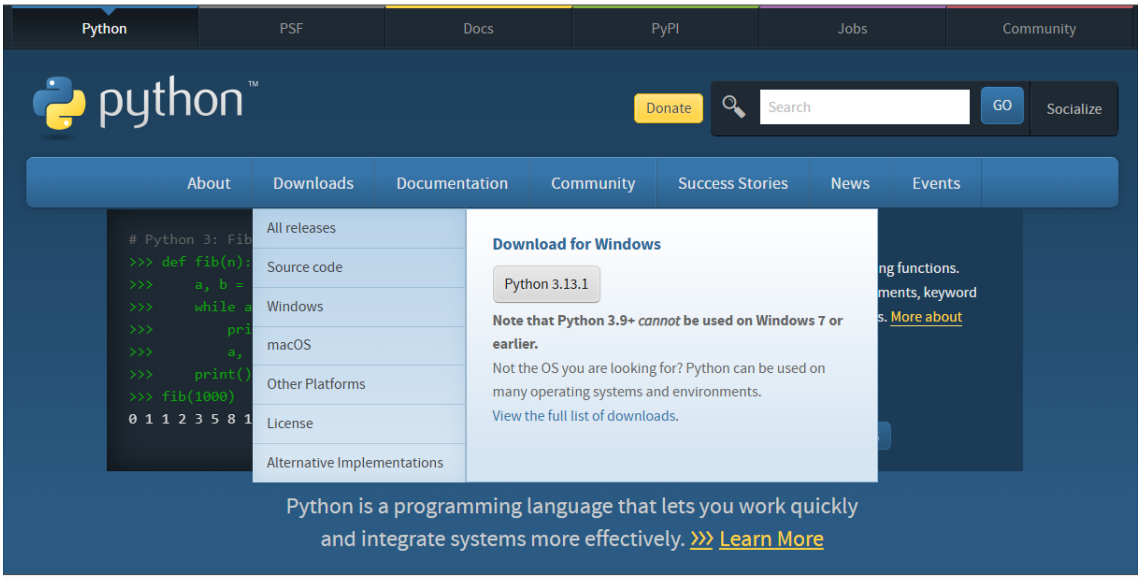
BLUKODE Academy

Day 1: Python Live Session

1. Installation of Python

- Download and install Python (latest stable version).
- Download link: Download Python



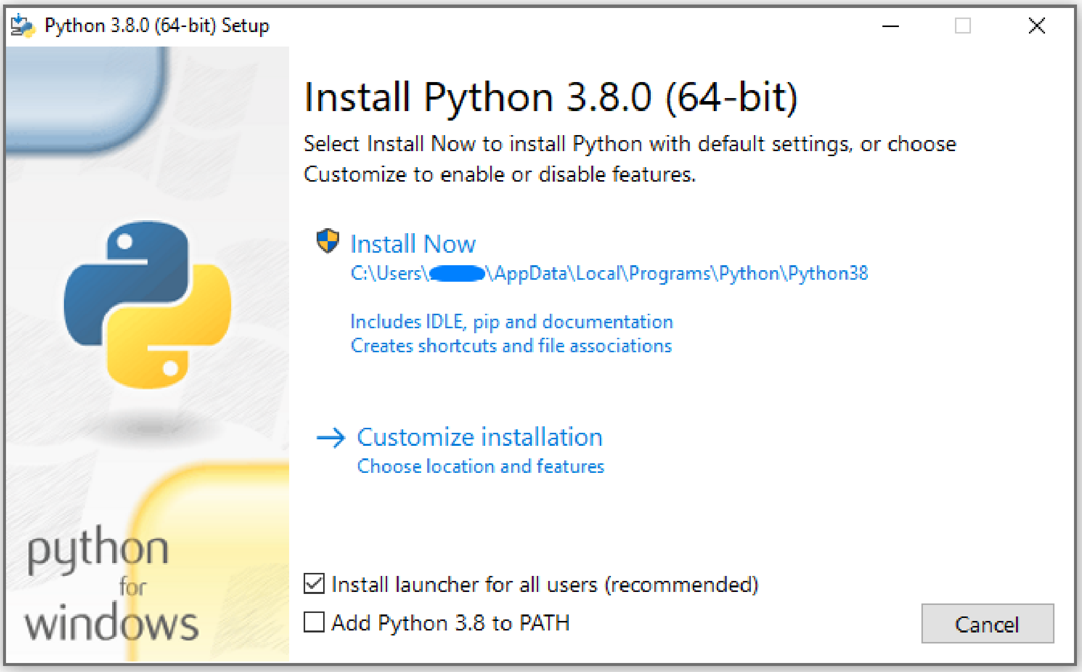
Note that Python 3.9+ *cannot* be used on Windows 7 or earlier.

Not the OS you are looking for? Python can be used on many operating systems and environments.

View the full list of downloads.

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

- Setting up the environment (PATH variable, pip).



Install Python 3.8.0 (64-bit)

