1. Explain OOPS Concepts in Python

Encapsulation : Encapsulation means binding of data (attributes) and methods (behaviors) that operate on the data into a single unit, called a class. This helps keep the implementation details hidden from the outside world, allowing for better organization and security.

By encapsulation we can keep data related to two different Department separate.
For example in a company there are two department Sales and Accounts. So here sales data and people are separate from Accounts data and people.

Abstraction : Abstraction means showing only the necessary details to user and hiding the implementation of it.

For example : We use ATM , we just press some buttons and cash comes out , but we dont know what actually is happening behind machine.

Polymorphism : It means existing in various forms . Two important types of Polymorphism are CompileTime(method overloading) and Runtime Polymorphism(method overriding).

Inheritance : Inheritance is a way to create new classes based on existing ones. It allows a new class (called a derived class or subclass) to inherit properties and behaviors from an existing class (called a base class or superclass). This promotes code reuse and helps establish a hierarchy of classes.

2. Explain Access modifiers in Python

Public:

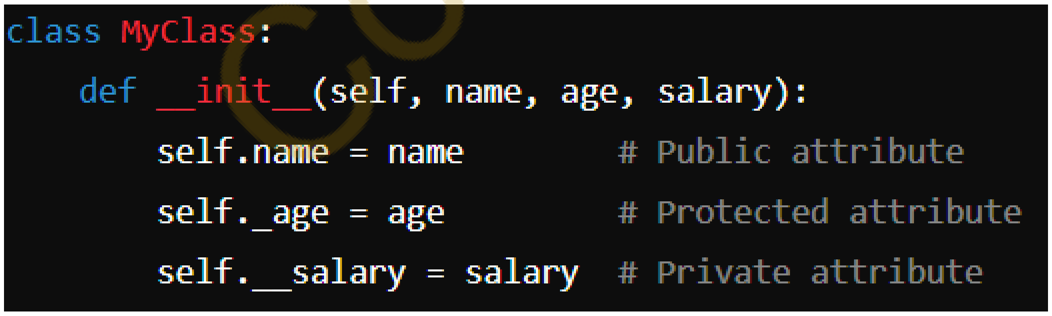
- Syntax: No underscores (_) before the name.
- Example: self.name
- Description: Public members are accessible from anywhere, both inside and outside the class. By default, all class members are public.

Protected:

- Syntax: A single underscore (_) before the name.
- Example: self._name
- Description: Protected members are intended to be accessed only within the class and its subclasses. This is just a convention, and technically, they can still be accessed from outside the class.

Private:

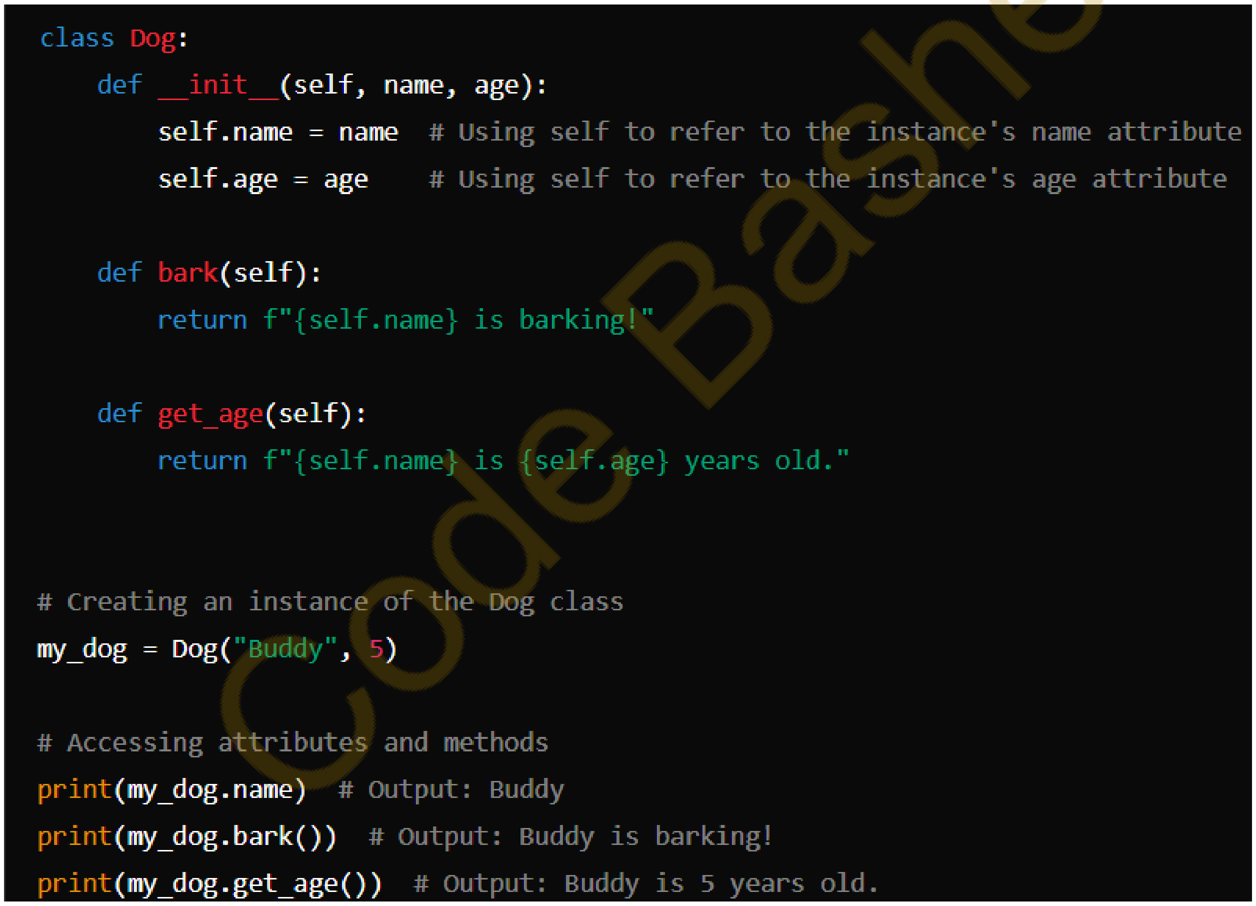
- Syntax: Double underscores (__) before the name.
- Example: self.__name
- Description: Private members are intended to be accessed only within the class where they are defined. Python performs name mangling to make it harder to access private members from outside the class. They can still be accessed using a specific name-mangled syntax.



```python
class MyClass:
    def __init__(self, name, age, salary):
        self.name = name        # Public attribute
        self._age = age         # Protected attribute
        self.__salary = salary  # Private attribute
```

