Python Basic my

Author : Arnab Dana

▼ Resource

1. Python Tutorial | Learn Python Programming (geeksforgeeks.org)

▼ Content

1. **Python Input/Output**
2. **Python Data Types**
3. **Python List**
4. **Python Tuple**
5. **Some Questions**
6. **Python Sets**
7. **Python Dictionaries**
8. **Python Arrays**
9. **Python String**
10. **Python Variables**
11. **Python Operators**
12. **Python Control Flow**
13. **Generators in Python**
14. **Python Functions**
15. **Python Built in Functions**
16. **Python Class**
17. **Decorators in Python**
18. **Error Handling and Debugging**
19. **Python Numpy**

I am providing my Python notes for you to arrange in a structured, organized format. Please follow these guidelines:

1. **Formatting & Syntax**: Ensure all content is formatted properly and syntax errors are corrected.
2. **Topic Order**: Organize the notes by topic, ensuring a logical flow for learning.
3. **Examples**: For topics with existing examples, format them in proper code blocks. If a topic lacks an example, add a simple use case in a code block.
4. **Index**: Create an index at the top, listing all topics for easy navigation.

Thank you

Summary of python Tuple Characteristics:

▼ 1. Python Input/Output

```
# Basic Input/Output
val = input("Enter your value: ")
print(val)

# Converting input type
int(input())
float(input())
str(input())
```

Taking Multiple Inputs from User in Python

1. **Using `split()` method**

```
# Taking two values separated by space
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)
print("First number is {} and second number is {}".format(x, y))
```

```
# Taking multiple values and converting to a list of integers
z = list(map(int, input("Enter multiple values: ").split()))
print("List of students: ", z)

# Example input/output:
# Enter multiple values: 5 10 15 20 25
# List of students: [5, 10, 15, 20, 25]
```

1. **Using List Comprehension**

```
# Taking two integer values in a single line
x, y = [int(x) for x in input("Enter two values: ").split()]

# Taking multiple values separated by commas
z = [int(x) for x in input("Enter multiple values separated by commas: ").
split(",")]
print("List of numbers is: ", z)

# Taking multiple values separated by spaces, limiting to 3 elements
x = [val for val in input("Enter multiple values separated by spaces: ").sp
lit(" ", 2)]
print("List of numbers:", x)

# Example input/output:
# Enter multiple values separated by spaces: 52 52 52 63 4 15 26 3
# List of numbers: ['52', '52', '52 63 4 15 26 3']
```

Output Formatting

```
# Basic print statement with multiple variables
name = "Alice"
age = 25
print("Hello, my name is", name, "and I am", age, "years old.")

# Using format() to print variables
a, b = 10, 1000
print('The value of a is {} and b is {}'.format(a, b))
```

```
# Concatenating strings in print
print('GeeksforGeeks is a Wonderful ' + 'Website.')

# Using sep parameter in print
print(5, "aaa", "cc", sep="-")  # Output: 5-aaa-cc

# Using end parameter in print
print("aaa", end=" ")
print("bbb")  # Output: aaa bbb

# Formatting floating-point numbers with specified width and precision
print("first one:{0:10.2f}".format(123.45678))  # Output: first one:    123.4
6
```

```
# Rounding a floating-point number to 2 decimal places
val = round(float(12345.6789), 2)
print(val)  # Output: 12345.68
```

Summary of Python Input/Output Characteristics:

- **Input Functions**:
  - `input()`:
    - Used to take user input from the console.
    - Always returns the input as a string, even if the user enters numbers.
    - Can accept a prompt message as an argument to guide the user.
    - Example: `user_input = input("Enter your name: ")`
- **Output Functions**:
  - `print()`:
    - Used to display output to the console.
    - Can accept multiple arguments and prints them in sequence.
    - Supports various data types, including strings, integers, floats, and more.

        - Automatically adds a newline at the end of the output unless specified otherwise.
        - Can also format output using f-strings or other formatting methods (e.g., `%`, `str.format()`).
        - Example: `print("Hello, World!")`
- **File Input/Output**:
    - `open()`:
        - Used to open a file for reading, writing, or appending.
        - Supports different modes like `'r'` (read), `'w'` (write), `'a'` (append), `'rb'` (read binary), etc.
        - Returns a file object, which can be used to read or write the file.
        - Example: `file = open("file.txt", "r")`
    - `read()`:
        - Reads the content of a file.
        - Can read the entire file or a specific number of characters.
        - Example: `content = file.read()`
    - `write()`:
        - Writes content to a file.
        - If the file is opened in write mode (`'w'`), it will overwrite existing content.
        - Example: `file.write("Hello, World!")`
    - `close()`:
        - Closes the file after reading or writing.
        - Always ensure to close the file to release system resources.
        - Example: `file.close()`
    - `with` **statement**:
        - Ensures the file is automatically closed after its block of code has been executed, even in case of an error.
        - Example:

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

- **Formatted Output**:
  - `f-string`:
    - Provides a convenient way to embed expressions inside string literals, introduced in Python 3.6.
    - Example: `name = "Alice"; print(f"Hello, {name}!")`
  - `str.format()`:
    - Allows formatting of strings by placing placeholders `{}` in the string and using the `.format()` method to replace them.
    - Example: `"Hello, {}".format(name)`
  - `%` **formatting**:
    - An older method of string formatting, often referred to as the "modulo" operator.
    - Example: `"Hello, %s" % name`
- **Error Handling in I/O**:
  - `try...except` **block**:
    - Used for handling errors during input or output operations.
    - Ensures the program does not crash if an error occurs (e.g., file not found).
    - Example:

```
try:
    file = open("file.txt", "r")
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()
```

- **Binary Input/Output**:

- `open()` **with binary mode**:
  - Used to read or write binary files (e.g., images, audio files).
  - Modes like `'rb'` (read binary) and `'wb'` (write binary) are used.
  - Example:

```
with open("image.png", "rb") as file:
    data = file.read()
```

▼ 2. Python Data Types

Here's a formatted overview of Python's built-in data types:

1. Numeric Types

- **int:** Whole numbers (e.g., -2, 0, 10, 1000).

```
age = 30
```

- **float:** Floating-point numbers (e.g., 3.14, -2.5, 1.0e-4).

```
pi = 3.14159
print(round(pi, 2))  # Precision: 3.14
```

- **complex:** Complex numbers with real and imaginary parts (e.g., 3 + 2j, -1.5 - 4j).

```
z = 2 + 3j
```

2. Text Type

- **str:** String of characters (e.g., "Hello, world!", "Python", "123").

```
name = "Bard"
```

3. Sequence Types

- **list:** Ordered, mutable collections of items.

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

- **tuple:** Ordered, immutable collections of items.

```
coordinates = (10, 20)
```

- **range:** Represents a sequence of numbers, often used in loops.

```
for i in range(5):
    print(i)  # Output: 0, 1, 2, 3, 4
```

Syntax: `range(start, stop, step)`

4. Boolean Type

- **bool:** Represents True or False values.

```
is_valid = True
```

5. Set Types

- **set:** Unordered collections of unique items.

```
unique_letters = {"a", "b", "a", "c"}  # Duplicates are removed
print(unique_letters)  # Output: {'a', 'b', 'c'}
```

- **frozenset:** Immutable version of a set.

```
frozen_set = frozenset({1, 2, 3})
print(frozen_set)  # Output: frozenset({1, 2, 3})
```

6. Mapping Type

- **dict:** Unordered collections of key-value pairs.

```
person = {"name": "Alice", "age": 30, "city": "New York"}
print(person["name"])  # Output: Alice
person["email"] = "alice@example.com"  # Add new key-value pair
```

```
print(person)  # Output: {'name': 'Alice', 'age': 30, 'city': 'New York', 'email': 'alice@example.com'}
```

7. Binary Types

i. bytes

```
data = b"Hello"
print(data)  # Output: b'Hello'
```

ii. bytearray

```
mutable_data = bytearray(b"Hello")
mutable_data[0] = 74  # Changes 'H' to 'J' (ASCII 74)
print(mutable_data)  # Output: bytearray(b'Jello')
```

iii. memoryview

```
data = bytes([10, 20, 30])
view = memoryview(data)
print(view[1])  # Output: 20 (accesses the second byte)
```

These concise examples show the basics of each type.

Key Points

- **Dynamic Typing**: Python is dynamically typed, so variable types are inferred automatically.
- **Type Checking**: Use `type()` to check a variable's type.
- **Data Type Operations**: Each data type has specific operations and methods.
- **Choosing Data Types**: Selecting the appropriate type improves efficiency and clarity in programming.

Summary of Python Datatype Characteristics:

- **Integers (`int`)**:

    - **Whole Numbers**: Represents whole numbers without a fractional part.
    - **Immutable**: Cannot be modified after creation.
    - **Supports Arithmetic Operations**: Can be used in addition, subtraction, multiplication, division, etc.
    - **Unlimited Size**: Can represent large numbers, limited only by available memory.
- **Floating Point Numbers (`float`)**:
    - **Decimal Numbers**: Represents numbers with a fractional part.
    - **Immutable**: Cannot be modified after creation.
    - **Supports Arithmetic Operations**: Can be used in mathematical operations.
    - **Limited Precision**: Precision can be lost with very large or very small numbers due to floating-point representation.
- **Strings (`str`)**:
    - **Immutable**: Cannot be modified after creation.
    - **Ordered**: Elements are in a specific order, and you can access them by indexing.
    - **Heterogeneous**: Can contain any Unicode character, including numbers and symbols.
    - **Supports Slicing and Concatenation**: Can be sliced to extract parts and concatenated to combine.
    - **Iterable**: Can be looped over like a list.
- **Booleans (`bool`)**:
    - **True/False Values**: Represents truth values, either `True` or `False`.
    - **Immutable**: Cannot be modified after creation.
    - **Used for Conditional Logic**: Commonly used in control flow (e.g., `if` statements, loops).
- **Lists (`list`)**:
    - **Ordered**: Elements are stored in a specific order.

    - **Mutable**: Can be modified (e.g., add/remove elements).
    - **Allow Duplicates**: Can contain repeated elements.
    - **Heterogeneous**: Can store different data types.
    - **Indexed**: Supports indexing to access elements by position.
- **Tuples (`tuple`)**:
    - **Ordered**: Elements are stored in a specific, predictable order.
    - **Immutable**: Cannot be modified after creation.
    - **Allow Duplicates**: Can contain repeated elements.
    - **Heterogeneous**: Can store elements of different data types.
    - **Indexed**: Supports indexing to access elements by position.
    - **Faster than Lists**: More efficient for read-only operations.
- **Dictionaries (`dict`)**:
    - **Unordered**: Elements are stored as key-value pairs and do not maintain a specific order.
    - **Mutable**: Can be modified by adding, removing, or changing key-value pairs.
    - **Key-Value Pairs**: Each element has a unique key, which is used to access its corresponding value.
    - **Keys are Unique**: Keys must be immutable (e.g., strings, numbers, tuples), while values can be any datatype.
    - **Efficient for Lookup**: Optimized for retrieving values based on their keys.
- **Sets (`set`)**:
    - **Unordered**: Elements are not stored in a specific order.
    - **Mutable**: Can be modified by adding or removing elements.
    - **Unique Elements**: Does not allow duplicates.
    - **Heterogeneous**: Can store elements of different data types.
    - **Supports Mathematical Operations**: Union, intersection, difference, and symmetric difference.

- **NoneType (`None`)**:
  - **Represents Absence**: Used to represent the absence of a value or a null value.
  - **Immutable**: Cannot be modified.
  - **Commonly Used for Default Values**: Often used as a default value in function parameters or to indicate that a variable does not contain a value.

▼ 3. Python List

1. Creating a List
2. Printing Lists
3. List Methods
   - append()
   - clear()
   - copy()
   - count()
   - extend()
   - index()
   - insert()
   - pop()
   - remove()
   - reverse()
   - sort()
   - max()/min()
4. List Operations

---

1. Creating a List

```
# Example of creating a list of fruits
fruits = ['apple', 'banana', 'cherry']
```

2. Printing Lists

To print list elements, use the following methods:

```
l = [1, 2, 3, 4, 5, 6]

# Unpacked print (prints each element individually)
print(*l)  # Output: 1 2 3 4 5 6

# Regular print (prints the list as it is)
print(l)  # Output: [1, 2, 3, 4, 5, 6]
```

3. List Methods

Below are common methods available for lists in Python with descriptions and examples.

Method	Description	Example Code
`append()`	Adds an element to the end of the list.	`fruits.append("orange")`
`clear()`	Removes all elements from the list.	`fruits.clear()`
`copy()`	Returns a shallow copy of the list.	`x = fruits.copy()`
`[:]`	Returns a shallow copy of the entire list.	`copy_list = fruits[:]`
`count()`	Counts occurrences of a specified value.	`x = fruits.count("cherry")`
`len()`	Returns the number of elements in the list.	`length = len(fruits)`
`extend()`	Adds elements of another iterable to the list.	`fruits.extend(["kiwi", "mango"])`
`index()`	Returns index of first element with a value.	`x = fruits.index("cherry")`
`insert()`	Inserts element at specified position.	`fruits.insert(1, "orange")`
`pop()`	Removes element at a specified position.	`fruits.pop(1)`
`remove()`	Removes first item with specified value.	`fruits.remove("banana")`
`reverse()`	Reverses the order of the list.	`fruits.reverse()`

`sort()`	Sorts the list.	`fruits.sort()`
`max()` / `min()`	Returns the maximum/minimum value.	`print(max([1.2, 1.3, 0.1]))`

4. List Operations

Index

---

1. Concatenation

Concatenation combines two or more lists using the `+` operator.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print(combined_list)  # Output: [1, 2, 3, 4, 5, 6]
```

2. Repetition

Repeating elements in a list can be done using the `*` operator.