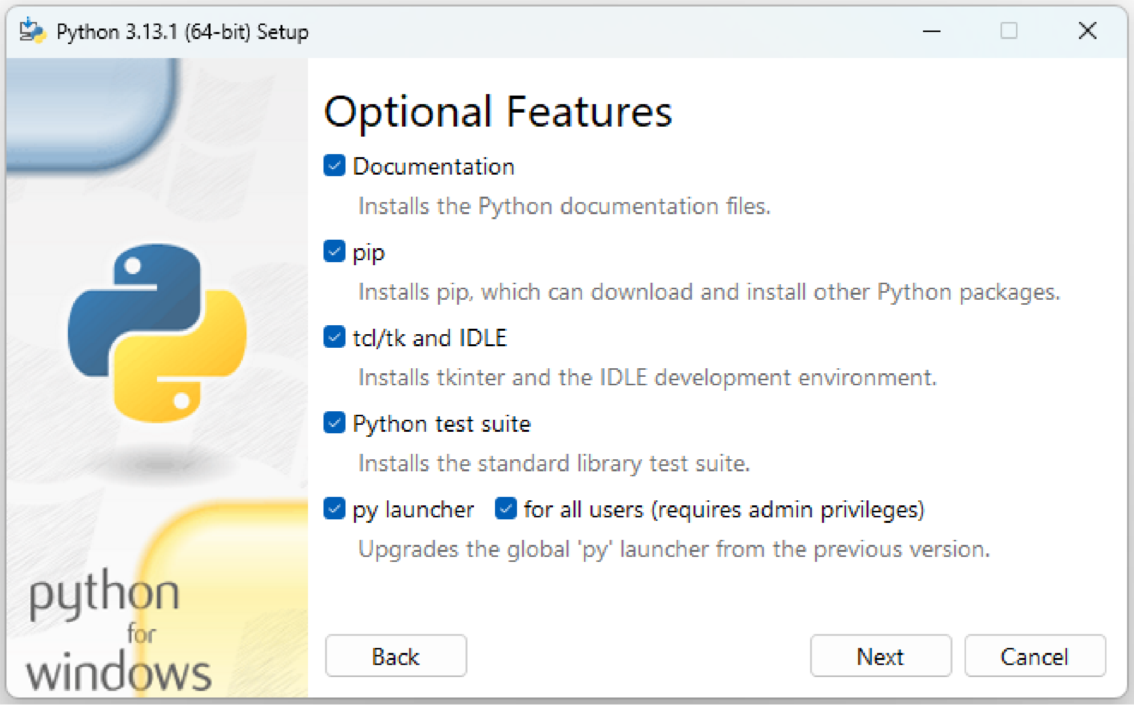
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python38

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
□ Add Python 3.8 to PATH



Optional Features

☑ Documentation

Installs the Python documentation files.

☑ pip

Installs pip, which can download and install other Python packages.

☑ tcl/tk and IDLE

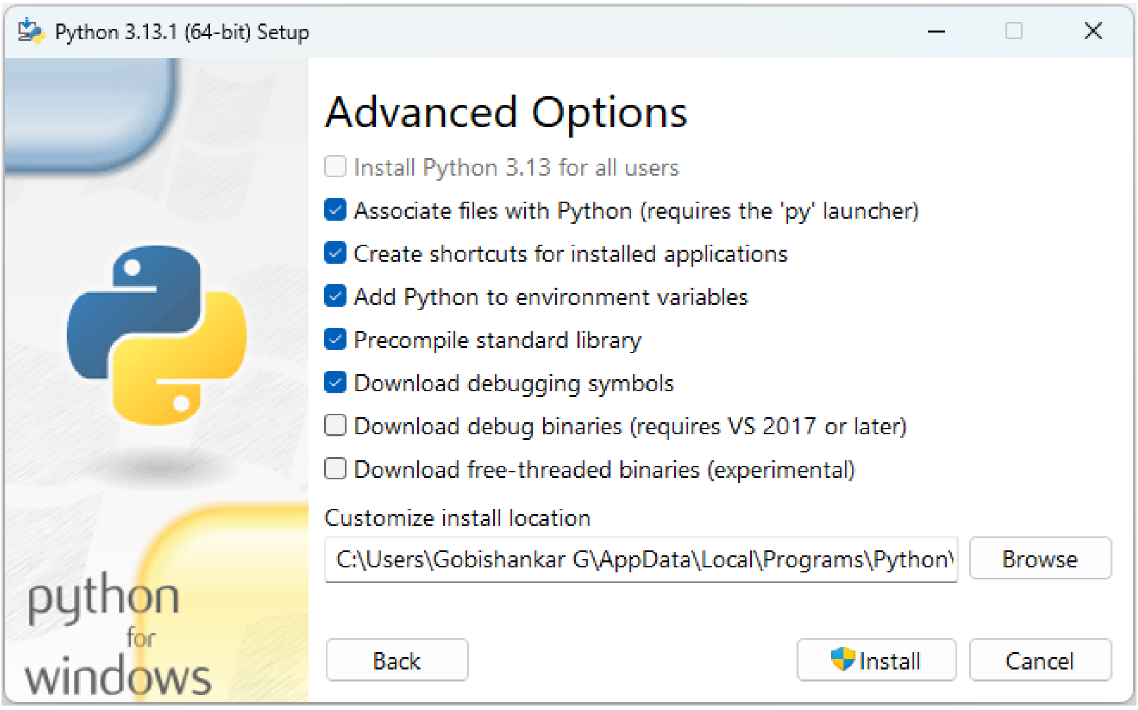
Installs tkinter and the IDLE development environment.

☑ Python test suite

Installs the standard library test suite.

☑ py launcher ☑ for all users (requires admin privileges)

Upgrades the global 'py' launcher from the previous version.



Advanced Options

☐ Install Python 3.13 for all users

☑ Associate files with Python (requires the 'py' launcher)

☑ Create shortcuts for installed applications

☑ Add Python to environment variables

☑ Precompile standard library

☑ Download debugging symbols

☐ Download debug binaries (requires VS 2017 or later)

☐ Download free-threaded binaries (experimental)

Customize install location

C:\Users\Gobishankar G\AppData\Local\Programs\Python\

- Introduction to Python IDEs (e.g., VSCode, Jupyter Notebook).

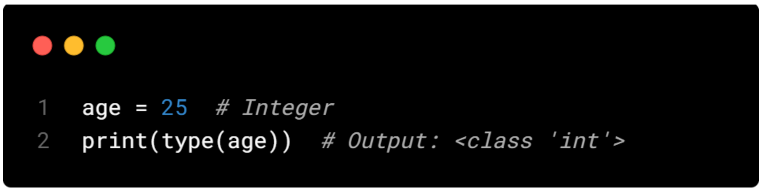
2. Data Types

In Python, **data types** represent the kind of value a variable holds. The data type determines how much space it occupies in memory and what kind of operations can be performed on it.