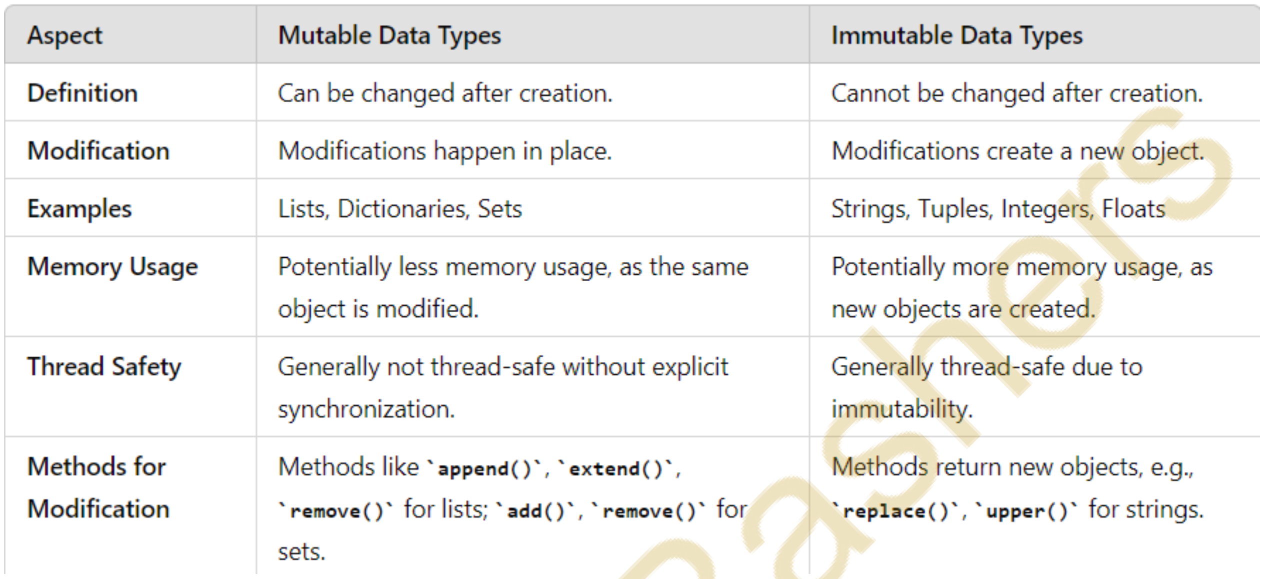
3. Explain Self Keyword In python

The `self` keyword in Python is used in class methods to refer to the instance of the class that is calling the method. It allows you to access the attributes and other methods of the class within its methods. Here’s a simple explanation with an example:

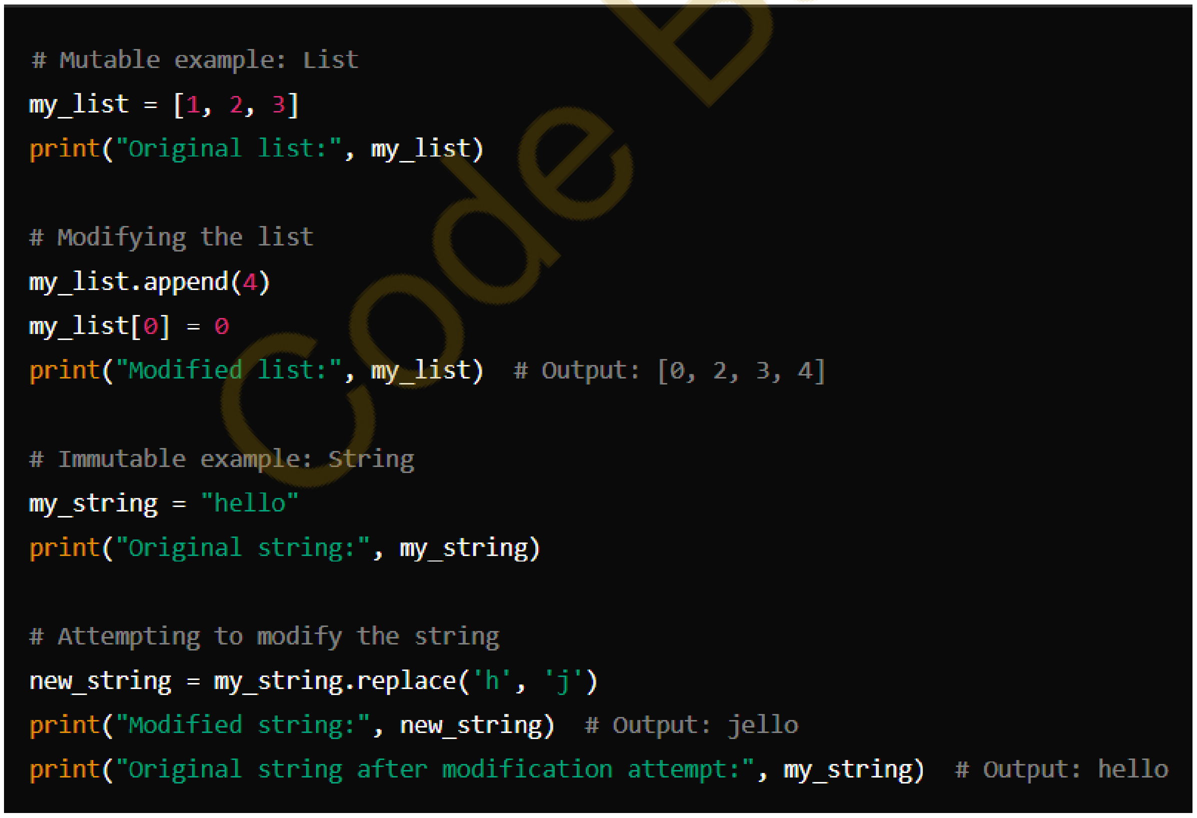


```python
class Dog:
    def __init__(self, name, age):
        self.name = name  # Using self to refer to the instance's name attribute
        self.age = age    # Using self to refer to the instance's age attribute

    def bark(self):
        return f"{self.name} is barking!"

    def get_age(self):
        return f"{self.name} is {self.age} years old."


# Creating an instance of the Dog class
my_dog = Dog("Buddy", 5)

# Accessing attributes and methods
print(my_dog.name)  # Output: Buddy
print(my_dog.bark())  # Output: Buddy is barking!
print(my_dog.get_age())  # Output: Buddy is 5 years old.
```

