# Python Data Types:

## Numeric Type:

### Integers (int):

* Represents whole numbers, both positive and negative
* No fractional or decimal parts
* Example: 5, -12, 100, 50, 104

#### What built-in functions can I use with Integers?

* abs(): Returns the absolute value of the integer

print (abs(-10)) # Output = 10

* pow(): Raises an integer to the power of another number. It also allows modulo operations as an optional third argument.

print (pow(2,3)) # Output: 8

print (pow(2,3,5) # Output: 3 #2^3 % 5

* divmod(): Returns a tuple containing the quotient and the remainder of the integer division.

print(divmod(10,3)) # Output: (3,1)

* round(): Rounds an integer (or float) to a specified number of decimal places. Its useful when integers are mixed with other types during calculations.

print(round(5.98) # Output: 6

* bin(): converts an Integer to its binary representation

print(bin(10)) # Output: 0b1010

* hex(): Converts an integer to its hexadecimal representation

print(hex(255)) # Output: ‘0xff’

* oct(): Converts an integer to its Octadecimal representation

print(oct(8)) # Output: ‘0010’

* isinstance: Checks if the variable belongs to the int class

print(isinstance(42,int)) # Output: True

#### Performance Considerations while using Integers

When working with integer data type in Python, performance considerations often revolve around how integers are represented, manipulated and utilized in operations. Here are some advanced points to keep in mind.

1. Memory Usage: Python integers are not limited by specific sizes (like int32 or int64 in other languages) because Python uses arbitrary-precision integers. While this allows operations on very large integers, it can increase memory usage, especially dealing with extremely large numbers.

Optimization Tip: Minimize the use of unnecessary large integers by carefully choosing datatype for performance-critical applications.

1. CPU Performance

Operations on integers are generally fast because Python optimizes basic arithmetic. However, integer division (using / or //) is slower than addition and subtraction due to the complexity of division algorithms

Optimization Tip: Prefer simpler arithmetic operations over division and modulus in performance-critical loops

1. Type Conversion

Frequent conversion between int, float, and complex data types can introduce performance overhead, especially if done repeatedly in loops or large datasets.

Optimization Tip: Avoid unnecessary type conversion by ensuring variables remain in the same numeric type throughout calculations.

1. Large Integer Arithmetic

While Python can handle large integers, computations like factorials or modular exponentiation may become computationally expensive due to their inherent complexity

Optimization Tip: For large-scale operations, leverage libraries such as gmpy2, which provides faster integer arithmetic for very large numbers.

1. Bitwise Operations

Python supports bitwise operations on integers, which are incredibly efficient for tasks like toggling bits, creating flags, or encoding data. However improper use can lead to bugs and reduce readability

Optimization Tip: Use bitwise operations judiciously and document their purpose for maintainability.

1. Integer Caching

Python internally caches small integers (typically -5 to 256) for performance reasons. Reusing integers within this range is faster compared to creating new integer objects outside this range.

Optimization Tip: When possible, reuse small integers rather than creating large one unnecessarily.

1. Parallelism

When performing bulk integer operations, single-threaded execution may become a bottleneck. Using multithreading or multiprocessing can improve performance, especially for mathematical simulations.

Optimization Tip: Use libraries like concurrent.futures or multiprocessing for parallel computation.

1. Performance Debugging

Performance monitoring tools like cProfile and timeit can be used to benchmark integer-related operations to identify bottlenecks in your code

Optimization Tip: Profile your code to find slow areas, optimize loops and avoid redundant calculations.

1. Hardware Limitations

Certain integer operations may benefit from the hardware’s specific capabilities such as faster arithmetic on GPUs or specialized CPUs. This is particularly relevant in scientific computing.

Optimization Tip: For high-performance needs, use libraries like NumPy that can leverage hardware optimization.

1. Python Version

Some performance optimizations for integers are tied to Python versions. Ensure you are using the latest version of Python to take advantage of Improvements.

### Floating-Point Numbers (float):

* Represents real numbers that can have decimal points
* Can also be expressed using scientific notations (e.g. 1.2e3 for 1200)
* Example: 5.74, 90.12, -5.3, 1.34e4

#### Performance Considerations while using Float

Using the float data type in Python is convenient for handling real numbers, but there are a few performance and accuracy considerations to keep in mind.

1. Precision Loss: Floats are represented in binary format internally, which can cause rounding errors, especially with very large or very small numbers. For instance:

0.1 + 0.2 # Output: 0.30000000004

These inaccuracies can propagate in complex calculations

1. Memory Usage: Floats consume more memory than integers. In Python, a flot typically uses 64 bits, which might not be ideal if memory is a constraint and you can work with integers instead.
2. Performance: Arithmetic operations on floats can be slower than on integers, as floating-point arithmetic requires computational resources. If your application involves intensive numerical computations, this could impact performance.
3. Comparisons: Comparing floating-point numbers for equality can be unreliable due to precision issues. It’s better to check if the numbers are “close enough” using a small tolerance:

a =0.1 + 0.2

b = 0.3

print (a == b) # Output: False

import math

math.isclose(0.1 + 0.2, 0.3) # Output: True

1. Alternatives:

For high precision: Use the decimal module, which avoids some floating-point inaccuracies by representing numbers exactly.

For large-scale numerical computations: Use specialized libraries like NumPy, which optimize floating-point performance for arrays and matrices.

### Complex Numbers (complex):

* Represents numbers with a real and imaginary part
* Written in the form a + bj, where a is the real part and b is the imaginary part, j is the imaginary unit (SquareRoot of -1)
* Example: 3+4j, 1-2j

C1 = 5 + 3j

C2 = -2 + 4j

Addition = C1 + C2

Subtraction = C1 - C2

Multiplication = C1 \* C2

print(“Value of Addition variable is: “, Addition) # Output: (3+7j)

print(“Value of Subtraction variable is: “, Subtraction) #Output: (7-1j)

print(“Value of Multiplication variable is: “, Multiplication) #Output: (-22+14j)

* Accessing Real and Imaginary Part
* C = 7 + 2j

print(“Real Part: “, C.real) # Output: 7.0

print(“Imaginary Part: “, C.imag) #Output: 2.0

#### Performance Considerations while using Complex Numbers

When working with complex numbers in Python, their performance considerations depend on the scale of operations, the size of data being processed, and the libraries being used. Here are some key points:

1. Built-In Python Performance: Python’s built-in support for complex numbers is efficient for basic operations (addition, subtraction, multiplication etc) on small-scale datasets. However, for high-performance applications involving large datasets or repeated computations, it may not be optimal:
   1. Built-In complex number operations are performed in software rather than hardware, which may not be as fast.
   2. For small datasets, the difference is negligible, but for larger-scale computations, performance bottlenecks can occur
2. Using NumPy for efficiency: NumPy provides optimized support for complex numbers and is much faster than native Python for vectorized operations.

Benefits of NumPy:

* Vectorization: Performs operation on entire arrays without loops
* Low-level optimizations: Uses C and hardware acceleration for efficient computation

1. Memory Usage: Complex number in Python are stored as two float values (real and imaginary parts), which means they consume more memory compared to standard numbers. For large arrays of complex numbers, memory usage can become significant. If memory usage is critical, you can use NumPy arrays (dtype=np.complex64) to manage memory better.
2. Alternatives for advanced use cases: For scientific and engineering applications that require extensive complex operations, specialized libraries like SciPy or performance libraries like TensorFlow might be better suited. SciPy’s signal processing module uses complex number for Fourier Transforms. TensorFlow is optimized for GPU/TPU-based computation.

Each of these numeric types is a class in Python, and variables of these types are instances of their respective classes. You can use the **type()** function to check the data type of a value. For instance:

a = 10 # Integer

b = 3.14 # Float

c = 1 + 2j # Complex

print(type(a)) # Output: <class 'int'>

print(type(b)) # Output: <class 'float'>

print(type(c)) # Output: <class 'complex'>

## Sequence Type:

Sequence data types in Python are used to store collections of items in an ordered manner, allowing for efficient organization and retrieval of elements. Here are the main sequence data types

### String (str):

* Strings are arrays of Unicode characters, used for text data
* Immutable, meaning you cannot change a string after creation
* Example: s = “Hello World”

#### Key characteristics of Strings in Python

1. **Immutable**
   1. Strings cannot be changed after they are created. Any modification creates a new string

Example: s = “Hello World”

s = s + “ Of Python” # Creates a new string

print (s) # Output: Hello World Of Python

1. **Single or Double Quotes**
   1. Strings can be created using either single or double quotes
2. **Triple Quotes for multi-line strings**
   1. Strings spanning multiple lines can be created using triple quotes (‘’’ or “””)
3. **Accessing Characters:**
   1. Strings are indexed and support slicing. Index starts at 0
   2. Negative index starts from the end. (-1 is the last character of the string)

Example:

S = “Python”

S[0] # Output: P

S[-1] # Output: n

#### Indexing in Strings

Indexing in Strings refers to the process of accessing individual characters in a string using their position, also known as their index. Each character in a string is assigned a unique index number, starting from 0. Python supports both positive indexing (from the start) and negative indexing (from the end)

#### Types of Indexing

1. Positive Indexing

Starts from 0 for the first characters

Each subsequent character increases the index by 1

Example:

Text = “Allen”

print(Text[0]) # Output: A

print(Text[3]) # Output: e

1. Negative Indexing

Starts from -1 for the last character and goes backward

Useful for accessing characters relative to the end of the strings.

Example:

Text = “Allen”

print(Text[-1]) # Output: n

print(Text[-3]) # Output: l

#### String Slicing

String slicing in Python is a technique used to extract a portion (or “slice”) of a string based on specific start, stop and step parameters. The Syntax for slicing is:

String[start:stop:step]

Here’s an explanation of each parameter:

* start: The index where the slicing begins (inclusive). If omitted, the default is 0
* stop: The index where the slicing ends (exclusive). If omitted, the default is the length of the string.
* step: The interval between indices. If omitted, the default is 1

Examples:

text = “Programming”

# Basic Slicing

print(text[0:5]) # Output: Progr

print(text[3:8]) # Output: gramm

# Omitting start or stop

print(text[:6]) # Output: Progra

print(text[4:]) # Output: ramming

# Using Negative Indices

print(text[-4:]) # Output: ming

print(text[:-6]) # Output: Progr

# Using a step

print(text[0:10:2]) # Output: Pormig

print(text[::-1]) # Output: gnimmargorP

#### Common String Methods

* **.lower() :** Converts string to a lowercase. Example: “Hello”.lower() # Output: hello
* **.upper():** Converts string to uppercase. Example: “Hello”.upper() # Output : HELLO
* **.strip():** Removes leading or trailing white spaces or characters. Example: “ Hello “.strip -> Output: Hello
* **.split():** Splits string into lists based on the delimiter (default: Space). Example: “a,b,c”.split(“,”) -> # Output: [‘a’, ’ b’, ’ c’]
* **.join():** Joins elements of a list into a single string. Example: “,”.join([‘a’, ‘b’, ‘c’]) -> #Output: abc
* **.replace(old,new):** Replaces occurrences of a substring with another substring. Example: “Python”.replace(“Py”,”Cy”) -> # Output: Cython
* **.startswith():** Checks if a string starts with a given substring. Example: “Python”.startswith(“Py”) -> # Output: True
* **.endswith():** Check if a string ends with a given substring.

Example: “Python”.endswith(“on”) - > True

* **.count(substring):** Counts the number of times a substring appears.

Example: text = "hello hello world"

print(text.count("hello")) # Output: 2

* **.partition(separator):** Splits the string at the first occurrence of the separator from the left.

Example**:** text = “apple-orange-banana”

result = text.partition(“-“)

print (result) # Output: ‘apple’ , ‘-‘, “orange-banana”

* **.rpartition(separator):** Splits the string at the first occurrence of the separator from the right.

Example**:** text = “apple-orange-banana”

result = text.rpartition(“-“)

print (result) # Output: ‘apple-orange’ , ‘-‘, “banana”

* **zfill(width):** width is the total length of the string after padding with zeros

Example: number = 22

print(“On padding with width 5: “, number.zfill(5))

#### String Formatting

1. **Using f-strings (Python 3.6+):**
   1. Embed variables directly into string using {}

name = “Allen”

print (f“Name is {name}”) # Output: Name is Allen

1. **Using format():**
   1. Another way to insert variables into strings

print (“I love {}”.format(“Python”)) # Output: I love Python

1. **Old-style(% Operator):**
   1. Still supported but less commonly used

print (“I scored %d out of %d” % (95, 100)) # Output: I scored 95 out of 100

#### Escape Characters

Special Characters can be included using escape sequences:

* \n : Newline
* \t : Tab
* \\ : Backslash
* \’ : Single quote
* \” : Double quote

#### Performance Considerations while using String

When working with strings in Python, performance considerations often arise due to their immutable nature and how they are handled during operations. Here are key factors to keep in mind:

1. Immutability and Memory Usage: Strings in Python are immutable, meaning once created, they cannot be modified. This immutability can lead to performance issues in scenarios where strings are repeatedly modified or concatenated:

Problem: Repeated concatenation creates a new string each time, leading to higher memory usage and slower performance.

Solution: Use join() method instead of + for concatenating multiple strings efficiently

Example:

# Ineffiecient

result = “”

for i in range (10000):

result += str(i)

# Efficient

result = “”.join(str(i) for i in range(10000))

1. Large-Scale String Operations: When working with large strings or performing frequent manipulations:

Problem: Operations like slicing, splitting, or replacing strings may impact performance.

Solution: Use libraries like NumPy or pandas for handling large-scale text data efficiently

1. Memory Considerations: Strings consume more memory compared to some other data types:

Use generators or streaming techniques to handle large datasets without loading the entire string into memory at once.

1. Regular Expressions: Using Regular expressions (re module) for complex string matching can improve performance for certain tasks but may introduce overhead if not optimized properly:

import re

pattern = r”\d+”

text = “Python 123 and 456”

matches = re.findall(pattern, text)

print(matches) # Output: [‘123’, ‘456’]

Precompile regular expressions for better efficiency when used repeatedly

Compiled = re.compile(r”\d+”)

Matches = compiled.findall(text)

1. Alternatives: For extremely large-scale string processing tasks, consider specialized tools like: Pandas for processing tabular text data. Text-based libraries like spaCy for NLP tasks.

### List (list):

In Python, a list is a built-in data type that is used to store collections of items. It is an ordered, mutable (changeable), and heterogenous data structure, meaning it can contain elements of different data types (e.g., integers, strings or even other lists).

Characteristics of Lists:

* Ordered: Elements in a list maintain the order in which they are inserted. You can access them using their index.
* Mutable: A mutable collection that can hold items of various data types. Allows insertion, deletion, and modification of elements

Example: l = [1, “hundred”, 5.9, “7”]

* Heterogeneous: A single list can contain items of different data types
* Dynamic Size: Lists in Python can grow or shrink dynamically as elements are added or removed.
* A List is created by enclosing elements in a square brackets [], separated by commas.

