Introduction to Python

- **Overview of Python:**
  - Python is a high-level, interpreted programming language known for its simplicity and readability.
  - It was developed by Guido van Rossum and first released in 1991.
  - Python emphasizes code readability with its notable use of significant whitespace.
- **Why Learn Python?**
  - **Simplicity:** Easy to read and write.
  - **Versatility:** Used in web development, data analysis, AI, scientific computing, and more.
  - **Large Community:** Extensive libraries and frameworks available.
  - **Career Opportunities:** High demand in various industries.
- **Real-world Applications:**
  - Web Development (e.g., Django, Flask)
  - Data Science and Machine Learning (e.g., Pandas, scikit-learn)
  - Automation and Scripting
  - Game Development (e.g., Pygame)
  - Embedded Systems

Installing Python

- **Step-by-Step Installation:**
  - **Windows:**
    1. Download the installer from the official Python website.
    2. Run the installer and check the box to add Python to your PATH.
    3. Click “Install Now” and follow the prompts.
  - **macOS:**
    1. Download the installer from the Python website.
    2. Open the `.pkg` file and follow the instructions.
    3. Verify installation by opening the terminal and typing `python3 --version`.
  - **Linux:**
    1. Open your terminal.
    2. Update your package list: `sudo apt update`.

3. Install Python 3: `sudo apt install python3`.

- **Verifying Installation:**
  - Open a terminal or command prompt.
  - Type `python --version` or `python3 --version` to check the installed version.

Install Pycharm

PyCharm is a popular Integrated Development Environment (IDE) for Python development. Here's a step-by-step guide to installing PyCharm on your computer:

Step 1: Download PyCharm

1. Go to the official PyCharm website: JetBrains PyCharm
2. You will see two versions: Professional and Community. The Community edition is free and open-source, while the Professional edition offers more features but requires a license. Choose the version that suits your needs and click the "Download" button.

Step 2: Install PyCharm

For Windows:

1. Once the download is complete, open the installer (`pycharm-community-*.exe` for the Community edition).
2. Follow the installation wizard:
   - Click "Next" to continue.
   - Choose the installation location and click "Next."
   - Select the installation options you prefer, such as creating a desktop shortcut or associating `.py` files with PyCharm.
   - Click "Install" to begin the installation process.
3. After the installation is complete, click "Finish" to exit the installer. You can choose to run PyCharm immediately if you wish.

Writing and Running Your First Python Program

Hello, World! Program

```
print("Hello, World!")
```

Running the Program:

- Save the code in a file named `hello.py`.
- Open a terminal or command prompt and navigate to the directory containing `hello.py`.
- Run the script by typing `python hello.py` or `python3 hello.py`.

Using the Python Interactive Shell:

- Open a terminal or command prompt.
- Type `python` or `python3` to enter the interactive shell.
- Type the code directly:

```
print("Hello, World!")
```

Understanding How Python Code Works

To understand how Python code works, we'll look at a simple example and explain each step involved in its execution.

Example: Greeting Program

```
# greeting.py

# Step 1: Get the user's name
name = input("Enter your name: ")

# Step 2: Print a personalized greeting
print("Hello, " + name + "!")
```

Steps Involved:

1. **Reading the Source Code:**
   - The Python interpreter reads the source code from the file `greeting.py`.
2. **Bytecode Compilation:**
   - The source code is translated into bytecode by the interpreter.
   - Bytecode is a set of instructions that can be executed by the Python Virtual Machine (PVM).
3. **Execution by PVM:**
   - The PVM executes the bytecode instructions line-by-line.

Understanding Code Execution & Introduce with debugging

- Debugging goes beyond finding bugs; it's crucial from development to production and understanding code.
- It allows you to see what's happening at each line, making it easier to understand complex logic step-by-step.
- Small mistakes causing many errors can be quickly identified and fixed through debugging.
- Debugging helps break down and test large functions incrementally, avoiding the need to write and test all at once.
- It's useful for understanding other people's code, especially in varied coding styles and unfamiliar projects.
- Debugging improves testing, performance, and code quality across multiple languages, not just Python, including JavaScript, Java, and C#

```
# Calculate the area of a rectangle
length = 5  # Length of the rectangle
width = 3  # Width of the rectangle
area = length * width  # Area formula: length * width
print("Area:", area)
```

Python Comments

Single-line Comments: Use the `#` symbol.

```
# This is a single-line comment
```

Multi-line Comments: Enclose comments in triple quotes.

```
"""
This is a multi-line comment
that spans multiple lines.
"""
```

Best Practices:

- Write clear and concise comments.

- Use comments to explain the purpose of the code, not obvious details.

```
# Calculate the area of a rectangle
length = 5  # Length of the rectangle
width = 3  # Width of the rectangle
area = length * width  # Area formula: length * width
print("Area:", area)
```

Python Variables

Definition:

- Variables store data values.
- Python is dynamically typed, so you don't need to declare a variable type explicitly.

Assigning Values:

```
x = 5
name = "Alice"
is_student = True
```

Naming Conventions:

- **Descriptive Names**: Use meaningful and descriptive names to make your code self-explanatory. For example, use `total_cost` instead of `tc`.
- **Lowercase with Underscores**: Variable names should be written in lowercase letters and words should be separated by underscores for readability. For example, `student_name` instead of `studentName`.
- **Avoid Reserved Words**: Do not use Python reserved keywords as variable names, such as `class`, `for`, `if`, etc.
- **Start with a Letter or Underscore**: Variable names must start with a letter (a-z, A-Z) or an underscore (_). They cannot start with a number.
- **No Special Characters**: Variable names should only contain letters, numbers, and underscores. Avoid using special characters like `!`, `@`, `#`, etc.
- **Case Sensitivity**: Remember that variable names are case-sensitive. For example, `myVariable` and `myvariable` are two different variables.
- **Short but Meaningful**: While being descriptive, try to keep variable names reasonably short. For example, `num_students` is better than `number_of_students_in_the_class`.

- **Use Singular Nouns**: Use singular nouns for variables that hold a single value, and plural nouns for variables that hold collections. For example, `student` for a single student, and `students` for a list of students.
- **Consistency**: Be consistent with your naming conventions throughout your code to maintain readability and ease of understanding.
- **Avoid Double Underscores**: Do not use double underscores at the beginning and end of variable names, as these are reserved for special use in Python (e.g., `__init__`, `__main__`).