```
list1 = ['a', 'b', 'c']
repeated_list = list1 * 3
print(repeated_list)  # Output: ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

3. Slicing

Slicing extracts a subset of elements from a list.

```
my_list = [10, 20, 30, 40, 50]

# Slicing from index 1 to 3 (exclusive of 3)
print(my_list[1:3])  # Output: [20, 30]

# Slicing with step
print(my_list[::2])  # Output: [10, 30, 50]

# Slicing in reverse
print(my_list[::-1])  # Output: [50, 40, 30, 20, 10]
```

4. Membership Testing

Use `in` and `not in` to check for the presence of an element in a list.

```
fruits = ['apple', 'banana', 'cherry']

# Check if 'banana' is in the list
print('banana' in fruits)  # Output: True

# Check if 'mango' is not in the list
print('mango' not in fruits)  # Output: True
```

5. List Comprehension

List comprehensions provide a concise way to generate lists by iterating and applying conditions.

```
# Generating a list of squares
squares = [x**2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]

# Using a condition within a list comprehension
even_squares = [x**2 for x in range(1, 6) if x % 2 == 0]
print(even_squares)  # Output: [4, 16]
```

6. Shallow Copying

A shallow copy creates a new list but does not duplicate the nested objects within it.

```
# Using slicing to create a shallow copy
list1 = [1, 2, 3, 4]
list_copy = list1[:]
print(list_copy)  # Output: [1, 2, 3, 4]
```

7. Deep Copying

A deep copy duplicates the list, including all nested objects within it. This requires the `copy` module.

```
import copy

list1 = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(list1)
print(deep_copy)  # Output: [[1, 2], [3, 4]]

# Modifying deep_copy does not affect list1
deep_copy[0][0] = 9
print(list1)      # Output: [[1, 2], [3, 4]]
print(deep_copy)  # Output: [[9, 2], [3, 4]]
```

Summary of Python List Characteristics:

- **Ordered**: Elements are stored in a specific, predictable order.
- **Mutable**: Can be modified after creation (e.g., adding, removing, or changing elements).
- **Allow Duplicates**: Can contain duplicate elements; the same element can appear multiple times.
- **Heterogeneous**: Can store elements of different data types (e.g., integers, strings, objects).
- **Indexed**: Elements can be accessed using an index, starting from 0.
- **Iterable**: Can be iterated over using loops.
- **Supports Nesting**: Can contain other lists or complex data structures.

- **Supports Slicing**: Allows for slicing to access sub-portions of the list.
- **Dynamic Size**: The size of the list can grow or shrink as elements are added or removed.
- **Used for Sequential Data**: Ideal for storing sequences of items where order matters.

▼ 4. Python Tuple

Index

1. Creating a Tuple

Tuples are immutable, meaning their elements cannot be modified after creation.

```
# Empty tuple
empty_tuple = ()

# Tuple with single element (a trailing comma is required)
```

```
single_element_tuple = (5,)

# Tuple with multiple elements
multiple_elements_tuple = (1, 2, 3, 4)

packed_tuple = 1, 2, 3, 4  # Also a valid tuple
print(packed_tuple)  # Output: (1, 2, 3, 4)

tuple_from_list = tuple([1, 2, 3, 4])
print(tuple_from_list)
```

2. Tuple Methods

Below are the methods available for tuples in Python, along with descriptions and examples.

Method	Description	Example Code
`count()`	Returns the number of times a specified value occurs in the tuple.	`x = thistuple.count(5)`
`index()`	Searches for a specified value and returns the position of its first occurrence.	`x = thistuple.index(8)`
`all()`	Returns `True` if all elements are `True` or if the tuple is empty.	`print(all((True, True, False)))`
`any()`	Returns `True` if any element of the tuple is `True`. If the tuple is empty, returns `False`.	`print(any((False, False, True)))`
`len()`	Returns the length of the tuple (number of elements).	`print(len(thistuple))`
`enumerate()`	Returns an enumerate object containing index-value pairs from the tuple.	`for i, v in enumerate(thistuple): print(i, v)`
`max()`	Returns the maximum element in the tuple.	`print(max(thistuple))`
`min()`	Returns the minimum element in the tuple.	`print(min(thistuple))`
`sum()`	Returns the sum of all elements in the tuple.	`print(sum((1, 2, 5, 3)))`

`sorted()`	Returns a new sorted list from the elements of the tuple.	`sorted_tuple = sorted(thistuple)`
`tuple()`	Converts an iterable (like a list) to a tuple.	`t = tuple([1, 2, 3])`

3. Example Code for Tuple Operations

Here are examples demonstrating each of the above tuple operations:

```
# Example tuple
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)

# count() - Number of times 5 appears
print(thistuple.count(5))  # Output: 2

# index() - First position of 8
print(thistuple.index(8))  # Output: 3

# all() - Returns True if all elements are True
print(all((True, True, False)))  # Output: False
print(all((4, 5, 1)))            # Output: True

# any() - Returns True if any element is True
print(any((False, False, True)))  # Output: True

# len() - Number of elements in the tuple
print(len(thistuple))  # Output: 10

# enumerate() - Enumerate object for tuple
l1 = ("a", "b", "c", "d", "e")
for i, v in enumerate(l1, start=5):
    print(i, v)
# Output:
# 5 a
# 6 b
# 7 c
# 8 d
# 9 e
```

```
# max() - Maximum element in the tuple
print(max(thistuple))  # Output: 8

# min() - Minimum element in the tuple
print(min(thistuple))  # Output: 1

# sum() - Sum of elements in a tuple
l1 = (1, 2, 5, 3)
print(sum(l1))  # Output: 11

# sorted() - Sorts tuple elements into a new list
sorted_tuple = sorted(thistuple)
print(sorted_tuple)  # Output: [1, 3, 4, 5, 5, 6, 7, 7, 8, 8]

# tuple() - Convert a list to a tuple
t = tuple([1, 2, 3])
print(t)  # Output: (1, 2, 3)
```

Summary of Tuple Characteristics:

- **Ordered**: Elements are in a specific order.
- **Immutable**: Cannot be modified after creation.
- **Allow Duplicates**: Can contain repeated elements.
- **Heterogeneous**: Can store different data types.
- **Fixed Size**: Size cannot be changed once created.
- **Indexed and Iterable**: Can be accessed using indexing and can be iterated over.
- **Supports Nesting**: Can contain other tuples or complex data structures.
- **Hashable**: Can be used as dictionary keys.
- **Faster than Lists**: More efficient in terms of performance.
- **Slicing**: Supports slicing to access sub-portions.

▼ 5. Some Questions

Index

1. Built-in Functions vs Built-in Methods

2. Python | a += b is not always a = a + b

1. Built-in Functions vs Built-in Methods

Feature	Built-in Functions	Built-in Methods
Association	Not tied to specific objects	Tied to specific object types
Calling	Called directly: `len(list_name)`	Called on objects using dot notation: `list_name.append()`
Focus	Often work on diverse data types	Usually focus on actions related to the object's type
Examples	`print()`, `len()`, `input()`, `type()`	`append()`, `upper()`, `keys()`

2. Python | a += b is not always a = a + b

The expression `a += b` is not always equivalent to `a = a + b` because of how Python handles memory references and object mutability. Let's break it down:

- `a = a + [2, 3]`:

  This creates a **new object** and assigns it to `a`. The original `a` list is not modified, and the reference to `a` changes to the new list, which results in a new memory location for `a`.

- `b += [2, 3]`:

  This modifies the **same object**. The list `b` is updated in place without changing its reference. Thus, `b` retains the same memory address.

Code Example:

```
a = [0, 1]
print("id of a: ", id(a), "Value: ", a)
a = a + [2, 3]
print("id of a: ", id(a), "Value: ", a)

b = [0, 1]
print("id of b: ", id(b), "Value: ", b)
```

```
b += [2, 3]
print("id of b: ", id(b), "Value: ", b)
```

Output:

```
id of a:  132885299534016 Value:  [0, 1]
id of a:  132885299531072 Value:  [0, 1, 2, 3]
id of b:  132885299544000 Value:  [0, 1]
id of b:  132885299544000 Value:  [0, 1, 2, 3]
```

- **Explanation**:
  - For `a = a + [2, 3]`, the id of `a` changes because a new list is created.
  - For `b += [2, 3]`, the id of `b` remains the same because the operation modifies the list in place without creating a new reference.

This difference arises from the fact that `+=` is an **in-place operation** for mutable types (like lists), whereas `+` creates a **new object** and reassigns it.

▼ 6. Python Sets

Index

In Python, a set is an unordered collection of unique items. Sets are mutable, but they do not allow duplicates. They are commonly used for operations like union, intersection, and difference.

Example of Creating a Set:

```
fruits = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
```

Set Operations:

Method	Description	Example
add()	Adds an element to the set	`fruits.add("orange")`
clear()	Removes all the elements from the set	`fruits.clear()`
copy()	Returns a copy of the set	`x = fruits.copy()`
difference()	Returns a set containing the difference between two or more sets	`x = {"a","b","c"}; y = {"c","d"}; m = {"a","d"}; z = x.difference(y, m); print(z)` **Output:** `{'b'}`
difference_update()	Removes the items in this set that are also included in another, specified set	`x = {"a","b","c"}; y = {"c","d"}; x.difference_update(y); print(x)` **Output:** `{'b', 'a'}`
discard()	Remove the specified item from the set without raising an error if the item does not exist	`fruits.discard("banana")`
intersection()	Returns a set that is the intersection of two or more sets	`fruits.intersection(y)`
intersection_update()	Removes items in this set that are not present in other, specified set(s)	`x.intersection_update(y)`
isdisjoint()	Returns `True` if two sets have no intersection, otherwise returns `False`	`x.isdisjoint(y)`

issubset()	Returns `True` if another set contains this set	`x.issubset(y)`
issuperset()	Returns `True` if this set contains another set	`x.issuperset(y)`
pop()	Removes and returns an arbitrary element from the set	`fruits.pop()`
remove()	Removes the specified element from the set. Raises an error if element not found	`fruits.remove("banana")`
symmetric_difference()	Returns a set with the symmetric differences of two sets (items that are in one set but not the other)	`x = {"a","b","c"}; y = {"c","d"}; z = x.symmetric_difference(y); print(z)` **Output:** `{'d', 'a', 'b'}`
symmetric_difference_update()	Inserts the symmetric differences from this set and another	`x.symmetric_difference_update(y)`
union()	Return a set containing the union of sets (all elements from both sets)	`x = {"a","b","c"}; y = {"c","d"}; z = x.union(y); print(z)` **Output:** `{'d', 'b', 'c', 'a'}`
update()	Updates the set with another set or any iterable	`x = {"a","b","c"}; y = {"c","d"}; x.update(y); print(x)` **Output:** `{'d', 'b', 'c', 'a'}`
frozenset()	Returns an immutable frozenset object	`frozen_set = frozenset([1, 2, 3])`

Example Use Cases:

- **Adding an element to the set:**

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits)
# Output: {'apple', 'banana', 'cherry', 'orange'}
```

- **Getting the difference between two sets:**

```
x = {"a", "b", "c"}
y = {"c", "d"}
z = x.difference(y)
print(z)
# Output: {'a', 'b'}
```

- **Performing a union of two sets:**

```
x = {"a", "b", "c"}
y = {"c", "d"}
z = x.union(y)
print(z)
# Output: {'a', 'b', 'c', 'd'}
```

Summary of Set Characteristics:

- **Unordered**: Elements are not stored in any particular order.
- **Mutable**: Can add or remove elements after creation.
- **Unique Elements**: Does not allow duplicates; each element must be unique.
- **Heterogeneous**: Can store elements of different data types.
- **No Indexing**: Does not support indexing or accessing elements by position.
- **Iterable**: Can be iterated over using loops.

- **Supports Mathematical Operations**: Can perform operations like union, intersection, difference, and symmetric difference.
- **Cannot Contain Mutable Elements**: Only immutable data types (e.g., numbers, strings, tuples) can be stored.
- **Optimized for Membership Testing**: Efficient at checking if an element is in the set.
- **Faster than Lists for Lookup**: More efficient than lists for membership checks and operations like union and intersection.

▼ 7. Python Dictionaries

Index

---

1. Introduction to Python Dictionaries

A dictionary in Python is a collection of unordered, changeable, and indexed data. It is used to store data in key-value pairs.

Example:

```
dictionary = {
   "key1": "value1",
   "key2": "value2",
   "key3": "value3"
}
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
```

2. Dictionary Methods

Method	Description	Example
clear()	Removes all the elements from the dictionary	`car.clear()`
copy()	Returns a copy of the dictionary	`x = car.copy()`
fromkeys()	Returns a dictionary with the specified keys and a default value	`x = ('k1', 'k2', 'k3')   y = "a"   thisdict = dict.fromkeys(x, y)   print(thisdict)`
get()	Returns the value of the specified key	`x = car.get("model")`
items()	Returns a list containing a tuple for each key-value pair	`x = car.items()` Output: `dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])`
keys()	Returns a list containing the dictionary's keys	`x = car.keys()`
values()	Returns a list of all the values in the dictionary	`x = car.values()`
pop()	Removes the element with the specified key	`car.pop("model")`
popitem()	Removes the last inserted key-value pair	`car.popitem()`
setdefault()	Returns the value of the specified key. If the key	`car = { "a": "1", "b": "2", "c": "3" }   x = car.setdefault("b", "99")   y = car.setdefault("d", "4")   print(car)   print(x, y)` Output: `{'a': '1', 'b':`

	does not exist, insert the key with a specified value	`'2', 'c': '3'}`
		 `{'a': '1', 'b': '2', 'c': '3', 'd': '4'}`
		 `2 4`

update()	Updates the dictionary with the specified key-value pairs	`car.update({"color": "White"})`

Summary of Python Dictionary Characteristics (Linear List)

1. **Unordered**: Dictionaries do not maintain the order of elements (though Python 3.7+ maintains insertion order as an implementation detail).
2. **Mutable**: You can modify a dictionary after creation (add, remove, or change key-value pairs).
3. **Indexed by Keys**: Elements are accessed using unique keys, not by index like lists.
4. **No Duplicate Keys**: Each key must be unique. Assigning a new value to an existing key will overwrite the old value.
5. **Key-Value Pair**: Every element is a key-value pair, where the key is unique and used to access the corresponding value.
6. **Key Types**: Keys must be immutable types (e.g., strings, integers, tuples), but values can be any data type.
7. **Dynamic**: The size of a dictionary can change dynamically by adding or removing elements.
8. **Accessing Elements**: Elements are accessed using keys. If a key is missing, it raises a `KeyError` unless methods like `get()` are used.
9. **Built-in Methods**: Python dictionaries have many methods like `clear()`, `copy()`, `get()`, `pop()`, and `update()` to modify or retrieve information.
10. **Memory Efficient**: Dictionaries are implemented using hash tables, which allow for efficient operations like lookups, additions, and deletions.

▼ 8. Python Arrays

Arrays in Python are implemented using the `array` module, which provides a space-efficient way to store data of the same type. They are typically used

when you need to work with large datasets of homogeneous types (e.g., integers, floats).

Syntax:

```
python
Copy code
import array as arr

# Creating arrays with different typecodes
int_array = arr.array('i', [1, 2, 3, 4, 5])  # Array of integers
float_array = arr.array('d', [3.14, 2.71, 1.61])  # Array of floats
```

- **Typecode**: Specifies the type of elements in the array.
  - `'i'`: Signed integer
  - `'d'`: Double-precision float

Common Array Methods:

Method	Description	Example
`append(x)`	Adds an item `x` to the end of the array.	`int_array.append(6)`
`insert(i, x)`	Inserts an item `x` at the specified index `i`.	`int_array.insert(2, 2.5)`
`pop(i)`	Removes and returns the item at index `i`.	`popped_value = int_array.pop(2)`
`remove(x)`	Removes the first occurrence of the value `x`.	`int_array.remove(2)`
`index(x)`	Returns the index of the first occurrence of value `x`.	`index = int_array.index(5)`
`count(x)`	Counts the number of occurrences of value `x`.	`count = int_array.count(5)`
`reverse()`	Reverses the order of elements in the array.	`float_array.reverse()`
`sort()`	Sorts the array in ascending order.	`float_array.sort()`

Summary of Python Array Characteristics

1. **Ordered**: Arrays maintain the insertion order of elements.

2. **Fixed Size**: The size of an array is fixed upon creation and cannot be changed directly.
3. **Homogeneous**: Arrays can only store elements of the same data type.
4. **Efficient Memory Usage**: Arrays use less memory than lists due to storing homogeneous data types.
5. **Mutable**: Elements in an array can be changed, but the size is fixed.
6. **Indexed**: Elements are accessed using zero-based indices.
7. **Support for Mathematical Operations**: Arrays support element-wise operations, especially with libraries like NumPy.
8. **Limitations on Data Types**: Arrays restrict the types of data they can hold, based on the specified type.
9. **Accessing Elements**: Elements are accessed using indices, and out-of-range access results in an error.
10. **Slower Than Lists (Without NumPy)**: Regular arrays are slower than lists for general-purpose tasks.
11. **Array Module**: Arrays are created and manipulated using Python's `array` module.

```
import array
```

▼ 9. Python String

Index

1. Introduction to Python Strings

Strings are sequences of characters enclosed in quotes. In Python, strings can be created using single quotes, double quotes, or triple quotes (for multi-line strings).

Example:

```
a = "Hello"
print(a)  # Output: Hello

a = '''a<br>b<br>c'''
print(a)  # Output: a<br>b<br>c
```

2. **Basic String Operations**

Code Snippet	Description	Output
`a = "Hello"`	Assigns the string "Hello" to variable `a`	`Hello`
`a = """a b c"""` or `a = '''a b c'''`	Assigns a multi-line string to `a` using triple quotes	`a b c`
`print(a[1])`	Prints the character at index 1 (second character)	`e`

`for x in "abcd":`	Iterates through each character in "abcd"	`a`
		 `b`
		 `c`
		 `d`

`txt = "The best things in life are free!"` `print("free" in txt, "free" not in txt)`	Checks for the presence and absence of "free" in `txt`	`True False`
`x = "abcd"` `print(x[-1])`	Prints the last character using negative indexing	`d`
`print("abcd"[::-1])`	Reverses the string using slicing	`dcba`

3. **String Methods**

Method	Description	Example

`capitalize()`	Converts the first character to upper case	`name = "john"`
		 `print(name.capitalize())`
		Output: "John"

`casefold()`	Converts string into lower case, for caseless comparisons	`text = "HELLO WORLD"`
		 `print(text.casefold())`
		Output: "hello world"

`center()`	Returns a centered string	`title = "Welcome"`
		 `print(title.center(50))`
		Output: `" Welcome "`

`count()`	Returns the number of times a specified value occurs in a string	`message = "Hello, world!"`
		 `print(message.count("o"))`
		Output: `2`

`encode()`	Returns an encoded version of the string	`text = "Python rocks!"`
		 `print(text.encode("utf-8"))`
		Output: `b"Python rocks!"`

`endswith()`	Returns True if the string ends with the specified value	`filename = "data.txt"`
		 `print(filename.endswith(".txt"))`
		Output: `True`

`expandtabs()`	Sets the tab size of the string	`code = "print(\\t'Hello')"`
		 `print(code.expandtabs(4))`
		Output: `"print 'Hello'"`

`find()`	Searches the string for a specified value and returns the position	`text = "Hello, world!"`
		 `print(text.find("world"))`
		Output: `7`

`format()`	Formats specified values in a string	`name = "Alice"`
		 `age = 30`
		 `print("Hello, {}! You are {} years old.".format(name, age))`
		Output: `"Hello, Alice! You are 30 years old."`

`format_map()`	Formats specified values in a string using a mapping	`data = {"name": "Bob", "age": 25}`
		 `print("Name: {name}, Age: {age}".format_map(data))`
		Output: `"Name: Bob, Age: 25"`

`index()`	Searches the string for a specified value and returns the position	`text = "Python"`
		 `print(text.index("y"))`
		Output: `1`

`isalnum()`	Returns True if all characters in the string are alphanumeric	`text = "hello123"`
		 `print(text.isalnum())`
		Output: `True`

`isalpha()`	Returns True if all characters in the string are alphabetic	`text = "hello"`
		 `print(text.isalpha())`
		Output: `True`

`isascii()`	Returns True if all characters in the	`text = "hello"`
		 `print(text.isascii())`
		Output: `True`

	string are ASCII characters	

`isdecimal()`	Returns True if all characters in the string are decimal	`price = "123.45"`
		 `print(price.isdecimal())`
		Output: `False` (due to the decimal point)

`isdigit()`	Returns True if all characters in the string are digits	`number = "12345"`
		 `print(number.isdigit())`
		Output: `True`

`isidentifier()`	Returns True if the string is a valid identifier	`name = "my_variable"`
		 `print(name.isidentifier())`
		Output: `True`

`islower()`	Returns True if all characters in the string are lowercase	`text = "hello world"`
		 `print(text.islower())`
		Output: `True`

`isnumeric()`	Returns True if all characters in the string are numeric	`num = "12345"`
		 `print(num.isnumeric())`
		Output: `True`

`isprintable()`	Returns True if all characters in the string are printable	`text = "Hello, world!"`
		 `print(text.isprintable())`
		Output: `True`

`isspace()`	Returns True if all characters in the string are whitespace	`spaces = " \\t\\n"`
		 `print(spaces.isspace())`
		Output: `True`

`istitle()`	Returns True if the string follows title capitalization rules	`name = "John Doe"`
		 `print(name.istitle())`
		Output: `True`

`isupper()`	Returns True if all characters in the string are uppercase	`text = "HELLO WORLD"`
		 `print(text.isupper())`
		Output: `True`

`join()`	Converts the elements of an iterable into a string	`words = ["Hello", "world"]`
		 `sentence = " ".join(words)`
		 `print(sentence)`
		Output: `"Hello world"`

`ljust()`	Returns a left justified version of the string	`text = "Python"`
		 `print(text.ljust(10))`
		Output: `"Python "`

`lower()`	Converts a string into lowercase	`text = "HELLO WORLD"`
		 `print(text.lower())`
		Output: `"hello world"`

`lstrip()`	Returns a left trim version of the string	`text = " Hello, world! "`
		 `print(text.lstrip())`
		Output: `"Hello, world!"`

`maketrans()`	Returns a translation table for custom string translations	`table = str.maketrans("aeiou", "12345")`
		 `text = "hello"`
		 `print(text.translate(table))`
		Output: `"h2ll5"`

`partition()`	Returns a tuple with the string partitioned by a separator	`text = "apple, banana, cherry"`
		 `parts = text.partition(", ")`
		 `print(parts)`
		Output: `("apple", ", ", "banana, cherry")`

`replace()`	Returns a string with specified values replaced	`text = "Hello, world!"`
		 `print(text.replace("world", "Python"))`
		Output: `"Hello, Python"`

`rfind()`	Searches the string for a value from the right, returning its last position	`text = "Hello, hello!"`
		 `print(text.rfind("hello"))`
		Output: `7`

Method	Description	Example	Output
`rindex()`	Searches the string for a value from the right, returning its last position, raising an error if not found	`text = "Python rocks!" \\nprint(text.rindex("o"))`	`10`
`rjust()`	Returns a right justified version of the string	`text = "Python" \\nprint(text.rjust(10))`	`" Python"`
`rpartition()`	Returns a tuple with the string partitioned by a separator from the right	`text = "apple, banana, cherry" \\nparts = text.rpartition(", ") \\nprint(parts)`	`("apple, banana", ", ", "cherry")`
`rsplit()`	Splits the string at the specified separator, starting from the right	`text = "apple, banana, cherry" \\nfruits = text.rsplit(", ") \\nprint(fruits)`	`["apple", "banana", "cherry"]`
`rstrip()`	Returns a right trim version of the string	`text = " Hello, world! " \\nprint(text.rstrip())`	`" Hello, world!"`

`split()`	Splits the string at the specified separator and returns a list	`text = "apple, banana, cherry" \\nfruits = text.split(", ") \\nprint(fruits)`	`["apple", "banana", "cherry"]`
`splitlines()`	Splits the string at line breaks and returns a list	`text = "Line 1\\nLine 2\\nLine 3" \\nlines = text.splitlines() \\nprint(lines)`	`["Line 1", "Line 2", "Line 3"]`
`startswith()`	Returns True if the string starts with the specified value	`text = "Hello, world!" \\nprint(text.startswith("Hello"))`	`True`
`strip()`	Returns a trimmed version of the string, removing leading and trailing whitespace	`text = " Hello, world! " \\nprint(text.strip())`	`"Hello, world!"`
`swapcase()`	Swaps the case of letters in the string	`text = "Hello, WoRlD!" \\nprint(text.swapcase())`	`"hELLO, wOrLd!"`
`title()`	Converts the first character of each word to upper case	`name = "john doe" \\nprint(name.title())`	`"John Doe"`
`translate()`	Returns a translated string using a translation table	`table = str.maketrans("aeiou", "12345") \\ntext = "hello" \\nprint(text.translate(table))`	`"h2ll5"`
`upper()`	Converts a string to uppercase	`text = "hello world" \\nprint(text.upper())`	`"HELLO WORLD"`
`zfill()`	Fills the string with leading zeros to a specified width	`number = "123" \\nprint(number.zfill(5))`	`"00123"`

Summary of python String Characteristics:

1. **Immutable**: Strings cannot be modified after creation.
2. **Sequence Type**: Strings are ordered sequences of characters.

3. **Ordered**: The characters in a string have a defined order.
4. **Iterable**: Strings can be iterated through using loops.
5. **Supports String Operations**: Strings support operations like concatenation and membership checks.
6. **Unicode Support**: Strings handle Unicode characters by default.
7. **Length**: The length of a string can be obtained using `len()`.
8. **Escape Characters**: Strings can include special characters like newlines (`\\n`) using escape sequences.
9. **Supports String Methods**: Strings come with built-in methods for manipulation, like `upper()`, `replace()`, etc.
10. **Can Be Formatted**: Strings can be formatted dynamically using f-strings or `format()`.

▼ 10. Python Variables

Index

1. Python Variables

Scope of Variable

In Python, variables can have different scopes: local, global, or built-in. The following example demonstrates variable scope:

```
def f():
    s = "bb"  # local variable
    print(s)

# Global variable
s = "aa"
f()
print(s)
```

Output:

```
bb
aa
```

- `s` inside the function `f()` is a local variable, whereas the `s` outside the function is a global variable. The function prints the local `s`, and the last line prints the global `s`.

---

2. Packing and Unpacking Arguments

You can pack arguments into a tuple and unpack them back into individual variables. This is useful when passing multiple values to a function.