#### Common List Methods

* Accessing List Elements:

my\_list = [10, 20, 30]

print (my\_list[1]) # Output: 20

* Modifying List Elements:

my\_list[1] = 40

print (my\_list) # Output: 10, 40, 30

* Adding List Elements:
  + Append Method append(): Adding element to the end of the list

my\_list.append(50)

print(my\_list) # Output: 10,40,30, 50

* + Insert Method insert(): Add element at a specific index of list

my\_list.insert(1,15)

print(my\_list) # 10, 15, 40, 30, 50

* + extend method extend(): Extends the list by appending elements from another iterable (e.g., list or tuple)

my\_list = [1, 2]

my\_list.extend([3, 4])

print(my\_list) # Output: [1, 2, 3, 4]

* Removing List Elements:
  + Using remove(): To remove or delete an element by its value

my\_list.remove(15)

print(my\_list) # 10,40,30,50

* + Using pop(): Removes and returns the element at the specified index (default is the last element).

my\_list.pop(2)

print(my\_list) # Output: 10,40,50

* Iterate through List:
  + for i in my\_list:

print (i)

* len(list): Gives the length of the list/number of elements in a list
* sum(list): Calculates the sum of all numeric elements in a list
* min(list): Returns the smallest element in the list
* max(list): Return the largest element in the list
* sorted(list): Returns sorted version of the list (does not modify the original)
* index(item[, start[, end]]): Returns the index of the first occurrence of the specified element

my\_list = [1, 2, 3]

print(my\_list.index(2)) # Output: 1

* count(item): Returns the number of occurrences of the specified element.

my\_list = [1, 2, 2, 3]

print(my\_list.count(2)) # Output: 2

* sort([key[, reverse]]): Sorts the list in place
* reverse(): Reverses the elements in the list

my\_list = [1, 2, 3]

my\_list.reverse()

print(my\_list) # Output: [3, 2, 1]

* clear(): Removes all elements in the list

my\_list = [1, 2, 3]

my\_list.clear()

print(my\_list) # Output: []

* copy(): Returns a shallow copy of a list

my\_list = [1, 2, 3]

new\_list = my\_list.copy()

print(new\_list) # Output: [1, 2, 3]

The copy() function in Python is used to create a **shallow copy** of a list. A shallow copy means that a new list is created, but the elements within the list are **references** to the original objects. If the original list contains mutable objects like other lists or dictionaries, changes to these objects will affect both the original and the copied list.

**# Example of shallow copy**

original\_list = [1, 2, [3, 4]]

copied\_list = original\_list.copy()

# Modifying the original nested list

original\_list[2][0] = 99

print(original\_list) # Output: [1, 2, [99, 4]]

print(copied\_list) # Output: [1, 2, [99, 4]] (nested list is affected)

Deep Copy

import copy

# Example of deep copy

original\_list = [1, 2, [3, 4]]

deep\_copied\_list = copy.deepcopy(original\_list)

# Modifying the original nested list

original\_list[2][0] = 99

print(original\_list) # Output: [1, 2, [99, 4]]

print(deep\_copied\_list) # Output: [1, 2, [3, 4]] (nested list is unaffected)

#### Performance Considerations while using List

1. Time and Complexity:
   1. Appending to a list is generally O(1), thanks to Python’s dynamic resizing.
   2. Accessing elements (indexing) is O(1)
   3. Inserting or Deleting elements can be O(n) due to shifting elements
   4. Inserting over list in O(n), which is expected.
2. Memory Usage:
   1. List in Python are dynamic arrays. They allocate extra memory to accommodate future growth, which can lead to overhead.
   2. For large lists, this memory usage can be significant. If memory efficiency is crucial, consider alternatives like numpy arrays or deque from collections module.
3. Using List Comprehension:
   1. List comprehensions are often faster than equivalent loops because they are optimized at the C Level.
   2. However, they may consume more memory temporarily.
4. Avoid Excessive Resizing:
   1. Repeated resizing during appends or inserts can cause overhead. If you know the list size in advance, prealllocate or use methods to build efficiently.
5. Copying and Slicing:
   1. Copying and slicing lists can be memory and time-consuming, especially for large lists, since these operations create new objects.
6. Alternative Data Structures:
   1. For frequent insertions and deletions, consider using deque, which is optimized for such operations.
   2. If you need constant-time membership checks, use sets or dictionaries instead of lists.
7. Itertools for Efficiency:
   1. For generating combinations, permutations, or iterating, use itertools to avoid crating large lists in memory

#### List Comprehension

List comprehension in Python is a concise and elegant wat to create or transform lists. It allows you to perform operations on items in an iterable (like list, range, or string) and generate a new list in a single line of code. Here’s the breakdown:

Syntax:

[Expression for Item in Iterable if Condition]

- Expression: What you want to do with each time (e.g., transform it, modify it).

- Item: The variable representing each element from the iterable.

- Iterable: The data source you are looping over (e.g., a list, range)

- Condition (optional): A filter to decide which items are included in the result.

Example:

1. Squares = [x \*\*2 for x in range(10)]

print(Squares) # Output: [0,1,4,9,16,25,36,49,64,81]

2. Evens = [x for x in range(10) if x%2 == 0]

print(Evens) # Output: [0,2,4,6,8]

3. Words = [“hello”,”world”,”Python”]

uppercase = [word.upper() for word in Words ]

print(uppercase) # Output: [“HELLO”, “WORLD”, “PYTHON”]

4. Creating a 2D matrix (list of lists) and transposing it:

Create a 3 x 3 matrix

matrix = [[1,2,3], [4,5,6], [7,8,9]]

# Transpose Matrix

transpose = [row[i] for row in matrix for i in range(len(matrix[0]))]

5. Flattening a Nested List

nested = [[1,2,3,4], [5,6], [7,8,9]]

flattened = [item for sublist in nested for item in sublist]

print(flattened) # Output: [1,2,3,4,5,6,7,8,9]

data = {'a': 1, 'b': 2, 'c': 3}

6. Dictionary to List Conversion

# List of keys

keys = [key for key in data]

print(keys) # Output: ['a', 'b', 'c']

# List of values

values = [value for value in data.values()]

print(values) # Output: [1, 2, 3]

# List of key-value pairs

key\_value\_pairs = [(key, value) for key, value in data.items()]

print(key\_value\_pairs) # Output: [('a', 1), ('b', 2), ('c', 3)]

7. Prime Numbers with List Comprehension

n = 30

primes = [x for x in range(2, n + 1) if all(x % i != 0 for i in range(2, int(x \*\* 0.5) + 1))]

print(primes)

# Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

8. Working with Enumerations

words = ["apple", "banana", "cherry"]

# Create a list of tuples with index and word

indexed = [(index, word) for index, word in enumerate(words)]

print(indexed)

# Output: [(0, 'apple'), (1, 'banana'), (2, 'cherry')]

#### Generator

A generator is a type of iterable, like a list, but instead of holding all its elements in memory at once, it yields them one at a time, on demand. Generators are defined using functions and the yield keyword

How do Generators Work?

1. Yield keyword: When a generator function is called, it returns a generator object but doesn’t execute the function. Execution happens when the generator is iterated.

2. State Preservation: The function “remembers” where it left off after each “yield”, allowing it to resume from the last point when called again.

**Creating a Generator**

Def generate\_numbers()

For i in range(5):

yeild(i)

# Using Generator

gen = generate\_numbers()

for num in gen:

print(num)

# Output: 0 1 2 3 4

 Explanation: Each call to yield pauses the function and returns a value. The next iteration resumes from the previous point.

Example:

1. Infinite Sequence:

Def infinite\_counter():

n=0

while True:

yield n

n += 1

counter = infinite\_counter()

for i in range(5):

print (next(counter))

# Output: 0 1 2 3 4

**2. Fibonacci Sequence:**

def Fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

fib = fibonacci()

for \_ in range(5):

print (next(fib))

# Output: 0 1 1 2 3

**3. File Processing:** Read a file line by line without loading the entire file into memory:

def read\_file(filename):

with open (filename, ‘r’) as file:

for line in file:

yield line.strip()

for line in read\_file(‘example.txt’):

print(line)

### Tuple (tuple):

A tuple in Python is an immutable, ordered collection of items. Unlike lists, tuples cannot be modified after creation, making them ideal for storing data that should remain constant. Here is a breakdown of its characteristics.

1. Immutable:

Similar to lists, but immutable – values cannot be altered after creation

Useful for storing fixed data

Example: t = (1, “two”, 3/0)

1. Ordered:

Tuples maintain the order of elements, meaning you can access items using their index.

1. Defined Using Parenthesis:

A tuple is created using parentheses () or the tuple() function.

Example:

My\_tuple = (1, “Hello”, 3.14)

1. Flexible Data Types:

Tuples can contain elements of different data types: Integers, strings, floats other tuples etc.,

1. Unpacking:

You can unpack tuple elements directly into variables

a, b, c = my\_tuple

1. Memory Efficiency:

Tuples often require less memory than lists because they are immutable

**Common Features:**

All sequence types support slicing and indexing to access individual items or ranges of items.

You can iterate over sequences using loops

Example: l = [1,2,3,4]

Print(l[1:3]) # output [2,3]

#### Examples of Tuples

1. **Creating a Tuple**

# A tuple with mixed data types

mixed\_tuple = (1, "Hello", 3.14, True)

# A tuple containing another tuple

nested\_tuple = (1, 2, (3, 4))

1. **Accessing Elements**

# Accessing elements by index

print(mixed\_tuple[1]) # Output: Hello

# Negative indexing

print(mixed\_tuple[-1]) # Output: True

1. **Tuple Operations**

# Concatenation

tuple1 = (1, 2)

tuple2 = (3, 4)

result = tuple1 + tuple2 # Output: (1, 2, 3, 4)

# Repeating

repeated\_tuple = ("Hi",) \* 3 # Output: ('Hi', 'Hi', 'Hi')

1. **Unpacking Tuples:**

# Unpacking into variables

x, y, z = (5, 10, 15)

print(x, y, z) # Output: 5 10 15

1. **Tuple Methods**

# Count and index methods

sample\_tuple = (1, 2, 3, 2, 2)

print(sample\_tuple.count(2)) # Output: 3

print(sample\_tuple.index(3)) # Output: 2

1. **Tuples as Function Returns**

# Example function

def calculate(x, y):

return x + y, x \* y

sum\_result, product\_result = calculate(3, 5)

print(sum\_result, product\_result) # Output: 8 15

1. **Immutable Nature**

immutable\_tuple = (1, 2, 3)

# immutable\_tuple[0] = 10 # Uncommenting this line will raise an error

1. **Tuple for Data Integrity**

coordinates = (45.0, 92.0) # Immutable latitude and longitude values

print(f"Coordinates: {coordinates}")

## Mapping Type:

In Python, the Mapping Data type is represented by dictionaries, which are an unordered collection of key-value pairs. Here’s a deeper look at it:

### Dictionary (dict):

* A dictionary maps key to values. Each key is unique and immutable, while the values can be of any data type and are not required to be unique.
* Dictionaries are mutable, meaning you can add, remove, or update key-value pairs

#### Creation of Dictionary

Dictionaries can be created using curly braces {} or the dict() constructor:

Example:

# Using Curly Braces

my\_dict = {‘name’: ‘Alice’, ‘age’ : 23, ‘Place’: ‘Bangalore’}

# Using dict contructor

my\_dict = dict(name=”Bob”, Age=23)

#### Accessing Element

You can access values in a dictionary by their keys:

print (my\_dict[‘name’]) # output: Bob

Alternatively you can use get() method to avoid errors if a key does not exist

print(my\_dict.get(‘age’)) # Output: 23

print(mydict.get(‘name’,’age’,’place’) # Output: Alice, 23, Bangalore

#### Updating Dictionary

You can add, update or remove key-value pairs

# Adding a new key-value pair

my\_dict[‘country’] = ‘India’

# Updating an existing key

my\_dict['name’] = ‘Allen’

# Removing key-value pair

del my\_dict[‘country’]

#### Common dictionary Methods

1. keys(): Returns a view object of all keys in the dictionary

my\_dict = {"name": "Bob", "age": 30}

print(my\_dict.keys()) # Output: dict\_keys(['name', 'age'])

1. values(): Returns a view object of all values

print(my\_dict.values()) # Output: dict\_values[”Bob”, 30]

1. items(): Returns a view object of key-value pairs
2. update(): Updates the dictionary with another dictionary or iterable of key-value pair
3. clear(): Removes all elements in the dictionary
4. copy(): creates a shallow copy of dictionary

#### Key Features

* Keys are case-sensitive (‘Name’ and ‘name’ are different)
* Unordered, Dictionaries are unordered
* Dictionaries are mutable, meaning you can change, add, or remove key-value pairs after creation.
* Keys must be unique and immutable (e.g., strings, numbers or tuples)
* Values can be of any data type
* Dictionary is defined using {}

#### Common Methods

.keys(): Returns all keys in the dictionary

My\_dict.keys() -> [‘name’, ‘age’]

.values(): Returns all values in the dictionary

My\_dict.values() - > [‘Alice’, 23]

.items(): Returns key-value pairs as tuples

My\_dict.items() -> [(‘name’ ,’Alice’, ‘age’,23)]

.pop(key): Removes a key and returns its value

My\_dict.pop(‘age’) -> 23

.update(other\_dict): Updates the dictionary with another

My\_dict.update(‘Country’: ‘USA’)

#### Performance consideration while using Dictionaries:

Dictionaries in Python are highly optimized for key-value lookups but come with their own performance considerations. Here is what you need to keep in mind:

1. Time Complexity of Operations

Accessing Values: Retrieving a value by its key is O(1) on average due to the underlying hash table implementation

Adding or updating elements: These operations are also O(1) on average.

Deleting Elements: Deleting a specific key-value pair is O(1), while clearing the dictionary using clear() is O(n)

However, in the worst case (hash collisions), these operations can degrade to O(n)

1. Memory Usage

Dictionaries can consume significant memory because of the underlying hash table.

If memory efficiency is a concern, alternatives like collections.Counter (for counting) or defaultdict can sometimes offer optimized use cases.

1. Hash Collisions

Performance can degrade if there are many hash collisions, where multiple keys map to the same hash bucket.

Using keys with a good distribution in their hash function (e.g., integers or strings) reduces the risk of collisions.

1. Iterations Costs

Iterating over a dictionary using methods like items(), keys(), or values() is O(n), where n is the number of items in the dictionary.

Iteration can be memory-efficient, but copying large dictionaries can be expensive.

1. Dynamic Resizing

Python dictionaries dynamically resize themselves when the number of elements grows.

Resizing is computationally expensive, so preallocating dictionary size (if possible) can help in performance-critical applications

1. Immutability of Keys

Only hashable (immutable) data types like strings, numbers, or tuples can be used as keys. Using mutable keys, such as lists, will throw an error

1. Optimization for Membership Checks

Checking if a key exists using the “in” keyword is O(1) on average.

1. Alternatives for specialized use cases

For highly memory-intensive applications, consider using alternatives like:

* collections.defaultdict: automatically provides default values for missing keys
* collections.OrderedDict: Preserves the order of insertion (through standard dictionaries also do so from Python 3.7+)
* collections.Counter: Ideal for counting elements
* set: for scenarios focused purely on membership testing.