- **Primitive Data Types**

I.Integer (int):

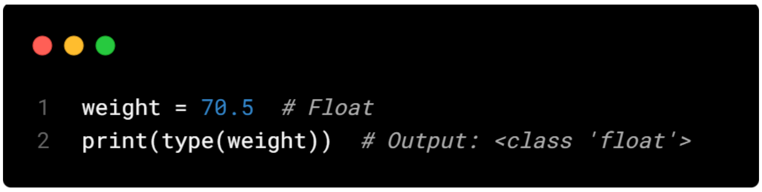
Represents whole numbers, both positive and negative, without decimals.



```python
age = 25   # Integer
print(type(age))   # Output: <class 'int'>
```

II.Float (float):

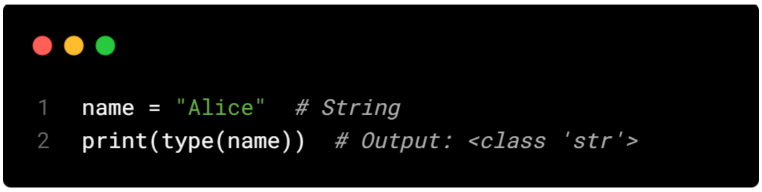
Represents real numbers or numbers with decimal points.



```python
weight = 70.5   # Float
print(type(weight))   # Output: <class 'float'>
```

III.String (str):

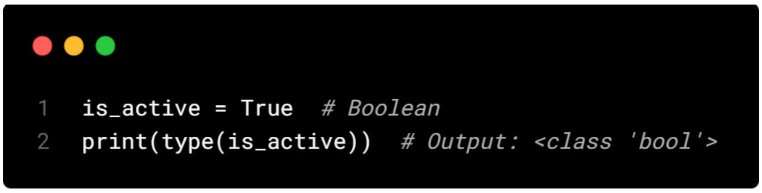
Represents a sequence of characters enclosed in quotes (single, double, or triple quotes).



```python
name = "Alice"   # String
print(type(name))   # Output: <class 'str'>
```

IV.Boolean

Represents two values, either **True** or **False**.

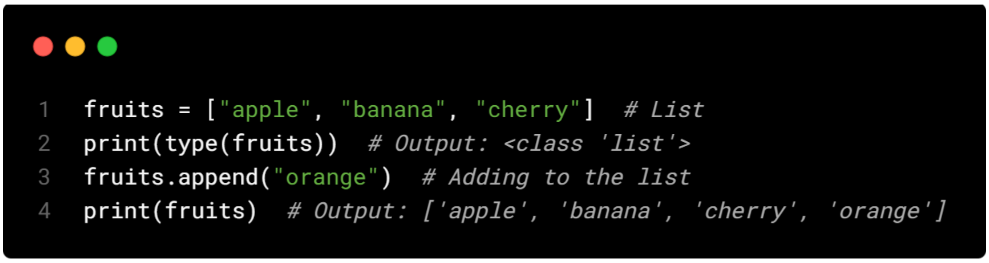


```python
is_active = True   # Boolean
print(type(is_active))   # Output: <class 'bool'>
```

- **Collection Types**

I.List (list):

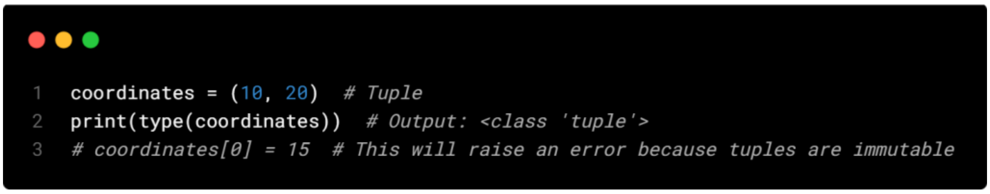
An ordered collection of elements that can be of any data type. Lists are mutable (can be changed).



```python
fruits = ["apple", "banana", "cherry"]  # List
print(type(fruits))  # Output: <class 'list'>
fruits.append("orange")  # Adding to the list
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange']
```

II.Tuple (tuple):

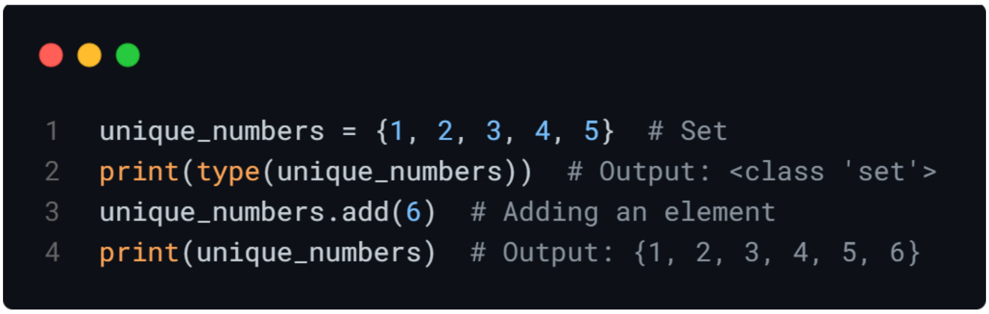
An ordered collection of elements, similar to a list, but immutable (cannot be changed).



```python
coordinates = (10, 20)  # Tuple
print(type(coordinates))  # Output: <class 'tuple'>
# coordinates[0] = 15  # This will raise an error because tuples are immutable
```

III.Set (set):

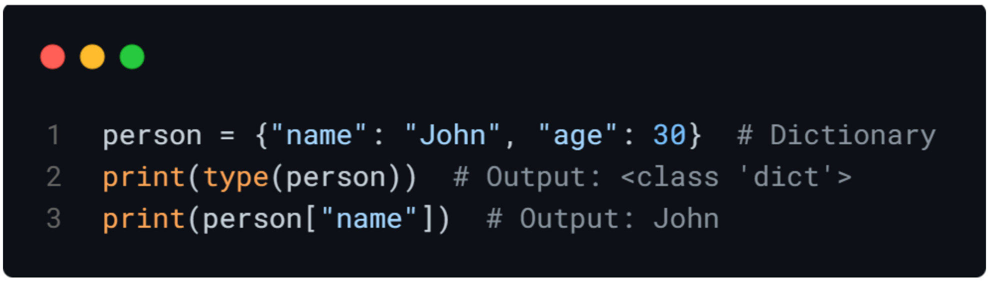
An unordered collection of unique elements. Sets do not allow duplicate items.



```python
unique_numbers = {1, 2, 3, 4, 5}  # Set
print(type(unique_numbers))  # Output: <class 'set'>
unique_numbers.add(6)  # Adding an element
print(unique_numbers)  # Output: {1, 2, 3, 4, 5, 6}
```

IV.Dictionary (dict):

A collection of key-value pairs. Each key is unique, and values can be any data type.



