)

self.has\_fur = has\_fur

def give\_birth(self):

print(f"{self.name} gives birth to live young.")

class Dog(Mammal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name, has\_fur=True) # Initialize parent (Mammal)

self.breed = breed

def bark(self):

print(f"{self.name} says Woof!")

# Create an object of Dog

dog = Dog("Buddy", "Golden Retriever")

dog.eat() # Output: Buddy is eating. (from Animal)

dog.give\_birth() # Output: Buddy gives birth to live young. (from Mammal)

dog.bark() # Output: Buddy says Woof! (from Dog)

1. **Multiple Inheritance**

In multiple inheritance, a child class inherits from more than one parent class.

Pc

Pc

Cc

Code

# Parent classes

class A:

def display(self):

print("Class A")

class B:

def show(self):

print("Class B")

# Child class inheriting from A and B

class C(A, B):

pass

# Create an object of C

obj = C()

obj.display() # Output: Class A (from A)

obj.show() # Output: Class B (from B)

1. **Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance. It allows for complex relationships between classes.

A

C

B

D

Code

# Base class

class A:

def display(self):

print("Class A")

# Hierarchical Inheritance (B and C inherit from A)

class B(A):

def show(self):

print("Class B")

class C(A):

def show(self):

print("Class C")

# Multiple Inheritance (D inherits from B and C)

class D(B, C):

pass

# Create an object of D

d = D()

d.display() # Output: Class A (from A)

d.show() # Output: Class B (due to MRO)

**# Polymorphism.**

* The word "polymorphism" means "many forms"
* In python the ability of a function to behave differently is called polymorphism.
* The process of the functions exhibiting or showing different behaviour is called as polymorphism.
* The method overloading and method overriding are example of polymorphism.

In python we achieve polymorphism by.

* Method overloading
* Method overriding
* Operator overloading

Code

# Parent class

class Animal:

def speak(self):

raise NotImplementedError("Subclasses must implement this method")

# Child classes

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

class Cow(Animal):

def speak(self):

return "Moo!"

# Polymorphism in action

def animal\_sound(animal):

# Calls the appropriate method based on the object type

print(animal.speak())

# Creating objects of different types

dog = Dog()

cat = Cat()

cow = Cow()

# Same function used for different object types

animal\_sound(dog) # Output: Woof!

animal\_sound(cat) # Output: Meow!

animal\_sound(cow) # Output: Moo!

**#Method overriding**

* The process of inheriting the method and changing the implementation of the inherited method is called as method overriding.
* Using method overriding the method names should be always same but change in parameters.

**#Method overloading**

* It is the process of invoking/calling a function in different ways is called as method overloading.

**# Abstraction**

* The process of hiding all the unnecessary implementation and showing only the important functionalities to the user with the help of abstract class is called as abstraction.
* In simple words hiding implementation and showing important functionalities to the user is called as abstraction.

**Rules to achieve abstraction.**

* Abstract class or interface with abstract methods.
* Is-A relationship (Inheritance)
* Method overriding
* Upcasting.

Code:

from abc import ABC, abstractmethod

# Abstract Class

class Animal(ABC):

@abstractmethod

def make\_sound(self):

"""Abstract method - must be implemented by subclasses."""

pass

def sleep(self):

"""Concrete method - can be inherited directly."""

print("Sleeping...")

# Concrete Subclasses

class Dog(Animal):

def make\_sound(self):

print("Woof!")

class Cat(Animal):

def make\_sound(self):

print("Meow!")

# Example Usage

# animal = Animal() # Cannot instantiate abstract class

dog = Dog()

cat = Cat()

dog.make\_sound() # Output: Woof!

cat.make\_sound() # Output: Meow!

dog.sleep() # Output: Sleeping...

cat.sleep() # Output: Sleeping...

**#Exceptional handling**

Exception:

* It is an event or an issue occurs during program execution.
* These exceptions can be handled by the exception handlers.
* Some exceptions can be handled, some exception cannot be handled.

**What is exception handlers:**

* It is a part of a program executed only when exception occurs.
* To handle exceptions, there are 3 exception handlers named:
* try → code that might cause an error
* except → handles the error if it happens
* finally → always runs (cleanup or closing actions)

code:  
 try:

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

result = 10 / num

except ZeroDivisionError:

print("Cannot divide by zero!")

except ValueError:

print("Please enter a valid number.")

finally:

print("This always runs.")

* Code inside try runs first.
* If an error occurs, Python jumps to the matching except block.
* The finally block runs **no matter what**, even if there’s an error.