Basic Operations:

```
a = 10
b = 20
sum = a + b
print(sum)  # Output: 30
```

Data Types in Python

Numeric Types

- **int**: Integer numbers, e.g., `5`, `-3`, `42`.
- **float**: Floating-point numbers, e.g., `3.14`, `-0.001`, `2.0`.
- **complex**: Complex numbers with real and imaginary parts, e.g., `1 + 2j`, `3 - 4j`.

```
x = 5       # int
y = 3.14    # float
z = 1 + 2j  # complex
```

Numeric Types Practical Use Case

- `int`: Whole numbers without decimal points. Used for counting and indexing.
- `float`: Numbers with decimal points. Used for precise calculations and measurements.
- `complex`: Numbers with real and imaginary parts. Used for advanced mathematical computations.

String Type

- **str**: A sequence of characters, e.g., `"hello"`, `'world'`.

```
greeting = "Hello, world!"
```

String Types Practical Use Case

- **Collect and Store Feedback**: Gather customer feedback and store it in a list of strings.
- **Extract Useful Information**: Identify key phrases or sentiments to understand customer opinions.
- **Format Responses**: Prepare feedback data for reporting or display, enhancing readability.

Sequence Types

- **list**: Ordered, mutable collection of items, e.g., `[1, 2, 3]`, `['a', 'b', 'c']`.

```
fruits = ['apple', 'banana', 'cherry']
# It may have diff types of data
fruits = [1, 3.4, True ,'cherry']

# May have duplicate data
fruits = ['apple', 'apple', 'apple']

# List has index
print(fruits[0])
```

- **tuple**: Ordered, immutable collection of items, e.g., `(1, 2, 3)`, `('a', 'b', 'c')`.

```
coordinates = (10, 20,40)
# It may have diff types of data
coordinates = (10, "20",4.0)

# May have duplicate data
coordinates = (10, 10,10)

# has index
print(coordinates[0])
```

- **range**: Represents an immutable sequence of numbers, commonly used in loops, e.g., `range(5)`, `range(1, 10, 2)`.

```
numbers = range(1, 10)

# Using Loop
numbers = range(1, 10)
for number in numbers:
    print(number)

# Converting List
print(list(numbers))

# Use Star
print(*numbers)

# Means Default Start from 0
numbers = range(10)


# Means Range After 2 Step
numbers = range(1, 10, 2)
```

String Types Practical Use Case

- **List**: Used for storing a collection of items that can be modified. Ideal for tasks where you need to add, remove, or change items frequently.
- **Tuple**: Used for storing a collection of items that should not be changed. Perfect for read-only data or fixed collections of items, like coordinates or configuration settings.
- **Range**: Used for generating a sequence of numbers. Commonly used in loops for iterating a specific number of times or creating sequences of numbers efficiently.

Mapping Type

- **dict**: Unordered, mutable collection of key-value pairs, e.g., `{'name': 'Alice', 'age': 25}`

```
person = {'name': 'Alice', 'age': 25}
print(person['name'])
```

```
print(person)
```

Mapping Type Practical Use Case

- **Storing Employee Data**: Use dictionaries to store employee information with unique IDs as keys and details (name, position, salary) as values.
- **Accessing Employee Data**: Retrieve specific employee details quickly using their unique ID as the key.
- **Updating Employee Records**: Easily update or modify employee information in the dictionary by accessing the relevant key.

Set Types

- **set**: Unordered, mutable collection of unique items, e.g., `{1, 2, 3}`, `{'a', 'b', 'c'}`.

```
# Must have unique data
unique_numbers = {1, 2, 3}

# Duplicate data avoided
unique_numbers = {1, 2,2, 3,3,3}
```

- **frozenset**: Unordered, immutable collection of unique items, e.g., `frozenset([1, 2, 3])`.

```
# Must have unique data
immutable_set = frozenset([1, 2, 3])

# Duplicate data avoided
immutable_set = frozenset([1,2, 2, 3])
```

Set Types Practical Use Case

- **Set**: Used for storing a collection of unique items. Ideal for tasks that require eliminating duplicates or performing mathematical set operations like unions, intersections, and differences.
- **Frozenset**: An immutable version of a set. Suitable for scenarios where a set of unique items needs to be hashable, such as using sets as dictionary keys or elements of another

set.

Boolean Type

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

```
is_active = True
```

Boolean Type Practical Use Case

- **Authentication Status**: Use a boolean variable to track if a user is logged in (`True`) or not (`False`).
- **Conditional Statements**: Use booleans in `if` statements to execute different code blocks based on conditions, such as granting access to certain features only if the user is authenticated.
- **Validation Checks**: Use booleans to validate user inputs or data integrity, such as checking if an input meets specific criteria (`True`) or not (`False`).

None Type

- **NoneType**: Represents the absence of a value or a null value.

```
result = None
```

None Type Practical Use Case

- **Function with No Return Value**: Use `None` to indicate that a function does not return a value. This is useful for functions that perform actions rather than calculations.
- **Default Parameter Values**: Use `None` as a default parameter value to signify that no argument was passed, allowing for flexible function definitions and behavior.
- **Placeholder for Optional Data**: Use `None` as a placeholder for optional or missing data, making it clear when a variable is intentionally left unset or waiting for a value.

Checking Data Types

```python
x = 10
print(type(x))  # Output: <class 'int'>

y = 3.14
print(type(y))  # Output: <class 'float'>

message = "Hello"
print(type(message))  # Output: <class 'str'>

is_valid = True
print(type(is_valid))  # Output: <class 'bool'>
```

Checking Data Types Use Case

- **Input Validation**: Ensure that user inputs are of the expected type before processing them.
- **Function Arguments**: Validate function arguments to prevent type errors and ensure correct operation.
- **Data Processing**: Confirm data types during processing to apply appropriate operations and avoid errors.
- **Configuration Loading**: Verify the types of configuration settings loaded from files or environment variables.
- **Dynamic Data Handling**: Handle data that can come in various types (e.g., JSON parsing) by checking types before processing.

Mutable vs. Immutable Data Types:

- **Mutable:** Can be changed after creation (e.g., lists, dictionaries).
- **Immutable:** Cannot be changed after creation (e.g., strings, tuples).