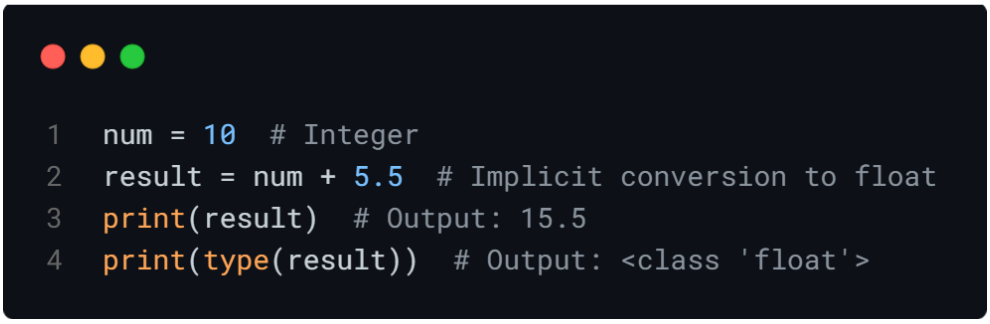
```python
person = {"name": "John", "age": 30}  # Dictionary
print(type(person))  # Output: <class 'dict'>
print(person["name"])  # Output: John
```

- **Typecasting function**

Typecasting is the process of converting one data type into another. Python allows implicit typecasting (automatically by the interpreter) and explicit typecasting (using functions like **int()**, **float()**, **str()**).

I.Implicit Typecasting:

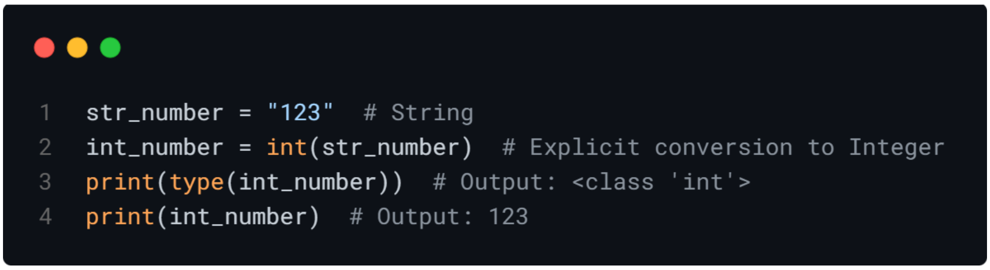
Happens automatically when converting a lower data type to a higher one.



```python
num = 10  # Integer
result = num + 5.5  # Implicit conversion to float
print(result)  # Output: 15.5
print(type(result))  # Output: <class 'float'>
```

II.Explicit Typecasting:

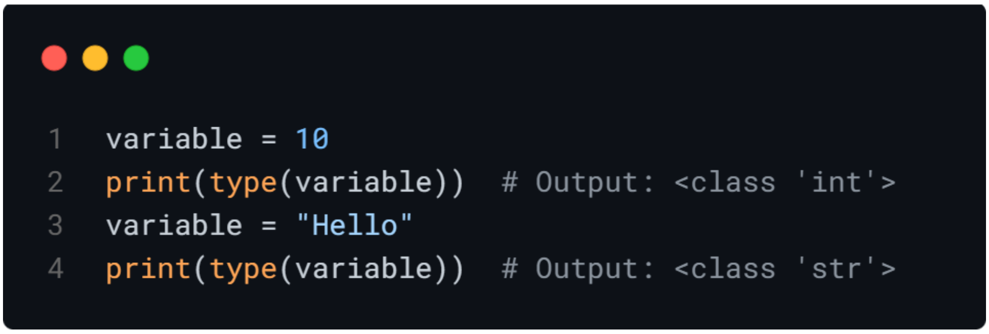
Requires the user to manually convert data types.



```python
str_number = "123"  # String
int_number = int(str_number)  # Explicit conversion to Integer
print(type(int_number))  # Output: <class 'int'>
print(int_number)  # Output: 123
```

- **Type() Function**

The **type()** function in Python is used to check the data type of a given variable or value.

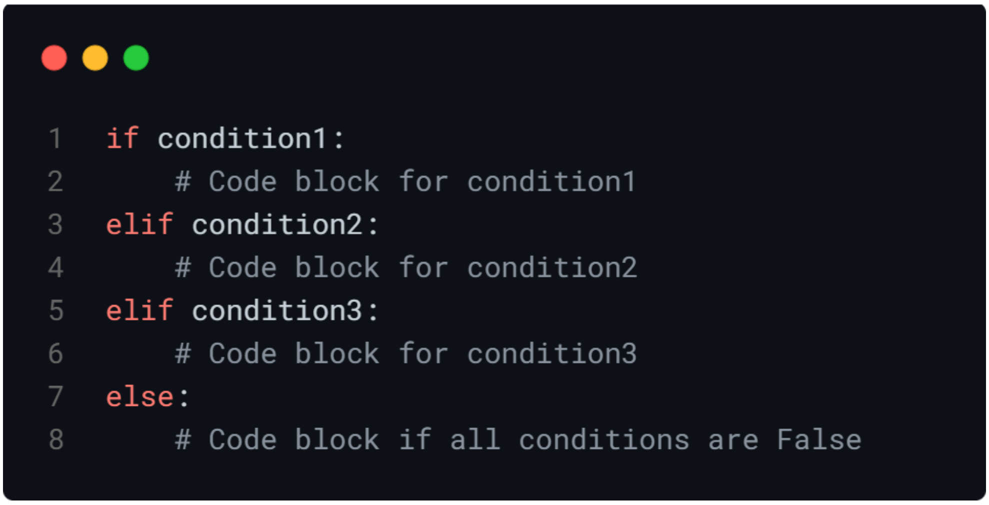


```python
variable = 10
print(type(variable))  # Output: <class 'int'>
variable = "Hello"
print(type(variable))  # Output: <class 'str'>
```

3. Conditions

In Python, **conditions** are used to execute a block of code only if a specified condition is true. Conditions rely on comparison operators (e.g., ==, >, <, etc.) and logical operators (and, or, not).