```
def fun(a, b, c, d):
    print(a, b, c, d)

my_list = [1, 2, 3, 4]
fun(*my_list)
```

Output:

```
1 2 3 4
```

- The `my_list` syntax unpacks the list `my_list` and passes its elements as individual arguments to the function.

---

3. Type Conversion in Python

Implicit Type Conversion

Python automatically converts smaller data types to larger ones when necessary (i.e., from `int` to `float`).

```
num1 = 10  # int
num2 = 3.14  # float
result = num1 + num2  # num1 is implicitly converted to float
print(result)  # Output: 13.14
```

Explicit Type Conversion

You can explicitly convert between types using functions like `int()`, `float()`, etc.

```
num1 = 10  # int
num2 = 3.14  # float
result = float(num1) + num2  # num1 is explicitly converted to float
print(result)  # Output: 13.14
```

---

4. Underscore (_) in Python

The underscore (`_`) in Python has several special uses:

Single Underscore Usage

- **In the interpreter**: Represents the result of the last evaluated expression.

```
>>> 5 + 5
10
>>> _ + 5  # Uses the last result
15
```

- **Before a variable name**: It indicates that the variable is intended for internal use only.

```
_internal_variable = "Internal use only"
```

- **In numeric literals**: Used to improve readability of large numbers.

```
one_million = 1_000_000
print(one_million)  # Output: 1000000
```

Double Underscore Usage

- **Before a variable name**: Used for private variables, which are not intended to be accessed directly.

```
__private_variable = "Private variable"
```

---

5. Special Methods (Dunder Methods)

These methods are surrounded by double underscores and define special behaviors for classes and objects. Here's an example:

```
class Person:
    def __init__(self, name):
        print("__init__")
        self.name = name  # Constructor for creating objects

    def __str__(self):  # String representation of the object
        print("__str__")
        return f"Person(name={self.name})"

    def fun(self):  # Regular method
        print("inside fun")

    def __add__(self, other):  # Custom addition behavior
        print("__add__")
        return Person(f"{self.name} and {other.name}")
```

```
person1 = Person("Alice")
print(person1)  # __str__ called here
print(person1.fun())
person2 = Person("Bob")
print(person1 + person2)  # __add__ called here
```

Output:

```
__init__
__str__
Person(name=Alice)
__init__
inside fun
None
__init__
__str__
Person(name=Alice and Bob)
```

---

6. Name Mangling

Name mangling in Python is a technique to prevent name conflicts in subclasses by changing the names of variables. It is done automatically for variables with double leading underscores.

```
class MyClass:
    def __init__(self):
        self.__mangled_variable = "I'm mangled!"

    def get_mangled_variable(self):
        return self.__mangled_variable

class MySubclass(MyClass):
    def __init__(self):
        super().__init__()
        self.__mangled_variable = "I'm in the subclass!"

    def get_mangled_variable_in_subclass(self):
        return self.__mangled_variable
```

```
obj_subclass = MySubclass()
print(obj_subclass.get_mangled_variable())  # Output: I'm mangled!
print(obj_subclass.get_mangled_variable_in_subclass())  # Output: I'm in the subclass!
```

- Name mangling changes the attribute name in the parent class and subclass, which avoids naming conflicts.

---

7. `__name__` Special Variable

The special variable `__name__` helps distinguish if a Python file is run directly or imported as a module.

- **When running a script directly**: `__name__` is set to `"__main__"`.
- **When importing a script**: `__name__` is set to the script/module name.

```
# File1.py
print("File1 __name__ = %s" % __name__)

if __name__ == "__main__":
    print("File1 is being run directly")
else:
    print("File1 is being imported")
```

Output when running File1.py directly:

```
File1 __name__ = __main__
File1 is being run directly
```

Output when importing File1.py into another module:

```
File1 __name__ = File1
File1 is being imported
```

▼ 11. Python Operators

Index

---

1. Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, etc.

Operator	Description	Syntax	Example
`+`	Addition	`x + y`	`5 + 3 = 8`
`-`	Subtraction	`x - y`	`5 - 3 = 2`
`*`	Multiplication	`x * y`	`5 * 3 = 15`
`/`	Division	`x / y`	`5 / 2 = 2.5`
`//`	Floor division	`x // y`	`5 // 2 = 2`
`%`	Modulo	`x % y`	`5 % 3 = 2`
`**`	Exponentiation	`x ** y`	`5 ** 3 = 125`

---

2. Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Description	Syntax	Example
`=`	Assign a value	`x = 5`	`x = 5`
`+=`	Add and assign	`x += 3` (equiv. to `x = x + 3`)	`x = 5, x += 3 → x = 8`

`-=`	Subtract and assign	`x -= 2` (equiv. to `x = x - 2`)	`x = 5, x -= 2 → x = 3`
`*=`	Multiply and assign	`x *= 4` (equiv. to `x = x * 4`)	`x = 3, x *= 4 → x = 12`
`/=`	Divide and assign	`x /= 2` (equiv. to `x = x / 2`)	`x = 8, x /= 2 → x = 4.0`
`//=`	Floor divide and assign	`x //= 3` (equiv. to `x = x // 3`)	`x = 10, x //= 3 → x = 3`
`%=`	Modulo and assign	`x %= 2` (equiv. to `x = x % 2`)	`x = 5, x %= 2 → x = 1`
`**=`	Exponentiate and assign	`x **= 2` (equiv. to `x = x ** 2`)	`x = 4, x **= 2 → x = 16`

3. Comparison Operators

Comparison operators are used to compare two values.

Operator	Description	Syntax	Example
`==`	Equal to	`x == y`	`5 == 5` → `True`
`!=`	Not equal to	`x != y`	`5 != 3` → `True`
`<`	Less than	`x < y`	`5 < 3` → `False`
`>`	Greater than	`x > y`	`5 > 3` → `True`
`<=`	Less than or equal to	`x <= y`	`5 <= 3` → `False`
`>=`	Greater than or equal to	`x >= y`	`5 >= 5` → `True`

4. Logical Operators

Logical operators are used to perform logical operations.

Operator	Description	Syntax	Example
`and`	Logical AND	`x and y`	`True and False` → `False`
`or`	Logical OR	`x or y`	`True or False` → `True`
`not`	Logical NOT	`not x`	`not True` → `False`

5. Identity Operators

Identity operators are used to compare the memory location of two objects.

Operator	Description	Syntax	Example
`is`	Object identity (same object)	`x is y`	`a is b` → `True`
`is not`	Negation of object identity	`x is not y`	`a is not b` → `True`

6. Membership Operators

Membership operators are used to test if a value is a member of a sequence.

Operator	Description	Syntax	Example
`in`	Membership in a sequence	`x in y`	`5 in [1, 2, 3, 4, 5]` → `True`
`not in`	Negation of membership	`x not in y`	`5 not in [1, 2, 3, 4]` → `True`

7. Bitwise Operators

Bitwise operators are used to perform bit-level operations.

Operator	Description	Syntax	Example
`&`	Bitwise AND	`x & y`	`5 & 3` → `1`
`` ` ``	`` ` ``	Bitwise OR	`` `x ``
`^`	Bitwise XOR (exclusive OR)	`x ^ y`	`5 ^ 3` → `6`
`~`	Bitwise NOT (invert bits)	`~x`	`~5` → `-6`
`<<`	Left shift	`x << y`	`5 << 1` → `10`
`>>`	Right shift	`x >> y`	`5 >> 1` → `2`

8. Ternary Operator

The ternary operator provides a shorthand for an `if-else` statement.

Using `if-else`:

```
a, b = 10, 20
print("Both a and b are equal" if a == b
      else "a is greater than b" if a > b
      else "b is greater than a")
```

Using Tuples:

```
a, b = 10, 20
print((b, a)[a < b])  # Prints b when a < b is False
```

Using Dictionary:

```
a, b = 10, 20
print({True: a, False: b}[a < b])  # Returns a if a < b is True, otherwise b
```

Using Lambda Functions:

```
a, b = 10, 20
print((lambda: b, lambda: a)[a < b]())  # Executes the appropriate lambd
a based on condition
```

Print in Ternary Operator:

```
a, b = 10, 20
print(a, "is greater") if (a > b) else print(b, "is Greater")
```

▼ 12. **Python Control Flow**

Index

1. **Conditional Statements**

Statement	Description	Syntax	

`if`	Executes a block of code if a condition is true.	`if condition: statement`	`if age >= 18: print(" eligible " )`
`else`	Executes a block of code if the preceding `if` condition is false.	`else: statement`	`if temperature >= 30: print("hot day.") else: print("not a hot day.")`
`elif`	Stands for "else if" and is used for multiple conditional checks.	`elif condition: statement`	`grade = 85 if grade >= 90: print("Excellent!") elif grade >= 80: print("Very good!") else: print("Good job!")`
`nested if`	Using an `if` statement inside another `if` statement.	`if condition1: if condition2: statement`	`if has_ticket: if has_id: print("enter the concert.") else: print("You need an ID to enter.")`
`pass`	Acts as a placeholder; does nothing and lets the program continue.	`if condition: pass`	`if age < 18: pass`
`ternary operator`	A concise way to write a simple `if-else` statement in a single line.	`value_if_true if condition else value_if_false`	`age = 25 can_vote = True if age >= 18 else False`

2. **Loops**

Loop Type	Description	Syntax	Example
`for` loop	Iterates over a sequence (e.g., list, tuple, string) or other iterable objects.	`python for variable in sequence: statement`	`for num in range(5): print(num)`

`while` loop	Repeatedly executes a block of code as long as the specified condition is true.	`python while condition: statement`	`count = 0`
			`while count < 5:`
			`print(count) count += 1`

`nested loops`	Using one or more loops inside another loop.	`python for variable1 in sequence1: for variable2 in sequence2: statement`	`for i in range(3):`
			`for j in range(2):`
			`print(i, j)`

`break`	Terminates the loop prematurely based on a certain condition.	`python for variable in sequence: if condition: break`	`for num in range(10):`
			`if num == 5: break`
			`print(num)`

`continue`	Skips the rest of the code inside a loop for the current iteration and continues with the next iteration.	`python for variable in sequence: if condition: continue statement`	`for num in range(5):`
			`if num == 2: continue`
			`print(num)`

Else With loop	*The else block just after for/while is executed only when the loop is NOT terminated by a break statement.*		`for i in range(1, 4):`
			`print(i)`
			`else:`
			`print("No Break")`

3. **Switch Case**

```
switcher = {
  0: "zero",
  1: "one",
  2: "two",
}
print(switcher[2]) #two
print(switcher.get(1,"none")) #one
print(switcher.get(99,"none")) #none
```

4. Using Iterations in Python Effectively

```
cars = ["Aston", "Audi", "McLaren"]
i = 0
```

Method	Description	Syntax	Example

C-style approach	Requires prior knowledge of total iterations	`while (i < len(iterable)):`	`while (i < len(cars)):`
			`print(cars[i])`
			`i += 1`

for-in loop	Automatically iterates through elements	`for item in iterable:`	`for car in cars:` `print(car)`
Indexing with range()	Uses indices for accessing elements	`for i in range(len(iterable)):`	`for i in range(len(cars)):` `print(cars[i])`
enumerate()	Returns index-value pairs	`for index, item in enumerate(iterable):`	`for i, v in enumerate(cars):` `print(i, v)`

Looping with multiple iterators	Using multiple iterators in a single loop	`for item1, item2 in zip(iterable1, iterable2):`	`m = ["a", "b", "c"]`
			`n = ["x", "y"]`
			`for m, n in zip(m, n):`
			`print(m, n)`

Unzipping iterators	Reversing zip using `*` operator	`list1, list2 = zip(*zipped_iterable)`	`l1, l2 = zip(*[('a', 'b'),`
			`('aa', 'bb'),`
			`('aaa', 'bbb')])`
			`print(l1)`
			`print(l2)`

▼ 13. Generators in Python

Index

1. Introduction to Generators

In Python, a **generator** is a special type of iterator that allows you to iterate over a sequence of data. Unlike lists, generators do not store all values in memory at once. Instead, they generate items on the fly, which makes them memory-efficient, especially when working with large datasets or streams of data.

2. Creating Generators

Generators can be created in two ways:

1. **Generator Functions**: These use the `yield` keyword.
2. **Generator Expressions**: These are like list comprehensions but use parentheses instead of square brackets.

---

3. Using `yield`

The `yield` keyword is used to produce values from a generator function. When a function contains `yield`, it becomes a generator function and returns a generator iterator when called.

- The generator function can be paused and resumed, maintaining its state between calls.

Example 1: Simple Generator Function

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
for number in counter:
    print(number)
```

Output:

```
1
2
```

```
3
4
5
```

In this example, the `count_up_to` function generates numbers from 1 to the `max` value. The `yield` pauses the function, saving its state and allowing iteration over the sequence.

---

4. Generator Expressions

Generator expressions are similar to list comprehensions, but instead of returning a list, they return a generator.