4. Difference between a Mutable data type and an Immutable data type?



Aspect	Mutable Data Types	Immutable Data Types
Definition	Can be changed after creation.	Cannot be changed after creation.
Modification	Modifications happen in place.	Modifications create a new object.
Examples	Lists, Dictionaries, Sets	Strings, Tuples, Integers, Floats
Memory Usage	Potentially less memory usage, as the same object is modified.	Potentially more memory usage, as new objects are created.
Thread Safety	Generally not thread-safe without explicit synchronization.	Generally thread-safe due to immutability.
Methods for Modification	Methods like `` `append()` ``, `` `extend()` ``, `` `remove()` `` for lists; `` `add()` ``, `` `remove()` `` for sets.	Methods return new objects, e.g., `` `replace()` ``, `` `upper()` `` for strings.



```python
# Mutable example: List
my_list = [1, 2, 3]
print("Original list:", my_list)

# Modifying the list
my_list.append(4)
my_list[0] = 0
print("Modified list:", my_list)  # Output: [0, 2, 3, 4]

# Immutable example: String
my_string = "hello"
print("Original string:", my_string)

# Attempting to modify the string
new_string = my_string.replace('h', 'j')
print("Modified string:", new_string)  # Output: jello
print("Original string after modification attempt:", my_string)  # Output: hello
```

5. What is the difference between a Set and Dictionary?

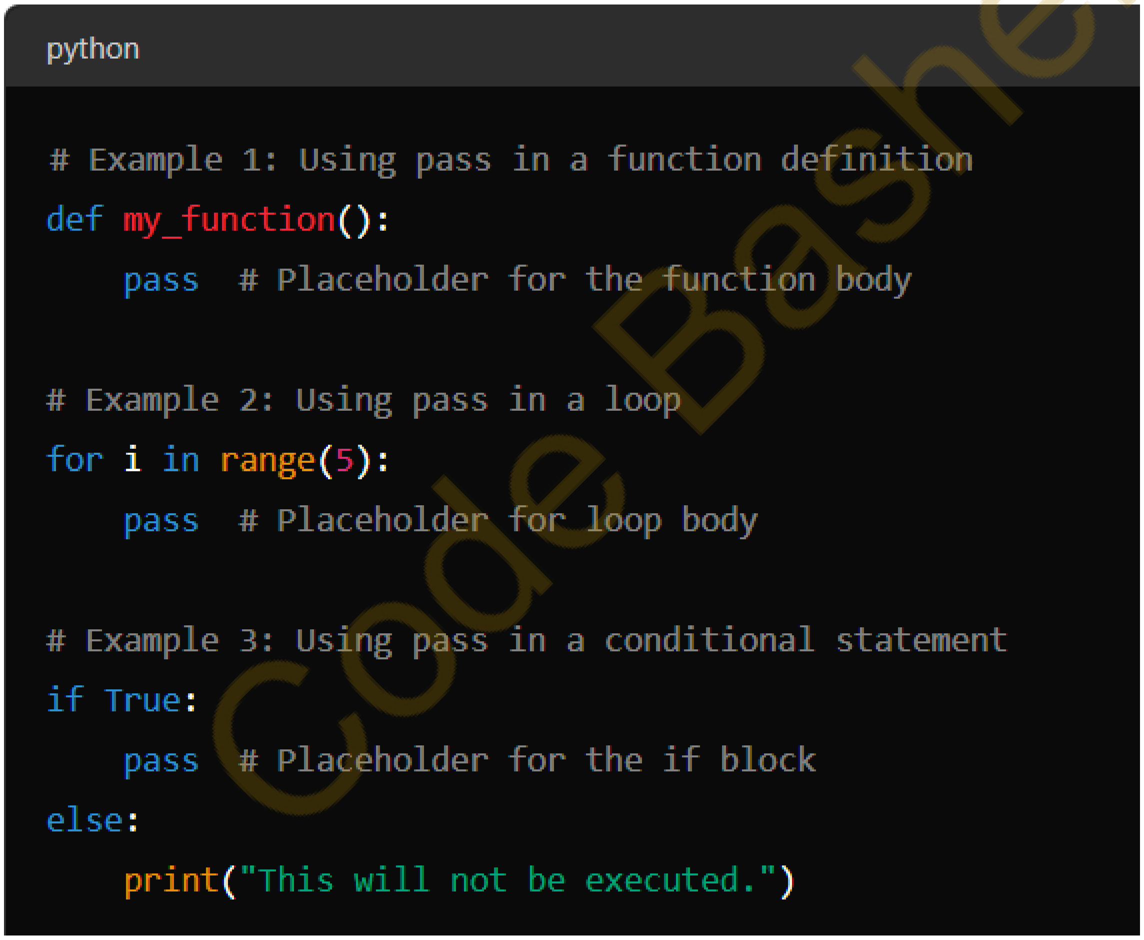
Aspect	Set	Dictionary
Definition	An unordered collection of unique elements.	An unordered collection of key-value pairs.
Syntax	`{1, 2, 3}` or `set([1, 2, 3])`	`{'a': 1, 'b': 2, 'c': 3}`
Access	Elements accessed directly, but not by index.	Values accessed by keys, e.g., `dict['key']`.
Mutability	Mutable (can add or remove elements).	Mutable (can add, remove, and change key-value pairs).
Duplicates	Does not allow duplicate elements.	Keys must be unique, values can be duplicated.
Methods	`add()`, `remove()`, `union()`, `intersection()`, `difference()`, etc.	`get()`, `keys()`, `values()`, `items()`, `update()`, etc.
Use Case	When you need a collection of unique items.	When you need to map unique keys to values.