Immutable Data Types

Immutable objects cannot be modified after their creation. Any operation that seems to modify an immutable object will actually create a new object. Immutable types include.

Integers (`int`): Whole numbers, positive or negative.

```
a = 5
initial_id = id(a)
a = 10  # Creates a new integer object with value 10
new_id=id(a)
```

Floating-point numbers (`float`): Numbers with a decimal point.

```
b = 3.14
initial_id = id(b)
b = 2.71  # Creates a new float object with value 2.71
new_id=id(b)
```

Strings (`str`): Sequences of characters.

```
s = "hello"
initial_id = id(s)
s = "world"  # Creates a new string object with value "world"
new_id=id(s)
```

Tuples (`tuple`): Ordered collections of items.

```
t = (1, 2, 3)
initial_id = id(t)
t = (4, 5, 6)  # Creates a new tuple object with different values
new_id=id(t)
```

Frozen Sets (`frozenset`): Immutable sets.

```
fs = frozenset([1, 2, 3])
initial_id = id(fs)
fs = frozenset([4, 5, 6])  # Creates a new frozenset object with different
values
new_id=id(fs)
```

Immutable Practical Use Cases

- **Configuration Settings**: Store application settings in tuples to ensure they are not accidentally modified.

- **User Roles**: Define fixed user roles (e.g., admin, editor, viewer) using tuples for security and integrity.
- **API Endpoints**: Use tuples to store API endpoints, ensuring the URLs remain constant.
- **Coordinates**: Store geographical coordinates as tuples to maintain their integrity throughout the application.
- **Cache Keys**: Use frozensets for cache keys to ensure that key combinations remain consistent and hashable.

Mutable Data Types

Mutable objects can be modified after their creation. Operations that modify mutable objects do not create new objects but rather change the existing object. Mutable types include:

Lists (`list`): Ordered collections of items.

```
l = [1, 2, 3]
initial_id = id(l)
l[0] = 4  # Modifies the existing list object
new_id = id(l)
```

Dictionaries (`dict`): Collections of key-value pairs.

```
d = {'a': 1, 'b': 2}
initial_id = id(d)
d['a'] = 3  # Modifies the existing dictionary object
new_id = id(d)
```

Sets (`set`): Unordered collections of unique items.

```
s = {1, 2, 3}
initial_id = id(s)
s.add(4) # Modifies the existing set object
new_id = id(s)
```

Mutable Practical Use Cases

- **User Sessions**: Use dictionaries to store session data, allowing dynamic updates of user-specific information.

- **Shopping Cart**: Implement shopping carts using lists to add, remove, or modify items based on user actions.
- **Form Data**: Collect and modify form inputs using dictionaries, making it easy to validate and process user submissions.
- **Real-time Notifications**: Maintain a list of notifications for users, allowing additions and deletions as new events occur.
- **Dynamic UI Elements**: Use lists or dictionaries to manage dynamic elements like user-generated content or interactive components that change based on user interaction.

Type Conversion

Explicit Type Conversion: The programmer manually converts a data type using functions like int(), float(), or str().

```
x = "123"
y = int(x)  # Convert string to integer
z = float(x)  # Convert string to float
a = str(456)  # Convert integer to string

print(y)  # Output: 123
print(z)  # Output: 123.0
print(a)  # Output: "456"
```

Implicit Type Conversion: Python automatically converts one data type to another during operations without explicit instruction from the programmer.

```
x = 10
y = 3.14
z = x + y  # x is converted to float
print(z)  # Output: 13.14
```

Handling Conversion Errors

```
try:
    x = "abc"
    y = int(x)
```

```
except Exception as e:
    print(f"An error occurred: {e}")
```

Type Conversion Use Case

- **User Input Handling**: Convert string inputs from forms into integers or floats for calculations.
- **Data Processing**: Convert data types when reading from or writing to files to ensure correct data formats.
- **Mathematical Operations**: Convert data to appropriate numeric types for accurate mathematical operations.
- **JSON Parsing**: Convert data types when parsing JSON to ensure correct types for further processing.
- **Database Interaction**: Convert data types to match database schema requirements when inserting or retrieving data.

Example: Simple Calculator

```
# Simple Addition
num1 = input("Enter first number: ")
num2 = input("Enter second number: ")

# Convert input strings to integers
num1 = int(num1)
num2 = int(num2)

# Calculate the sum
sum = num1 + num2

# Print the result
print("The sum is:", sum)
```

Example: Greeting Program

```
# Greeting Program
name = input("Enter your name: ")
```

```
# Print a personalized greeting
print("Hello, " + name + "!")
```

Example: Temperature Converter (Celsius to Fahrenheit)

```
# Temperature Converter (Celsius to Fahrenheit)
celsius = input("Enter temperature in Celsius: ")

# Convert input string to float
celsius = float(celsius)

# Calculate Fahrenheit
fahrenheit = (celsius * 9/5) + 32

# Print the result
print("Temperature in Fahrenheit:", fahrenheit)
```

Example: Even or Odd Checker

```
# Even or Odd Checker
num = input("Enter a number: ")

# Convert input string to integer
num = int(num)

# Check if the number is even or odd
if num % 2 == 0:
    print(num, "is even")
else:
    print(num, "is odd")
```

Example: Simple Interest Calculator

```
# Simple Interest Calculator
principal = input("Enter the principal amount: ")
rate = input("Enter the rate of interest: ")
time = input("Enter the time (in years): ")

# Convert input strings to float
principal = float(principal)
rate = float(rate)
time = float(time)

# Calculate simple interest
interest = (principal * rate * time) / 100

# Print the result
print("The simple interest is:", interest)
```

Strings

- Strings in Python are sequences of characters enclosed within single (`' '`), double (`" "`), or triple quotes (`''' '''` or `""" """`).
- They are immutable, meaning they cannot be changed once created.

```
# Single quotes
string1 = 'Hello, World!'

# Double quotes
string2 = "Hello, World!"

# Triple quotes
string3 = '''Hello,
World!'''

# Triple quotes can span multiple lines
string4 = """Hello,
World!"""
```

Single Quotes (`' '`)