#### Examples of how to use Dictionaries in Python

**Example-1: Mapping Values:**

Dictionaries are commonly used for mapping unique key to values

students\_scores = {“Allen”: 100, “Alice”: 95, “Bob”: 85}

print(students\_scores[“Allen”]) #Output: 100

**Example-2: Counting Occurrences**

Dictionaries can be used to count occurrences of items in a list or a sequence

fruits = [“apple”, “banana”, “orange”, “apple”, “orange”, “apple”, “mango”]

count= {}

for fruit in fruits:

count[fruit] = count.get(fruit, 0) + 1

print (count) # Output: {“apple” : 3, “banana”: 1, “Orange”:2, “mango”: 1}

**Example-3: Updating Values**

You can update values dynamically based on some logic

inventory = {“apples”: 100, “oranges” : 80, “bananas”: 50}

# Update Stock

inventory[“apples”] += 20

inventory[“bananas”] -= 10

print (inventory) # Output: {“apples”: 120, “oranges”:80, “bananas”:40}

**Example-4: Dictionary Comprehensions**

Create dictionaries efficiently using comprehensions

squares = { x: x \*\* 2 for x in range(1,6)}

print(squares) # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

**Example-5: Fetching API Data**

Dictionaries are often used to parse and store JSON data fetched from APIs

import requests

response = requests.get(<https://jsonplaceholder.typicode.com/todos/1>)

data = response.json() # converted JSON to Python Dictionary

print (data[“Title”] # Output: prints the title from the API response

## Boolean Type:

In Python, the Boolean data type represents one of two values: Ture or False. These values are used to evaluate conditions and make decisions in your code.

Key Features of Booleans in Pythons

1. Data Type: Booleans are a fundamental data type in Python and are subclassed from integers. Internally:

True is equivalent to 1

False is equivalent to 0

1. Logical Operations: Booleans are used in logical operations like and, or, and not.

Example:

a = True

b = False

print (a and b) # Output: False

print (a or b) # Output: True

print (not a) # Output: False

1. Comparisons Result in Booleans: When you compare values in Python, the result is a Boolean (True or False). Example

x = 10

y = 20

print (x < y) # Output: True

print(x == y) # Output: False

1. Truthy and Falsy Values: Many data types can be evaluated as True, or False:

Falsy values: 0, None, empty objects ([],{},””)

Truthy values: Any non-zero number or non-empty object. Example:

print(bool(0)) # Output: False

print(bool(42)) # Output: True

print(bool([])) #Output: False

print(bool([1,2,3])) # Output: True

#### Why Booleans Matter:

Booleans play a critical role in conditional statements (if, while, etc.,), allowing programs to make decisions and execute code based on conditions.

#### Performance considerations while using Boolean in Python:

Boolean operations in Python are generally lightweight and efficient since they involve basic logical computations. However, in performance-critical scenarios or larger programs, here are some considerations to keep in mind.

1. Short-circuit Evaluation:

How it works: Python uses short-circuit evaluation for and or operators

* For and, if the first operand is False, Python does not evaluate the second operand
* For or, if the first operand is True, Python does not evaluate the second operand

Why it matters: you can leverage this behavior to avoid unnecessary computations

1. Truthy and Falsy Evaluation:

Evaluating objects in a Boolean context (e.g., if obj: ) may have overhead depending on the type of obj.

Example: Using large lists or complex objects can be slower than direct Boolean values

Optimization Tip: Use explicit comparisons (is None, len(obj > 0), etc) when the context involves complex objects

1. Avoid Overusing Boolean Chaining:

Combining multiple Boolean operations unnecessarily can make your code harder to read and slightly impact performance.

# Overcomplicated

Result = a and b and c or d

# Simplify

if a and b and c:

result = d

1. Boolean operations vs Conditional Branching:

Boolean operations: Use for simple logic

Conditionals Operations: Use if statements for complex conditions

# less efficient for complex conditions

Result = a and b and (c or d)

# more efficient

if a and b:

if c or d:

result = True

1. Avoid redundant Computations:

If and expression is reused in Boolean condition, calculate it once and store the result.

# Inefficient

if len(items) > 0 and len(items) < 10:

print (“valid range”)

# Optimized

Item\_count = len(items)

If 0 < item\_count < 10:

print(“Valid Range”)

1. Use built-in functions for clarity and speed:

Built-in Functions like any() and all() are optimized for Boolean operations on iterables

# check if any value is true

print(any([False,False,True]) # Output: True

#Check if all values are True

print(all([True, True, True])) # Output: True

In most cases, boolean operations are highly efficient in Python. However, for complex scenarios or performance-critical applications, consider optimizing your code for readability and reducing unnecessary computations.

## Set Type:

### Set

In Python, the set data type is an unordered collection of unique elements. It is useful for storing and performing operations on items without duplicates. Here is an explanation of the key features, operations and use case of sets:

#### Key Features of a Set:

1. Unordered:

Sets do not maintain any specific order of elements

my\_set = {1,3,0,2}

print(my\_set) # Output : {1,2,3,0}

1. Unique Elements:

Sets automatically removes duplicate values

my\_set = {1, 2,3,1,0}

print(my\_set) # Output: {0,1,2,3}

1. Mutable:

Sets can be modified (elements can be added or removed)

my\_set = {1,2,3}

my\_set.add(4)

print(my\_set) # Output: {1,2,3,4}

1. Non-Indexable:

You cannot access set elements using an index because they are unordered.

Creating a Set:

A set is created using curly braces {} or the set() function

my\_set = {1,2,3}

empty\_set = set() # Creates an empty set, Note: {} creates an empty dictionary

Set Operations:

1. Add Elements

my\_set = {1,2,3}

my\_set.add(4)

print (my\_set) # Output: {1,2,3,4}

1. Remove Elements

Use remove() (raises an error if the element does not exist) or discard() (not error if the element does not exist)

my\_set = {1, 2, 3}

my\_set.remove(2)

print(my\_set) #Output: {1, 3}

1. Set Union

set1 = {1, 2, 3}

set2 = {2, 3, 4}

print (set1 | set2) # Output: {1,2,3,4}

1. Set Intersection

set1 = {1, 2, 3}

set2 = {2, 3, 4}

print (set1 & set2) # Output: {2,3}

1. Set Difference

set1 = {1, 2, 3}

set2 = {2, 3, 4}

print (set1 - set2) # Output: {1}

1. Set Symmetric Difference

Elements in either set but not both

set1 = {1, 2, 3}

set2 = {2, 3, 4}

print (set1 ^ set2) # Output: {1, 4}

Use cases/ Examples of Sets

1. Removing Duplicates from a list:

my\_list = [1,2,2,3,3]

my\_set = set(my\_list)

print(my\_list) # Output: {1,2,3}

1. Membership Testing:

Sets offer fast lookups using in keyword

my\_set = {1,2,3}

print(2 in my\_set) # Output: True

print(4 in my\_set) # Output: False

1. Mathematical Operations:

Union, intersection and difference are helpful for tasks like finding shared elements between groups, that can be helpful for venn diagrams

#### Performance considerations while using sets in Python

When using sets in Python, there are several performance considerations to keep in mind. Sets are implemented as hash tables, which makes them highly efficient for certain operations but less suitable for others.

1. Fast Membership Testing:

Performance: Checking for membership using the in keyword is very fast in sets, with an average time complexity of O(1)

my\_set = {1,2,3,4}

print(3 in my\_set) # Output: True

1. Fast Add and Remove Operations

Performance: Adding or removing elements has an average time complexity of O(1). This is because sets use a hash-based mechanism to store elements

my\_set = {1,2,3}

my\_set.add(4)

my\_set.remove(2)

print(my\_set) # Output: {1,3,4}

1. Avoid Unhashable elements

Sets only work with hashable elements (e.g., numbers, strings, tuples). Attempting to use unhashable types like lists or other sets will raise a TypeError

Optimization Tip: Use immutable structure like tuple instead of list if you need to store similar data in a set.

1. Duplicate Elimination is Efficient

When creating a set from an iterable (like list), duplicates are automatically removed. This is useful for duplication tasks

Performance: Converting a list to a set has an average time complexity of O(n)

my\_list = [1,2,2,3,4]

my\_set = set(my\_list)

print (my\_set) # Output: {1,2,3,4}

1. Cost of Iteration

Iterating over a set is slower than iterating over a list because the elements in a set are unordered and stored in hash table.

Use Case: If you need to iterate frequently and order is important, a list or tuple may be more suitable

1. Set Operations:

Python provides several set operations like union, intersection, and difference, which are optimized

Union (|): Combines two sets;

Intersection(&)

Difference(-)

1. Memory Usage

Sets can be more memory-intensive than lists due to the overhead of the hash table. If memory is a constraint, consider using list or tuples if their functionality suffices

1. Hash Collisions

The performance of sets depends on the quality of the hash function. Too much hash collisions (rare but possible) can degrade performance O(n) for operations like adding, removing and membership string.

1. Immutable Sets (frozenset)

Use frozenset when you need an immutable set for operations like being a dictionary key or an element of another set. It provides the same performance characteristics as a regular set but ensures immutability.

### Frozenset

In Python, a frozenset is an immutable version of a set. Once created, its elements cannot be changed (no addition or removal). This makes it useful for scenarios where an immutable and hashable collection of unique items is needed.

#### Key Features of Frozenset

1. Immutable:

Unlike a regular set, a frozenset cannot be modified after its creation. This means you cannot add or remove elements

1. Hashable:

Since its immutable, a frozenset can be used as a key in a dictionary or as an element of another set.

1. Unordered:

Like a set, a frozenset does not maintain any specific order of elements

1. No Duplicates

A Frozenset automatically removes duplicate values, just like a regular set.

#### Creating a Frozenset

You can create a frozenset using the frozenset() constructor

my\_frozenset = frozenset([1,2,3,3])

print(my\_frozenset) # Output: frozenset({1,2,3})

#### Operations on Frozenset

Frozensets support most set operations, but any operation that modifies a set is not allowed.

1. Membership Testing

my\_frozenset = frozenset([1,2,3])

print(2 in my\_frozenset) # Output: True

1. Union (|)

frozenset1 = frozenset([1,2,3])

frozenset2 = frozenset([3,4,5])

print(frozenset1 | frozenset2) # Output: frozenset({1,2,3,4,5})

1. Intersection(&)

frozenset1 = frozenset([1,2,3])

frozenset2 = frozenset([3,4,5])

print(frozenset1 & frozenset2) # Output: frozenset({3})

1. Difference(-)

frozenset1 = frozenset([1,2,3])

frozenset2 = frozenset([3,4,5])

print(frozenset1 - frozenset2) # Output: frozenset({1,2})

1. Symmetrical Difference(^)

frozenset1 = frozenset([1,2,3])

frozenset2 = frozenset([3,4,5])

print(frozenset1 ^ frozenset2) # Output: frozenset({1,2,4,5})

#### Use Cases / Examples for Frozenset

1. As dictionary Keys

my\_dict = {frozenset([1,2,3]): “value”}

print(my\_dict) # Output: {frozenset({1,2,3}): ‘value’}

1. Membership in Sets

A frozenset can be added to a regular set because it is immutable

my\_set = {frozenset([1,2]), frozen([3,4])}

print(my\_set) # Output: {frozenset({1,2}), frozenset({3,4})}

1. Data Integrity

Frozensets are great when you want to ensure that a set of data remains unchanged.

#### Performance considerations while using frozenset:

When using frozenset in Python, there are a few performance considerations to keep in mind due to its immutability and unique characteristics.

1. Hashability:

Frozensets are hashable, meaning they can be used as keys in dictionaries or elements in other sets

Performance Benefit: Since frozensets are immutable, their hash values are fixed, making lookups in has-based data structure (like dictionaries and sets) efficient.

my\_dict = {frozenset([1,2,3]) : “value”}

print(my\_dict) # Output: {frozenset({1,2,3}): ‘value’}

1. Memory Usage:

Frozensets generally have lower memory overhead comparted to mutable sets because Python does’nt need to store data for operations like additions or deletions

Consideration: While Frozensets are lightweight, larger frozenset more memory due to their content

1. Speed of Operations:

Frozensets support set operations such as union, intersection, and difference with comparable performance of regular sets.

1. Use in Hash-Based Structures:

Using frozensets as keys in dictionaries or elements of other sets ensures data integrity and immutability, which improves performance in scenarios requiring frequent lookups or comparisons.

1. Immutability and Thread Safety:

Frozensets are immutable, making them inherently thread-safe. This is significant advantage in concurrent programming, where mutable objects might lead to race conditions.

Example: In multi-thread environments, frozensets can be shared across threads without worrying about unintended modifications.

1. Initialization from Iterable

Converting an iterable to a frozenset is an O(n) operation, where n is the size of the iterable. Ensure that iterable is not unnecessarily large to avoid memory and performance bottlenecks

My\_frozenset = frozenset(range(10000))

1. Membership Testing

Like sets, frozenset allows fast membership testing using O(1) average complexity due to their has table implementation.

1. Tradeoff with Regular Sets

Frozenset lack methods for modifying data (add, remove), making them less flexible than regular sets. Use frozensets only when immutability is essential or when the frozenset needs to be used as a dictionary key or within another set.

#### Use Cases

Frozensets shine in scenarios where immutability, hashability, and data integrity are priorities, such as:

* + **Configuration Management**: Storing fixed sets of options.
  + **Data Caching**: Using frozensets as unique keys for caching results.
  + **Concurrency**: Sharing immutable collections across threads safely.

## Binary Type:

The binary data type in Python is used to deal with binary data, which is essentially data stored in bytes. It is particularly useful when working with binary files like images, audio, or video files, as well as for tasks such as encoding or networking.

Python provides three main classes to handle binary data:

### bytes

This is an immutable sequence of bytes. Once created, it cannot be modified. Each byte is a number ranging from 0 to 255. This type is commonly used to handle binary data such as file, images or other data that requires direct byte-level manipulation. Once bytes object is created, its content cannot be changed.

#### Key Characteristics of

* 1. Immutable: You cannot modify the content of a bytes object after creation.
  2. Sequence: It behaves like a sequence, so you can access individual bytes using indexing or slicing
  3. Binary Representation: Each byte is represented in its binary form, prefixed with b (e.g., b’Hello’)

#### Creating bytes Object

You can create a bytes object in several ways:

1. From a Literal:

data = b’Hello’

print(data) # Output: b’Hello’

1. From a String using Encoding:

text = “Hello”

data = bytes(text, encoding=’utf-8’)

print(data) # Output: b’Hello’

1. From an Iterable of Integers:

numbers = [65, 66, 67]

data = bytes(numbers)

print(data) # Output: b’ABC’

1. Empty bytes Object:

empty\_data = bytes(5) # Createsa ‘bytes’ object with 5 zeroed bytes

print(empty\_data) # Output: b’\x00\x00\x00\x00\x000’

#### Common Operations:

1. Indexing and Slicing

data = b’Hello’

print(data[0]) # Output: 72 (ASCII value of ‘H’)

print(data[:2]) # Output: b’He’

1. Concatenation and Repetition

data = b’Hello’

new\_data = data + b’ World’

print(new\_data) # Output: b’Hello World’

1. Length

data = b’Hello’

print(len(data)) # Output: 5

1. Writing Binary Data to File

You can use the bytes type to write binary data into a file.

data = b’This is some binary data.’

# Open the file in binary write mode and write the data

with open (‘binary\_file.bin’, ‘wb’) as file:

file.write(data)

This creates a file named binary\_file.bin and writes the binary data to it.

1. Reading Binary Data from a File

You can read binary data from a file using the rb mode

# Open the file in binary read mode and read the data

with open(‘binary\_file.bin’, ‘rb’) as file:

data = file.read()

print(data) # Output: b’This is some binary data.’

1. Handling Binary Files (e.g., Image)

You can read and write image files in binary mode to copy them.