6. Differentiate between List and Tuple?

Aspect	List	Tuple
Definition	Ordered collection of elements.	Ordered collection of elements.
Syntax	`` `[1, 2, 3]` `` or `` `list((1, 2, 3))` ``	`` `(1, 2, 3)` `` or `` `tuple([1, 2, 3])` ``
Mutability	Mutable (can add, remove, and change elements).	Immutable (cannot be modified after creation).
Access	Elements accessed by index, e.g., `` `my_list[0]` ``.	Elements accessed by index, e.g., `` `my_tuple[0]` ``.
Modification	Add: `` `append()` ``, `` `extend()` ``, `` `insert()` ``. Remove: `` `pop()` ``, `` `remove()` ``, `` `del` ``. Change: Assign to index, e.g., `` `my_list[0] = 5` ``.	Cannot be modified after creation.
Memory Usage	Potentially more memory usage due to mutability.	Potentially less memory usage due to immutability.
Use Case	When the collection of elements may change over time.	When the collection of elements should remain constant.
Examples	`` `my_list = [1, 2, 3]` ``	`` `my_tuple = (1, 2, 3)` ``

7. What is a pass in Python?

`pass` is a null statement, meaning it does nothing when executed. It's typically used as a placeholder in situations where a statement is syntactically required but no action is needed. Here's a simple example:

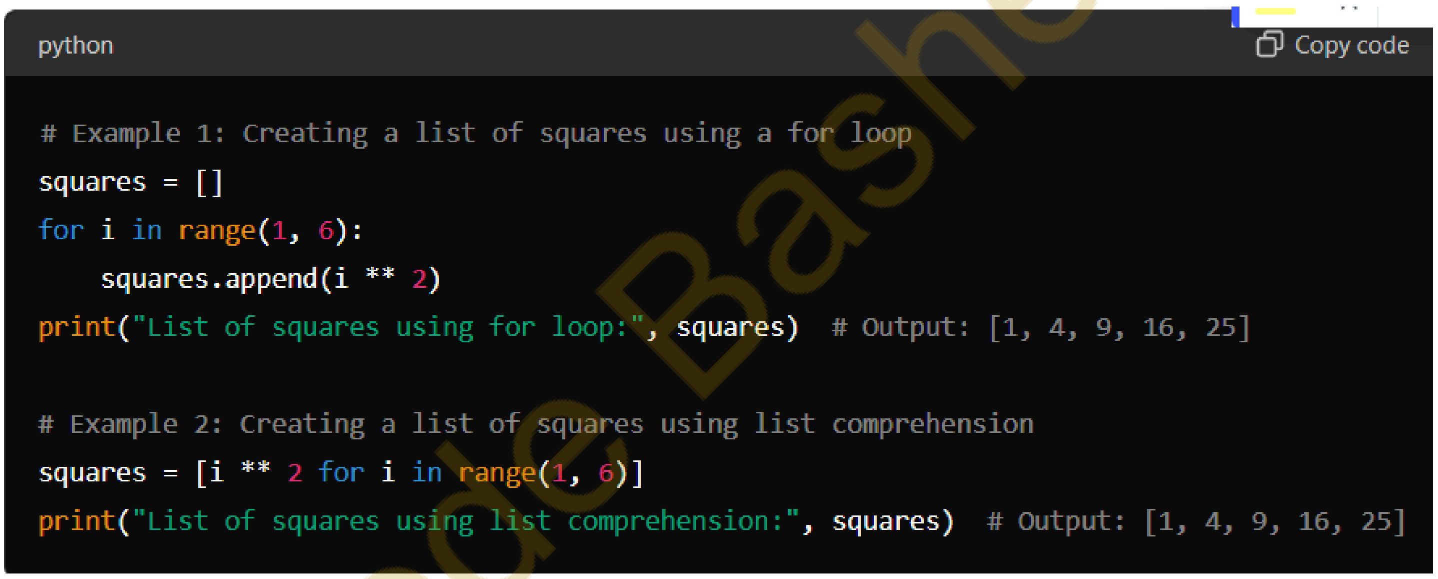


python

```python
# Example 1: Using pass in a function definition
def my_function():
    pass  # Placeholder for the function body

# Example 2: Using pass in a loop
for i in range(5):
    pass  # Placeholder for loop body

# Example 3: Using pass in a conditional statement
if True:
    pass  # Placeholder for the if block
else:
    print("This will not be executed.")
```