**#File handling / File programming**

* A process of writing a data into a file and reading the data from a file is called as file handling.

**Different modes used in file handling**

‘W’ - writing only

‘R’ - reading only

‘A’ - append only

**Import and functions used in file handling.**

Open() : It is used to open the file in specified mode.

Syntax: open(‘filename’,’mode’)

Close(): It is used to close the specified file.

Syntax: fileobject.close()

Write(): it is used to write data into file.

Syntax: fileobject.write(data)

Read(): it is used to read the data from file.

Syntax: fileobject.read(data)

**Import and variables used in file handling:**

**Name:** return the file name.

**Mode:** returns the mode of specified file.

**Closed:** check if the file closed or not.

**Note:** when we open a file in write mode(‘w’)

there are 2 possibilities.

1.**file present:** If the file is already present, open() opens the same file in write mode and already existing data is last.

2.**file not present:** If the file is not present, open() creates a new file and open in write mode.

**Program to write data into a file.**

F = open(‘Employee : txt’, ’w’)

Data = ‘Good morning’

f.write(data)

f.close()

**ex:**

f = open(‘Employee.txt’ , ‘w’)

f.write(‘Good morning’)

f.close()

**Note:** when we open a file in read mode (‘r’) there are 2 possibilities.

1. **File present:** If the file is already present open()opens the same file in read mode and data can be read successfully.

2. **File not present:** If the file is not present, open() throws an exception called as file not found error.

**Program to read data from file.**

F = open(‘Employee.txt’, ‘r’)

Print(f.read())

f.close()

f = open(‘Pen.txt’,’w’)

f.write(“Good Moring”)

f.close()

f = open(‘pen.txt’,’r’)

Print(f.read()) #Good moring.

f.close()

f = open(“Pen.txt”,’w’)

print(f.name) #Pen.txt

print(f.mode) #’W’

print(f.closed) #’False’

f.close()

print(f.closed) #True.

**Program to append the data into a file.**

F = open(‘demo.txt’,’w’)

F.write(‘Hello’)

F.close()

F = open(‘Demo.txt’,’a’)

f.write(‘Tom’)

f.close()

Note: In normal file handling it is possible to store only string data into a file and we can’t write or store list, set, tuple, dictionary etc into a file we have to use Json.

**Json (JavaScript object Notation)**

* Json is a data format using which we can store any type of data into a file.
* To work with ‘Json’ we have to import a built-in module or predefined module called as “json”

**Program:**

Import json

F = open(‘demo.txt’,’w’) #open the file in write mode.

Sdata = json\_dumps([30,40,50]) # convert data into

f.write(sdata) #store json data into file.

f.close()

# read()

Import json

f.open(‘demo.txt’,’r’) #open file in read mode.

Data = json.loads(sdata) #convert the json data into original data.

Print(data) #[30,40,50] #orignial data.

f.close()

**emp.txt**

**i**mport json

f = open(‘Emp.txt’,’w’)

data = json.dumps([1:’hi’,2:’hello’])

f.write(data)

f.close()

f = open(‘Emp.txt’,’r’)

sdata = json.loads(f.read())

print(sdata)

f.close()

**File Handling in Python:**

File handling allows Python programs to read from and write to files on the system. Python provides built-in functions and methods to handle files efficiently.

**1. File Basics**

- Files are used to store data permanently.  
- Python can handle various types of files such as:  
- Text files (.txt)  
- CSV files (.csv)  
- JSON files (.json)  
- Binary files (.bin)  
- File handling operations include: Create, Read, Write, Append, and Delete.

**2. Opening Files**

To work with a file, you must first open it using the built-in open() function:  
file = open("filename.txt", "mode")

**Common File Modes:**

|  |  |
| --- | --- |
| **Mode** | **Description** |
| r | Read (default mode) |
| w | Write (overwrites existing content) |
| a | Append (adds to the end) |
| x | Create a new file (fails if exists) |
| b | Binary mode |
| t | Text mode (default) |

**3. Reading Files**

**Read content from a file using methods:**  
file = open("file.txt", "r")  
content = file.read()  
print(content)  
file.close()

**Reading Methods:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| read() | Reads entire content as a string |
| readline() | Reads one line at a time |
| readlines() | Returns a list of lines |

**4. Writing Files**

**To write into a file:**  
  
file = open("file.txt", "w")  
file.write("Hello, world!")  
file.close()

- If the file doesn't exist, it creates one.  
- If it exists, it overwrites existing content.