# Copy an image file

with open(‘source\_image.jpg’, ‘rb’) as source:

binary\_data = source.read()

with open(‘copy\_image.jpg’, ‘wb’) as copy:

copy.write(binary\_data)

1. Appending Binary Data to File

additional\_data = b’ More binary data. ‘

with open (‘binary\_file.bin’, ‘ab’) as file: # ‘ab’ is append binary mode

file.write(additional\_data)

1. Reading Specific Bytes from a File

with open(‘binary\_file.bin’, ‘rb’) as file:

file.seek(5) # Move to the 5th byte

specific\_data = file.read(10) # Read the next 10 bytes

print(specific\_data)

### bytearray

This is mutable sequence of bytes, meaning you can modify it after creation. It is similar to the bytes data type, but unlike bytes, you can modify the content of a bytearray object after it has been created. This makes bytearray especially useful when you need to work with binary data that requires editing.

#### Key Characteristics of bytearray

1. Mutable: You can modify its contents, such as replacing, inserting, or deleting bytes.
2. Sequence: Similar to bytes, it supports indexing, slicing, and iteration.
3. Efficient for Modifications: It’s ideal for cases where you need to manipulate byte data.

#### Creating bytearray Objects

You can create a bytearray object in various ways:

1. From a Bytes Literal

data = bytearray(b’Hello’)

print(data) # Output: bytearray(b’Hello’)

1. From a String (with Encoding)

text = “Hello”

data = bytearray(text, encoding=’utf-8’)

print(data) # Output: bytearray(b’Hello’)

1. From an Iterable of Integers

numbers = [65, 66, 67]

data = bytearray(numbers)

print(data) # Output: bytearray(b’ABC’)

1. Empty bytearray of a Specified Size

data = bytearray(5) # Creates a bytearray with 5 zeroed bytes

print(data) # bytearray(b'\x00\x00\x00\x00\x00')

#### Common Operations

1. Modifying Contents

data = bytearray(b'Hello')

data[0] = 72 # ASCII value of 'H'

print(data) # Output: bytearray(b'Hello')

1. Appending Bytes

data = bytearray(b'Hello')

data.append(33) # ASCII value of '!'

print(data) # Output: bytearray(b'Hello!')

1. Deleting Bytes

data = bytearray(b'Hello')

del data[0]

print(data) # Output: bytearray(b'ello')

1. Slicing and Concatenation

data = bytearray(b'Hello')

sliced = data[:3]

print(sliced) # Output: bytearray(b'Hel')

new\_data = data + bytearray(b' World')

print(new\_data) # Output: bytearray(b'Hello World')

#### Common Use Case

The `bytearray` data type has several practical applications, particularly when working with binary data. Its mutability makes it ideal for situations where you need to modify data directly. Here are some common use cases:

* **Efficient Data Manipulation**

The `bytearray` is often used for scenarios where you need to modify data in memory without creating new objects, which is faster and uses less memory.

Example: Editing chunks of binary data from a stream or file.

* **Networking**

In network programming, `bytearray` can be used as a buffer to store incoming or outgoing binary data and modify it before transmitting or processing.

Example: Handling data packets in socket programming.

* **File Handling**

When working with binary files, such as images or videos, `bytearray` is useful for reading, modifying, and writing byte-level content.

Example: Manipulating metadata within binary files.

* **Data Serialization**

`bytearray` is used in serialization protocols (e.g., pickling, protobuf, etc.) to encode or decode structured binary data efficiently.

* **Text Encoding**

You can use `bytearray` to work with text encoded in binary formats (e.g., UTF-8). It allows you to manipulate the text data directly at the byte level.

Example: Encoding strings or modifying parts of encoded text.

* **Custom Protocols**

`bytearray` is ideal for implementing custom communication protocols that require direct manipulation of binary data, such as IoT messaging or sensor data processing.

* **Memory Buffers**

In performance-critical applications, `bytearray` serves as a buffer for temporary storage and manipulation of raw binary data.

Example: Audio processing, image manipulation, or video streaming.

#### When to Use bytearray

1. **When Binary Data Needs Modifications**: Unlike bytes, you can directly change the contents of a bytearray.
2. **For Performance Optimization**: It’s efficient when working with large binary data that you need to alter in place, such as buffers or streams.

### memoryview

This provides a view of the binary data without creating a copy. It allows you to access data efficiently, especially when working with large binary files.

# Python Control Statements:

Python provides conditional statements (if, elif, else) and loops (for, while) for controlling the flow of programs.

## Conditional Statement

### if

The “if” statement in Python is a control statement used to execute a block of code based on a condition. If the condition evaluates to “True”, the indented block of code under the “if” statement is executed. If the condition evaluates to “False”, the block is skipped.

#### Basic Syntax of “if”

**if condition:**

# Block of code to execute if the condition is “True”

statements(s)

**Condition:** This is an expression that evaluates to “True” or “False”

**Indentation:** Python requires the block of code under the “if” statement to be indented.

**Example 1 : Simple if Statement**

x = 10

if x > 5:

print(“x is greater then 5”)

# Output: x is greater than 5

**Example 2: if with else**

You can use else to execute a block of code when the condition is False.

x = 3

if x > 5:

print (“x is greater than 5”)

else:

print(“x is less than or equal to 5”)

# Output: x is less than or equal to 5

**Example 3: if, elif and else**

You can use elif (short of “else if”) to check multiple conditions

x = 10

if x > 20:

print (“x is greater than 20”)

elif x > 10:

print (“x is greater than 10 but less than or equal to 20”)

else:

prtint(“x is 10 or less”)

# Output: x is 10 or less

Example 4: Nested if

You can nest if statements to check conditions within another if

x = 15

if x > 10:

if x % 2 == 0:

print (“x is greater than 10 and even”)

else:

print(“x is greater than 10 and odd”)

# Output: x is greater than 10 and odd

**Example 5: Using Logical Operators in if**

You can combine conditions using logical operators like and, or, and not

x = 7

if x > 5 and x <10:

print(“x is between 5 and 10”)

# Output: x is between 5 and 10

#### Key Points to Remember

1. Use Indentation to define the block of code under the if statement
2. Python treats conditions like 0, None, False, or empty collections (e.g., [], {} “”) as False.
3. You can have multiple elif blocks, but only one else block (optional)

### elif

The elif conditional statement in Python, short for “else if”, allows you t check multiple conditions in a sequence. It provides an elegant way to handle decision-making with more than two possible outcomes. By using elif, you avoid deeply nested if statements, making your code cleaner and more readable.

### Syntax

If condition1:

statement(s)

elif condition2:

statement(s)

else:

statement(s)

#### How it works

1. The first if condition is evaluated.
2. If the if condition is True, the code inside its block is executed, and the rest of the conditions are ignored.
3. If the if condition is False, the elif condition(s) are evaluated in order.
4. The else block is optional and executed if none of the conditions (from if or elif) are met

**Example-1: Checking Multiple Conditions**

X = 20

if x > 30:

print (“x is greater than 3-0”)

elif x > 20:

print(“x is greater than 20 but less than or equal to 30”)

elif x == 20:

print(“x is exactly 20”)

else:

print(“x is 20 or less”)

# Output: x is exactly 20

**Example-2: Checking Multiple elif Conditions**

You can use multiple elif conditions to handle complex decision-making.

score = 85

if score >= 90:

print("Grade: A")

elif score >= 80:

print("Grade: B")

elif score >= 70:

print("Grade: C")

elif score >= 60:

print("Grade: D")

else:

print("Grade: F")

# Output: Grade: B

**Example 3: Using Multiple elif blocks**

You can use multiple elif conditions to handle complex decision-making

score = 85

if score >= 90:

print("Grade: A")

elif score >= 80:

print("Grade: B")

elif score >= 70:

print("Grade: C")

elif score >= 60:

print("Grade: D")

else:

print("Grade: F")

# Output: Grade: B

#### Key Points to Remember

1. **Only One Block Executes**: As soon as a condition is True, the corresponding block runs, and the rest are skipped.
2. **Order of Conditions Matters**: Conditions are evaluated from top to bottom. If a condition is met, Python doesn’t check the ones below it.
3. else **is Optional**: The else block can be omitted if not needed.
4. **Readability**: Avoid placing too many elif blocks, as it can become harder to read. Consider refactoring using dictionaries or other structures for more complex logic.

### else

The else conditional statement in Python is used as the final fallback block in an if-elif-else structure. It allows you to execute a block of code when none of the preceding if or elif conditions are met. The else block does not have a condition—it runs unconditionally when all previous conditions fail.

#### Basic Syntax

if condition1:

# Code to execute if condition1 is True

statement(s)

elif condition2:

# Code to execute if condition2 is True

statement(s)

else:

# Code to execute if none of the conditions are True

statement(s)

**Example 1: Basic else statement**

x = 5

if x > 10:

print(“x is greater than 10”)

else:

print(“x is 10 or less”)

# Output: x is 10 or less

**Example 2: else in and if-elif Structure**

score = 50

if score >= 90:

print("Grade: A")

elif score >= 80:

print("Grade: B")

elif score >= 70:

print("Grade: C")

else:

print("Grade: F")

# Output: Grade: F

#### Key Points to Remember

1. **Unconditional Execution**: The else block is always executed if none of the preceding conditions are True.
2. **Optional**: The else block is not mandatory—it can be omitted if no fallback action is needed.
3. **Placement**: The else block must come after all if and elif blocks.

## Control Statements

### for

The “for” control statement in Python is used to iterate over a sequence (such as a list, tuple, string, dictionary, or range) and execute a block of code for each element in the sequence. It is a widely used looping construct that allows you to process collections or repeat actions a specified number of times.

#### Basic Syntax

for variable in sequence:

# Code block to execute for each element in the sequence

statement(s)

* variable: This represents the current element in the sequence during each iteration.
* sequence: The collection or iterable object (e.g., list, string, range, etc.) to iterate over.

**Example-1 Iterating over a list**

fruits = [“apple”, “orange”, “fig”]

for fruit in fruits:

print(fruit)

# Output:

apple

orange

fig

**Example-2 Iterating over a String**

Word = “Hello”

for char in Word:

print(char)

# Output:

H

e

l

l

o

**Example-3 Using Range Operator**

The range() function generates a sequence of numbers, which is often used with for loops.

for i in range(5): # Loops from 0 to 4

print(i)

# Output:

# 0

# 1

# 2

# 3

# 4

**Example-4 Accessing Indexes with enumerate()**

When iterating over a sequence, you can use the enumerate() function to access both the index and the element.

colors = ["red", "green", "blue"]

for index, color in enumerate(colors):

print(f”Index is {index} : {color}”)

# Output:

Index 0: red

Index 1: green

Index 2: blue

**Example-5 Iterating over Dictionaries**

When working with dictionaries, you can loop through keys, values, or both.

my\_dict = {"a": 1, "b": 2, "c": 3}

for key, value in my\_dict.items():

print(f"{key}: {value}")

# Output:

# a: 1

# b: 2

# c: 3

#### Key Points About for Loops in Python

* **Iterables**: The for loop works with any iterable, such as lists, tuples, strings, dictionaries, sets, or custom objects that implement the iterator protocol.
* **Else Clause**: A for loop can optionally have an else clause, which executes if the loop completes without encountering a break.

for i in range(3):

print(i)

else:

print("Loop completed!")

# Output:

# 0

# 1

# 2

# Loop completed!

* **Nested Loops**: You can nest for loops to iterate over multiple dimensions or sequences.

### While

The while control statement in Python is used to create loops that execute a block of code as long as a specified condition is true. It provides flexibility when the number of iterations isn't known in advance and depends on dynamic conditions during runtime.

#### Basic Syntax

while condition:

# Block of code to execute while the condition is True

statement(s)

**Condition**: This is an expression that evaluates to either True or False.

The loop stops when the condition becomes False.

**Example-1: Basic while loop**

count = 0

while count < 5:

print(f"Count is {count}")

count += 1

# Output:

# Count is 0

# Count is 1

# Count is 2

# Count is 3

# Count is 4

**Example-2: Using break to Exit the Loop**

You can use the break statement to exit the loop prematurely.

count = 0

while True: # Infinite loop

print(f"Count is {count}")

count += 1

if count == 5:

break # Exit the loop when count reaches 5

# Output:

# Count is 0

# Count is 1

# Count is 2

# Count is 3

# Count is 4

**Example-3: Using continue to Skip Iterations**

The continue statement skips the rest of the code in the loop and starts the next iteration.

count = 0

while count < 5:

count += 1

if count == 3:

continue # Skip printing when count is 3

print(f"Count is {count}")

# Output:

# Count is 1

# Count is 2

# Count is 4

# Count is 5

**Example-4: Loop with else**

The else block in a while loop executes if the loop terminates normally (not through break).

count = 0

while count < 3:

print(f"Count is {count}")

count += 1

else:

print("Loop finished")

# Output:

# Count is 0

# Count is 1

# Count is 2

# Loop finished

#### Key Points to Remember

1. Infinite Loops: Be cautious to avoid infinite loops where the condition never becomes False

# Infinite loop example (use with care)

while True:

print("This will run forever unless stopped!")

1. Dynamic Conditions: Use while loops when the stopping condition depends on values that change during runtime.
2. Performance: Ensure the condition is updated properly within the loop to prevent unnecessary iterations
3. The while statement is versatile and especially useful for iterative tasks where the number of iterations is not predefined.

# Functions

Functions in Python are blocks of reusable code that performs specific tasks. They help make your code modular, organized, and easier to maintain by allowing you to avoid repetition. Python supports user-defined functions, as well as built-in functions like print(), len() and others

## Syntax of a Function:

To define a function in Python, you use the def keyword followed by the function name, parentheses (which may contain parameters), and a colon. The function’s block is indented.

def function\_name(parameters):

# Block of code

return value

### Key Components:

1. Function Definition: Use of def keyword to create a function
2. Function Name: Used to identify the function
3. Parameters: These are inputs the function can take (Optional)
4. Code Block: The logic enclosed within the function
5. Return Statement: The return keyword specifies the output of the function

**Defining and Calling a function:**

def greet(name):

return (f“Welcome {name}”)

message = greet(“Allen”)

print(message) # Output: Welcome Allen

### Types of Functions:

1. Built**-in functions:** These are predefined by Python and include functions like len(), sum(), max(), etc.,

Example: print(“Hello World”)

1. **User-defined functions**: Functions written by developers for specific tasks

Example:

def square(x):

return x \* x

print(square(4)) # Output : 16

1. **Lambda Functions:** Anonymous functions created with the lambda keyword for short-term use.

square = lambda x: x \* x

print(square(4)) # Output: 16

1. **Recursive Functions:** Functions that call themselves to solve smaller instances of the same problem

def factorial(n):

if n == 1:

return 1

return n \* factorial (n-1)

print(factorial(5)) # Output: 120

### Benefits of using functions:

1. Code Reusability: Write once, use multiple times.
2. Modularity: Divide your program into smaller, manageable pieces
3. Readability: Functions make your code easier to read and understand
4. Easy Debugging: Errors can be isolated to specific functions.

### Best Practices for Writing Functions

1. **Keep Functions Small:** Each function should perform one specific task.
2. **Use Meaningful Names:** Function names should clearly indicate what the function does.
3. **Minimize Side Effects:** Avoid modifying global variables within functions.
4. **Document Your functions:** Use docstrings to explain the purpose, inputs, and outputs of the function
5. **Handle Exceptions:** Include error-handling mechanisms when dealing with external inputs.

### Writing Functions with Documentation

Python uses docstrings to document function. A docstring is a multiline string enclosed within triple quates (“””) placed directly below the function definition.