8. What is List Comprehension?

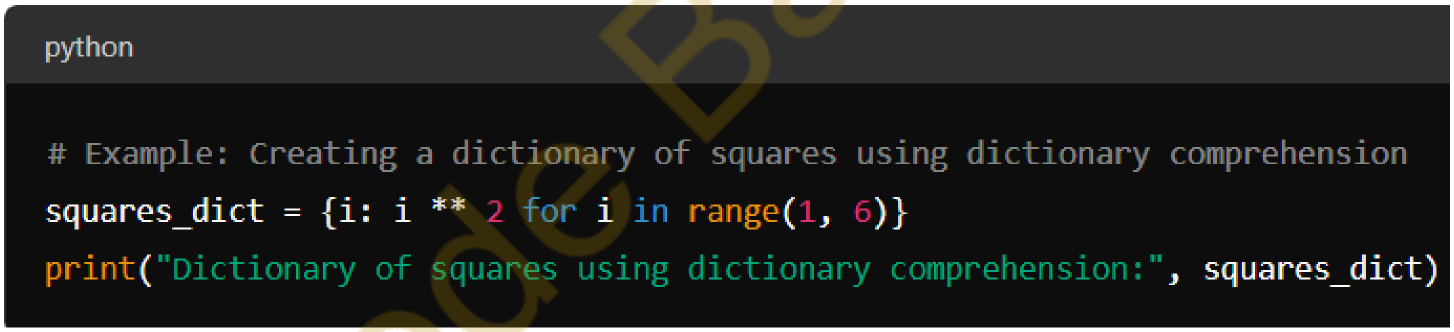
List comprehension is a concise way of creating lists in Python. It allows you to generate a new list by applying an expression to each item in an existing iterable (like a list, tuple, or range) and optionally filtering the items based on a condition. Here's a simple explanation with an example:



```python
# Example 1: Creating a list of squares using a for loop
squares = []
for i in range(1, 6):
    squares.append(i ** 2)
print("List of squares using for loop:", squares)  # Output: [1, 4, 9, 16, 25]

# Example 2: Creating a list of squares using list comprehension
squares = [i ** 2 for i in range(1, 6)]
print("List of squares using list comprehension:", squares)  # Output: [1, 4, 9, 16, 25]
```

9. What is Dictionary Comprehension? Give an Example

Dictionary comprehension is similar to list comprehension, but instead of creating lists, it allows you to create dictionaries in a concise and efficient way. You can generate a new dictionary by specifying key-value pairs based on an expression and optionally filtering them based on a condition. Here's a simple explanation with an example:



```python
# Example: Creating a dictionary of squares using dictionary comprehension
squares_dict = {i: i ** 2 for i in range(1, 6)}
print("Dictionary of squares using dictionary comprehension:", squares_dict)
```

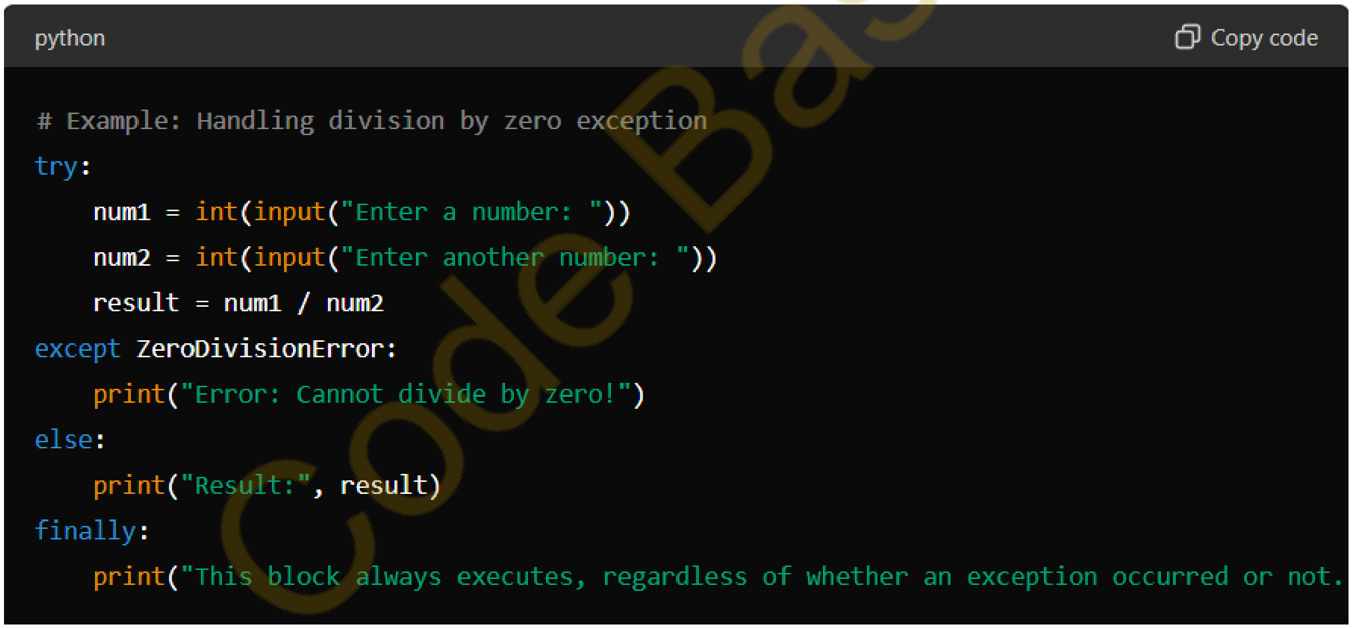
10. What is a lambda function?