- Used to create string literals.
- Typically used for short strings or when the string itself contains double quotes.
- Can be escaped using a backslash (`\`).
- Best for short strings, especially when the string contains double quotes.

```
single_quote_str = 'Hello, World!'
print(single_quote_str)  # Output: Hello, World!

# Using single quotes inside the string
quote_in_str = 'He said, "Hello, World!"'
print(quote_in_str)  # Output: He said, "Hello, World!" - no need to escape
since the inner quotes are double quotes.

# Using single quotes inside the string with escaping
escaped_quote_in_str = 'He said, \'Hello, World!\''
```

```
print(escaped_quote_in_str)  # Output: He said, 'Hello, World!' - escaped using a backslash (\).
```

Double Quotes (`" "`)

- Also used to create string literals.
- Preferred when the string contains single quotes to avoid escaping.
- Can be escaped using a backslash (`\`).
- Best for short strings, especially when the string contains single quotes

```
double_quote_str = "Hello, World!"
print(double_quote_str)  # Output: Hello, World!

# Using single quotes inside the string
quote_in_str = "It's a wonderful day!"
print(quote_in_str)  # Output: It's a wonderful day! - no need to escape since the inner quotes are single quotes.

# Using double quotes inside the string with escaping
escaped_quote_in_str = "He said, \"Hello, World!\""
print(escaped_quote_in_str)  # Output: He said, "Hello, World!" - escaped using a backslash (\).
```

Triple Single Quotes (`''' '''`)

- Used for multi-line strings or docstrings.
- Can contain both single and double quotes without escaping.
- Preserves the formatting, including line breaks and indentation.
- Ideal for multi-line strings and when the string contains both single and double quotes

```
triple_single_quote_str = '''This is a string
that spans multiple lines.
It can contain both "double quotes" and 'single quotes' without escaping.'''
print(triple_single_quote_str)

# Output:
# This is a string
# that spans multiple lines.
# It can contain both "double quotes" and 'single quotes' without escaping.
```

Triple Double Quotes (`""" """`)

- Functionally identical to triple single quotes.
- Often used for docstrings (multi-line comments) in functions, classes, and modules.
- Preserves the formatting, including line breaks and indentation.
- Also ideal for multi-line strings and commonly used for docstrings

```
triple_double_quote_str = """This is another string
that spans multiple lines.
It also can contain both "double quotes" and 'single quotes' without
escaping."""
print(triple_double_quote_str)

# Output:
# This is another string
# that spans multiple lines.
# It also can contain both "double quotes" and 'single quotes' without
escaping.
```

String Indexing

Positive Indexing

- Starts from `0` and goes up to `len(string) - 1`.
- Index `0` refers to the first character, index `1` to the second character, and so on.

```
text = "Hello, World!"
print(text[0])  # Output: 'H'  (first character)
print(text[7])  # Output: 'W'  (eighth character)
```

Negative Indexing

- Starts from `-1` and goes backwards from the end of the string.
- Index `-1` refers to the last character, index `-2` to the second last character, and so on.

```
text = "Hello, World!"
print(text[-1])  # Output: '!'  (last character)
```

```
print(text[-2])  # Output: 'd'  (second last character)
```

String Indexing Use Case

- **Extracting Substrings**: Retrieve specific parts of a string, such as a substring or a single character.
- **Reversing Strings**: Access characters in reverse order.
- **Manipulating User Input**: Modify or analyze parts of user-provided strings, like form inputs.
- **Parsing Data**: Extract specific fields from structured data formats.
- **Validation and Formatting**: Check and adjust the format of strings, such as dates or IDs.

String Slicing

String slicing in Python allows you to extract a portion of a string using a colon (`:`) syntax. The basic form of slicing is `string[start:stop:step]`, where:

- `start` is the index where the slice starts (inclusive).
- `stop` is the index where the slice ends (exclusive).
- `step` determines the step size or the increment between each index.

Here are the detailed examples based on the given string `text = "Hello, World!"`:

- Extracts a Substring from Index 0 to 4

```
text = "Hello, World!"
print(text[0:5])
```

- Extracts from Index 7 to the End

```
print(text[7:]) # Output: 'World!'
```

- Extracts from the Start to Index 4

```
print(text[:5])
```

- Extracts Every Second Character

```
 print(text[::2])
```

- Reverses the String

```
print(text[::-1])
```

- Extracting a Substring with a Specific Step

```
text = "Hello, World!"
# Extract every third character starting from index 0
print(text[0::3])  # Output: 'Hl r!'
```

- Extracting a Substring from the Middle

```
text = "Hello, World!"
# Extract substring from index 3 to 8
print(text[3:8])  # Output: 'lo, W'
```

Slicing Use Case

- **Extracting Substrings**: Retrieve specific parts of a string, such as words or sentences.
- **Reversing Strings**: Easily reverse the entire string or specific parts of it.
- **Formatting Strings**: Modify parts of a string to fit a certain format or extract meaningful data.
- **Analyzing Data**: Extract specific fields from structured data formats like dates or file paths.
- **Cleaning Data**: Remove unwanted parts of a string or reformat it.

String Concatenation

String concatenation is the process of combining two or more strings into one. In Python, this can be done using the `+` operator.

Using the `+` operator:

```
string1 = "Hello"
string2 = "World"
combined = string1 + ", " + string2 + "!"
```

```
print(combined)  # Output: Hello, World!
```

Using `join()` method:

```
string1 = "Hello"
string2 = "World"
combined = ", ".join([string1, string2]) + "!"
print(combined)  # Output: Hello, World!
```

Using formatted string literals (f-strings) (Python 3.6+):

```
string1 = "Hello"
string2 = "World"
combined = f"{string1}, {string2}!"
print(combined)  # Output: Hello, World!
```

Using the `format()` method:

```
string1 = "Hello"
string2 = "World"
combined = "{}, {}!".format(string1, string2)
print(combined)  # Output: Hello, World!
```

Using `%` formatting:

```
string1 = "Hello"
string2 = "World"
combined = "%s, %s!" % (string1, string2)
print(combined)  # Output: Hello, World!
```

String Concatenation Use Case

- **Building Dynamic Messages**: Combine strings to create dynamic text for user messages or logs.
- **URL Construction**: Assemble URLs from different parts, such as base URLs and query parameters.
- **File Paths**: Construct file paths by combining directory names and file names.

- **Template Strings**: Create templates by merging fixed text with dynamic data.
- **Data Formatting**: Combine multiple pieces of data into a formatted string for display or storage.