**Example-1:**

def calculate\_area(length, width):

"""

Calculate the area of a rectangle.

Parameters:

length (float): The length of the rectangle.

width (float): The width of the rectangle.

Returns:

float: The calculated area of the rectangle.

"""

return length \* width

**# Using the function**

print(calculate\_area(10, 5)) # Output: 50

Example-2:

from statistics import mean, median, mode

def calculate\_statistics(numbers):

"""

Calculate mean, median, and mode for a list of numbers.

Parameters:

numbers (list): A list of integers or floats.

Returns:

dict: A dictionary with keys 'mean', 'median', and 'mode' containing their respective values.

Raises:

ValueError: If the input list is empty.

"""

if not numbers:

raise ValueError("The list of numbers cannot be empty.")

return {

"mean": mean(numbers),

"median": median(numbers),

"mode": mode(numbers)

}

# Example usage

data = [1, 2, 3, 4, 4]

stats = calculate\_statistics(data)

print(stats)

# Output: {'mean': 2.8, 'median': 3, 'mode': 4}

This function has:

* A clear docstring to describe its purpose, parameters, return value, and exceptions.
* Error handling using raise to ensure input validation.

### Writing Functions with Test Cases

Test cases help ensure that your function behaves as expected. Python's unittest module is commonly used for this purpose.

Example:

import unittest

# Function to test

def add(a, b):

"""Add two numbers and return the result."""

return a + b

# Test cases

class TestAddFunction(unittest.TestCase):

def test\_positive\_numbers(self):

self.assertEqual(add(3, 7), 10)

def test\_negative\_numbers(self):

self.assertEqual(add(-3, -7), -10)

def test\_mixed\_numbers(self):

self.assertEqual(add(-3, 7), 4)

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

unittest.main()

Example-2:

import unittest

class TestCalculateStatistics(unittest.TestCase):

def test\_valid\_data(self):

data = [1, 2, 3, 4, 4]

expected = {"mean": 2.8, "median": 3, "mode": 4}

self.assertEqual(calculate\_statistics(data), expected)

def test\_empty\_list(self):

with self.assertRaises(ValueError):

calculate\_statistics([])

def test\_single\_value(self):

data = [10]

expected = {"mean": 10, "median": 10, "mode": 10}

self.assertEqual(calculate\_statistics(data), expected)

def test\_identical\_values(self):

data = [5, 5, 5]

expected = {"mean": 5, "median": 5, "mode": 5}

self.assertEqual(calculate\_statistics(data), expected)

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

unittest.main()

**Key Features of the Test Cases**

1. **Positive Cases**: Ensure that the function works with typical inputs.
2. **Edge Cases**: Test with unusual inputs, like an empty list or single-item lists.
3. **Error Handling**: Verify that appropriate exceptions are raised for invalid inputs.

### Key Benefits of Documentation and Test Cases

* **Documentation**: Helps other developers (and your future self) understand the function's purpose, usage, and behavior.
* **Test Cases**: Ensures reliability and correctness of the function across different scenarios.

#### try and except in Error handling

Python’s try and except blocks are used for error handling, allowing you to manage runtime exceptions gracefully without crashing the program. This feature enable you to predict and handle errors that might occur during the execution of your code.

How it works:

1. try Block:
   * The code you suspect might throw an error is placed inside the try block.
   * If an error occurs, Python skips the rest of the try block and moves to except block.
2. except Block:
   * The except block catches and handles the specific error type (or all errors if no type is specified)

#### Syntax for try and except

try:

# Code that might raise an exception

Statement(s)

except ExceptionType:

# Code to handle the exception

Statement(s)

Example: Basic Error Handing

try:

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

print(f"The square of the number is {num\*\*2}")

except ValueError:

print("Error: Please enter a valid integer!")

**Explanation:**

* The try block attempts to convert user input into an integer.
* If the input is invalid (e.g., a string instead of a number), a ValueError is raised, which is caught in the except block.

# Error Handling

In Python, error handling is primarily managed using try, except, else and finally blocks. This mechanism allows you to anticipate and handle exceptions – unexpected events or errors that occur during program execution – without crashing your program.

Here’s a breakdown

## try-except

The try block is used to write code that may raise an exception. If an exception occurs, the corresponding except block handles it.

try:

num = int(input(“Enter a number : “))

print(10/num) # Risk of ZeroDivisionError

except ZeroDivisionError:

print(“You cannot divide by Zero!”)

except ValueError:

print(“Invalid Input: Please enter a Number”)

## else

The else block is executed only if no exceptions are raised in the try block.

try:

num = int(input(“Enter a number : “))

print(10/num) # Risk of ZeroDivisionError

except ZeroDivisionError:

print(“You cannot divide by Zero!”)

except ValueError:

print(“Invalid Input: Please enter a Number”)

else:

print(“Input processed successfully”)

## finally

The finally block is used to write code that should execute no matter what – whether an exception occurs or not. This is useful for cleanup actions (e.g., closing files or releasing resources)

try:

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

content = file.read()

print(content)

except FileNotFoundError:

print("File not found.")

finally:

file.close()

print("File operation completed.")

Key Points:

* Exceptions are represented by specific error types like TypeError, ValueError, and ZeroDivisionError.
* You can raise your own exceptions using the raise statement
* Custom exception can be defined using classes that inherit from Python’s Exception class

## Examples:

**Example-1: Handling Multiple Exceptions**

def process\_input(data):

try:

number = int(data)

print(f"Square of the number is: {number\*\*2}")

except ValueError:

print("Error: Input is not a valid integer!")

except TypeError:

print("Error: Input type is not supported!")

# Testing

process\_input("5") # Valid input

process\_input("abc") # ValueError

process\_input(None) # TypeError

**Example-2: Catching all Exceptions**

Sometimes, you may not know the specific error type, so you can use a generic exception handler.

def divide(a, b):

try:

result = a / b

except Exception as e:

print(f"An error occurred: {e}")

return None

else:

return result

# Testing

print(divide(10, 2)) # Works fine

print(divide(10, 0)) # ZeroDivisionError

print(divide(10, "x")) # TypeError

**Example-3: Using Custom Message**

Add custom messages to exceptions for clarity.

try:

raise ValueError("This is a custom error message!")

except ValueError as e:

print(f"Caught an exception: {e}")

**Example-4: Using Finally for Cleanup**

When working with resources like files or databases, use finally to ensure cleanup.

def read\_file(file\_name):

try:

file = open(file\_name, "r")

content = file.read()

print(content)

except FileNotFoundError:

print(f"Error: File '{file\_name}' not found!")

finally:

print("Closing the file.")

file.close()

# Testing

read\_file("example.txt")

**Example-5: Functions with built-in error handling**

def safe\_cast(value, target\_type):

try:

return target\_type(value)

except (ValueError, TypeError) as e:

print(f"Error: {e}")

return None

# Testing

print(safe\_cast("123", int)) # Converts to 123

print(safe\_cast("abc", int)) # Fails and handles the error

## Common built-in exceptions

1. **ValueError**

Occurs when an operation or function receives an argument of the correct type but an in appropriate value.

try:

number = int(“abc”) # Invalid value for integration conversion

except ValueError as e:

print(f”ValueError: {e}”)

1. **TypeError**

Occurs when an operation or function is applied to an object of inappropriate type.

try:

result = “2” + 3 # Adding a string to an integer

except TypeError as e:

print(f“TypeError: {e}”)

1. **ZeroDivisionError**

Occurs when division or module by zero is attempted

try:

result = 100/0

except ZeroDivisionError as e:

print(f”ZeroDivisionError : {e}”)

1. **IndexError**

Occurs when attempting to access an invalid index in a sequence (e.g., list or string)

try:

my\_list = [1,2,3]

print([my\_list[5]) # Invalid Index

except IndexError as e:

print(f”IndexError: {e}”)

1. **KeyError**

Occurs when a dictionary key that doesn’t exist is accessed.

try:

my\_dict = {“name”: “Allen”}

print(my\_dict[“age”]) # Non-existent key

except KeyError as e:

print(f”KeyError: {e}”)

1. **AttributeError**

Occurs when an invalid attribute is accessed or an assignment is attempted on an object

try:

num = 5

num.append(6) # Integers don’t have an ‘append’ method

except AttributeError as e:

print(f”AttributeError: {e}”)

1. **FileNotFoundError**

Occurs when trying to access or open a file that does not exist

try:

with open(“nonexistent\_file.txt”, “r”) as file:

content = file.read()

except FileNotFoundError as e:

print(f”FileNotFoundError: {e}”)

1. **ImportError**

Occurs when an import statement fails to find the specified module or cannot load it.

try:

import nonexistent\_module # Non-existent module

except ImportError as e:

print(f“ImportError: {e}”)

1. **RuntimeError**

Occurs when an error is detected that does’nt fal into any specific category

try:

raise RunTimeError(“This is a custom runtime error.”)

except RunTimeError as e:

print(f”RunTimeError: {e}”)

1. **StopIteration**

try:

my\_iter = iter([1,2,3])

while True:

print(next(my\_iter)) # Keep calling ‘next’ until exhausted

except StopIteration as e:

print(f“StopIteration: {e}”)

1. **AssertionError**

Raised when an assert statement fails

try:

x = 10

assert x < 5, “value should be less than 5”

except AssertionError as e:

print(f”AssertionError : {e}”)

1. **NameError**

Occurs when a variable or function name that hasn’t been defined is accessed.

try:

print(undefined\_variable) # Variable is not defined

except NameError as e:

print(f”NameError: {e}”)

1. **EOFError**

Occurs when the input() function reaches the end of a file (EOF) and no more data is available.

try:

user\_input = input(“Enter something: “) # Trigger with end-of-file

except EOFError as e:

print(f”EOFError: {e}”)

# Lambda (Anonymous) Functions

These are small, anonymous functions created using the lambda keyword. They are typically used for short, single-expression operations.

Syntax: lambda arguments: expression

Lambda functions can take any number of arguments but only have a single expression.

**Example-1**

square = lambda x : x \*\* 2

print(square(5) # Output: 25

Explanation: The lambda function takes one argument “x” and returns x \*\* 2

**Example-2**

Using lambda function with python’s filter() to get even number from a list

numbers = [1,2,3,4,5,6,7,8]

even\_numbers = list(filter(lambda x : x % 2 == 0, numbers))

print(even\_numbers) # Output: [2,4,6,8]

**Example-3**

Use a lambda function with Python’s map() to convert temperature from celsius to Fahrenheit

celsius = [0,10,20,30]

fahrenheit = list(map(lambda x : (x \* 9/5) + 32, celsius))

print(fahrenheit) # Output: [32.0, 50.0, 68.0, 86.0]

**Example-4**

Use a lambda function with Python’s reduce() to compute the product of numbers in a list.

from functools import reduce

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

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 120

# Generator Functions

These are functions that return an iterator object and use “yield” instead of “return”. They are memory efficient for iterating over large datasets. When a function contains “yield”, it becomes a generator.

## Step-by-Step Explanation

1. **Defining a Generator Function**
   * A generator function looks like a normal function but contains one or more “yield” statements.

def my\_generator():

yield 1

yield 2

yield 3

1. **Using Generator**
   * To use the generator, you need to call the function. It returns a generator object rather than executing the function.

gen = my\_generator() # Creates a generator object

print(next(gen)) # Outputs: 1

print(next(gen)) # Outputs: 2

print(next(gen)) # Outputs: 3

1. **Yield Instead of Return**
   * The yield keyword pauses the function, saving its state for the next iteration. When next() is called again, the generator resumes from where it left off.
2. **Iterating Through a Generator**
   * Generators can be looped through using a “for” loop, which automatically handles calling next()

for value in my\_generator():

print(value)

## Why are Generators Memory Efficient?

* **On-the-fly computation:** Generators produce items one at a time only when needed. This means they don’t store the entire sequence in memory.
* **No Data Duplication:** Instead of loading a large dataset into memory, generators compute each time as required.

def infinite\_sequence():

n = 0

while True:

yield n

n += 1

* **Comparison with lists:** Let’s consider generating a list of numbers

# List

numbers = [x for x in range(1000000)] # Takes up a lot of memory

# Generator

numbers = ( x for x in range(1000000)) # Uses generator expression

* The list comprehension allocated memory for all 1000000 items upfront.
* The generator expression computes each value only when required, saving memory

## Visualizing the Memory Efficiency

Here’s an example to illustrate memory usage:

import sys

# List

num\_list = [x for x in range(1000000)]

print(“Memory used by list: “, sys.getsizeof(num\_list), “bytes”)

# Generator

num\_gen = (x for x in range(1000000))

print(“Memory used by generator:”, sys.getsizeof(num\_gen), “bytes”)

Output:

A list may take tens or hundreds of times more memory compared to a generator.

## Generators are excellent for scenarios like:

* Handling large datasets (E.g., reading files line by line)
* Infinite sequences
* Streaming real-time data.
* print("\n#################################################################################")
* print ("################### Example-3 (Comparison of Time efficieny Generator) #######")
* print ("""
* List Comprehension:
* Time Efficiency:
* Faster for small datasets, as the entire list is created in memory upfront.
* All values are computed and stored immediately, making subsequent access very fast.
* Generator:
* Time Efficieny:
* More time-efficient for large datasets or when you don’t need to access all items.
* Values are computed lazily (on the fly), which might take slightly longer per individual item but saves memory.
* In general:
* List comprehensions might be faster when you need to repeatedly access or modify the entire dataset.
* Generators are better for streaming or when you don’t need all items at once.
* Which one to use depends on your use case. Memory efficiency or lazy evaluation? Generators win.
* Speed with small, manageable data? List comprehensions excel. 😊
* """)
* import timeit
* # Code to be measured for list comprehension
* code\_to\_listComprehension = """
* num\_list = [x for x in range(10)]
* print(num\_list)
* """
* # Code to be measured for generator
* code\_to\_generator = """
* gen\_list = (x for x in range(10))
* for value in gen\_list:
* print(value)
* """
* # Measure execution time
* execution\_time\_listComprehension = timeit.timeit(code\_to\_listComprehension, number=1)
* execution\_time\_generator = timeit.timeit(code\_to\_generator, number=1)
* print(f"Average time taken over 1 runs for list comprehension: {execution\_time\_listComprehension} seconds")
* print(f"Average time taken over 1 run for generator: {execution\_time\_generator} seconds")
* print("#################################################################################\n")

# Recursive Functions

These are functions that call themselves to solve smaller instances of a problem. Recursive functions in Python are functions that call themselves in order to solve problems. They break down complex problems into smaller sub-problems by repeatedly calling themselves until reaching a base case (the stopping condition)

## Structure of a Recursive Function

1. Base Case: The condition that stops the recursion
2. Recursive Case: The function calls itself to solve a smaller problem

**Example: Factorial Calculation**

def factorial (n):

# Base Case: Stop recursion when n is 0

If n == 0

return 1

# Recursive Case: Multiply n by factorial of (n -1)

return n \* factorial(n-1)

print (factorial(5)) # Output: 120

**Example: Fibonacci**

def Fibonacci(n):

# Base Cases

if n == 0:

return 0

elif n == 1:

return 1

# Recursive Case

return fibonacci(n -1) + Fibonacci(n-2)

print(Fibonacci(6)) # Output: 8

**Advantages of Recursion:**

* Simplifies code for problems involving repetitive tasks or divide-and-conquer strategies
* Useful for tasks like traversing trees or solving mathematical problems

**Disadvantages:**

* Can lead to excessive memory usage and slower performance for deep recursion.
* Python imposes a recursion limit (usually 1000). You can check or modify it using.