**5. Editing Files**

**To update or edit a file:**  
  
file = open("file.txt", "r")  
lines = file.readlines()  
file.close()  
  
lines[0] = "Updated line\n"  
  
file = open("file.txt", "w")  
file.writelines(lines)  
file.close()

**6. Working with Different File Extensions**

**Text Files:**  
open("example.txt", "r")

**CSV Files:**  
import csv  
with open("data.csv", newline='') as file:  
 reader = csv.reader(file)  
 for row in reader:  
 print(row)

**JSON Files:**  
import json  
with open("data.json", "r") as file:  
 data = json.load(file)  
 print(data)

**Binary Files:**  
file = open("image.png", "rb")  
binary\_data = file.read()  
file.close()

**7. With Statement (Context Manager)**

Using "with" ensures the file is automatically closed, even if an error occurs.  
  
with open("file.txt", "r") as file:  
 content = file.read()  
 print(content)

Benefits of "with" statement:

- Cleaner and more readable syntax

- Auto resource management (closes the file automatically)

**Summary Table**

|  |  |
| --- | --- |
| Task | Method / Function |
| Open a file | open() |
| Read content | read(), readline(), readlines() |
| Write content | write(), writelines() |
| Append content | Open file in 'a' mode |
| Close a file | file.close() |
| Auto-close file | with open(...) as ...: |

**Advance:**

**# Decorator.**

* It is a function that takes another function as argument and returns a function.
* In other words one function is providing additional functionality to another function is called as decorates.

syntax: @function\_name.

Rules to achieve decorator function.

* + - create a nested function.
    - The outer function should accept one parameter which stores

the address of function to the decorator.

* + - The outer function should return address of the inner function.
    - Define the decorator logic inside inner function.

Code:

def decorator\_name(func):

def wrapper(\*args, \*\*kwargs):

# Add functionality before the original function call

result = func(\*args, \*\*kwargs)

# Add functionality after the original function call

return result

return wrapper

@decorator\_name

def function\_to\_decorate():

# Original function code

pass

code.

# A simple decorator function

def decorator(func):

def wrapper():

print("Transaction Initiated")

func()

print("Transaction completed.")

return wrapper

# Applying the decorator to a function

@decorator

def hello():

print("Executing all the steps of Transaction")

#hello1 = decorator(hello)

hello()

**# Iterator.**

* Iterator are functions which is used to traverse objects or elements from a collection or (Group of data).

**Important methods used in iterators.**

* iter()- it is used to return iterable object address.
* next()- it is used to return the object/element.

code.

mytuple = ("apple", "banana", "cherry")

myit = iter(mytuple)

print(next(myit))

print(next(myit))

print(next(myit))

#second

mystr = "banana"

myit = iter(mystr)

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

**# Generator**.

* Generator are functions which is used to generate a sequence of values at one time.

**syntax**: def function\_name():

yield data.

Code:

# A generator function that yields 1 for first time,

# 2 second time and 3 third time

def fun():

yield 1

yield 2

yield 3

# Driver code to check above generator function

for val in fun():

print(val)

**#Special Function:**

1. **What is Lambda() function in python?**

* A **lambda** function is a small anonymous function.
* An anonymous functions means that the function is without a name.
* A lambda function can take any number of arguments, but can only have one expression.

**Syntax:** lambda arguments: expression

Code:

# Add 10 to argument a, and return the result:

x = lambda a : a + 10

print(x(5)) #op 15

# Multiply argument a with argument b and return the result:

x = lambda a, b : a \* b

print(x(5, 6)) #op 30

# Summarize argument a, b, and c and return the result:

x = lambda a, b, c : a + b + c

print(x(5, 6, 2)) #op 13

# Example: Check if a number is even or odd

check = lambda x: "Even" if x % 2 == 0 else "Odd"

print(check(4)) #op Even

print(check(7)) #op odd

# Example: Perform addition and multiplication in a single line

calc = lambda x, y: (x + y, x \* y)

res = calc(3, 4)

print(res) #op 7,12

# Example: Filter even numbers from a list

n = [1, 2, 3, 4, 5, 6]

even = filter(lambda x: x % 2 == 0, n)

print(list(even)) #op [2, 4, 6]

# Example: Double each number in a list

a = [1, 2, 3, 4]

b = map(lambda x: x \* 2, a)

print(list(b)) #op [2, 4, 6, 8]

1. **What is the map() function in Python?**

The map() function applies a given function to all items in an iterable and returns a map object (which is an iterator). It’s a clean way to apply transformations to a list or tuple without writing a loop.