String Repetition

String repetition is the process of repeating a string a specified number of times. This can be done using the `*` operator.

```
# Defining a string
repeat_str = "Hello! "

# Repeating the string 3 times
repeat = repeat_str * 3

# Printing the repeated string
print(repeat)  # Output: 'Hello! Hello! Hello! '
```

String Repetition Use Case

- **Generating Patterns**: Create repeated patterns or borders for text-based interfaces or displays.
- **Formatting Output**: Repeat characters or strings to format output consistently, like underlining headings.
- **Initialization**: Quickly initialize a string with repeated characters for placeholders or data preparation.
- **Creating Repeated Messages**: Generate repeated warning or notification messages for emphasis.
- **Visual Separators**: Use repeated strings as visual separators in logs or reports.

String Methods

```
# Define a string for demonstration
text = "hello world"
```

```python
# Convert to uppercase
print("Uppercase:", text.upper())  # Output: 'HELLO WORLD'


# Convert to lowercase
text = "HELLO WORLD"
print("Lowercase:", text.lower())  # Output: 'hello world'


# Capitalize the first letter
text = "hello world"
print("Capitalize:", text.capitalize())  # Output: 'Hello world'


# Title case (capitalize first letter of each word)
print("Title case:", text.title())  # Output: 'Hello World'


# Swap case (invert case of each letter)
text = "Hello World"
print("Swap case:", text.swapcase())  # Output: 'hELLO wORLD'


# Replace a substring
text = "hello world"
print("Replace:", text.replace("world", "Python"))  # Output: 'hello Python'


# Split the string into a list
text = "hello-world"
words = text.split("-") # Splits on hyphen print(words) # Output: ['hello',
'world']


# Join a list into a string
words = ['hello', 'world']
print("Join:", ' '.join(words))  # Output: 'hello world'


# Strip whitespace from both ends
text = "   hello world   "
print("Strip:", text.strip())  # Output: 'hello world'


# Remove leading whitespace
print("Left strip:", text.lstrip())  # Output: 'hello world   '
```

```python
# Remove trailing whitespace
print("Right strip:", text.rstrip())  # Output: '   hello world'


# Check if string starts with a substring
text = "hello world"
print("Starts with 'hello':", text.startswith("hello"))  # Output: True


# Check if string ends with a substring
print("Ends with 'world':", text.endswith("world"))  # Output: True


# Find the position of a substring
print("Find 'world':", text.find("world"))  # Output: 6


# Count occurrences of a substring
print("Count 'o':", text.count("o"))  # Output: 2


# Check if all characters are alphanumeric
print("Is alphanumeric:", text.isalnum())  # Output: False


# Check if all characters are alphabetic
text = "hello"
print("Is alphabetic:", text.isalpha())  # Output: True


# Check if all characters are digits
text = "12345"
print("Is digit:", text.isdigit())  # Output: True


# Check if the string contains only whitespace
text = "   "
print("Is whitespace:", text.isspace())  # Output: True


# Check if the string is titlecased
text = "Hello World"
print("Is titlecased:", text.istitle())  # Output: True
```

```
# Example of combining methods
# Capitalizing each word in a sentence
sentence = "this is a sample sentence."
capitalized_sentence = sentence.title()
print("Capitalized sentence:", capitalized_sentence)  # Output: 'This Is A
Sample Sentence.'


# Removing extra spaces and converting to uppercase
text = "   hello world   "
cleaned_text = text.strip().upper()
print("Cleaned and uppercase:", cleaned_text)  # Output: 'HELLO WORLD'
```

String Methods Practical Use Case

- **Data Cleaning**: Remove unwanted characters, trim whitespace, and standardize text formats.
- **Text Analysis**: Count occurrences, find substrings, and analyze text content.
- **User Input Processing**: Validate and sanitize user inputs from forms or other sources.
- **Formatting Output**: Prepare and format strings for display or reporting.
- **Generating Dynamic Text**: Construct dynamic messages, URLs, or file paths based on variable data.

Numbers

Python supports several types of numbers: integers, floating-point numbers (floats), and complex numbers.

Basic Arithmetic Operations

```
# Define some numbers
a = 10
b = 3
c = 3.14

# Addition
print("Addition:", a + b)  # Output: 13

# Subtraction
print("Subtraction:", a - b)  # Output: 7
```

```
# Multiplication
print("Multiplication:", a * b)  # Output: 30

# Division
print("Division:", a / b)  # Output: 3.3333333333333335

# Floor Division (integer division)
print("Floor Division:", a // b)  # Output: 3

# Modulus (remainder)
print("Modulus:", a % b)  # Output: 1

# Exponentiation (power)
print("Exponentiation:", a ** b)  # Output: 1000
```

Arithmetic Operations Use Case

- **Financial Calculations**: Calculate interest, total payments, and loan amortization schedules.
- **Data Analysis**: Perform statistical calculations like mean, median, and standard deviation.
- **Graphics and Gaming**: Calculate positions, velocities, and accelerations for animations.
- **Unit Conversion**: Convert units, such as from miles to kilometers or Celsius to Fahrenheit.
- **Recipe Scaling**: Adjust ingredient quantities based on the number of servings.

Type Conversion

```
x = 10    # Integer
y = 3.14  # Float

# Convert int to float
print("Convert int to float:", float(x))  # Output: 10.0

# Convert float to int
print("Convert float to int:", int(y))  # Output: 3

# Convert int to complex
print("Convert int to complex:", complex(x))  # Output: (10+0j)
```

Math

```
import math

# Square root
print("Square root:", math.sqrt(16))  # Output: 4.0

# Power
print("Power:", math.pow(2, 3))  # Output: 8.0

# Trigonometric functions
print("Sine of 90 degrees:", math.sin(math.radians(90)))  # Output: 1.0
print("Cosine of 0 degrees:", math.cos(math.radians(0)))  # Output: 1.0

# Logarithmic functions
print("Natural log of 10:", math.log(10))  # Output: 2.302585092994046
print("Log base 10 of 10:", math.log10(10))  # Output: 1.0

# Factorial
print("Factorial of 5:", math.factorial(5))  # Output: 120

# Greatest common divisor
print("GCD of 48 and 180:", math.gcd(48, 180))  # Output: 12

# Absolute value
print("Absolute value of -7.5:", math.fabs(-7.5))  # Output: 7.5

# Floor and Ceiling
print("Floor of 3.7:", math.floor(3.7))  # Output: 3
print("Ceiling of 3.7:", math.ceil(3.7))  # Output: 4

# Constants
print("Pi:", math.pi)  # Output: 3.141592653589793
print("Euler's number:", math.e)  # Output: 2.718281828459045
```

Math Functions Use Case

- **Financial Calculations**: Compute compound interest, loan amortization schedules, and investment growth using exponential and logarithmic functions.
- **Data Analysis**: Perform statistical analyses such as calculating mean, median, standard deviation, and correlation coefficients.