Example 2: Simple Generator Expression

```
squares = (x * x for x in range(5))
for square in squares:
    print(square)
```

Output:

```
0
1
4
9
16
```

---

5. Advantages of Generators

- **Memory Efficiency**: Generators only produce items one at a time, so they are more memory efficient compared to lists or other data structures that hold the entire sequence in memory.
- **Lazy Evaluation**: Values are generated only when needed, which can improve performance when dealing with large datasets.
- **Infinite Sequences**: Generators can model infinite sequences (e.g., generating Fibonacci numbers), something that would be impossible with lists or arrays.

---

6. Examples

Example 3: Generator for Fibonacci Sequence

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

fib = fibonacci(10)
for number in fib:
    print(number)
```

Output:

```
0
1
1
2
3
5
8
```

In this example, the `fibonacci` generator yields Fibonacci numbers up to a specified limit, showing how generators can be used to model complex sequences.

▼ 14. Python Functions

https://www.geeksforgeeks.org/python-programming-language/

Index

1. Sample Function
2. Default Arguments
3. Keyword Arguments
4. Arbitrary Keyword Arguments

- args in Python (Non-Keyword Arguments)
- *kwargs in Python (Keyword Arguments)

5. Anonymous Functions / Lambda in Python
6. Return Statement in Python Function
7. Pass by Reference and Pass by Value
8. yield

---

1. **Sample Function**

A basic function that adds two numbers and returns the result.

```
def add(num1: int, num2: int) -> int:
    return num1 + num2

print(add(5, 6))
```

Output:

```
11
```

---

2. **Default Arguments**

Functions can have default values for parameters, making them optional during function calls.

```
def myFun(x, y=50):
    print("x: ", x, "y: ", y)

myFun(10)
```

Output:

```
x:  10 y:  50
```

---

3. **Keyword Arguments**

Arguments can be passed by specifying their names explicitly.

```
def student(s1, s2):
    print(s1, s2)

student(s1='aa', s2='bb')
```

Output:

```
aa bb
```

---

4. **Arbitrary Keyword Arguments**

- `args` **in Python (Non-Keyword Arguments)**: Allows passing a variable number of non-keyword arguments.

```
def myFun(*s1):
    for s in s1:
        print(s)

myFun('aa', 'bb', 'cc', 'dd')
```

Output:

```
aa
bb
cc
dd
```

- `*kwargs` **in Python (Keyword Arguments)**: Allows passing a variable number of keyword arguments.

```
def myFun(**s1):
    for key, value in s1.items():
        print(key, value)

myFun(k1='v1', k2='v2', k3='v3')
```

Output:

```
k1 v1
k2 v2
k3 v3
```

---

5. **Anonymous Functions / Lambda in Python**

Lambda functions are small anonymous functions defined with the `lambda` keyword.

```
fun_cube = lambda x: x * x * x
print(fun_cube(7))
```

Output:

```
343
```

---

6. **Return Statement in Python Function**

Functions can return values using the `return` keyword.

```
def fun():
    d = dict()
    d['str'] = "aa"
    d['x'] = 20
    return d

print(fun())
```

Output:

```
{'str': 'aa', 'x': 20}
```

---

7. **Pass by Reference and Pass by Value**

- **Pass by Value**: When a function modifies a variable, it only affects the local copy.

- **Pass by Reference**: Modifying the elements of a mutable object inside a function affects the original object.

```
lst = [11, 22, 33]

# Pass by Value: Reassigning x breaks the link with the original list.
def myFun1(x):
    x = [20, 30, 40]

myFun1(lst)
print(lst)

# Pass by Reference: Modifying the elements of the list affects the origi
nal list.
def myFun2(x):
    x[0] = 20

myFun2(lst)
print(lst)
```

Output:

```
[11, 22, 33]
[20, 22, 33]
```

---

8. yield

The `yield` keyword allows a function to return a generator, which can be used to iterate over values lazily (on-demand).

```
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

for value in simpleGeneratorFun():
    print(value, end=" ")
```

Output:

```
1 2 3
```

Another example demonstrating a generator function producing square numbers:

```
def nextSquare():
    i = 1
    while True:
        yield i * i
        i += 1

for num in nextSquare():
    if num > 100:
        break
    print(num, end=" ")
```

Output:

```
1 4 9 16 25 36 49 64 81 100
```

Lastly, using `yield` to return different types of values:

```
def get_values():
    yield 42
    yield 'hello'
    yield [1, 2, 3]

result = get_values()
print(next(result))  # print 42
print(next(result))  # print 'hello'
print(next(result))  # print [1, 2, 3]
```

Output:

```
42
hello
```

```
[1, 2, 3]
```

Summary of Python Function Characteristics:

- **Defined with `def`**: Functions are defined using the `def` keyword followed by the function name and parentheses.
- **Accept Parameters**: Functions can take parameters (arguments) to process input values.
- **Return Values**: Functions can return values using the `return` keyword. If no return is specified, `None` is returned.
- **Local Scope**: Variables defined inside a function are local to that function, meaning they are not accessible outside.
- **Global Scope**: Functions can access global variables but need the `global` keyword to modify them.
- **Default Arguments**: Functions can have default argument values that are used if no value is provided during the call.
- **Variable-Length Arguments**: Functions can accept a variable number of arguments using `args` (for non-keyword arguments) and `*kwargs` (for keyword arguments).
- **Lambda Functions**: Anonymous functions that are defined using the `lambda` keyword, typically used for small, one-line operations.
- **Recursive Functions**: Functions that can call themselves within their body, requiring a base case to prevent infinite recursion.
- **Higher-Order Functions**: Functions that can accept other functions as arguments or return functions as results.
- **Nested Functions**: Functions can be defined within other functions, and the inner function can access variables from the outer function's scope.
- **Docstrings**: Functions can include documentation strings (docstrings) to describe their purpose and usage, improving code readability.

▼ 15. Python Built in Functions

https://www.geeksforgeeks.org/python-built-in-functions/

Index

27. frozenset()
28. getattr()
29. globals()
30. hasattr()
31. hash()
32. help()
33. hex()
34. id()
35. input()
36. int()
37. isinstance()
38. issubclass()
39. iter()
40. len()
41. list()
42. locals()
43. map()
44. max()
45. memoryview()
46. min()
47. next()
48. object()
49. oct()
50. open()
51. ord()
52. pow()
53. print()

54. property()
55. range()
56. repr()
57. reversed()
58. round()
59. set()
60. setattr()
61. slice()
62. sorted()
63. staticmethod()
64. str()
65. sum()
66. super()
67. tuple()
68. type()
69. vars()
70. zip()
71. **import**()

Topic	Description	Example Code
abs()	Returns the absolute value of a number.	`print(abs(-5)) # Output: 5`
aiter()	Returns an asynchronous iterator for the object.	`async def example():\\n async for item in aiter(some_object):\\n print(item)`
all()	Returns `True` if all elements in the iterable are true, otherwise returns `False`.	`print(all([True, True, False])) # Output: False`
any()	Returns `True` if any element in the iterable is true, otherwise returns `False`.	`print(any([False, True, False])) # Output: True`

anext()	Returns the next item from an asynchronous iterator.	`async def example():\\n item = await anext(iterator)`
ascii()	Returns a string containing a printable representation of an object, escaping non-ASCII characters.	`print(ascii("Hello!")) # Output: 'Hello!'`
bin()	Converts an integer to its binary string representation.	`print(bin(10)) # Output: '0b1010'`
bool()	Converts a value to a boolean (True or False).	`print(bool(0)) # Output: False`
breakpoint()	Enters the debugger at the call location.	`breakpoint()`
bytearray()	Returns a new bytearray object.	`print(bytearray([65, 66, 67])) # Output: bytearray(b'ABC')`
bytes()	Returns a new bytes object.	`print(bytes([65, 66, 67])) # Output: b'ABC'`
callable()	Returns `True` if the object appears callable, otherwise returns `False`.	`print(callable(len)) # Output: True`
chr()	Converts an integer to its corresponding character.	`print(chr(65)) # Output: 'A'`
classmethod()	Converts a method to a class method.	`class MyClass:\\n @classmethod\\n def method(cls):\\n print("Class method")`
compile()	Compiles a source into a code object that can be executed.	`code = compile('a = 5', '<string>', 'exec')\\nexec(code)`
complex()	Creates a complex number from a real and an imaginary part.	`print(complex(2, 3)) # Output: (2+3j)`
delattr()	Deletes an attribute from an object.	`class MyClass:\\n x = 5\\ndelattr(MyClass, 'x')`
dict()	Creates a dictionary.	`my_dict = dict(a=1, b=2)\\nprint(my_dict)`
dir()	Returns a list of attributes and methods of an object.	`print(dir([]))`
divmod()	Returns a tuple of the quotient and remainder of dividing two numbers.	`print(divmod(10, 3)) # Output: (3, 1)`

Topic	Description	Example Code
enumerate()	Adds a counter to an iterable and returns it as an enumerate object.	`for index, value in enumerate(['apple', 'banana']):\\n print(index, value)`
eval()	Executes a Python expression within the program and returns the result.	`print(eval('3 + 2')) # Output: 5`
exec()	Executes dynamically created Python code, which can be a string or a code object.	`exec('x = 10\\nprint(x)') # Output: 10`
filter()	Filters elements from an iterable based on a function that returns a boolean value.	`print(list(filter(lambda x: x > 5, [1, 2, 6, 8]))) # Output: [6, 8]`
float()	Converts a number or a string to a floating-point number.	`print(float('3.14')) # Output: 3.14`
format()	Formats a string using placeholders.	`print("Hello, {}".format('World')) # Output: Hello, World`
frozenset()	Returns an immutable set.	`fset = frozenset([1, 2, 3])\\nprint(fset)`
getattr()	Returns the value of an attribute of an object, given its name as a string.	`class MyClass:\\n x = 5\\nprint(getattr(MyClass, 'x')) # Output: 5`
globals()	Returns a dictionary representing the current global symbol table.	`print(globals())`
hasattr()	Returns `True` if an object has a specified attribute.	`class MyClass:\\n x = 5\\nprint(hasattr(MyClass, 'x')) # Output: True`
hash()	Returns the hash value of an object, if it is hashable.	`print(hash('hello'))`
help()	Invokes the built-in help system.	`help(print)`
hex()	Converts an integer to a hexadecimal string.	`print(hex(255)) # Output: '0xff'`
id()	Returns the identity of an object.	`print(id('string'))`
input()	Reads a line of input from the user.	`name = input("Enter your name: ")`

int()	Converts a number or string to an integer.	`print(int('10')) # Output: 10`
isinstance()	Checks if an object is an instance of a specified class or a subclass thereof.	`print(isinstance(10, int)) # Output: True`
issubclass()	Checks if a class is a subclass of another class.	`print(issubclass(bool, int)) # Output: True`
iter()	Returns an iterator object for an iterable.	`my_iter = iter([1, 2, 3])\\nprint(next(my_iter)) # Output: 1`
len()	Returns the length (number of items) of an object.	`print(len([1, 2, 3])) # Output: 3`

Topic	Description	Example Code
list()	Converts an iterable to a list.	`print(list((1, 2, 3))) # Output: [1, 2, 3]`
locals()	Returns a dictionary representing the current local symbol table.	`x = 5\\ny = 10\\nprint(locals())`
map()	Applies a function to all items in an iterable and returns an iterator.	`print(list(map(lambda x: x ** 2, [1, 2, 3]))) # Output: [1, 4, 9]`
max()	Returns the largest item in an iterable or the largest of two or more arguments.	`print(max([1, 2, 3])) # Output: 3`
memoryview()	Returns a memory view object of a byte-like object.	`x = bytearray([1, 2, 3])\\nprint(memoryview(x))`
min()	Returns the smallest item in an iterable or the smallest of two or more arguments.	`print(min([1, 2, 3])) # Output: 1`
next()	Retrieves the next item from an iterator.	`my_iter = iter([1, 2, 3])\\nprint(next(my_iter)) # Output: 1`
object()	Returns a new object, the base class for all classes.	`print(object())`
oct()	Converts an integer to an octal string.	`print(oct(8)) # Output: '0o10'`
open()	Opens a file and returns a file object.	`file = open('example.txt', 'r')\\nprint(file.read())`
ord()	Returns the Unicode code point for a single character.	`print(ord('A')) # Output: 65`

Topic	Description	Example Code
pow()	Returns the value of the first argument raised to the power of the second argument.	`print(pow(2, 3)) # Output: 8`
print()	Prints objects to the text stream (console).	`print("Hello, World!") # Output: Hello, World!`
property()	Returns a property attribute for getting, setting, and deleting an attribute.	`class MyClass:\\n def __init__(self, x):\\n self._x = x\\n x = property(lambda self: self._x)`
range()	Returns an iterable of numbers, often used in loops.	`for i in range(3):\\n print(i) # Output: 0 1 2`
repr()	Returns a string representation of an object.	`print(repr('Hello')) # Output: "'Hello'"`
reversed()	Returns a reverse iterator for a sequence.	`print(list(reversed([1, 2, 3]))) # Output: [3, 2, 1]`
round()	Returns a rounded value of a floating-point number.	`print(round(3.14159, 2)) # Output: 3.14`
set()	Creates a set, an unordered collection of unique elements.	`print(set([1, 2, 3, 3])) # Output: {1, 2, 3}`
setattr()	Sets the value of an object's attribute.	`class MyClass:\\n pass\\nobj = MyClass()\\nsetattr(obj, 'x', 10)`

Topic	Description	Example Code
slice()	Returns a slice object, which can be used to slice sequences like lists or tuples.	`s = slice(1, 5)\\nprint([1, 2, 3, 4, 5][s]) # Output: [2, 3, 4]`
sorted()	Returns a new sorted list from the elements of any iterable.	`print(sorted([3, 1, 2])) # Output: [1, 2, 3]`
staticmethod()	Returns a static method for a class, which does not need a class instance to be called.	`class MyClass:\\n @staticmethod\\n def greet():\\n print("Hello")\\nMyClass.greet()`
str()	Converts an object to a string.	`print(str(123)) # Output: '123'`
sum()	Returns the sum of all items in an iterable.	`print(sum([1, 2, 3])) # Output: 6`

super()	Returns a proxy object that represents the parent classes.	`class A:\\n def hello(self):\\n print("Hello from A")\\nclass B(A):\\n def hello(self):\\n super().hello()`
tuple()	Converts an iterable to a tuple.	`print(tuple([1, 2, 3])) # Output: (1, 2, 3)`
type()	Returns the type of an object or creates a new type (class).	`print(type("Hello")) # Output: <class 'str'>`
vars()	Returns the **dict** attribute of an object, which contains its attributes.	`class MyClass:\\n x = 5\\nobj = MyClass()\\nprint(vars(obj)) # Output: {'x': 5}`
zip()	Combines multiple iterables element-wise into a single iterator.	`print(list(zip([1, 2], ['a', 'b']))) # Output: [(1, 'a'), (2, 'b')]`
import	Used to import a module, or a part of a module, into the current namespace.	`import math\\nprint(math.sqrt(16)) # Output: 4.0`

▼ 16. Python **Class**

1. **Class**
2. **Objects**
3. **Polymorphism**
4. **Encapsulation**
5. **Inheritance**
6. **Data Abstraction**
7. **Class Definition Syntax/Create a Class**
8. **Create Object**
9. `__init__()` **Function**
10. `__str__()` **Function**
11. **Function Inside Class**
12. **self Parameter**
13. **Delete Object Properties/Delete Objects**

14. **pass Statement**
15. **Method Overriding in Inheritance**

1. Class

A **class** is a blueprint for creating objects (instances). It defines a set of attributes and methods that the created objects will have.

Syntax:

```
class ClassName:
    # Attributes and Methods go here
```

Example:

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

    def display_info(self):
        print(f"Car Make: {self.make}, Model: {self.model}")

# Creating an object of the class
car1 = Car("Toyota", "Corolla")
car1.display_info()
```

---

2. Objects

An **object** is an instance of a class. It holds the data (attributes) and methods defined in the class.

Example:

```
# Object 'car1' is created from the class 'Car'
car1 = Car("Toyota", "Corolla")
```

---

3. Polymorphism

Polymorphism allows different classes to provide a method with the same name but different implementations. It enables methods to be used interchangeably.

Overloading: different number of parameter.

Overriding: same number of parameter.(run time Polymorphism)

Syntax:

```
class Parent:
    def speak(self):
        print("Parent speaking")

class Child(Parent):
    def speak(self):
        print("Child speaking")

# Demonstrating polymorphism
child = Child()
child.speak()  # Outputs: Child speaking
```

---

4. **Encapsulation**

Encapsulation is the concept of restricting access to some of an object's attributes and methods. It hides the internal state and protects the data by using private or protected access modifiers.

Syntax:

```
class MyClass:
    def __init__(self):
        self.__private_var = 10  # Private variable

    def get_private_var(self):
        return self.__private_var
```

Example:

```
obj = MyClass()
print(obj.get_private_var())  # Accessing private variable via a public me
thod
```

---

5. **Inheritance**

Inheritance allows a new class to inherit attributes and methods from an existing class, enabling code reuse and extending functionality.

Syntax:

```
class Parent:
    def speak(self):
        print("Parent speaking")

class Child(Parent):
    def greet(self):
        print("Child says hello")
```

Example:

```
child = Child()
child.speak()  # Inherited from Parent class
child.greet()  # Defined in Child class
```

---

6. **Data Abstraction**

Data abstraction is the concept of hiding the complex implementation details and showing only the necessary information. It is achieved using abstract classes or methods.

Syntax:

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass
```

```
class ConcreteClass(AbstractClass):
    def abstract_method(self):
        print("Implemented abstract method")
```

Example:

```
obj = ConcreteClass()
obj.abstract_method()  # Outputs: Implemented abstract method
```

---

7. Class Definition Syntax/Create a Class

The class is defined using the `class` keyword followed by the class name and a colon. Inside the class, you can define methods and attributes.

Syntax:

```
class ClassName:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
```

Example:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking!")
```

---

8. Create Object

An object is created by calling the class as if it were a function. The constructor `__init__()` is invoked automatically.

Syntax:

```
object_name = ClassName(arguments)
```

Example:

```
dog1 = Dog("Buddy", "Golden Retriever")
dog1.bark()  # Outputs: Buddy is barking!
```

---

9. `__init__()` Function

The `__init__()` method is a special method (constructor) used to initialize an object's state. It is called automatically when an object is created.

Syntax:

```
def __init__(self, parameters):
    self.attribute1 = parameter1
    self.attribute2 = parameter2
```

Example:

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

person1 = Person("John", 30)
print(person1.name, person1.age)  # Outputs: John 30
```

10. `__str__()` Function

The `__str__()` method is used to define a human-readable string representation of an object. It is called when you use `print()` or `str()` on an object.

Syntax:

```
def __str__(self):
    return "String representation of the object"
```

Example:

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

    def __str__(self):
        return f"{self.make} {self.model}"

car1 = Car("Toyota", "Corolla")
print(car1)  # Outputs: Toyota Corolla
```

---

11. **Function Inside Class**

You can define functions (methods) inside a class to perform operations on class attributes or perform actions.

Syntax:

```
class ClassName:
    def function_name(self):
        # Function code
```

Example:

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

    def speak(self):
        print(f"{self.name} says woof!")

dog1 = Dog("Buddy")
dog1.speak()  # Outputs: Buddy says woof!
```

---

12. **self Parameter**

The `self` parameter in class methods refers to the current instance of the class. It is used to access instance variables and methods.

Syntax:

```
def method_name(self):
    # Code that uses self
```

Example:

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

    def greet(self):
        print(f"Hello, my name is {self.name}")

person1 = Person("Alice")
person1.greet()  # Outputs: Hello, my name is Alice
```

---

13. **Delete Object Properties/Delete Objects**

You can delete object properties using the `del` keyword. You can also delete an entire object.

Syntax:

```
del object.property
del object
```

Example:

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

car1 = Car("Toyota")
print(car1.make)  # Outputs: Toyota
```

```
del car1.make
# print(car1.make)  # This will raise an error as 'make' is deleted.
```

---

14. **pass Statement**

The `pass` statement is a placeholder that does nothing. It is often used when a statement is required syntactically but you don't want any code to execute.

Syntax:

```
class MyClass:
    pass  # No code inside the class
```

Example:

```
class MyClass:
    pass

# Creating an object of the empty class
obj = MyClass()
```

---

15. **Method Overriding in Inheritance**

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its parent class. The child class method overrides the parent class method.

Syntax:

```
class Parent:
    def method(self):
        print("Parent method")

class Child(Parent):
    def method(self):
        print("Child method")
```

Example:

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

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

dog = Dog()
dog.sound()  # Outputs: Dog barks
```

▼ 17. **Decorators in Python**

https://www.geeksforgeeks.org/decorators-with-parameters-in-python/https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/

https://www.geeksforgeeks.org/memoization-using-decorators-in-python/

1. **Basic Functions in Python**

- Definition of functions in Python
- Function arguments and return values
- Lambda functions (Anonymous functions)

2. **Higher-Order Functions**

- Functions that take other functions as arguments
- Functions that return other functions

3. **Understanding Closures**

- What is a closure?
- How closures work with functions inside other functions
- Lexical scoping

4. **Decorators in Python**

- Definition and purpose of decorators
- Syntax of decorators: `@decorator_name`
- How decorators modify the behavior of a function
- How to apply decorators manually and automatically

5. **Nested Functions**

- Functions inside other functions (used in decorators)
- How a decorator is a higher-order function that takes another function as an argument

6. **Wrapping Functions**

- Using `functools.wraps()` to preserve metadata of the original function
- The importance of `__name__`, `__doc__`, and other attributes when working with decorators

7. **Common Use Cases for Decorators**

- Logging
- Timing functions (e.g., measuring execution time)
- Authentication and permission checking
- Memoization / Caching results (e.g., `functools.lru_cache`)
- Function arguments validation

8. **Class-based Decorators**

- How to create decorators using classes
- Using `__call__` method in classes for decorators

9. **Chaining Multiple Decorators**

- How to apply multiple decorators to a single function
- The order of decorator application

10. **Decorators with Arguments**

- Creating decorators that accept parameters

- Using arguments in your decorator function for more dynamic behavior

11. **Practical Examples**

- Implementing and using real-world decorators
- Writing your own decorators for specific needs

12. **Best Practices**

- When to use decorators and when to avoid them
- Avoiding decorator misuse and making code more readable

1. **Basic Functions in Python**

- **Definition of Functions in Python**

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

print(greet("Alice"))
```

2. **Higher-Order Functions**

- **Functions that take other functions as arguments**

```
def apply_function(func, value):
    return func(value)

def square(x):
    return x * x

print(apply_function(square, 4))  # Output: 16
```

- **Functions that return other functions**

```
def outer_function(name):
    def inner_function():
        return f"Hello, {name}!"
```

```
    return inner_function

greet = outer_function("Alice")
print(greet())  # Output: Hello, Alice!
```

3. **Understanding Closures**

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
print(add_five(10))  # Output: 15
```

4. **Decorators in Python**

- **Definition and Syntax of Decorators**

A decorator is a function that takes another function and extends its behavior without explicitly modifying it.

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

def say_hello():
    print("Hello!")

decorated_hello = decorator(say_hello)
decorated_hello()
```

- **Using the `@` Syntax**

You can apply a decorator using the `@` syntax, which is shorthand for the above example.

```
@decorator
def say_hello():
    print("Hello!")

say_hello()
```

5. **Nested Functions**

- A decorator itself is a higher-order function. It takes a function and returns a new function.

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@decorator
def greet():
    print("Greetings!")

greet()
```

6. **Wrapping Functions (Using `functools.wraps`)**

- To preserve metadata of the original function like the name and docstring when using decorators.

```
from functools import wraps

def decorator(func):
    @wraps(func)
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
```

```
@decorator
def say_hello():
    """This function says hello."""
    print("Hello!")

print(say_hello.__name__)  # Output: say_hello
print(say_hello.__doc__)   # Output: This function says hello.
```

7. Common Use Cases for Decorators

- **Logging**

```
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log
def add(a, b):
    return a + b

print(add(2, 3))  # Output: Calling function add, 5
```

- **Timing Functions**

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time} seconds")
        return result
    return wrapper
```

```
@timer
def slow_function():
    time.sleep(2)
    return "Finished"

print(slow_function())  # Output: Execution time: 2.0xx seconds, Finishe
d
```

8. **Class-based Decorators**

- You can also use classes to create decorators.

```
class MyDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        print("Before function call")
        self.func()
        print("After function call")

@MyDecorator
def greet():
    print("Hello!")

greet()
```

9. **Chaining Multiple Decorators**

- You can apply multiple decorators to a single function.

```
def decorator1(func):
    def wrapper():
        print("Decorator 1 before")
        func()
        print("Decorator 1 after")
    return wrapper
```

```
def decorator2(func):
    def wrapper():
        print("Decorator 2 before")
        func()
        print("Decorator 2 after")
    return wrapper

@decorator1
@decorator2
def say_hello():
    print("Hello!")

say_hello()
```

10. **Decorators with Arguments**

- You can pass arguments to decorators to make them more flexible.

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

11. **Practical Examples**

Here are some real-world scenarios where decorators can be useful:

Logging Decorator

A logging decorator that logs every function call, which can be useful for debugging:

```
python
Copy code
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} finished")
        return result
    return wrapper

@log
def process_data(data):
    print(f"Processing data: {data}")
    return len(data)

process_data("Sample data")
```

Output:

```
kotlin
Copy code
Calling function: process_data
Processing data: Sample data
Function process_data finished
```

Authorization Decorator

This example checks if the user has the necessary permission to execute a function.

```
python
Copy code
def requires_permission(permission):
```

```
    def decorator(func):
        def wrapper(user, *args, **kwargs):
            if permission in user['permissions']:
                return func(user, *args, **kwargs)
            else:
                print("Permission denied!")
                return None
        return wrapper
    return decorator

@requires_permission("admin")
def delete_account(user):
    print(f"Deleting account for {user['name']}")

user = {"name": "Alice", "permissions": ["admin", "user"]}
delete_account(user)  # Allowed

user2 = {"name": "Bob", "permissions": ["user"]}
delete_account(user2)  # Denied
```

Output:

```
rust
Copy code
Deleting account for Alice
Permission denied!
```

- In this case, `requires_permission` is a decorator that checks if a user has the necessary permissions to delete an account.

Memoization (Caching Decorator)

You can use decorators to cache the results of expensive function calls to optimize performance.