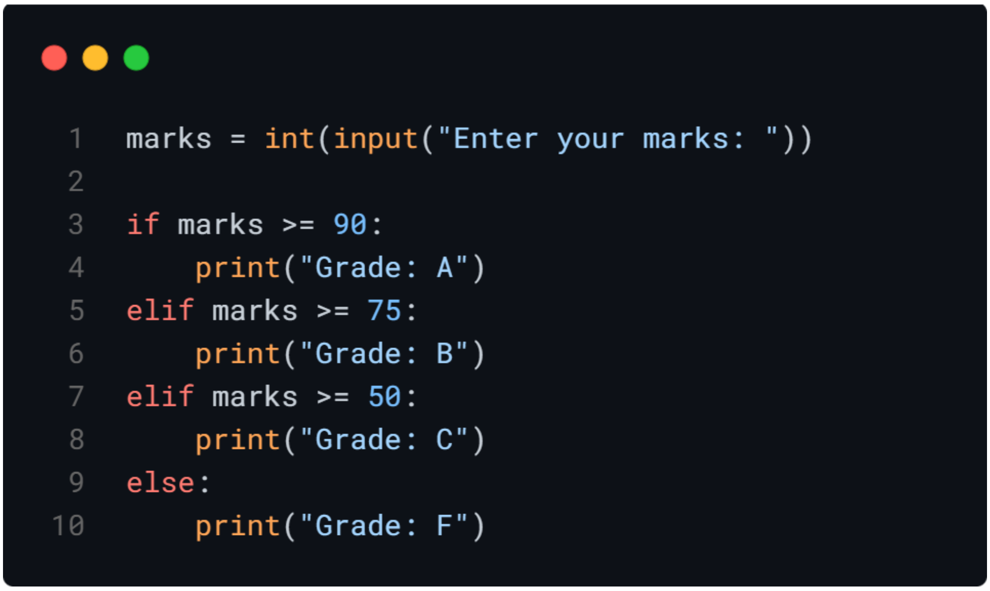
- **If-elif-else statements:**



```python
if condition1:
    # Code block for condition1
elif condition2:
    # Code block for condition2
elif condition3:
    # Code block for condition3
else:
    # Code block if all conditions are False
```

➢ **if**: Executes if **condition1** is True.
➢ **elif**: Checked if the previous condition(s) were False.
➢ **else**: Executes if none of the **if** or **elif** conditions are True.

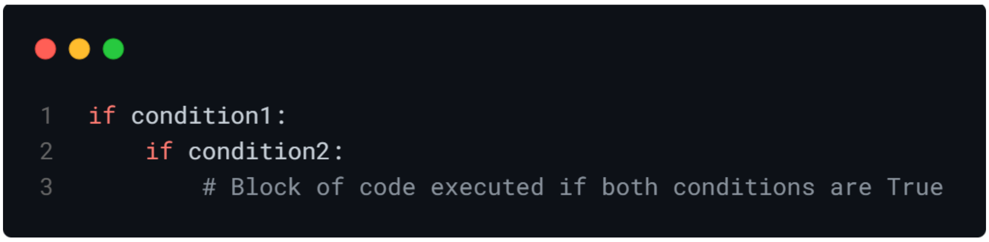
Example: Grading System



```python
marks = int(input("Enter your marks: "))

if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 50:
    print("Grade: C")
else:
    print("Grade: F")
```

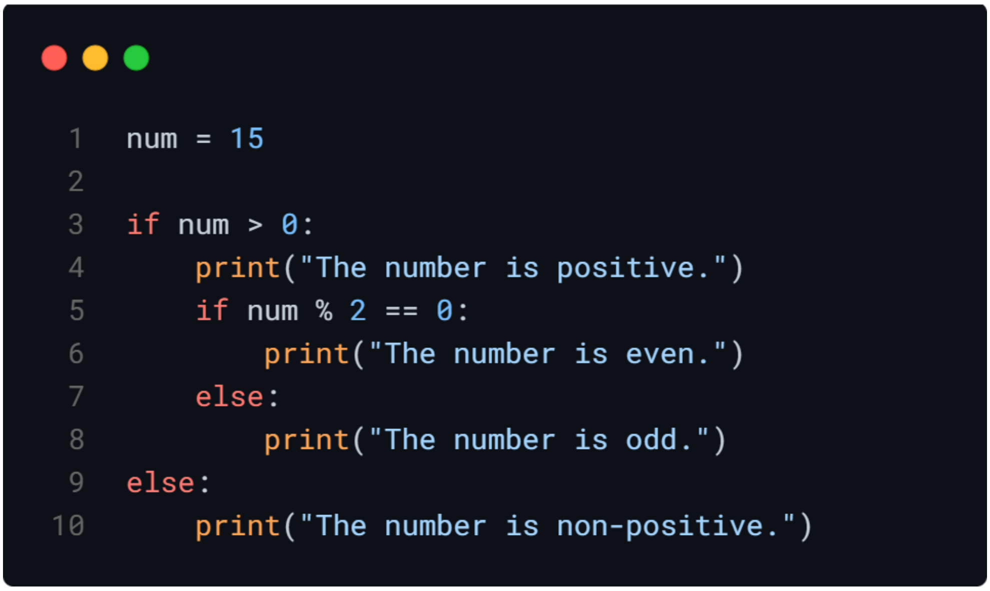
- **Nested if statements:**
  When one if statement is placed inside another, it is called nested if. This helps check multiple conditions.