Syntax:

map(function, iterable)

Example:

numbers = [1, 2, 3, 4]  
squared = map(lambda x: x\*\*2, numbers)  
print(list(squared)) # [1, 4, 9, 16]

map() is often used when you need to apply the same function to every element. It's efficient and can be combined with other functional programming tools like filter() or reduce(). It keeps code short and readable.

1. **What is the filter() function in Python?**

The filter() function is used to filter elements from an iterable based on a condition. It returns an iterator containing only the elements for which the function returns True.

Syntax: filter(function, iterable)

Example:

numbers = [1, 2, 3, 4, 5]  
even = filter(lambda x: x % 2 == 0, numbers)  
print(list(even)) # [2, 4]

It is useful when you want to keep elements that meet certain criteria and discard the rest. filter() makes your code more expressive and avoids manual loops with conditional checks.

1. **What is the reduce() function in Python?**

The reduce() function from the functools module applies a function to the items of a sequence and reduces it to a single value. It processes the sequence pairwise.

Syntax:

from functools import reduce  
reduce(function, iterable)

Example:

from functools import reduce  
numbers = [1, 2, 3, 4]  
result = reduce(lambda x, y: x + y, numbers)  
print(result) # 10

You can use reduce() for operations like summing, multiplying, or combining values. It's powerful, but sometimes less readable than a loop or sum(). Use it when you need to process a list into a single result step by step.

**Lambda Built-in Function.**

* In Python, lambda is a built-in anonymous function that allows you to create small, one-line functions without using the def keyword.

**Syntax: lambda arguments: expression**

* It can take any number of arguments but must have only one expression, which is evaluated and returned.

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case** | **Description** | **Syntax** | **Example & Output** |
| map() | Applies a function to all items in an iterable. | map(lambda x: x \*\* 2, numbers) | numbers = [1, 2, 3, 4, 5] squared = list(map(lambda x: x \*\* 2, numbers)) print(squared) # Output: [1, 4, 9, 16, 25] |
| filter() | Filters elements based on a condition. | filter(lambda x: x % 2 == 0, numbers) | numbers = [1, 2, 3, 4, 5, 6] even\_numbers = list(filter(lambda x: x % 2 == 0, numbers)) print(even\_numbers) # Output: [2, 4, 6] |
| sorted() | Sorts a list based on custom criteria. | sorted(students, key=lambda x: x[1]) | students = [('John', 80), ('Jane', 90), ('Dave', 75)] sorted\_students = sorted(students, key=lambda x: x[1]) print(sorted\_students) # Output: [('Dave', 75), ('John', 80), ('Jane', 90)] |
| reduce() | Reduces a sequence to a single value. | reduce(lambda x, y: x \* y, numbers) | from functools import reduce numbers = [1, 2, 3, 4, 5] product = reduce(lambda x, y: x \* y, numbers) print(product) # Output: 120 |
| max() / min() | Finds the maximum or minimum based on a custom key. | max(points, key=lambda x: x[1]) | points = [(2, 3), (5, 1), (1, 7)] max\_point = max(points, key=lambda x: x[1]) print(max\_point) # Output: (1, 7) |
| any() / all() | Checks conditions in a list. | all(lambda x: x % 2 == 0 for x in numbers) | numbers = [2, 4, 6, 8] print(all(lambda x: x % 2 == 0 for x in numbers)) # Output: True print(any(lambda x: x > 5 for x in numbers)) # Output: True |
| List Sorting | Sorts a list based on length or other custom criteria. | sorted(words, key=lambda x: len(x)) | words = ['apple', 'banana', 'kiwi', 'grape'] sorted\_words = sorted(words, key=lambda x: len(x)) print(sorted\_words) # Output: ['kiwi', 'grape', 'apple', 'banana'] |

Multithreading:

* It is used to perform multiple tasks concurrently (multitasking)
* Good for I/O bound tasks like reading files or fetching data from APIs.
* threading.Thread(target = my\_function)  
    
  Code:

import threading

import time

def walk\_dog(first):

time.sleep(8)

print(f"You finish walking {first}")

def take\_out\_trash():

time.sleep(2)

print("You take out the trash")

def get\_mail():

time.sleep(4)

print("You get the mail")

chore1 = threading.Thread(target = walk\_dog,args=("Scooby",))

chore1.start()

chore2 = threading.Thread(target = take\_out\_trash)

chore2.start()

chore3 = threading.Thread(target = get\_mail)

chore3.start()

chore1.join()

chore2.join()

chore3.join()

print("All chores are complete!")  
  
output:  
You take out the trash

You get the mail

You finish walking Scooby

All chores are complete!