```
python
Copy code
```

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def expensive_computation(x):
    print(f"Computing {x}...")
    return x * x

print(expensive_computation(4))  # Computes and caches
print(expensive_computation(4))  # Uses cached result
```

Output:

```
Copy code
Computing 4...
16
16
```

- The second time the function `expensive_computation(4)` is called, the result is returned from the cache, avoiding redundant computation.

---

12. **Best Practices**

When to Use Decorators

Decorators are useful for:

1. **Code reuse**: You can apply decorators to multiple functions to add the same behavior, such as logging or timing, without repeating the code.
2. **Separation of concerns**: Decorators allow you to separate auxiliary functionality (like logging, authentication, etc.) from the core logic of your function.

3. **Code readability**: Using decorators can make your code cleaner and more readable when used in moderation.

When to Avoid Decorators

Avoid using decorators when:

1. **Code complexity**: Overuse of decorators can make the code harder to follow, especially when chained or when they involve multiple layers.
2. **Misuse of functionality**: If you're applying a decorator to a function when the behavior should be part of the function itself (i.e., it doesn't logically belong as a decorator), it may lead to confusion.

Avoiding Misuse and Making Code More Readable

Here are some tips to ensure you're using decorators properly:

1. **Limit chaining**: Too many decorators can make the code difficult to trace. Keep the chaining minimal and simple.
2. **Use meaningful names**: Use descriptive names for decorators, so it's obvious what functionality they are adding. For example, `@authenticate_user` is better than `@decorator1`.
3. **Preserve function metadata**: Always use `functools.wraps` to preserve the original function's metadata (like name and docstring), which can be useful for debugging.
4. **Keep it simple**: When possible, keep your decorators simple and focused on one task. Don't try to do too much in a single decorator.

▼ 18. Error Handling and Debugging

Index

- **Example of Exception Handling**
  - File Handling Example with `FileNotFoundError`
- **Key Components of Exception Handling**
  - Explanation of `try`, `except`, `else`, `finally`
- **Raising Exceptions**
  - Example with `ValueError`
- **User-Defined Exceptions**
  - Creating Custom Exceptions
  - Example with `CustomError`

Basic Syntax of Exception Handling

```
try:
    # Code that may raise an exception
    risky_code()
except SomeException as e:
    # Code that runs if the specific exception occurs
    print("An error occurred:", e)
else:
    # Code that runs if no exception occurs
    print("Success!")
finally:
    # Code that always runs, regardless of whether an exception occurre
d or not
    print("Cleanup complete.")

#Example
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
else:
    print("File read successfully:", content)
finally:
```

```
    file.close()
    print("File closed.")
```

Key Components

1. `try` **block**: Code that might throw an exception is placed here.
2. `except` **block**: Code that executes if a specific exception occurs. You can catch specific exceptions by specifying the exception type.
3. `else` **block**: Optional. Runs only if no exceptions were raised in the `try` block.
4. `finally` **block**: Optional. Runs regardless of whether an exception occurred or not, often used for cleanup tasks.

Raising Exceptions

```
x = -1
if x < 0:
    raise ValueError("x cannot be negative")
```

User-defined exceptions

```
class CustomError(Exception):
    """Exception raised for a specific custom error condition."""
    pass

# Example usage
try:
    raise CustomError("This is a custom error!")
except CustomError as e:
    print("Caught a custom error:", e)
```

▼ 19. Python Numpy

NumPy is a powerful numerical computing library for Python that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

`Install : pip install numpy`

`Import : import numpy as np`

Arrays :

Method	Description	Example
`np.array( iterable)`	Creates array from list	`arr = np.array([1, 2, 3], dtype = int)`
`np.zeros(shape)` `np.ones(shape)`	Creates array of zeros or one	`arr = np.zeros((2, 3))`
`np.arange(start, stop, step)`	Creates evenly spaced array	arr = np.arange(5,15,2) o/p : `[ 5 7 9 11 13]`
`np.linspace(start, stop, num)`	Creates array with evenly spaced values	arr = np.linspace(0, 10, 5) o/p : [0. 2.5 5. 7.5 10.]
`np.random.rand(shape)` `np.random.randint(low, high, size)`	Creates array of random floats/integers	arr = np.random.rand(2, 2) arr = np.random.randint(1, 7, (3, 3))
`reshape(new_shape)`	Changes array shape	arr.reshape(3, 4)
`transpose()`	Transposes axes of an array	arr.transpose()
`flatten()`	Flattens array into one dimension	arr.flatten()

`concatenate((arr1, arr2), axis=0)`	Joins arrays along an axis	np.concatenate((arr1, arr2), axis=0)
`vstack((arr1, arr2))`	Stacks arrays vertically (row-wise)	np.vstack((arr1, arr2))
`vstack((arr1, arr2))` `hstack((arr1, arr2))`	Stacks arrays vertically/horizontally (row-wise)	np.vstack((arr1, arr2)) np.hstack((arr1, arr2))
`split(arr, indices_or_sections)`	Splits array into multiple sub-arrays	np.split(arr, 3)
shape	Returns shape of the array	arr.shape
size	Returns total number of elements	arr.size
`ndim`	Returns number of dimensions of the array	arr.ndim
`dtype`	Returns data type of the elements	arr.dtype
+, -, *, /, **	Arithmetic operations	result = arr1 + arr2
==, !=, <, >, <=, >=	Comparison operations	mask = arr > 5
`np.sin(), np.cos(), np.exp(), np.log(), etc.`	Mathematical functions	result = np.sin(arr)
Linear Algebra	Matrix multiplication, matrix inversion, etc.	np.dot(arr1, arr2), np.linalg.inv(arr)
Statistical Functions	Mean, standard deviation, sum, etc.	np.mean(arr), np.std(arr), np.sum(arr)
Copying	Create a deep copy of the array.	**arr.copy()**
Comparison	Element-wise comparison.	**`np.equal(arr1, arr2)`**, **`np.greater(arr1, arr2)`**
Indexing and Slicing	arr[index] arr[start:stop:step]	Indexing to access elements. Slicing to extract sub-arrays.

	Sorting	
np.sort(arr, axis=None)	Returns a sorted copy of the array.	sorted_arr = np.sort(arr, axis=1)
np.argsort(arr, axis=None)	Returns the indices that would sort the array.	sorted_indices = np.argsort(arr)
np.lexsort(keys, axis=None)	Indirect stable sort using a sequence of keys.	sorted_indices = np.lexsort((b, a))
	Searching	
np.argmax(arr, axis=None)	Returns indices of the maximum element(s).	max_idx = np.argmax(arr)
np.nanargmax(arr, axis=None)	Like argmax, ignoring NaNs.	
np.argmin(arr, axis=None)	Returns indices of the minimum element(s).	
np.nanargmin(arr, axis=None)	Like argmin, ignoring NaNs.	
np.where(condition)	Returns elements where the condition is True.	
np.searchsorted(arr, values)	Finds indices where to insert values to maintain order.	
	Counting	
np.count_nonzero(arr)	Counts non-zero elements in the array.	nonzero_count = np.count_nonzero(arr)
	Additional	
np.partition(arr, kth, axis=-1)	Partially sorts, placing the kth smallest element at its final position.	
np.argpartition(arr, kth, axis=-1)	Returns indices that would partially sort the array.	

np.percentile(array, 25%)	The 25th percentile is the value below which 25% of the data falls,	np.percentile(arr, 75)
	Matrix	
Determinant (`np.linalg.det()`):		det = np.linalg.det(matrix)
Matrix Inversion (`np.linalg.inv()`):		matrix_inv = np.linalg.inv(matrix)
Linear Equation Solver (`np.linalg.solve()`):	Solves a system of linear scalar equations specified by `a * x = b` for `x`, where `a` is the coefficient matrix and `b` is the right-hand side.	solution = np.linalg.solve(a, b)
Transpose (`np.transpose()` or array attribute `.T`):		transposed_arr = np.transpose(arr) transposed_arr = arr.T
Dot Product (`np.dot()`):	it performs matrix multiplication.	dot_product = np.dot(arr1, arr2)

1. reshape(new_shape)

```
import numpy as np

original_array = np.arange(12)
reshaped_array = original_array.reshape(3, 4)

print("Original Array:")
print(original_array)
print("\nReshaped Array:")
print(reshaped_array)

o/p :
Original Array:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

```
Reshaped Array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

2. transpose()

```
arr1 = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
arr2=arr1.transpose()
print("Array 1 \n",arr1)
print("Array 2 \n",arr2)

o/p :
Array 1
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Array 2
 [[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

3. concatenate((arr1, arr2), axis=0)

```
import numpy as np
arr1 = np.array([[1, 2, 3],
                 [4, 5, 6]])
print("Array 1:")
print(arr1)

arr2 = np.array([[7, 8, 9],
                 [10, 11, 12]])

print("\nArray 2:")
print(arr2)
```

```
result1 = np.concatenate((arr1, arr2), axis=0)
print("\nConcatenated Result along axis 0:")
print(result1)

result2 = np.concatenate((arr1, arr2), axis=1)
print("\nConcatenated Result along axis 1:")
print(result2)

o/p :
Array 1:
[[1 2 3]
 [4 5 6]]

Array 2:
[[ 7  8  9]
 [10 11 12]]

Concatenated Result along axis 0:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Concatenated Result along axis 1:
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

4. Splits array into multiple sub-arrays

```
import numpy as np

arr = np.arange(12)
print("Original Array:",arr)

# Split the array into 3 equal parts
result = np.split(arr, 3)
print("result : ",result)
```

```
print("\nSplit Result:")
for subarray in result:
    print(subarray)

o/p :
Original Array: [ 0  1  2  3  4  5  6  7  8  9 10 11]
result :  [array([0, 1, 2, 3]), array([4, 5, 6, 7]), array([ 8,  9, 10, 11])]

Split Result:
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

5. **Array Information**

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
print("arr.shape : ",arr.shape)
print("arr.size : ",arr.size)
print("arr.ndim : ",arr.ndim)
print("arr.dtype : ",arr.dtype)

o/p :
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
arr.shape :  (2, 2, 3)
arr.size :  12
arr.ndim :  3
arr.dtype :  int64
```

6. **Element-wise Operations:**

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 2, 6])
print(arr1 + arr2)  # [5 4 9]
print(arr1 - arr2)  # [-3  0 -3]
print(arr1 * arr2)  # [ 4  4 18]
print(arr1 / arr2)  # [0.25 1.   0.5 ]
print(arr1 ** 2)  # [1 4 9]
print(arr1 == arr2) # [False  True False]
print(arr1 < arr2) # [ True False  True]

arr3 = np.array([1, 7, 3, 9, 2, 8, 4, 6])
mask = arr3 > 5

print(mask) # [False  True False  True False  True False  True]
print(arr[mask]) # [7 9 8 6]
```

7. statistics

```
import numpy as np

# Create a NumPy array
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5])

# Calculate mean
mean_value = np.mean(arr)
print(f"Mean: {mean_value}")

# Calculate standard deviation
std_value = np.std(arr)
print(f"Standard Deviation: {std_value}")

# Find minimum and its index
min_value = np.min(arr)
min_index = np.argmin(arr)
print(f"Minimum Value: {min_value}, Index: {min_index}")
```

```
# Find maximum and its index
max_value = np.max(arr)
max_index = np.argmax(arr)
print(f"Maximum Value: {max_value}, Index: {max_index}")

# Calculate sum
sum_value = np.sum(arr)
print(f"Sum: {sum_value}")

# Calculate variance
variance_value = np.var(arr)
print(f"Variance: {variance_value}")

o/p :
Mean: 4.0
Standard Deviation: 2.449489742783178
Minimum Value: 1, Index: 1
Maximum Value: 9, Index: 5
Sum: 36
Variance: 6.0
```

8. Percentile

```
import numpy as np

# Create a NumPy array
arr = np.array([1,2,3,4])

#These lines calculate the 25th and 75th percentiles of the array using
#np.percentile(). The 25th percentile is the value below which 25% of
#the data falls, and the 75th percentile is the value below which 75%
#of the data falls.
# Calculate the 25th and 75th percentiles
percentile_25 = np.percentile(arr, 25)
percentile_75 = np.percentile(arr, 75)
```