  Syntax:



```
if condition1:
    if condition2:
        # Block of code executed if both conditions are True
```

Example:



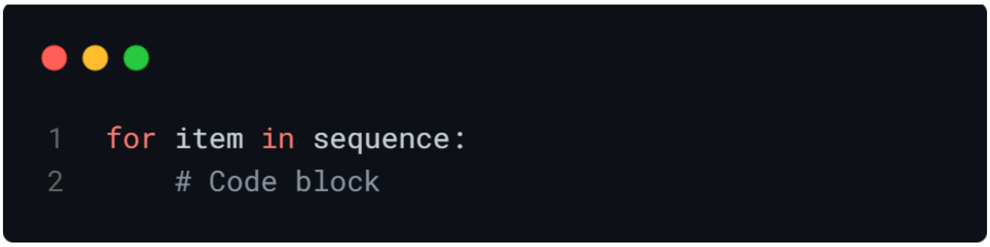
```
num = 15

if num > 0:
    print("The number is positive.")
    if num % 2 == 0:
        print("The number is even.")
    else:
        print("The number is odd.")
else:
    print("The number is non-positive.")
```

4. Loops

Loops are used in Python to execute a block of code multiple times until a specific condition is met.

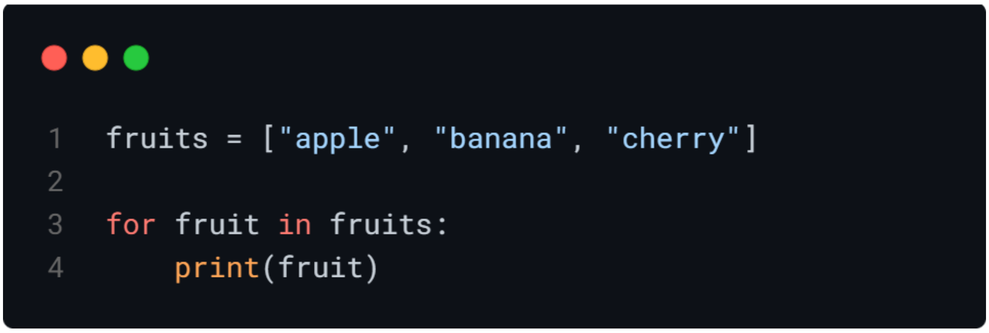
- **for loop**

  The **for** loop iterates over a sequence (like a list, tuple, string, or range) and executes the block of code for each element.



```
for item in sequence:
    # Code block
```

Example 1: Iterating over a list

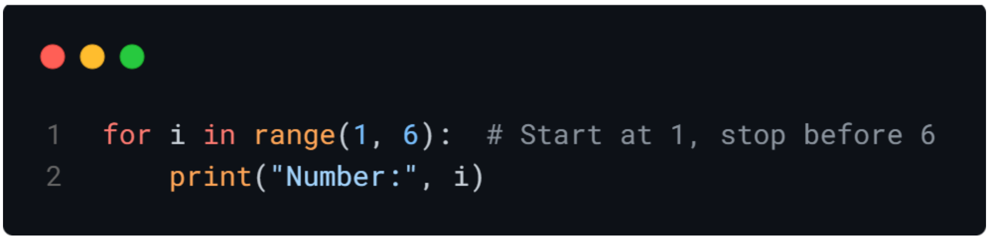


```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Explanation:

- The loop iterates through each element in the **fruits** list.
- On each iteration, the value of fruit changes to the current element ("**apple**", then "**banana**", then "**cherry**").

Example 2: Using range()



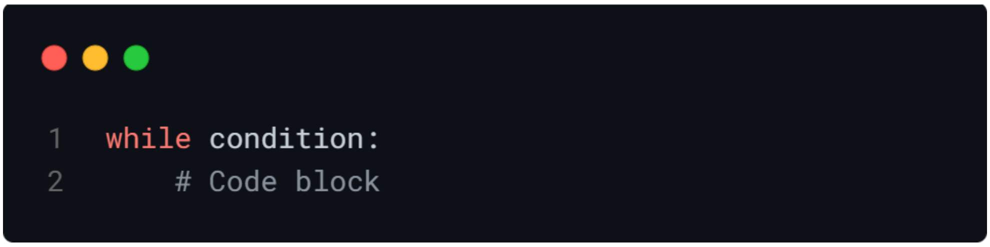
```
for i in range(1, 6):  # Start at 1, stop before 6
    print("Number:", i)
```

Explanation:

- The **range()** function generates numbers from 1 to 5.
- The loop prints each number during the iteration.

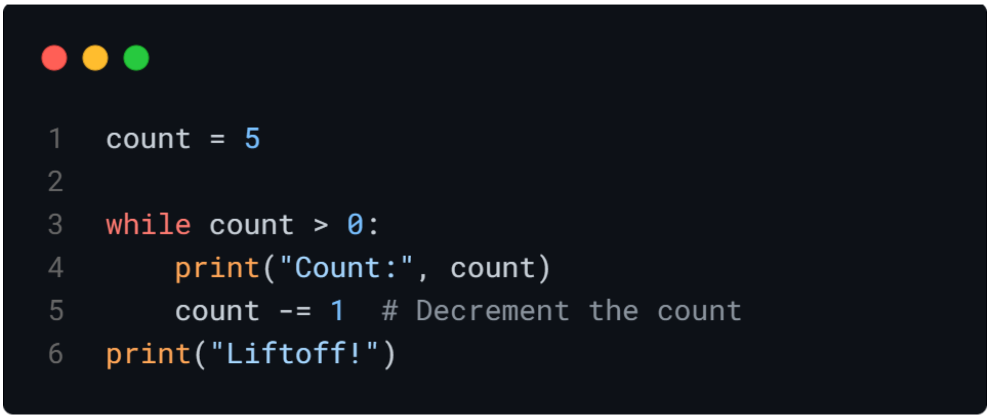
- **while loop**

  The **while** loop continues to execute as long as the given condition is **True**.