A lambda function in Python is a small anonymous function defined using the `lambda` keyword. It can take any number of arguments but can only have one expression. Lambda functions are often used when you need a short function for a short period of time. Here's a simple explanation with an example:

```python
# Example: Creating a lambda function to calculate the square of a number
square = lambda x: x ** 2

# Using the lambda function
print("Square of 5:", square(5))  # Output: 25
```

11. How is Exceptional handling done in Python?

Exception handling in Python allows you to gracefully handle errors or exceptional situations that may occur during program execution. It involves using `try`, `except`, `else`, and `finally` blocks to catch and handle exceptions.



```python
# Example: Handling division by zero exception
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
else:
    print("Result:", result)
finally:
    print("This block always executes, regardless of whether an exception occurred or not.
```

- We use a `try` block to execute the code that may raise an exception.
- If an exception occurs (e.g., division by zero), the control jumps to the corresponding `except` block.
- In the `except` block, we handle the exception (e.g., print an error message).
- If no exception occurs, the code inside the `else` block is executed.
- Finally, the `finally` block is executed regardless of whether an exception occurred or not. It's often used for cleanup operations.

12. What is the difference between a shallow copy and a deep copy?

Aspect	Shallow Copy	Deep Copy
Definition	Copies the top-level structure and references to nested objects.	Recursively copies the contents, including nested objects.
Syntax	`copy.copy()`	`copy.deepcopy()`
Example Code	```python	```python

```python
import copy

# Shallow copy example
original_list = [1, [2, 3], 4]
shallow_copied_list = copy.copy(original_list)

original_list[1][0] = 'x'

print("Original List (Shallow Copy):", original_list)  # Output: [1, ['x', 3], 4]
print("Shallow Copied List:", shallow_copied_list)  # Output: [1, ['x', 3], 4]

# Deep copy example
deep_copied_list = copy.deepcopy(original_list)

original_list[1][0] = 'y'

print("Original List (Deep Copy):", original_list)  # Output: [1, ['y', 3], 4]
print("Deep Copied List:", deep_copied_list)  # Output: [1, ['x', 3], 4]
```

13. What are *args and **Kwargs

Aspect	*args	**kwargs
Usage	Used to pass a variable number of positional arguments to a function.	Used to pass a variable number of keyword arguments (or named arguments) to a function.
Syntax	`*args` unpacks the arguments into a tuple within the function.	`**kwargs` unpacks the arguments into a dictionary within the function.
Example	```python	```python

python

```python
# *args example
def my_function(*args):
    for arg in args:
        print(arg)

my_function(1, 2, 3, 4)  # Output: 1 2 3 4

# **kwargs example
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(key, ":", value)

my_function(a=1, b=2, c=3)  # Output: a : 1, b : 2, c : 3
```

14. Explain Datatypes in python

List : Represents ordered collections of items enclosed within square brackets `[]`. Items can be of different data types and mutable (modifiable). Example: `[1, 'hello', 3.14]`.

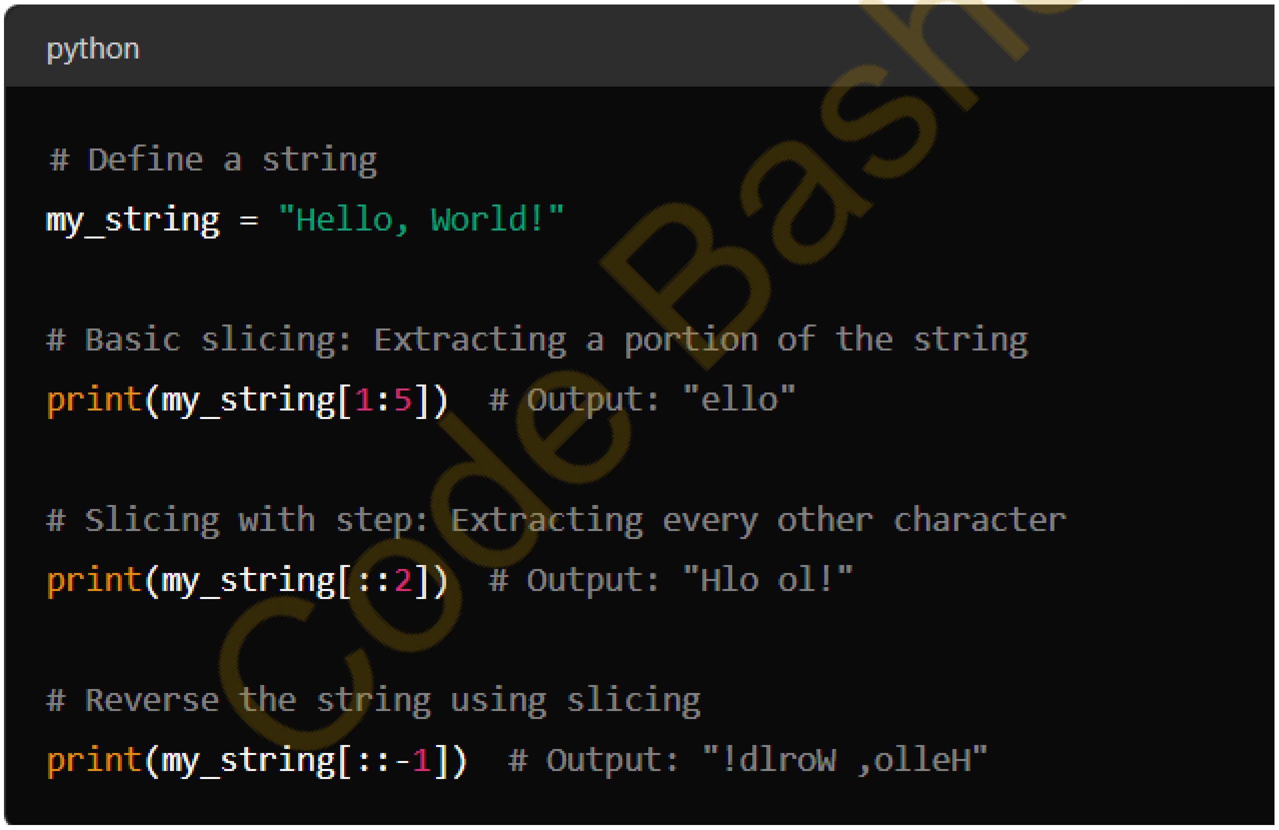
Tuple: Represents ordered collections of items enclosed within parentheses `()`. Items can be of different data types and immutable (cannot be modified after creation). Example: `(1, 'world', 3.14)`.