Import sys

print(sys.getrecursionlimit())

sys.setrecursionlimit(2000) # Set a higher limit

# Callback Functions

These are functions passed as arguments to other functions and executed at a later point. Callback functions in Python are functions that are passed as arguments to other functions, to be called (or "called back") at a later time. They are often used for asynchronous programming, event handling, or customization of behavior.

## Example-1

def greet(x):

    print(f"Hello {x}, Welcome to the world of callback functions!")

def execute\_callback(callbackfunction,\*args):

    callbackfunction(\*args)

print(execute\_callback(greet,"Allen"))

## Example-2:

def greet(x):

    print(f"Hello {x}, Welcome to the world of callback functions!")

def Add(\*args):

    T = sum(\*args)

    print("Sums is : ", T)

def execute\_callback(callbackfunction,\*args):

    callbackfunction(\*args)

print(execute\_callback(greet,"Allen"))

print(execute\_callback(Add,(1,2,3)))

## Example-3:

def execute\_callback(callbackfunction,\*args):

    callbackfunction(\*args)

def findtype(x):

    print("DataType is : ", type(x))

def findlength(x):

    print("Length is ", len(x))

print(execute\_callback(findtype,"Allen"))

print(execute\_callback(findlength,"Allen"))

print(execute\_callback(findtype,[1,2,3,4]))

print(execute\_callback(findlength,[1,2,3,4]))

## Example-4:

def sort\_by\_length(word):

    return len(word)

words = ["Apple", "Banana", "Fig"]

words.sort(key=sort\_by\_length)

print(words)

# File handling

File handling in Python refers to performing operations like reading, writing, appending and closing files stored on your system. Python provides built-in functions to handle files efficiently.

## Opening a File

Python uses the open() function to open a file. It requires a file name and mode.

### File Modes:

“r”: Read mode (default)

“w”: Write Mode (Overwrites if the file exist)

“a”: Append mode (adds contents to the end of the existing file)

“b”: Binary mode (e.g., “rb”, “wb” for reading/writing binary files)

### Syntax:

file = open(“file\_name.txt”, “mode”)

Example:

file = open(“example.txt”, “w”) # Opens file in write mode

file.write(“Hello World”) # Writes to file

file.close() # Closes the file

### Various File Methods

Let’s explore **all** the file methods used in Python file handling with detailed examples.

**1. open()**

The open() function is used to open a file. It requires the file name and mode.

**Example**:

python

file = open("example.txt", "w") # Opens the file in write mode

file.write("Hello, World!") # Writes to the file

file.close() # Closes the file

**2. read()**

The read() method reads the entire content of the file.

**Example**:

python

with open("example.txt", "r") as file:

content = file.read()

print(content) # Prints: Hello, World!

**3. readline()**

The readline() method reads one line at a time.

**Example**:

python

with open("example.txt", "r") as file:

first\_line = file.readline()

print(first\_line) # Prints the first line of the file

**4. readlines()**

The readlines() method reads all lines of the file and returns them as a list.

**Example**:

python

with open("example.txt", "r") as file:

lines = file.readlines()

print(lines) # Prints: ['Hello, World!\n', 'Second line']

**5. write()**

The write() method writes a string to the file.

**Example**:

python

with open("example.txt", "w") as file:

file.write("This is a new file.")

**6. writelines()**

The writelines() method writes multiple lines to the file. It accepts a list of strings.

**Example**:

python

lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

with open("example.txt", "w") as file:

file.writelines(lines)

**7. close()**

The close() method closes a file and frees system resources.

**Example**:

python

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

file.close()

**8. seek()**

The seek() method moves the file pointer to a specified position.

**Example**:

python

with open("example.txt", "r") as file:

file.seek(5) # Move the pointer to the 6th character

print(file.read()) # Reads from the new pointer position

**9. tell()**

The tell() method returns the current position of the file pointer.

**Example**:

python

with open("example.txt", "r") as file:

file.read(10)

print(file.tell()) # Prints: 10 (current pointer position)

**10. flush()**

The flush() method clears the internal buffer, ensuring that all data is written to the disk.

**Example**:

python

with open("example.txt", "w") as file:

file.write("Temporary data.")

file.flush()

**11. truncate()**

The truncate() method removes content from the file after the specified size.

**Example**:

python

with open("example.txt", "w") as file:

file.write("Hello World!")

file.truncate(5) # Keeps only the first 5 characters

**12. mode Attribute**

The mode attribute returns the mode in which the file was opened.

**Example**:

python

with open("example.txt", "r") as file:

print(file.mode) # Output: 'r'

**13. name Attribute**

The name attribute returns the name of the file.

**Example**:

python

with open("example.txt", "r") as file:

print(file.name) # Output: 'example.txt'

**14. closed Attribute**

The closed attribute checks whether a file is closed or open.

**Example**:

python

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

print(file.closed) # Output: False

file.close()

print(file.closed) # Output: True

## Handling Binary Files

Handling binary files in Python involves working with non-text data such as images, audio files, video files, or other files stored in binary format. In Python, you can use file modes like “rb” (read binary), “wb” (write binary) or “ab” (append binary) to manage binary files.

### Reading a Binary File

Example:

with open (“example.jpg”, “rb”) as file:

binary\_data = file.read()

print (binary\_data)

### Writing Binary Data

Example:

binary\_content = b”This is some binary data!”

with open(“binary\_output.bin”, “wb”) as file:

file.write(binary\_content)

**Explanation**:

* The file is opened in "wb" mode (write binary).
* Binary content (denoted with b) is written directly to the file.

### Key Considerations When Handling Binary Files

1. **Performance**:
   * For large binary files, always read/write in chunks to optimize memory usage.
2. **Binary vs. Text Mode**:
   * Binary mode ("rb", "wb") processes raw data without text encoding or decoding.
   * Text mode processes text data with encoding, like UTF-8.
3. **Cross-Platform Compatibility**:
   * Binary files should be handled carefully when sharing across different operating systems, as file formats may vary.

# Modules

Modules in Python are files that contain Python Code, such as functions, classes, and variables, which can be reused across multiple programs. They help organize code into manageable and reusable components.

## Key Features of Python Modules:

1. **Code Reusability:** Modules allow you to reuse code instead of rewriting it.
2. **Simplified Maintenance:** By splitting code into smaller files, it becomes easier to manage and debug.
3. **Built-in and Custom Modules:** Python provides built-in modules (e.g., math, os) and allows you to create your own.

## Types of Modules:

1. **Standard Modules**: Pre-installed modules like math, random, os
2. **Third-Party Modules:** Modules that can be installed using package managers like pip (e.g., numpy, pandas)
3. **Custom Modules:** Created by users for specific community

## Using Modules

You can use the import statement to include modules in your program

Example:

# Importing the math module

import math

print(math.sqrt(16)) # Output: 4.0

## Creating Custom Modules

You can create your own module by writing Python Code in a file (e.g., my\_module.py)

Example: my\_module.py

def greet(name):

return (f”Hello {name}, Welcome to Learning Python Programming with NaveenSilvester

!”)

## Using Custom Modules

import my\_module

print(greet(“Allen”) # Output: Hello Allen, Welcome to Learning Python Programming with NaveenSilvester!

## Best Practices

1. Keep modules small and focused on a single purpose.
2. Use descriptive names for modules to reflect their functionality
3. Avoid using \* imports to prevent name conflicts
4. Organize related modules in to packages for better maintainability

# Packages

You can organize multiple modules into directories called packages. A package is a directory that contains an \_\_init\_\_.py. file. A **package** in Python is a collection of related modules grouped together in a directory. It helps organize code into a hierarchical structure, making it more manageable and modular for larger applications.

## Key Features of a Package

1. **Directory-Based**: A package is essentially a directory that contains one or more Python module files (.py).
2. \_\_init\_\_.py **File**: Every package must include an \_\_init\_\_.py file (can be empty). It marks the directory as a Python package and can include initialization code.
3. **Nested Packages**: A package can contain sub-packages, creating a hierarchical structure.

## Structure of a Package

Imagine you're building an application for data processing. You can organize your code as follows:

data\_processing/

\_\_init\_\_.py # Indicates this is a package

reader.py # Module for reading data

writer.py # Module for writing data

processor.py # Module for processing data

## Creating and Using a Package

### Step 1: Create a Directory

Create a directory for the package (e.g., data\_processing).

### Step 2: Add Modules

Inside the directory, add modules such as reader.py, writer.py, and processor.py.

### Step 3: Add the \_\_init\_\_.py File

This file can be empty, or it can include code to initialize the package or expose specific components.

**Example:** \_\_init\_\_.py

### Example: Creating a Package

#### Directory Structure

DataProcessing/

\_\_init\_\_.py

reader.py

writer.py

#### Module: reader.py

def read\_data(file\_name):

with open (file\_name, "r") as file:

content = file.read()

return (content)

#### Module: writer.py

def write\_data(file\_name):

with open (file\_name, "w") as file:

file.write("Hello here are the contents that I have entered into Output.txt file\n")

return f"Writing data to {file\_name}"

#### Script to use the package

Save this file outside the package directory

import DataProcessing.reader as reader

import DataProcessing.writer as writer

print(reader.read\_data("input.txt")) # Output: Reading data from input.txt

print(writer.write\_data("output.txt")) # Output: Writing data to output.txt

### An Example of creating Package while following Best Practices

Creating a Python package involves structuring your code in a way that it can be easily reused and shared. I will explain the steps with an example package called “mathfun”, which contains utilities to perform mathematical operations like addition, subtraction, and multiplication.

#### Step-by-Step Guide

* 1. **Set Up the Directory Structure**

Organize your package files in a specific directory structure. For our mathfun package:

mathfun/

├── mathfun/

│ ├── \_\_init\_\_.py

│ ├── operations.py

├── setup.py

├── README.md

├── LICENSE

* The outer mathfun/ is the project folder.
* The inner mathfun/ folder contains the package code.
* \_\_init\_\_.py is an empty file that marks the folder as a package.
* operations.py contains the actual functionality.
  1. **Write the Package Code**

Inside operations.py, define the package functions:

python

# mathfun/operations.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

def multiply(a, b):

return a \* b

* 1. **Create the \_\_init\_\_.py File**

The \_\_init\_\_.py file allows you to expose the module functions directly. For example:

# mathfun/\_\_init\_\_.py

from .operations import add, subtract, multiply

Now, users can call functions like mathfun.add() directly after importing the package.

* 1. **Write a setup.py File**

The setup.py file contains metadata for your package. Here's an example:

# setup.py

from setuptools import setup, find\_packages

setup (

name="mathfun", # Package name

version="1.0.0", # Version

author="Your Name",

author\_email="your.email@example.com",

description="A simple math utilities package",

long\_description=open("README.md").read(),

long\_description\_content\_type="text/markdown",

url="https://github.com/yourusername/mathfun", # GitHub or other repository

packages=find\_packages(),

classifiers=[

"Programming Language :: Python :: 3",

"License :: OSI Approved :: MIT License",

"Operating System :: OS Independent",

],

python\_requires=">=3.6",

)

* 1. **Add Supporting Files**

README.md: Describes your package and its usage.

LICENSE: Specifies the open-source license (like MIT License).

* 1. **Build and Distribute the Package**

To distribute the package:

* + 1. Install build and twine if not already installed:

pip install build twine

* + 1. Build the package:

python -m build

This generates a dist/ folder with .tar.gz and .whl files.

* + 1. Upload to PyPI (Python Package Index):

twine upload dist/\*

* 1. **Test Your Package**

Install your package locally to test it:

pip install .

Then, create a test script:

import mathfun

print(mathfun.add(5, 3)) # Output: 8

print(mathfun.subtract(5, 3)) # Output: 2

print(mathfun.multiply(5, 3)) # Output: 15

* 1. **Best Practices**
* Use **docstrings** for each function to document its purpose.
* Add **unit tests** using a testing framework like unittest or pytest.
* Follow the Python style guide (**PEP 8**) for clean and readable code.

# Special Functions

## map(function, iterable)

Applies a function to every item in an iterable and returns a map object

nums =[1,2,3]

squared = map(lambda x : x \*\*2, nums) # Output: [1,4,9]

## zip(\* iterables)

Combines multiple iterables into a single iterator of tupels.

names = [“Allen”, “Bob”]

scores = [100,90]

combine = zip(names, scores) # Output: [(“Allen”, 100), (“Bob”, 90)]

## filter(function, iterable)

Filters elements of an iterable based on condition.

nums = [1 ,2 , 3, 4]

evens = filter(lambda x : x %2 == 0, nums) #Output: [2,4]

# Decorators

Decorators in Python are a powerful and elegant way to extend or modify the behavior of functions or methods without permanently changing them. A decorator is essentially a function that takes another function as input and returns a modified or enhanced version of that function.

## How Decorators Work

Decorators are applied using the @decorator\_name syntax, placed above the function definition. They are useful for tasks like logging, authentication, modifying output, etc.

1. Simple Example: Logging

Let’s create a decorator that logs when a function is called.

def simple\_logger(func):

def wrapper():

print(f”Calling function : {func.\_\_name\_\_} “)

return func()

return wrapper

# Function to be decorated

@simple\_logger

def greet():

print(“Hello World”)

greet()

print("\n############### Example-2: Decorator Function #################################")

import time

def timer(func):

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

start\_time = time.time()

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

end\_time = time.time()

print (f"Function called {func.\_\_name\_\_} completed in {end\_time - start\_time} seconds")

return result

return wrapper

@timer

def compute\_sum(n):

return sum(range(n))

#compute\_sum(1000)

print (compute\_sum(100000))

print("##################################################################################\n")

print("\n############### Example-3: Decorator Function (Authentication Example) ##########")

def authenticate (user\_role):

def decorator(func):

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

if user\_role == "Admin":

print(f"Access granted to {user\_role}")

return func(\*args, \*\*kwargs)

else:

print (f"Access Denied for {user\_role}")

return None

return wrapper

return decorator

@authenticate(user\_role="Admin")

def view\_sensitive\_data():

print("Here is Sensitive Data")

@authenticate(user\_role="Guest")

def view\_sensitive\_guest():

print("Here is Sensitive Data")

view\_sensitive\_data()

view\_sensitive\_guest()

# Object Oriented Programming (OOP)

Object Oriented Programming (OOP) is a programming paradigm based on the concept of “objects”, which can contain data and methods to operate on that data. In Python, OOP provides a structured approach to design programs by organizing related data and functions together.

## Key Concepts of OOP in Python

### Classes and Objects

A Class is a blueprint for creating objects. It defines the structure and behavior that the object will have.

An Object is an instance of class, representing a specific entity.

Both \_\_init\_\_ and self are essential parts of defining and working with classes in Python. Here's what they mean:

**\_\_init\_\_**

The \_\_init\_\_ method is a special method in Python, also known as the **constructor**. It is automatically called when you create a new object of a class. Its main purpose is to initialize the attributes (variables) of the object.

* Think of \_\_init\_\_ as the setup or preparation method where you define and assign the initial state of the object.
* It ensures that each object starts with the necessary attributes.

**self** is a reference to the current object of the class. It is used to access the attributes and methods of the object within the class.

* It allows you to distinguish between instance attributes/methods (specific to the object) and local variables within the methods.
* Think of self as a placeholder for the object that calls the method.