```
print(f"25th Percentile: {percentile_25}") # 25th Percentile: 1.75
print(f"75th Percentile: {percentile_75}") # 75th Percentile: 3.25
```

need to check

- **Linear Algebra:**
  - Matrix operations (`dot()` , `transpose()` , `solve()` , `inv()` , `det()`)
- **Random Numbers:**
  - Generating random numbers from various distributions (`random.rand()` , `random.randn()` , `random.normal()` , etc.)

Advanced Topics:

- **File I/O:**
  - Saving and loading arrays to/from files (`save()` , `load()`)

▼ 20. Scikit-Learn (INRIA)

Machine Learning Techniques

- **Linear Models**
  - `LinearRegression` – Linear regression for predicting a continuous target.
  - `LogisticRegression` – Logistic regression for binary or multiclass classification.
  - `Ridge` – Ridge regression (L2 regularization).
  - `Lasso` – Lasso regression (L1 regularization).
  - `ElasticNet` – Combines L1 and L2 penalties.
- **Support Vector Machines (SVM)**
  - `SVC` – Support vector classification.
  - `SVR` – Support vector regression.
  - `LinearSVC` – Linear SVM classification.
  - `LinearSVR` – Linear SVM regression.
- **Decision Trees and Ensembles**
  - `DecisionTreeClassifier` – Classification with decision trees.

    - `DecisionTreeRegressor` – Regression with decision trees.
    - `RandomForestClassifier` – Random forest classifier.
    - `RandomForestRegressor` – Random forest regressor.
    - `GradientBoostingClassifier` – Gradient boosting classifier.
    - `GradientBoostingRegressor` – Gradient boosting regressor.
    - `AdaBoostClassifier` – Adaptive boosting classifier.
    - `AdaBoostRegressor` – Adaptive boosting regressor.
- **Naive Bayes**
    - `GaussianNB` – Gaussian Naive Bayes for continuous data.
    - `MultinomialNB` – Multinomial Naive Bayes for discrete data.
    - `BernoulliNB` – Bernoulli Naive Bayes for binary features.
- **Nearest Neighbors**
    - `KNeighborsClassifier` – K-nearest neighbors classifier.
    - `KNeighborsRegressor` – K-nearest neighbors regressor.
- **Clustering**
    - `KMeans` – K-means clustering.
    - `DBSCAN` – Density-based clustering.
    - `AgglomerativeClustering` – Hierarchical clustering.
- **Dimensionality Reduction**
    - `PCA` – Principal component analysis.
    - `TruncatedSVD` – Truncated Singular Value Decomposition.
    - `KernelPCA` – Kernel Principal Component Analysis.
    - `TSNE` – t-distributed Stochastic Neighbor Embedding.

Deep Learning Techniques (Limited in Scikit-Learn)

- **Neural Network Models**
    - `MLPClassifier` – Multilayer perceptron classifier (feedforward neural network).

- `MLPRegressor` – Multilayer perceptron regressor.

▼ 21. TensorFlow (Google Brain)

Machine Learning Techniques

- **Linear Models**
  - `tf.keras.layers.Dense` – Basic dense layer that can be used to build linear models or neural networks.
- **Boosted Trees**
  - `tf.estimator.BoostedTreesClassifier` – Boosted tree classifier.
  - `tf.estimator.BoostedTreesRegressor` – Boosted tree regressor.

Deep Learning Techniques

- **Sequential and Functional API**
  - `tf.keras.Sequential` – Build a simple, linear stack of layers.
  - `tf.keras.Model` – Functional API for building complex neural networks.
- **Core Layers**
  - `tf.keras.layers.Dense` – Fully connected (dense) layer.
  - `tf.keras.layers.Conv2D` – 2D convolutional layer for CNNs.
  - `tf.keras.layers.Conv3D` – 3D convolutional layer.
  - `tf.keras.layers.LSTM` – Long short-term memory layer for sequence modeling.
  - `tf.keras.layers.GRU` – Gated recurrent unit for sequence modeling.
  - `tf.keras.layers.Bidirectional` – Wrapper to make RNNs bidirectional.
  - `tf.keras.layers.Embedding` – Embedding layer for NLP tasks.
  - `tf.keras.layers.Dropout` – Dropout layer for regularization.
  - `tf.keras.layers.BatchNormalization` – Batch normalization for faster training and improved stability.
- **Preprocessing Layers**
  - `tf.keras.layers.Rescaling` – Rescale inputs (e.g., image scaling).
  - `tf.keras.layers.Normalization` – Normalize inputs.

- **Attention Mechanisms**
  - `tf.keras.layers.Attention` – Basic attention layer.
  - `tf.keras.layers.MultiHeadAttention` – Multi-head attention layer for transformers.

▼ 22. Keras (**François Chollet**, initially as an independent project)

Machine Learning Techniques

- **Simple Layers for Linear Models**
  - `keras.layers.Dense` – Can be used to create basic linear or logistic regression models.

Deep Learning Techniques

- **Sequential and Functional API**
  - `keras.Sequential` – Build models layer-by-layer for simple architectures.
  - `keras.Model` – Functional API for more complex architectures, including multi-input and multi-output models.
- **Core Layers**
  - `keras.layers.Dense` – Fully connected layer.
  - `keras.layers.Conv1D` – 1D convolution layer, useful for time-series data.
  - `keras.layers.Conv2D` – 2D convolutional layer for image processing.
  - `keras.layers.Conv3D` – 3D convolutional layer for 3D data.
  - `keras.layers.MaxPooling2D` – 2D max pooling layer.
  - `keras.layers.GlobalAveragePooling2D` – Global average pooling layer for CNNs.
  - `keras.layers.LSTM` – Long short-term memory layer for RNNs.
  - `keras.layers.GRU` – Gated recurrent unit.
  - `keras.layers.Bidirectional` – For bidirectional RNNs.
  - `keras.layers.Embedding` – Embedding layer, commonly used for NLP.
- **Regularization Layers**
  - `keras.layers.Dropout` – Dropout layer for reducing overfitting.

- `keras.layers.BatchNormalization` – Batch normalization for stabilization.

- **Attention Mechanisms**
  - `keras.layers.Attention` – Simple attention mechanism.
  - `keras.layers.MultiHeadAttention` – Multi-head attention for advanced architectures like transformers.

▼ 23. PyTorch (Meta AI)

1. Machine Learning Techniques (Manually Implemented)

- **Linear Models**
  - `torch.nn.Linear` – Applies a linear transformation to input data, useful for linear regression or logistic regression.

2. Deep Learning Techniques

- **Core Layers**
  - `torch.nn.Linear` – Fully connected layer, commonly used in neural network architectures.
  - `torch.nn.Conv1d` – 1D convolution for processing sequential data like time series.
  - `torch.nn.Conv2d` – 2D convolution for image processing.
  - `torch.nn.Conv3d` – 3D convolution for volumetric data.
  - `torch.nn.MaxPool2d` – Reduces spatial dimensions of 2D data via max pooling.
  - `torch.nn.AvgPool2d` – Reduces spatial dimensions of 2D data via average pooling.
  - `torch.nn.AdaptiveAvgPool2d` – Dynamically adapts average pooling to any output size.

- **Recurrent Layers**
  - `torch.nn.RNN` – Basic RNN layer for sequential data.
  - `torch.nn.LSTM` – LSTM layer, ideal for long-sequence processing.
  - `torch.nn.GRU` – GRU layer, an alternative to LSTM with fewer parameters.

- **Normalization Layers**
  - `torch.nn.BatchNorm1d` – Batch normalization for 1D inputs to stabilize training.
  - `torch.nn.BatchNorm2d` – Batch normalization for 2D inputs, often used in CNNs.
  - `torch.nn.LayerNorm` – Normalizes inputs independently across features.
- **Regularization Layers**
  - `torch.nn.Dropout` – Randomly zeroes some weights to prevent overfitting.
  - `torch.nn.AlphaDropout` – Dropout that preserves self-normalization.
- **Attention Mechanisms**
  - `torch.nn.MultiheadAttention` – Multi-head attention mechanism for transformer models.
  - `torch.nn.Transformer` – Transformer module for sequence-to-sequence tasks.
  - `torch.nn.TransformerEncoder` – Transformer encoder module.
  - `torch.nn.TransformerDecoder` – Transformer decoder module.
- **Embedding Layers**
  - `torch.nn.Embedding` – Embedding layer for mapping discrete inputs (e.g., words) to vectors.
  - `torch.nn.EmbeddingBag` – Efficient embedding for bag-of-words data without padding.

3. Advanced Techniques and Utilities

- **Differentiation and Autograd**
  - `torch.autograd` – Automatic differentiation for backpropagation.
- **Custom Loss Functions**
  - `torch.nn.CrossEntropyLoss` – Computes cross-entropy loss for classification.
  - `torch.nn.MSELoss` – Computes mean squared error loss for regression.
- **Optimization**

- `torch.optim.SGD` – Stochastic Gradient Descent optimizer.
- `torch.optim.Adam` – Adaptive Moment Estimation optimizer for faster convergence.
- `torch.optim.RMSprop` – Optimizer with adaptive learning rate, good for RNNs.

- **Pre-trained Models**
  - `torchvision.models` – Contains a variety of pre-trained models for transfer learning.

Each function is designed to help build and train custom models, from simple architectures to complex neural networks. Let me know if you want more details on any specific function!

▼ 24. Python libraries used for **ML** and **DL**

Here is a comprehensive list of popular Python libraries used for **machine learning (ML)** and **deep learning (DL)** tasks, organized by category:

1. **Core Machine Learning Libraries**

- **Scikit-Learn**: A widely-used library for classical ML algorithms like regression, classification, clustering, and preprocessing.
- **StatsModels**: Offers statistical modeling and tests; great for regression analysis and statistical inference.
- **XGBoost**: An optimized gradient boosting library for performance and speed, especially popular in competitions.
- **LightGBM**: Developed by Microsoft, it's a fast, efficient gradient boosting library that's well-suited for large datasets.
- **CatBoost**: A gradient boosting library by Yandex, optimized for categorical features.
- **H2O.ai**: An open-source platform providing scalable ML models, including AutoML capabilities.

2. **Deep Learning Libraries**

- **TensorFlow**: A comprehensive, scalable library for deep learning with support for both high-level and low-level API control.

- **PyTorch**: Known for dynamic computation graphs, it's highly popular for DL research and production.
- **Keras**: A high-level neural networks API, now part of TensorFlow (`tf.keras`), designed for ease of use and fast prototyping.
- **MXNet**: Backed by Amazon, it’s optimized for performance and scalability, with Gluon API for an imperative-style interface.
- **Chainer**: A flexible, Pythonic DL framework with a dynamic computation graph, similar to PyTorch.
- **Theano**: One of the earliest deep learning libraries; foundational for many others like Keras (now largely obsolete but influential).

3. AutoML Libraries

- **AutoKeras**: An AutoML library built on top of Keras and TensorFlow, for building models with minimal code.
- **TPOT**: Uses genetic programming to automate the ML pipeline creation, optimizing models for best performance.
- **H2O AutoML**: Part of H2O.ai, this provides automated model selection and hyperparameter tuning.
- **MLBox**: An AutoML library focusing on data cleaning, model selection, and optimization.
- **AutoGluon**: An AutoML toolkit by Amazon, particularly effective for tabular data.

4. Natural Language Processing (NLP)

- **NLTK**: A comprehensive library for NLP tasks like tokenization, parsing, and text processing.
- **SpaCy**: Fast and efficient for production NLP, with strong support for named entity recognition (NER) and dependency parsing.
- **Transformers**: Developed by Hugging Face, it provides pre-trained transformer models (e.g., BERT, GPT) for NLP tasks.
- **Gensim**: A library for topic modeling, document similarity, and word embeddings.

- **Flair**: A framework from Zalando Research focused on sequence tagging and text classification, particularly NER.

5. **Computer Vision**

- **OpenCV**: A library for real-time computer vision and image processing.
- **torchvision**: A PyTorch library with datasets, model architectures, and transformations for computer vision tasks.
- **Detectron2**: Facebook's object detection and segmentation library based on PyTorch.
- **Albumentations**: A library for fast and flexible image augmentations for deep learning.
- **KerasCV**: A Keras-based library with specialized components and models for computer vision tasks.

6. **Reinforcement Learning**

- **Stable-Baselines3**: A popular RL library with standard algorithms implemented in PyTorch.
- **Ray RLlib**: A scalable RL library by Ray for distributed RL tasks.
- **TensorFlow Agents (TF-Agents)**: A flexible library for building RL algorithms in TensorFlow.
- **OpenAI Gym**: A toolkit for developing and comparing RL algorithms with standardized environments.
- **Dopamine**: Google's framework for flexible RL experimentation.

7. **Probabilistic Programming**

- **Pyro**: A probabilistic programming library built on PyTorch for Bayesian modeling.
- **TensorFlow Probability**: A TensorFlow library for probabilistic reasoning and statistical analysis.
- **PyMC3**: A probabilistic programming library for Bayesian inference using MCMC.
- **Edward**: Built on TensorFlow, focusing on Bayesian modeling and inference.

8. **Data Preprocessing and Visualization**

- **Pandas**: Essential for data manipulation and analysis, handling structured data.
- **NumPy**: The foundational library for numerical computations and matrix operations in Python.
- **Dask**: A parallel computing library that scales Pandas and NumPy to larger datasets.
- **Matplotlib**: A library for static, animated, and interactive visualizations.
- **Seaborn**: Built on top of Matplotlib, providing more aesthetically pleasing and complex visualizations.
- **Plotly**: An interactive plotting library that supports rich visualizations, including dashboards.

9. **Hyperparameter Optimization**

- **Optuna**: An automatic hyperparameter optimization software framework, suitable for DL and ML.
- **Hyperopt**: A library for hyperparameter tuning, using a variety of search strategies, including Bayesian optimization.
- **Ray Tune**: Part of Ray, providing scalable hyperparameter tuning for distributed training.
- **Scikit-Optimize (skopt)**: An optimization library designed to work with Scikit-Learn models.

10. **Model Deployment and Serving**

- **ONNX**: An open format for representing ML models that makes models portable across different frameworks.
- **TensorFlow Serving**: A library for serving TensorFlow models in production environments.
- **TorchServe**: A serving library for PyTorch models, developed by AWS and Facebook.
- **BentoML**: A flexible platform for ML model deployment across multiple frameworks.

- **MLflow**: An end-to-end machine learning lifecycle management tool, with capabilities for tracking, deployment, and model registry.

Each library specializes in certain types of tasks, and together they provide a comprehensive set of tools to cover the full spectrum of machine learning and deep learning workflows in Python.

▼ 25.