# 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 builtin 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:
   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

### 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):

* 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”]

### Tuple (tuple):

* Similar to lists, but immutable – values cannot be altered after creation
* Useful for storing fixed data
* Example: t = (1, “two”, 3/0)

**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]

## 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’]

### Key Features

* Keys must be unique and immutable (e.g., strings, numbers or tuples)
* Values can be of any data type
* Keys are case-sensitive (‘Name’ and ‘name’ are different)

### 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’)

Boolean Type:

Bool

Set Type:

Set

Frozenset

Binary Type:

Bytes

Bytearray

Memoryview