**Class Attributes**

* Class attributes are shared across all instances of the class.
* They are defined **inside the class but outside of any methods**, typically at the top of the class definition.
* These attributes belong to the class itself, not to any specific object (instance).
* If a class attribute is modified, the change is reflected in all instances unless overridden by an instance attribute.

class Car:

# Class attribute

wheels = 4

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

self.brand = brand # Instance attribute

self.color = color # Instance attribute

# Accessing class attributes

print(Car.wheels) # Output: 4

# Creating instances

car1 = Car("Tesla", "Red")

car2 = Car("Maruti", "Blue")

# Both objects share the class attribute

print(car1.wheels) # Output: 4

print(car2.wheels) # Output: 4

# Changing the class attribute

Car.wheels = 6

print(car1.wheels) # Output: 6 (reflected in all instances)

**Instance Attributes**

* Instance attributes are unique to each object (instance) of the class.
* They are typically defined inside the \_\_init\_\_ method using self.
* Changes to instance attributes affect only that specific object.

class Car:

wheels = 4 # Class attribute

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

self.brand = brand # Instance attribute

self.color = color # Instance attribute

# Creating instances

car1 = Car("Tesla", "Red")

car2 = Car("Maruti", "Blue")

# Instance attributes are unique to each object

print(car1.brand) # Output: Tesla

print(car2.brand) # Output: Maruti

# Modifying an instance attribute

car1.color = "Black"

print(car1.color) # Output: Black

print(car2.color) # Output: Blue (unchanged)

#### What happens if an instance attribute shares the same name as a class attribute?

When an instance attribute shares the same name as a class attribute, **the instance attribute takes precedence** over the class attribute for that specific object. This means that when you access the attribute via the object, Python will look for the instance attribute first and use it, ignoring the class attribute.

Here’s a detailed explanation and example:

**What Happens**

1. **Instance Attribute Overrules:** The instance-specific value is used whenever you access the attribute through the object.
2. **Class Attribute Remains Intact:** The class attribute is still accessible through the class itself or through objects that don’t override it.

class Animal:

species = "Mammal" # Class attribute

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

self.name = name # Instance attribute

# Creating an object

dog = Animal("Dog")

# Accessing attributes

print(dog.species) # Output: Mammal (inherits class attribute)

# Overriding the class attribute with an instance attribute

dog.species = "Reptile"

print(dog.species) # Output: Reptile (instance attribute takes precedence)

print(Animal.species) # Output: Mammal (class attribute remains unchanged)

#### How does inheritance affect class and instance attributes?

In Python, **inheritance** allows a child class to acquire attributes and methods from a parent class. It affects both **class attributes** and **instance attributes** in specific ways:

**Class Attributes in Inheritance**

1. **Shared Across Parent and Child Classes:**
   * Class attributes defined in the parent class are automatically inherited by the child class.
   * Both the parent and child class, as well as instances of both, can access these shared attributes.
2. **Overriding Class Attributes:**
   * If the child class defines a class attribute with the same name as the parent class, it overrides the parent’s class attribute for the child class and its instances.
   * The parent class’s attribute remains unchanged.

class Parent:

species = "Mammal" # Class attribute

class Child(Parent):

species = "Reptile" # Overriding the class attribute

# Accessing class attributes

print(Parent.species) # Output: Mammal

print(Child.species) # Output: Reptile

**Instance Attributes in Inheritance**

1. **Inherited Instance Attributes:**
   * When creating an object of the child class, the \_\_init\_\_ method in the parent class can initialize instance attributes, provided the child class does not override it.
2. **Overriding Instance Attributes:**
   * The child class can define its own \_\_init\_\_ method, which can modify or redefine the instance attributes inherited from the parent class.

class Parent:

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

self.name = name # Instance attribute

class Child(Parent):

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

super().\_\_init\_\_(name) # Inherits 'name' from Parent class

self.age = age # Additional attribute for Child class

# Creating objects

parent = Parent("Alice")

child = Child("Bob", 10)

print(parent.name) # Output: Alice

print(child.name) # Output: Bob (inherited from Parent)

print(child.age) # Output: 10 (unique to Child)

**In this example:**

* The Parent class initializes the name attribute.
* The Child class inherits name but also defines its own age attribute.

Example:

class Dog:

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

self.name = name

self.breed = breed

def bark(self):

return (f”{self.name} says Woof!”)

# Creating objects

dog1 = Dog(“Buddy”, “Golden Retriever”)

print(dog1.bark) # Output: Buddy says Woof!

### Encapsulation

Encapsulation is building data (attributes) and methods (functions) within a class. It hides the implementation details and allows controlled access using methods. Encapsulation involves bundling data (attributes) and methods that operate on the data within one unit (class). It helps protect object attributes using access modifiers like private (\_attributes) and public (attributes) attributes.

class BankAccount:

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

self.balance = balance # Protected Attribute

def get\_balance(self):

return self.\_balance

account = BankAccount(1000)

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

### Inheritance

Inheritance allows a class (child class) to derive properties and methods from another class (parent class)

It facilitates code reuse and provides hierarchical structure

Example:

class Animal:

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

self.species = species

def describe(self):

print(f”I am a {self.species}.”)

class Cat(Animal): # Inherits from Animal Class

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

super().\_\_init\_\_(“Cat”)

self.name = name

def speak(self):

print(f”{self.name} says Meow!”)

fluffy = Cat(“Fluffy”)

fluffy.describe() # Output : I am a Cat

fluffy.speac() # Output: Fluffy say Meow!

**Inheritance** is one of the key concepts in Object-Oriented Programming (OOP). It allows one class (called the child class or subclass) to inherit properties (attributes) and methods from another class (called the parent class or superclass). This promotes code reuse and enables developers to build upon existing functionality.

#### Syntax and Concepts

* 1. **Defining Parent and Child Classes**

To implement inheritance, you define a parent class, and then the child class inherits from it using parentheses.

# Parent Class (Super Class)

class Parent:

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

self.name = name # Instance attribute

self.location = location # Instance attribute

def display\_info(self):

print(f"Name: {self.name}, Location: {self.location}")

# Child Class (Sub Class)

class Child(Parent):

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

super().\_\_init\_\_(name, location) # Use super() to call the Parent's \_\_init\_\_ method

self.age = age # Additional attribute for Child class

def display\_age(self):

print(f"Age: {self.age}")

In this syntax:

* **Parent class** defines the common attributes (name and location) and methods (display\_info).
* **Child class** inherits the attributes and methods of the parent using the super() function, while also adding its own (age and display\_age).
  1. **Using Inheritance**

Inheritance enables you to create objects of both the parent and child classes. Child objects have access to both parent and child attributes/methods

# Creating objects

parent = Parent("Alice", "Bangalore")

child = Child("Bob", "Chennai", 10)

# Access Parent class methods and attributes

parent.display\_info() # Output: Name: Alice, Location: Bangalore

# Access Child class methods and attributes

child.display\_info() # Output: Name: Bob, Location: Chennai

child.display\_age() # Output: Age: 10

* 1. **Overriding Parent Method in the Child Class**

Sometimes, you may want the child class to modify or replace a method from the parent class. This is method overriding

class Parent:

def greet(self):

print("Hello from the Parent class!")

class Child(Parent):

def greet(self):

print("Hello from the Child class!")

# Creating objects

parent = Parent()

child = Child()

# Call the overridden method

parent.greet() # Output: Hello from the Parent class!

child.greet() # Output: Hello from the Child class!

* 1. **Adding Additional Methods to the Child Class**

Chid classes can have their own specific methods in addition to inherited ones.

class Parent:

def greet(self):

print("Hello from the Parent class!")

class Child(Parent):

def greet(self):

print("Hello from the Child class!")

def child\_specific\_method(self):

print("This is a method unique to the Child class.")

# Creating objects

child = Child()

child.greet() # Output: Hello from the Child class!

child.child\_specific\_method() # Output: This is a method unique to the Child class.

#### Summary of Key Points

1. Inheritance allows child classes to use attributes and methods of the parent class.
2. super() helps invoke the parent class's constructor (\_\_init\_\_) or methods within the child class.
3. Child classes can override methods and add additional functionality.
4. It promotes code reuse and provides a clear hierarchy of relationships between classes.

### Polymorphism

Polymorphism allows methods in different classes to be used interchangeably through method overriding or interface implementation.

**Polymorphism** is one of the fundamental concepts in Object-Oriented Programming (OOP). It allows objects of different classes to be treated as objects of a common superclass. In Python, polymorphism refers to the ability of different objects to be used interchangeably, typically by having methods with the same name but behaving differently depending on the object's class.

#### Key Concepts and Syntax

**1. Method Overriding**

* When a child class defines a method with the same name as a method in the parent class, the child's method overrides the parent's method.
* This allows the same method name to exhibit different behaviors depending on the object.

**class Animal:**

def sound(self):

print("Animals make different sounds.")

**class Dog(Animal):**

def sound(self):

print("Dog says: Woof!")

**class Cat(Animal):**

def sound(self):

print("Cat says: Meow!")

# Demonstrating polymorphism

animals = [Dog(), Cat()]

for animal in animals:

animal.sound()

Example:

class Bird:

def speak(self):

print(“Tweet!”)

class Dog:

def speak(self):

print(“Woof!”)

def animal\_sound(animal):

animal.speak()

animal\_sound(Bird()) # Output: Tweet!

animal\_sound(Dog()) # Output: Woof!

#### Difference between Inheritance and Polymorphism

Polymorphism and inheritance are fundamental concepts in object-oriented programming, but they serve different purposes and function in distinct ways.

**Inheritance:**

**Definition:** Inheritance allows a class (child or subclass) to inherit attributes and methods from another class (Parent or Superclass). It promotes code reuse and represents an “as-is” relationship

**Purpose:** It helps create hierarchical structure and eliminates redundancy

**Example:** If you have a Vehicle class, you can create subclasses like Car, Bike that inherit common properties such as speed, or fuel\_capacity.

Polymorphism:

Definition: Polymorphism enables a single interface (e.g., method) to behave differently for different objects. It allows to be used interchangeably for object of different types.

Purpose: It promotes flexibility and dynamic behaviour, as the ame method can adapt to the object its acting upon.

Example: Different objects like Dog and Cat, both have a make\_sound() method, but they perform different actions

class Animal:

def make\_sount(self):

pass

class Dog(Animal):

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

return “Woof!”

class Cat(Animal):

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

return “Meow”

# Polymorphism in action

animals = [Dog(), Cat()]

for animal in animals:

print(animal.make\_sound()) # Output: Woof! Meow!

**Summary**

Think of inheritance as defining *what something is* (e.g., a car *is a* vehicle), while polymorphism focuses on *how something behaves* (e.g., different animals make distinct sounds). Together, these concepts work to make object-oriented programming more efficient and flexible.

### Composition

Composition is a design principle in Object Oriented Programming (OOP) where one class is made up of one or more instance of other classes. Instead of inheriting properties and behavior from a parent class, a class “has-a” relationship with other classes by including them as attributes. This promotes code reusability while avoiding some of the pitfalls of inheritance, like tightly coupled designs.

Inheritance describes an “is-a” relationship. Example: A Dog is an Animal

Composition describes a “has-a” relationship. Example: Car has an Engine

#### Why use composition?

* 1. **Flexible Relationships**: Composition allows you to dynamically define relationship between objects
  2. **Reduce complexity:** Avoids deeply nested inheritance hierarchies that can lead to difficult-to-maintain code
  3. **Promotes Modularity:** Components can be reused in other classes without rewriting code.

Example:

Let’s say you want to create a “Car” class that includes an “Engine”. Instead of making “Car” inherit from “Engine”, you can use composition.

class Engine:

def start(self):

return “Engine Started”

def stop(self):

return “Engine Stopped”

class Car”

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

self.engine = engine # Composition: Car has an Engine

def start\_car(self):

return self.engine.start() # Delegating to Engine’s Method

def stop\_car(self):

return self.engine.stop() # Delegating to Engine’s Method

# Create an Engine Object

engine = Engine()

# Pass the Engine object to Car

my\_car = car(Engine)

print(my\_car.start\_car()) # Output: Engine started

print(my\_car.stop\_car()) # Output: Engine stopped

Here, Car does not directly inherit from Engine. Instead, it contains an instance of the Engine class, making it easier to replace or modify the Engine object without altering the Car class.

### Abstraction

Abstraction simplifies complex systems by hiding implementation details while exposing functionalities.

Example:

from abc import ABC, abstractmethod

class Vechicle(ABC):

@abstractmethod

def drive(self):

pass

class Car(Vehicle):

def drive(self):

print(“Driving a Car!”)

car = car()

car.drive() # Output: Driving a Car!

## Different Class methods in Python

In Python, classes can define different types of methods to manipulate or access data, depending on their use case. These methods include **instance methods**, **class methods**, **static methods**, and **magic methods**.

### Instance Method

Instance methods are the most common methods in Python. They take self as their first parameter, which refers to the instance of the class calling the method. Instance methods can access and modify instance attributes and invoke other instance methods.

class Example:

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

self.name = name

def greet(self): # Instance method

return f"Hello, {self.name}!"

obj = Example("Alice")

print(obj.greet()) # Output: Hello, Alice!

### Class Method

Class methods take cls as their first parameter, which refers to the class itself (rather than the instance). They are defined using the @classmethod decorator. Class methods can access class-level attributes and modify them.

class Example:

count = 0 # Class attribute

@classmethod

def increment\_count(cls): # Class method

cls.count += 1

return cls.count

print(Example.increment\_count()) # Output: 1

### Static Method

Static methods do not take any implicit first parameter like self or cls. They behave like regular functions within the context of a class. Static methods are defined using the @staticmethod decorator. They cannot modify or access class or instance attributes directly.

class Example:

@staticmethod

def add(a, b): # Static method

return a + b

print(Example.add(5, 3)) # Output: 8

**Real-World Example: Utility Methods in an E-Commerce Application**

Imagine an e-commerce app where a class handles items in the inventory, and you need a method to calculate tax on an item. The tax calculation doesn't depend on the instance or class attributes, so a static method is perfect.

class Item:

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

self.name = name

self.price = price

@staticmethod

def calculate\_tax(price, tax\_rate):

return price \* tax\_rate

# Create an instance of Item

item1 = Item("Laptop", 50000)

# Call the static method

tax = Item.calculate\_tax(item1.price, 0.18)

print(f"The tax on {item1.name} is: ₹{tax}")

### Magic Methods (Dunder Methods)

Magic methods, also known as dunder methods (short for "double underscore"), are special methods in Python with predefined behavior. They are usually used to define operator overloading, object representation, and other features.

**Common Magic Methods:**

* \_\_init\_\_: Initializes instance attributes.
* \_\_str\_\_: Represents the object as a string.
* \_\_add\_\_: Implements addition operator.
* \_\_len\_\_: Returns the length of the object.

## Special Methods of OOP in Python

Special methods in Python classes are predefined methods surrounded by double underscores (\_\_). These methods allow objects to exhibit special behaviors, like built-in operations or customizing functionality. Here’s a list of commonly used special methods:

### Categories of Special Methods

#### Initialization and Representation

1. **\_\_init\_\_(self, ...):** Constructor, initializes an object when it’s created.
2. **\_\_repr\_\_(self):** Defines the string representation of the object (used by repr()).

