⬛ 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:

```python
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

```python
# 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'])
```

```python
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'}`.

```python
# 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])`.

```python
# 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: ")
```

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

⬛ Example: Temperature Converter (Celsius to Fahrenheit)

```python
# 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

```python
# 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)
```