- **Scientific Computing**: Solve equations, perform trigonometric calculations, and analyze physical phenomena.
- **Game Development**: Calculate angles, distances, and collision detection using trigonometric and geometric functions.
- **Engineering**: Design and analyze systems, perform signal processing, and compute stress and strain using advanced mathematical functions.

Operator Precedence

Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence

Operator Precedence Table (from highest to lowest)

1. **Exponentiation (``)**`*`*
2. **Unary plus, Unary minus, Bitwise NOT (`+x`, `-x`, `~x`)**
3. **Multiplication, Division, Floor division, Modulus (`*`, `/`, `//`, `%`)**
4. **Addition, Subtraction (`+`, `-`)**
5. **Bitwise shift (`<<`, `>>`)**
6. **Bitwise AND (`&`)**
7. **Bitwise XOR (`^`)**
8. **Bitwise OR (`|`)**
9. **Comparisons, Identity, Membership (`==`, `!=`, `>`, `<`, `>=`, `<=`, `is`, `is not`, `in`, `not in`)**
10. **Logical NOT (`not`)**
11. **Logical AND (`and`)**
12. **Logical OR (`or`)**

Example 1: Exponentiation vs. Multiplication

```
result = 2 ** 3 * 2
print("2 ** 3 * 2:", result)  # Output: 16
# Explanation: 2 ** 3 is evaluated first (8), then 8 * 2 = 16
```

Example 2: Multiplication vs. Addition

```
result = 10 + 3 * 2
print("10 + 3 * 2:", result)  # Output: 16
# Explanation: 3 * 2 is evaluated first (6), then 10 + 6 = 16
```

Operator Precedence use case

- **Mathematical Expressions**: Ensure correct order of operations in complex calculations involving multiple arithmetic operators.
- **Data Analysis**: Accurately compute expressions in data processing pipelines where multiple operations are performed sequentially.
- **Programming Logic**: Implement conditional statements and loops with mixed logical and comparison operators.
- **Financial Calculations**: Calculate investment returns, loan payments, and other financial metrics accurately by respecting operator precedence.
- **Game Development**: Evaluate expressions involving multiple operations, such as calculating positions, velocities, and collision responses.

`If` Statements

The `if` statement executes a block of code if a specified condition is `True`.

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

`Else` Statements

The `else` statement executes a block of code if the `if` condition is `False`.

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

Else If `(Elif)` Statements

The `elif` statement allows you to check multiple conditions. It stands for "else if" and can be used when you need to check more than one condition.

```
age = 16
if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

Combining `If`, `Elif`, and `Else` Statements

```
score = 75

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")
```

Nested `If` Statements

You can also nest `if` statements within other `if` statements to check more complex conditions.

```
age = 20
has_permission = True

if age >= 18:
    if has_permission:
        print("You can enter the club.")
    else:
        print("You need permission to enter the club.")
else:
    print("You are not allowed to enter the club.")
```

if Else use case

- **User Authentication**: Check if the entered username and password match the stored credentials and grant or deny access.
- **Form Validation**: Validate user input in forms and provide feedback or error messages.
- **Payment Processing**: Determine if a payment transaction is successful or if an error occurred, and handle each case accordingly.
- **Data Filtering**: Filter data based on specific criteria, such as filtering out invalid entries from a dataset.
- **Weather Forecasting**: Display different messages or actions based on weather conditions, such as suggesting an umbrella if it's going to rain.
- **Inventory Management**: Check if stock levels are sufficient to fulfill an order and alert if more inventory is needed.
- **Game Logic**: Determine game outcomes based on player actions or states, such as winning, losing, or drawing a game.

- **Personalized Greetings**: Provide personalized greetings or messages based on the time of day or user preferences.
- **Discount Application**: Apply discounts to purchases based on customer status, such as member, non-member, or special promotions.
- **File Handling**: Check if a file exists before attempting to read or write to prevent errors and handle cases where the file is missing.

`for` Loop in Python

The `for` loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) or other iterable objects.

Iterating Over a List

```
# Example of iterating over a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

Iterating Over a String

```
# Example of iterating over a string
word = "hello"
for letter in word:
    print(letter)
```

Using `range()` Function

```
# Example of using range() function
for i in range(5):
    print(i)
```

Iterating Over a Dictionary

```
# Example of iterating over a dictionary
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
for student, score in student_scores.items():
```

```
    print(f"{student}: {score}")
```

Iterating Over a Set

```
# Example of iterating over a set
unique_numbers = {1, 2, 3, 4, 5}
for number in unique_numbers:
    print(number)
```

Using `break` Statement

```
# Example of using break statement
for number in range(10):
    if number == 5:
        break
    print(number)
```

Using `continue` Statement

```
# Example of using break statement
for number in range(10):
    if number == 5:
        break
    print(number)
```

for loop use case

- **Data Processing**: Iterate over a list of data points to perform calculations or transformations.
- **File Handling**: Read and process lines in a file sequentially.
- **Generating Reports**: Create summaries or reports by iterating over data records.
- **Batch Processing**: Apply operations to a batch of items, such as resizing images or processing transactions.
- **Automating Tasks**: Automate repetitive tasks like sending emails or making API calls.
- **Iterating Over Dictionaries**: Access keys and values in a dictionary for tasks like configuration or data analysis.

- **Matrix Operations**: Perform operations on matrices or 2D arrays, such as addition, multiplication, or transposition.
- **Building User Interfaces**: Generate dynamic UI components by iterating over data models.
- **Simulation and Modeling**: Run simulations by iterating over time steps or model parameters.
- **Web Scraping**: Extract information from web pages by iterating over HTML elements.