Dictionary : Represents unordered collections of key-value pairs enclosed within curly braces `{}`. Keys are unique and immutable, values can be of any data type. Example: `{'name': 'John', 'age': 25}`.

Set : Represents unordered collections of unique elements enclosed within curly braces `{}`. Sets do not allow duplicate elements. Example: `{1, 2, 3}`, `{ 'apple', 'banana', 'orange'}`.

15. What is slicing in Python?

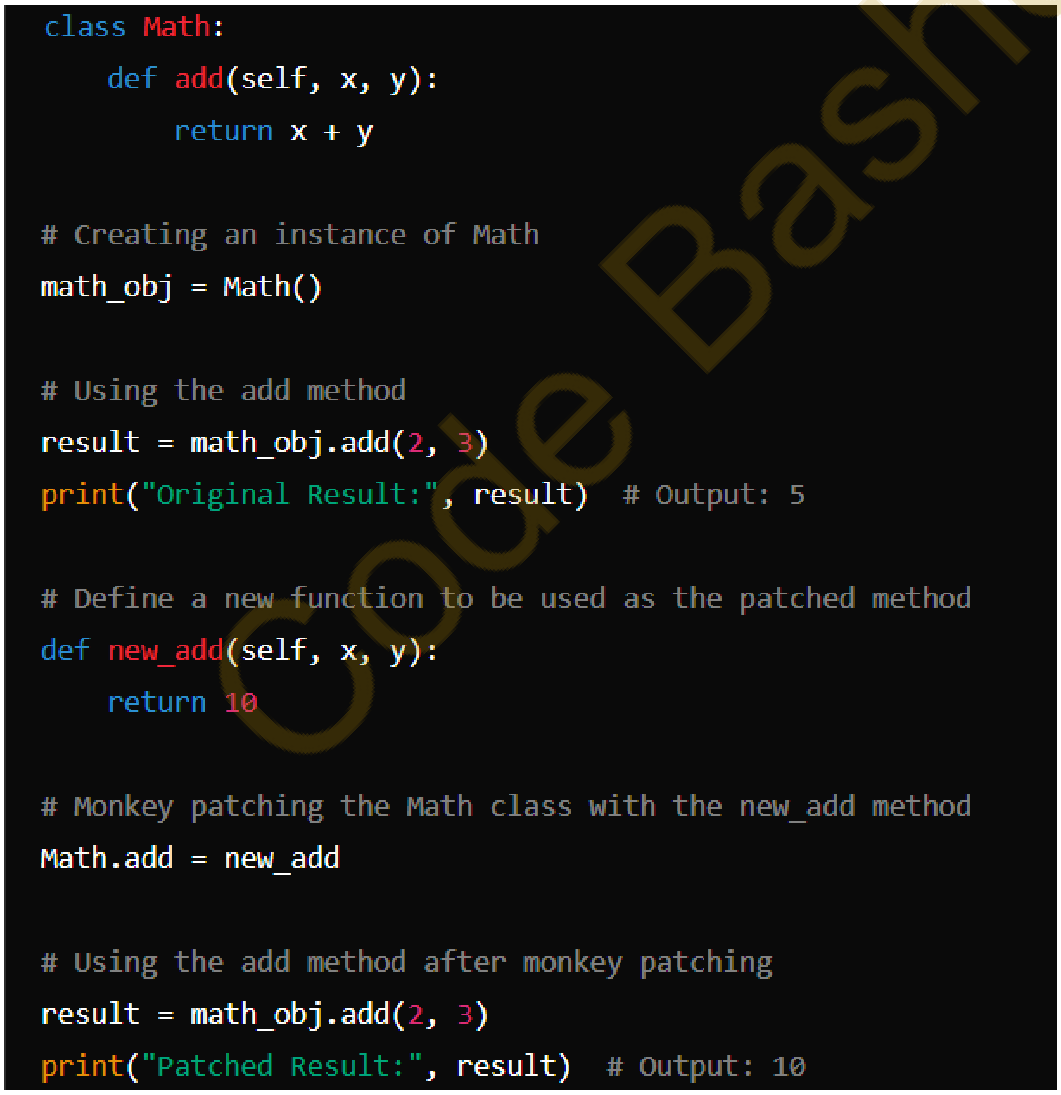
Slicing in Python allows you to extract a portion of a sequence (like a string, list, or tuple) by specifying a start index, stop index, and an optional step size. It's a powerful and efficient way to work with sequences. Here's a simple explanation with an example:



```python
# Define a string
my_string = "Hello, World!"

# Basic slicing: Extracting a portion of the string
print(my_string[1:5])  # Output: "ello"

# Slicing with step: Extracting every other character
print(my_string[::2])  # Output: "Hlo ol!"

# Reverse the string using slicing
print(my_string[::-1])  # Output: "!dlroW ,olleH"
```

