# 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

Binary Type:

Bytes

Bytearray

Memoryview