`While` Loops in Python

Iterating Over a List

```
# Iterating over a list with a while loop
fruits = ['apple', 'banana', 'cherry']
index = 0
while index < len(fruits):
    print(fruits[index])
    index += 1
```

Iterating Over a String

```
# Iterating over a string with a while loop
word = "hello"
index = 0
while index < len(word):
    print(word[index])
    index += 1
```

Using range() Function

```
# Simulating range() with a while loop
start = 0
end = 5
while start < end:
    print(start)
    start += 1
```

Iterating Over a Dictionary

```
# Iterating over a dictionary with a while loop
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
keys = list(student_scores.keys())
index = 0
while index < len(keys):
    key = keys[index]
    print(f"{key}: {student_scores[key]}")
    index += 1
```

Iterating Over a Set

```
# Iterating over a dictionary with a while loop
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
keys = list(student_scores.keys())
index = 0
while index < len(keys):
    key = keys[index]
    print(f"{key}: {student_scores[key]}")
    index += 1
```

Using break Statement

```
# Using break statement in a while loop
counter = 0
while counter < 10:
    if counter == 5:
        break
    print(counter)
    counter += 1
```

Using continue Statement

```
# Using continue statement in a while loop
counter = 0
while counter < 10:
    counter += 1
    if counter % 2 == 0:
        continue
```

```
    print(counter)
```

While loop use case

- **User Input Validation**: Continuously prompt the user for input until valid data is provided.
- **Reading Files**: Read data from a file until the end of the file is reached.
- **Polling for Changes**: Continuously check for changes in data or status until a condition is met.
- **Implementing Timers**: Create countdown timers or delay loops.
- **Game Loops**: Run the main loop of a game, which continues until the game is over.
- **Retry Logic**: Retry an operation until it succeeds or a maximum number of attempts is reached.
- **Simulations**: Run simulations that proceed until a certain condition is met.
- **Processing Queues**: Process items from a queue until it is empty.
- **Progress Tracking**: Track and update progress until a task is complete.
- **Generating Sequences**: Generate a sequence of numbers or data until a certain condition is reached.

What about other's type of loop

- In Python, there are `for` and `while` loops, but there is no direct equivalent to the `do-while` loop found in some other programming languages.
- Additionally, there is no `for in`, `for of`, or `forEach` loop syntax specifically like in JavaScript

---

Logical Operators in Python

Logical operators are used to combine conditional statements. The most common logical operators in Python are `and`, `or`, and `not`.

1. `and` Operator

The `and` operator returns `True` if both conditions are `True`. If either condition is `False`, the result is `False`.

```
age = 20
has_permission = True
```

```
if age >= 18 and has_permission:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

2. `or` Operator

The `or` operator returns `True` if at least one of the conditions is `True`. If both conditions are `False`, the result is `False`.

```
age = 16
has_permission = True

if age >= 18 or has_permission:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

3. `not` Operator

The `not` operator inverts the result of the condition. If the condition is `True`, `not` makes it `False`, and if the condition is `False`, `not` makes it `True`.

```
age = 16

if not age >= 18:
    print("You are not an adult.")
else:
    print("You are an adult.")
```

4. Combining Logical Operators

You can combine multiple logical operators to form more complex conditions.

```
age = 20
has_permission = False
is_vip = True

if (age >= 18 and has_permission) or is_vip:
    print("You can enter the club.")
```

```
else:
    print("You cannot enter the club.")
```

Logical Operators Use case

- **Access Control**: Check multiple conditions to grant or deny access to resources.
- **Input Validation**: Validate multiple input criteria simultaneously.
- **Search Functionality**: Filter search results based on multiple criteria.
- **Feature Toggles**: Enable or disable features based on various conditions.
- **Data Filtering**: Filter data records based on multiple conditions.
- **E-commerce**: Apply discounts and promotions based on combined conditions.
- **Game Development**: Determine game state changes based on multiple player actions or game conditions.
- **Scheduling**: Check for multiple availability conditions before scheduling an event.
- **Configuration Management**: Apply configuration settings based on multiple environment variables or settings.
- **Monitoring and Alerts**: Trigger alerts based on combined system monitoring conditions.

Comparison Operators in Python

Comparison operators are used to compare two values and return a Boolean result (`True` or `False`). These operators are essential for making decisions in your code using conditional statements.

1. Equal to (`==`)

The `==` operator checks if two values are equal.

```
x = 5
y = 5
print(x == y)  # True
```

2. Not equal to (`!=`)

The `!=` operator checks if two values are not equal.

```
x = 5
y = 3
print(x != y)  # True
```

3. Greater than (`>`)

The `>` operator checks if the value on the left is greater than the value on the right.

```
x = 7
y = 5
print(x > y)  # True
```

4. Less than (`<`)

The `<` operator checks if the value on the left is less than the value on the right.

```
x = 3
y = 5
print(x < y)  # True
```

5. Greater than or equal to (`>=`)

```
x = 5
y = 5
print(x >= y)  # True
```

6. Less than or equal to (`<=`)

```
x = 5
y = 7
print(x <= y)  # True
```

Comparison Operators Use case

- **User Authentication**: Verify if entered credentials match stored credentials.
- **Input Validation**: Ensure user input meets specific criteria, such as age or date range.
- **Sorting Data**: Compare elements to sort lists, tuples, or other data structures.

- **Conditional Formatting**: Apply different formatting based on data values, such as highlighting high scores.
- **Inventory Management**: Check stock levels and trigger reorder processes if inventory falls below a certain threshold.
- **Financial Transactions**: Validate if transactions exceed credit limits or fall within acceptable ranges.
- **Performance Monitoring**: Compare current system metrics against baseline values to trigger alerts.
- **Game Development**: Determine outcomes based on player scores or in-game conditions.
- **Access Control**: Grant or deny access based on user roles or permissions.
- **Data Analysis**: Filter and segment data based on comparison criteria.

Lists

- **Ordered**: Lists maintain the order of elements. This means that when you add elements to a list, they retain their position, and you can access elements using their index. The order of elements is preserved during iteration.
- **Mutable**: Lists are mutable, meaning you can modify them after creation. You can add, remove, or change elements within a list without creating a new list.
- **Allow duplicates**: Lists can contain duplicate elements. This allows you to have multiple occurrences of the same value within a list.
- **Heterogeneous**: Lists can hold elements of different data types. For example, a list can contain integers, strings, floats, and even other lists.
- **Dynamic size**: Lists in Python are dynamic, meaning their size can change as you add or remove elements.