```python
while condition:
    # Code block
```

Example: Counting down



```python
count = 5

while count > 0:
    print("Count:", count)
    count -= 1  # Decrement the count
print("Liftoff!")
```

Explanation:

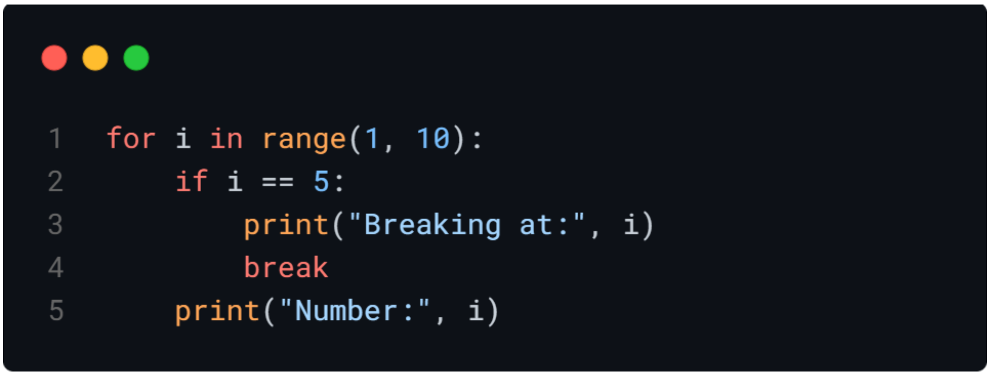
- The loop checks if count > 0. If true, it executes the block.
- After each iteration, count is decremented by 1.
- The loop stops when count reaches 0.

- **Loop control statements**

  Loop control statements modify the flow of the loop.

I.Break

Stop the loop when a condition is met



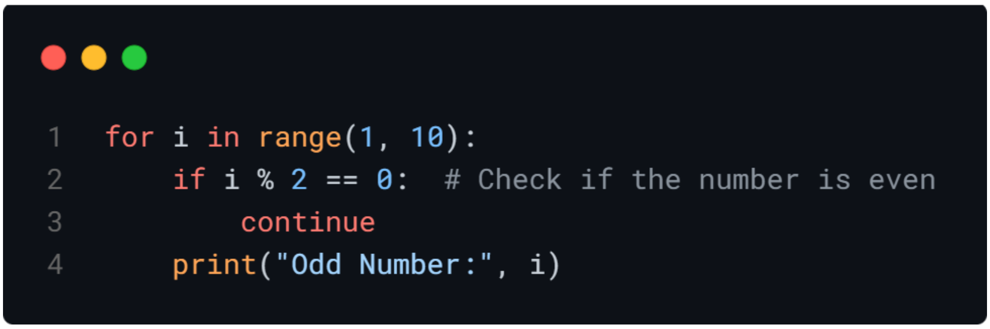
```python
for i in range(1, 10):
    if i == 5:
        print("Breaking at:", i)
        break
    print("Number:", i)
```

Explanation:

- The loop stops entirely when **i** equals 5.

II.Continue

Skip the current iteration and move to the next.



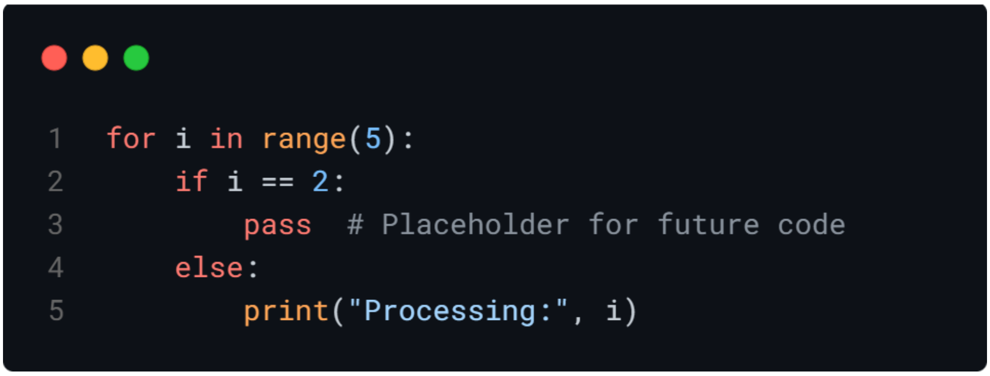
```python
for i in range(1, 10):
    if i % 2 == 0:  # Check if the number is even
        continue
    print("Odd Number:", i)
```

Explanation:

- When **i** is even, the **continue** statement skips printing the number and moves to the next iteration.

III.Pass

Do nothing (placeholder).



```python
for i in range(5):
    if i == 2:
        pass  # Placeholder for future code
    else:
        print("Processing:", i)
```

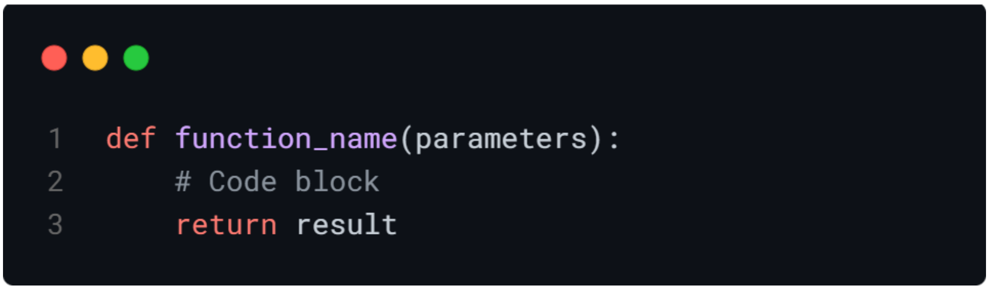
Explanation:

- When **i == 2**, the loop executes **pass** (does nothing) and continues.

5. Functions

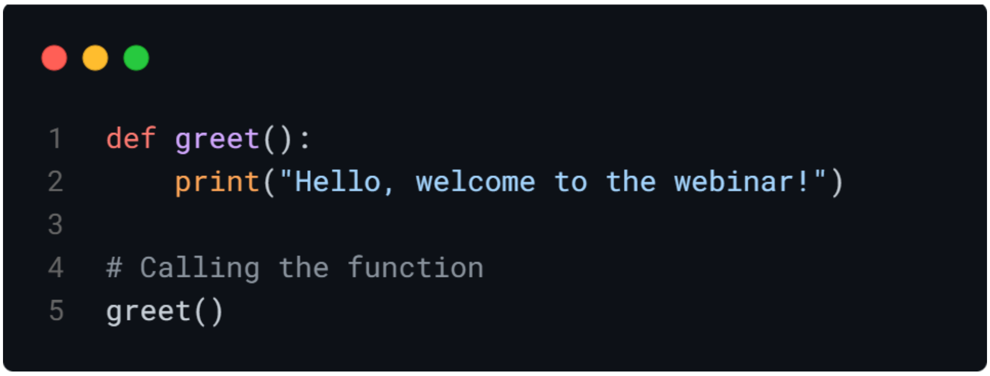
A **function** is a block of reusable code designed to perform a specific task. Functions make your code more organized, modular, and easy to maintain.

- **Function definition and calling.**