The \_\_repr\_\_(self) method in Python is a special method that defines the “official” string representation of an object. It is meant to provide a precise and unambiguous description of the object, primarily for developers and debugging purposes. The output of repr() is often meant to be evaluable by Python's interpreter (when possible), or at least descriptive.

class Book:

    def \_\_init\_\_(self, title, author):

        self.title = title

        self.author = author

    def get\_author(self):

        return (f"Author of the book is: {self.author}")

    def \_\_repr\_\_(self):

        return f"Book(title='{self.title}', author='{self.author}')"

book = Book("MyBook Title", "Allen")

print(book)

print(book.get\_author())

print(book.\_\_repr\_\_())

1. **\_\_str\_\_(self):** Defines the string representation used by print() and str().

The \_\_str\_\_(self) method in Python is a special method used to define the **string representation** of an object. Its primary goal is to provide a human-readable and user-friendly description of the object. The \_\_str\_\_ method is called when you pass an object to str(), or when using functions like print().

**Key Features of \_\_str\_\_(self)**

1. **Human-Readable Output**:
   * The focus is on presenting the object in a way that's clear and concise for the end-user.
   * This output is typically less detailed than \_\_repr\_\_.
2. **Default Behavior**:
   * If \_\_str\_\_ is not defined, Python will fall back to using the \_\_repr\_\_ method to represent the object.
3. **Invoked When**:
   * str(obj) is called.
   * The object is passed to print() or similar functions.

#### Attribute Access

1. **\_getattr\_\_(self, name):** Called when accessing an attribute that doesn’t exist.

The \_\_getattr\_\_() method in Python is a special method used to define behavior when an **attribute that does not exist** in an object is accessed. It acts as a fallback mechanism for undefined attributes and is invoked automatically.

**Key Features of \_\_getattr\_\_()**

1. **Triggered Only for Missing Attributes**:

It is called when attempting to access an attribute that is not defined in the object. For attributes that are already defined, \_\_getattr\_\_() is **not** invoked.

1. **Dynamic Behavior**:

It allows you to create dynamic or computed attributes on the fly.

1. **Common Use Cases**:
   1. Implementing proxies or fallback attributes.
   2. Dynamically generating values or delegating functionality.

Example:

class School:

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

self.name = name

self.location = location

def \_\_getattr\_\_(self, attr):

return f"'{attr}' attribute does not exist in the School class."

myschool = School("SJBHS", "Bangalore")

# Accessing existing attributes

print(myschool.name) # Output: SJBHS

print(myschool.location) # Output: Bangalore

# Accessing a missing attribute

print(myschool.principal) # Output: 'principal' attribute does not exist in the School class.

1. **\_\_setattr\_\_(self, name, value):** Called when setting an attribute.

The \_\_setattr\_\_ method in Python is a special method used to define how attribute assignment (self.attribute\_name = value) is handled in a class. When you override this method, you can customize how attributes are set in your class. This method intercepts all attempts to set attributes and provides you control over the assignment behavior.

Syntax:

def \_\_setattr\_\_(self, name, value):

# Custom logic here

super().\_\_setattr\_\_(name, value) # Ensures attributes are properly set

Example:

class School:

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

self.name = name

self.location = location

def \_\_setattr\_\_(self, name, value):

print(f"Setting {name} to {value}")

super().\_\_setattr\_\_(name, value) # Use the default behavior to set the attribute

# Example usage:

my\_school = School("Greenwood High", "Bengaluru")

print(my\_school.name) # Output: Greenwood High

**Explanation:**

* When you create an instance of the School class and set the attributes name and location, the overridden \_\_setattr\_\_ method is called.
* It prints a message each time an attribute is set.
* super().\_\_setattr\_\_(name, value) is used to ensure that attributes are actually set on the object. Without this line, attributes would not be saved to the instance.

**Use Cases:**

1. **Logging or Tracking**: Monitor all attribute changes in your object.
2. **Validation**: Check the value of the attribute before assigning it (e.g., only accept positive numbers or strings of a specific format).
3. **\_\_delattr\_\_(self, name):** Called when deleting an attribute.

The \_\_delattr\_\_ method in Python is a special method used to define how attributes are deleted from a class instance using the del keyword. When you override this method, you can customize the behavior that happens when an attribute is deleted.

Syntax:

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

# Custom logic here

super().\_\_delattr\_\_(name) # Ensures attribute deletion

Example:

class School:

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

self.name = name

self.location = location

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

print(f"Deleting the attribute: {name}")

super().\_\_delattr\_\_(name) # Perform the actual deletion

# Example usage:

my\_school = School("Greenwood High", "Bengaluru")

del my\_school.name # This will call the overridden \_\_delattr\_\_

**Explanation:**

* 1. When you use the del keyword, e.g., del my\_school.name, the overridden \_\_delattr\_\_ method is invoked.
  2. It prints a custom message before deleting the attribute.
  3. The super().\_\_delattr\_\_(name) ensures that the attribute is actually removed from the object. Without this line, the attribute would not be deleted.

**Use Cases:**

* 1. Logging or Tracking: You can log whenever an attribute is deleted.
  2. Validation: Prevent deletion of certain attributes or provide warnings if essential attributes are being removed.

#### Container Methods

1. \_\_getitem\_\_(self, key): Access items using square brackets (obj[key]).

The \_\_getitem\_\_(self, key) method in Python is used to retrieve an item from an object using the subscript notation (e.g., obj[key]). It is a special or magic method that enables custom behavior for indexing objects. You can define this method in your class to make objects of that class behave like dictionaries, lists, or other built-in sequence types.

class MyList:

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

self.items = items

def \_\_getitem\_\_(self, index):

return self.items[index]

# Create an instance of the MyList class

my\_list = My\_List([10, 20, 30, 40])

# Access items using subscript notation

print(my\_list[0]) # Output: 10

print(my\_list[2]) # Output: 30

In this example:

* 1. The MyList class takes a list as input and stores it in the self.items attributes
  2. The \_\_getitem\_\_ method is implemented to return an item from self.items based on the index provided
  3. Using subscript notation (my\_list[0] or my\_list[2]) calls the \_\_getitem\_\_ method internally

This method is particularly useful for custom container objects when you want to enable flexible indexing or slicing.

Example: Real world use case for \_\_getitem\_\_

class URLMapping:

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

self.urls = {“home” : “example.com/home”, “about”: “example.com/about”}

def \_getitem\_\_(self, key):

return self.urls.get(key, “404 Not found”)

urls = URLMapping()

print(urls[“home”]) # Output: example.com/home

1. \_\_setitem\_\_(self, key, value): Set items using square brackets (obj[key] = value).
2. \_\_delitem\_\_(self, key): Delete items using square brackets.
3. \_\_len\_\_(self): Return the length of an object (used by len()).
4. \_\_iter\_\_(self): Make an object iterable.
5. \_\_next\_\_(self): Define iteration behavior.

#### Arithmetic Operators

1. \_\_add\_\_(self, other): Implements addition (+).
2. \_\_sub\_\_(self, other): Implements subtraction (-).
3. \_\_mul\_\_(self, other): Implements multiplication (\*).
4. \_\_truediv\_\_(self, other): Implements division (/).

#### Comparison Operators

1. \_\_eq\_\_(self, other): Implements equality (==).
2. \_\_ne\_\_(self, other): Implements inequality (!=).
3. \_\_lt\_\_(self, other): Implements less-than (<).
4. \_\_gt\_\_(self, other): Implements greater-than (>).
5. \_\_le\_\_(self, other): Implements less-than-or-equal (<=).
6. \_\_ge\_\_(self, other): Implements greater-than-or-equal (>=).

#### Miscellaneous

1. \_\_call\_\_(self, ...): Makes an object callable like a function (obj()).
2. \_\_del\_\_(self): Destructor, called when an object is deleted.
3. \_\_contains\_\_(self, item): Checks membership (item in obj).
4. \_\_hash\_\_(self): Defines a hash value for the object (used in hash tables).
5. \_\_bool\_\_(self): Returns boolean value of the object (bool(obj)).

### Why Use Special Methods?

These methods allow you to:

1. Customize built-in operations for your objects (e.g., add support for arithmetic or string representation).
2. Make your objects behave like native types (e.g., dictionaries, lists, or functions).
3. Create more readable and maintainable code.

## Class Attributes in Python

In Python, attributes of classes can have different levels of accessibility: Public, Protected, and Private. These define how you can access and modify an attribute, and this is crucial for implementing encapsulation in object-oriented programming.

### Public Attributes

Public attributes are accessible from anywhere – in the class, subclasses, or even outside the class. By default, all attributes in Python are public unless explicitly stated otherwise.

Example:

class PublicExample:

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

self.public\_attribute = “I am Public”

example = PublicExample()

print(example.public\_attribute) # Accessible outside the class

### Protected Attributes

Protected attributes are meant to be accessed within the class and its subclasses. Python uses a single underscore \_ prefix to denote protected attributes. However, its not truly private – it’s more of a convention to indicated that the attribute should not be accessed directly.

**What are Protected Attributes?**

Protected attributes are declared with a single underscore (\_) prefix before the attribute name. For example: \_attribute\_name. This convention signals that the attribute should not be accessed directly outside the class or its subclasses. However, technically, they can still be accessed outside if needed.

**Characteristics of Protected Attributes**

**Access Visibility:**

* Accessible within the class where they are declared.
* Accessible in subclasses of the class.
* Technically accessible outside the class, but it’s discouraged as per convention.

**Soft Restriction:**

* The single underscore is a convention, not enforced by Python. It's more of a "gentle reminder" to developers.

class ProtectedExample:

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

self.\_protected\_attribute = "I am Protected"

def access\_protected(self):

return self.\_protected\_attribute # Accessible within the class

class Subclass(ProtectedExample):

def modify\_protected(self):

self.\_protected\_attribute = "Modified in Subclass"

example = ProtectedExample()

subclass = Subclass()

print(example.\_protected\_attribute) # Technically accessible, but discouraged

print(subclass.\_modify\_protected()) # Modified subclass or child class

**Why Use Protected Attributes?**

* 1. **Encapsulation:** Ensures the internal representation of an object is hidden from the outside.
  2. **Code Clarity:** Signals to other developers that certain attributes shouldn’t be accessed directly.
  3. **Prevents Unintended Changes:** Reduces the risk of accidental modification of critical attributes from outside the class

**Summary**

Protected attributes (\_attributes) are a Python convention to limit direct access from outside a class. They strike a balance between flexibility and encapsulation, making them an essential tool in object-oriented programming.

### Private Attributes

Private attributes are accessible only within the class where they are defined. Python uses a double underscore \_\_ prefix to denote private attributes. This triggers **name mangling**, which means the attribute name is internally changed to include the class name as a prefix.

Example:

class PrivateExample:

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

self.\_\_private\_attribute = "I am Private"

def access\_private(self):

return self.\_\_private\_attribute # Accessible within the class

example = PrivateExample()

# print(example.\_\_private\_attribute) # This will raise an AttributeError

print(example.\_PrivateExample\_\_private\_attribute) # Access using name mangling

In this case, \_\_private\_attribute cannot be accessed directly outside the class, To access it, you need to use the name-mangled version (\_ClassName\_\_atttibuteName).

To master Object-Oriented Programming (OOP) in Python and transition from a beginner to an expert, one needs to explore topics systematically, starting with basic concepts and gradually moving to advanced techniques. Here's a comprehensive roadmap:

---

### \*\*Beginner Level\*\*

1. \*\*Understanding Classes and Objects\*\*

- What are classes and objects?

- Creating and using classes.

- Attributes and methods.

2. \*\*Basic Principles of OOP\*\*

- Encapsulation: Why and how to bundle data with methods.

- Inheritance: Reusing and extending functionalities of classes.

- Polymorphism: Methods behaving differently based on object type.

3. \*\*Special Methods (`\_\_init\_\_`, `\_\_str\_\_`, etc.)\*\*

- Constructor (`\_\_init\_\_`) and string representation (`\_\_str\_\_` or `\_\_repr\_\_`).

4. \*\*Access Modifiers\*\*

- Private, protected, and public attributes.

---

### \*\*Intermediate Level\*\*

1. \*\*Inheritance and Composition\*\*

- Multiple inheritance.

- Composition vs inheritance: When to use each.

2. \*\*Decorators in OOP\*\*

- Class methods (`@classmethod`).

- Static methods (`@staticmethod`).

- Property decorators (`@property`, `@setter`).

3. \*\*Magic/Dunder Methods\*\*

- Operator overloading (`\_\_add\_\_`, `\_\_sub\_\_`, etc.).

- Object comparison (`\_\_eq\_\_`, `\_\_lt\_\_`, etc.).

- Customizing object behavior (`\_\_len\_\_`, `\_\_getitem\_\_`).

4. \*\*Understanding `self` and `cls`\*\*

- The role of `self` in instance methods.

- The role of `cls` in class methods.

5. \*\*Error Handling in OOP\*\*

- Raising exceptions in methods.

- Custom exception classes.

---

### \*\*Advanced Level\*\*

1. \*\*Design Patterns\*\*

- Singleton, Factory, Observer, and other patterns.

- Application of design patterns in Python.

2. \*\*Metaclasses\*\*

- What are metaclasses and why use them?

- Dynamic class creation with metaclasses.

3. \*\*Abstract Base Classes (ABC)\*\*

- Enforcing method implementation using abstract classes.

- The `abc` module in Python.

4. \*\*Inheritance vs Composition Best Practices\*\*

- Favoring composition over inheritance in complex projects.

5. \*\*Mixin Classes\*\*

- Extending functionality with mixins.

6. \*\*Serialization and Deserialization\*\*

- Using `pickle`, JSON, or other formats to serialize objects.

7. \*\*Understanding Python's Object Model\*\*

- The `type()` function and class creation.

- The `\_\_dict\_\_` attribute for objects and classes.

8. \*\*Testing OOP Code\*\*

- Unit testing OOP code.

- Mocking objects during tests.

---

### \*\*Real-World Applications\*\*

1. \*\*Building Real-World Projects\*\*

- Create projects like a library management system, game engines, or CRUD applications using OOP principles.

2. \*\*Understanding Frameworks\*\*

- Explore how frameworks like Django and Flask use OOP concepts.

3. \*\*Performance Optimization\*\*

- Profiling object-heavy applications.

- Memory management for objects.

---

### \*\*Expert Level\*\*

1. \*\*Dynamic Object Behavior\*\*

- Customizing attributes and methods dynamically.

- The `\_\_getattr\_\_`, `\_\_setattr\_\_`, and `\_\_delattr\_\_` methods.

2. \*\*Deep Dive into Object Internals\*\*

- Python’s garbage collection and reference counting.

- Slots (`\_\_slots\_\_`) for memory optimization.

3. \*\*Concurrency in OOP\*\*

- Thread-safe classes.

- Using `multiprocessing` or `asyncio` in object-oriented code.

4. \*\*Advanced Design Patterns\*\*

- Dependency Injection.

- Proxy and Adapter patterns.

5. \*\*Contributing to Open Source\*\*

- Contribute to large Python projects to understand advanced usage of OOP principles.

---

By following this roadmap, you'll not only gain a solid understanding of OOP concepts but also learn how to apply them effectively in real-world scenarios. Let me know if you'd like explanations or examples for any of these topics! 😊

Classes

Inheritance

Class instances

Methods

Multiple Instance objects

Customized by inheritance

Intercept python operators

Operator Overloading

Classes vs dictionaries

Constructors

Operator Overloading

Special Methods