**What is the Global Interpreter Lock (GIL) in Python?**

**Global Interpreter Lock (GIL) — Key Points**

1. **Definition:**  
   The GIL is a lock in Python that allows **only one thread** to execute Python code at a time.
2. **Purpose:**  
   It keeps Python’s memory and objects **safe from corruption** by preventing multiple threads from changing them simultaneously.
3. **Effect on threads:**  
   Even if you use multiple threads, **only one thread runs Python code at any given moment**.
4. **CPU-bound limitation:**  
   For **CPU-heavy tasks** (like complex calculations), the GIL **blocks true parallel execution**, so threads don’t make it faster.
5. **I/O-bound benefit:**  
   For **I/O tasks** (like reading files, making API calls, or downloading data), threads can still be **useful** because they spend time waiting.
6. **Workarounds for real parallelism:**
   * Use the **multiprocessing** module (creates separate processes, each with its own GIL).
   * Use **NumPy**, **C extensions**, or **JIT compilers (like PyPy)** which can bypass or minimize GIL effects.
7. **Applies to CPython only:**  
   The GIL exists in **CPython**, the main Python interpreter — not necessarily in others like **Jython** or **IronPython**.

**What does if \_\_name\_\_ == "\_\_main\_\_": mean?**

* Every Python file has a special variable called \_\_name\_\_.
* When you **run the file directly**, Python sets \_\_name\_\_ = "\_\_main\_\_".
* When you **import the file** into another file, \_\_name\_\_ becomes the **file’s name** instead.
* So, the code inside

def main():

print("This runs only when the file is executed directly.")

if \_\_name\_\_ == "\_\_main\_\_":

main()  
  
If you run this file → it prints the message.  
If you import it into another file → it **won’t print anything** automatically.

**72. How does Python's Global Interpreter Lock (GIL) affect multithreading and multiprocessing?**

The Global Interpreter Lock (GIL) is the Python interpreter's mechanism to ensure that only one thread can execute Python bytecode at a time.

1. In the case of multithreading, Python threads cannot take full advantage of multiple CPU cores to perform parallel processing. While threads can be helpful for I/O-bound tasks, they could be better suited for CPU-bound tasks that require intensive computation. In these cases, multiprocessing can be a better option.
2. Multiprocessing involves running multiple instances of the Python interpreter in parallel, each with its own GIL. It allows for parallel processing on multi-core machines, as each process can utilize a separate CPU core.

**73. What are some ways to improve a Python application's performance?**

There are several ways to improve the performance of a Python application:

1. **Use efficient algorithms and data structures:** Inefficient algorithms or data structures can lead to unnecessary computational overhead, slowing down the performance of your application.
2. **Optimise code with profiling**: Profiling measures your code's performance to identify bottlenecks and areas to be optimized. Python has several built-in profiling tools, such as cProfile and time, to help you identify performance issues.
3. **Utilise built-in functions and libraries:**Python has an extensive standard library with many built-in functions and modules optimized for performance.
4. **Implement parallelism with multiprocessing**: Python's Global Interpreter Lock (GIL) can limit the performance of multi-threaded programs. However, multiprocessing takes advantage of multiple CPU cores for parallel processing.

**74. What is the purpose of the "asyncio" library in Python?**

The "asyncio" library is a built-in library in Python that provides an infrastructure for writing asynchronous, concurrent, and parallel code. It is designed to help developers write highly efficient and scalable network servers and clients. Asyncio enables you to write code that can perform I/O operations without blocking the main thread of execution, which can significantly improve the performance and responsiveness of your applications.

**75. How can you optimize memory usage in a Python application?**

Some ways to optimize memory usage in a Python application:

1. **Use generators and iterators:** Generators and iterators can help reduce memory usage by allowing you to process data from one element simultaneously rather than loading the entire dataset into memory at once.
2. **Use built-in functions and modules**: Built-in functions and modules like "itertools" and collections can help optimize memory usage by providing efficient algorithms and data structures optimized for memory usage.
3. **Avoid unnecessary copying of data:** Python objects are often passed by reference, which can result in excessive data copying. To avoid this, you can use immutable objects like tuples or copy() functions to create shallow rather than deep copies.
4. **Use lazy loading:** Lazy loading is a technique in which data is loaded into memory only when needed rather than the entire dataset. It can help reduce memory usage and improve performance.

**################################### - END - ##################################**

* + - * Magic methods.
      * Threading.
      * Functional programming.
      * Expressions.
      * Closures.
      * Regex.
      * what is multithreading in python.
      * what is multiprocessing in python.