```python
def function_name(parameters):
    # Code block
    return result
```

Example: A Simple Greeting Function



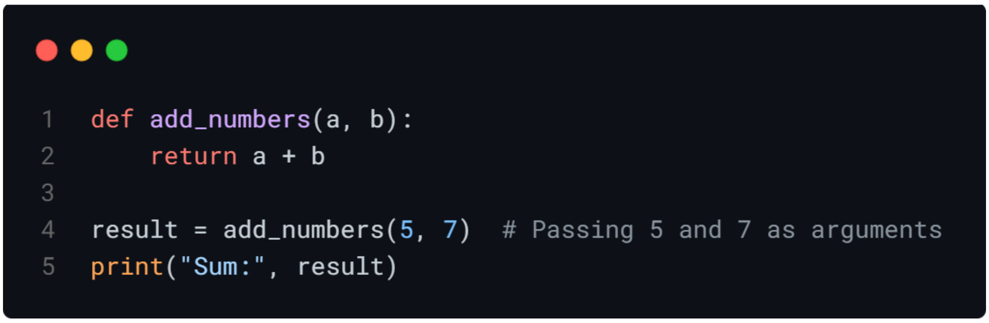
```python
def greet():
    print("Hello, welcome to the webinar!")

# Calling the function
greet()
```

Explanation:

- **def** is used to define the function **greet()**.
- Calling **greet()** runs the code inside it, printing the greeting message.

- **Parameters and return values.**
  - **Parameters** are inputs to a function.
  - **Return values** send results back to the caller.

Example: Function with Parameters and Return Value



```python
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 7)  # Passing 5 and 7 as arguments
print("Sum:", result)
```

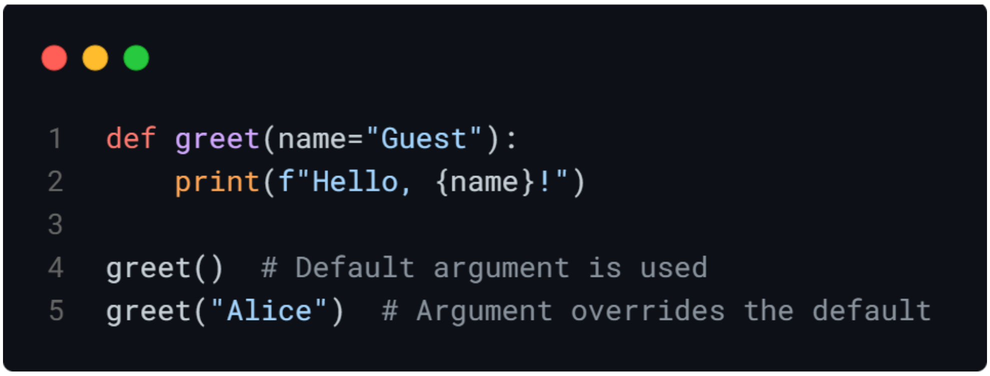
Explanation:

- **a** and **b** are parameters.
- **return** sends the result of **a + b** back to the caller.
- You can use the result (**12** in this case) elsewhere in the program.

- **Default arguments**

Default arguments provide default values to parameters if no argument is passed during the function call.

Example: Greeting with Default Name



```python
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()  # Default argument is used
greet("Alice")  # Argument overrides the default
```

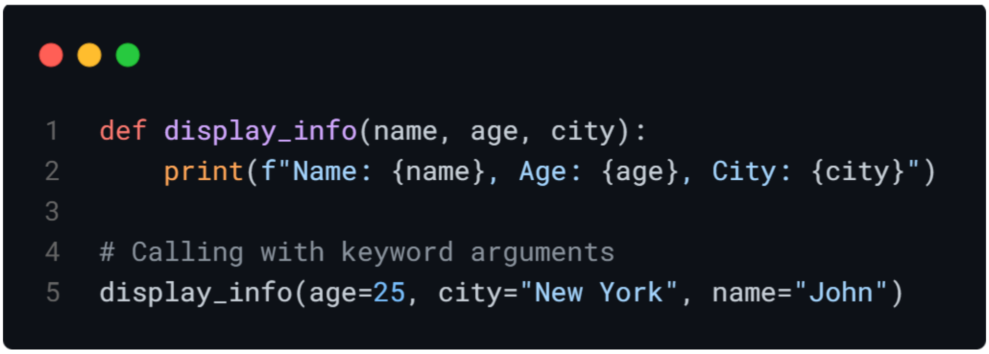
Explanation:

- If no argument is passed, the **name** defaults to "**Guest**".
- If an argument like "**Alice**" is passed, it replaces the default value.

- **Keyword arguments**

  Keyword arguments allow you to specify parameter names during the function call, making the code clearer and allowing arguments to be passed in any order.

  Example: Keyword Arguments



```python
def display_info(name, age, city):
    print(f"Name: {name}, Age: {age}, City: {city}")

# Calling with keyword arguments
display_info(age=25, city="New York", name="John")
```

Explanation:

- By specifying the parameter names (**age=25**), you can pass arguments in any order.

Task:

Write a program to:

- Accept a username and password.
- Verify if the entered data matches predefined credentials.
- Use loops to allow a maximum of three login attempts.