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

List Methods

1. `append()`

Adds an element to the end of the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)
```

2. `insert()`

Inserts an element at a specified position.

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "kiwi")
print(fruits)
```

3. `extend()`

Extends the list by adding elements from another list.

```
fruits = ["apple", "banana", "cherry"]
more_fruits = ["grape", "melon"]
fruits.extend(more_fruits)
print(fruits)
```

4. `remove()`

Removes the first occurrence of the specified element.

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)
```

5. `pop()`

Removes the element at the specified position (default is the last element) and returns it.

```
fruits = ["apple", "banana", "cherry"]
last_fruit = fruits.pop()
print(last_fruit)
print(fruits)
```

6. `clear()`

Removes all elements from the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits)
```

7. `index()`

Returns the index of the first occurrence of the specified element.

```
fruits = ["apple", "banana", "cherry"]
index = fruits.index("banana")
print(index)
```

8. `count()`

Returns the number of occurrences of the specified element.

```
fruits = ["apple", "banana", "cherry"]
count = fruits.count("apple")
print(count)
```

9. `sort()`

Sorts the list in ascending order by default.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.sort()
print(numbers)
```

10. `reverse()`

Reverses the order of the list.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.reverse()
print(numbers)
```

12. `len()`

Returns the number of elements in the list.

```
fruits = ["apple", "banana", "cherry"]
length = len(fruits)
print(length)
```

13. List Slicing

You can access a range of elements using slicing.

```
fruits = ["apple", "banana", "cherry", "date", "fig", "grape"]
print(fruits[1:4])  # ['banana', 'cherry', 'date']
print(fruits[:3])   # ['apple', 'banana', 'cherry']
print(fruits[3:])   # ['date', 'fig', 'grape']
print(fruits[-3:])  # ['date', 'fig', 'grape']
```

14. Looping Through a List

```
fruits = ["apple", "banana", "cherry", "date", "fig", "grape"]
for fruit in fruits:
    print(fruit)
```

List Use Case

- **Storing User Data**: Keep a list of user names, email addresses, or IDs for easy access and manipulation.
- **Managing To-Do Lists**: Track tasks and their statuses in a to-do list application.
- **Inventory Management**: Maintain a list of product items, quantities, and details in a store inventory system.
- **Processing Orders**: Store and process customer orders in an e-commerce application.
- **Collecting Survey Responses**: Gather and analyze survey responses from multiple participants.
- **Scheduling Events**: Organize and manage a list of events or appointments in a calendar application.
- **Data Analysis**: Store and manipulate datasets for statistical analysis or machine learning.
- **Playlist Management**: Keep track of songs, videos, or other media items in a playlist.
- **Shopping Cart**: Store items added to a shopping cart in an online shopping system.
- **Tracking Scores**: Maintain a list of scores or results for games or competitions.

Tuples

- **Ordered**: Like lists, tuples maintain the order of elements. The order in which elements are added is preserved, and they can be accessed using an index.
- **Immutable**: Once a tuple is created, it cannot be modified. You cannot add, remove, or change elements in a tuple after its creation. This immutability makes tuples suitable for use as keys in dictionaries and ensures that the data remains consistent.
- **Allow duplicates**: Tuples can contain duplicate elements. This allows multiple occurrences of the same value within a tuple.
- **Heterogeneous**: Tuples can hold elements of different data types. For example, a tuple can contain integers, strings, floats, and even other tuples.
- **Fixed size**: The size of a tuple is fixed upon creation. Unlike lists, you cannot change the size of a tuple after it is created.

```python
fruits = ("apple", "banana", "cherry")
print(fruits)
print(fruits[0])  # Output: apple
print(fruits[1])  # Output: banana
print(fruits[2])  # Output: cherry
```

Tuple Methods

1. `count()`

Returns the number of times a specified value appears in the tuple.

```python
numbers = (1, 2, 3, 2, 4, 2)
count_of_twos = numbers.count(2)
print(count_of_twos)  # Output: 3
```

2. `index()`

Returns the index of the first occurrence of the specified value.

```python
numbers = (1, 2, 3, 2, 4, 2)
index_of_three = numbers.index(3)
print(index_of_three)  # Output: 2
```

3. Looping Through a Tuple

You can loop through the elements of a tuple using a `for` loop.

```python
fruits = ("apple", "banana", "cherry")
for fruit in fruits:
    print(fruit)
```

4. Tuple Slicing

You can access a range of elements in a tuple using slicing.

```python
fruits = ("apple", "banana", "cherry")
print(fruits[1:3])  # Output: ('banana', 'cherry')
print(fruits[:2])   # Output: ('apple', 'banana')
print(fruits[1:])   # Output: ('banana', 'cherry')
print(fruits[-2:])  # Output: ('banana', 'cherry')
```

5. Converting Between Lists and Tuples

You can convert lists to tuples and vice versa using the `tuple()` and `list()` functions.

```
# List to tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)  # Output: (1, 2, 3)

# Tuple to list
my_tuple = (4, 5, 6)
my_list = list(my_tuple)
print(my_list)  # Output: [4, 5, 6]
```

Tuples Use case

- **Immutable Data Storage**: Store read-only configuration settings that should not be altered during program execution.
- **Return Multiple Values**: Return multiple values from a function efficiently.
- **Fixed Collections**: Store a fixed collection of related data, such as coordinates (x, y, z) or RGB color values.
- **Dictionary Keys**: Use tuples as keys in dictionaries for composite key lookups.
- **Database Records**: Represent rows of database records as tuples for easy and consistent access.
- **Data Integrity**: Ensure data integrity by using tuples for sequences of data that should remain constant.
- **Grouping Data**: Group heterogeneous data types together logically, such as an employee record with a name, ID, and position.
- **Efficient Iteration**: Iterate over a fixed set of elements without the overhead of mutable data structures.
- **Named Tuples**: Use named tuples to create self-documenting code and improve code readability.
- **Function Arguments**: Pass a fixed set of parameters to functions, ensuring the parameters remain unchanged.

Sets in Python

- **Unordered**: The elements in a set do not have a defined order. Unlike lists or tuples, when you iterate over a set, the items may appear in a different order each time, and they