16. What is monkey patching in Python?

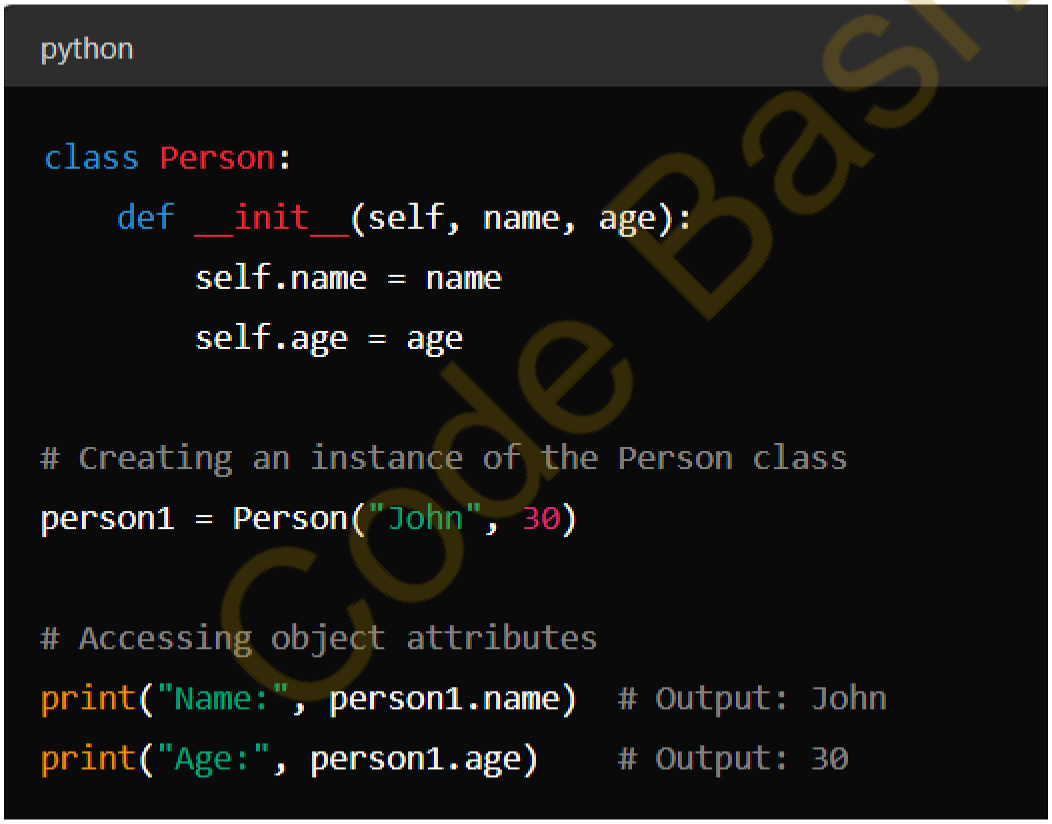
Monkey patching in Python refers to the practice of dynamically modifying or extending the behavior of a class or module at runtime. It allows you to change the behavior of existing code without altering its original source code. Here's a simple explanation with an example:



```python
class Math:
    def add(self, x, y):
        return x + y

# Creating an instance of Math
math_obj = Math()

# Using the add method
result = math_obj.add(2, 3)
print("Original Result:", result)  # Output: 5

# Define a new function to be used as the patched method
def new_add(self, x, y):
    return 10

# Monkey patching the Math class with the new_add method
Math.add = new_add

# Using the add method after monkey patching
result = math_obj.add(2, 3)
print("Patched Result:", result)  # Output: 10
```

17. What is __init__() in Python?

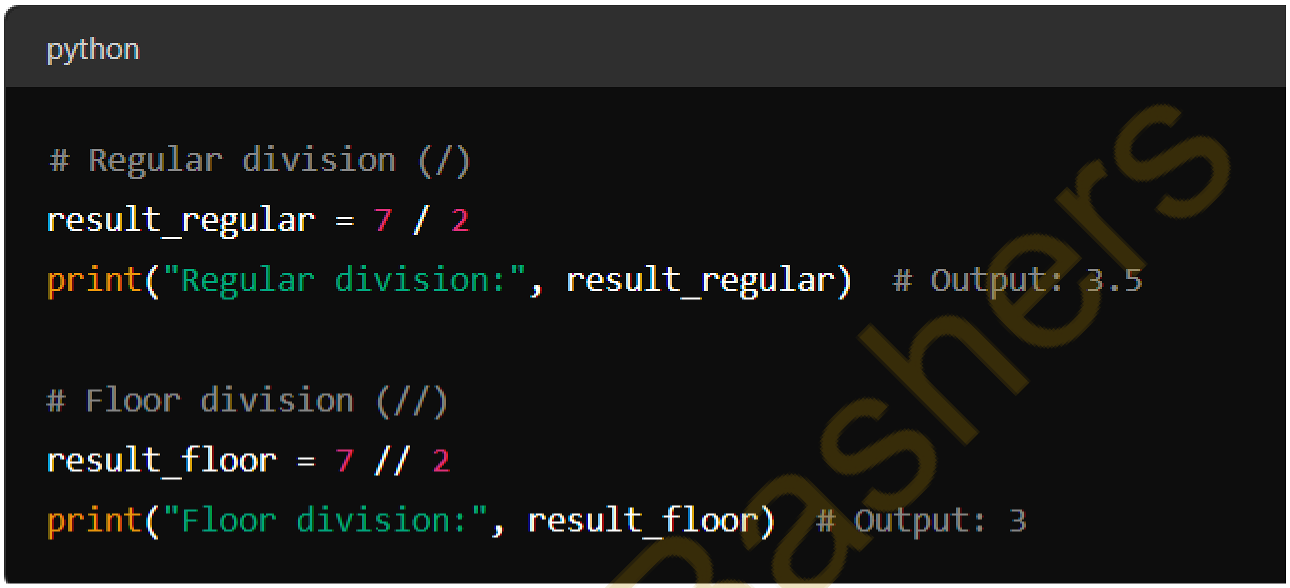
In Python, `__init__()` is a special method (also known as a constructor) that is automatically called when a new instance of a class is created. It is used to initialize the object's attributes or perform any setup that needs to be done when the object is created. Here's a simple explanation with an example:



python

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of the Person class
person1 = Person("John", 30)

# Accessing object attributes
print("Name:", person1.name)  # Output: John
print("Age:", person1.age)    # Output: 30
```

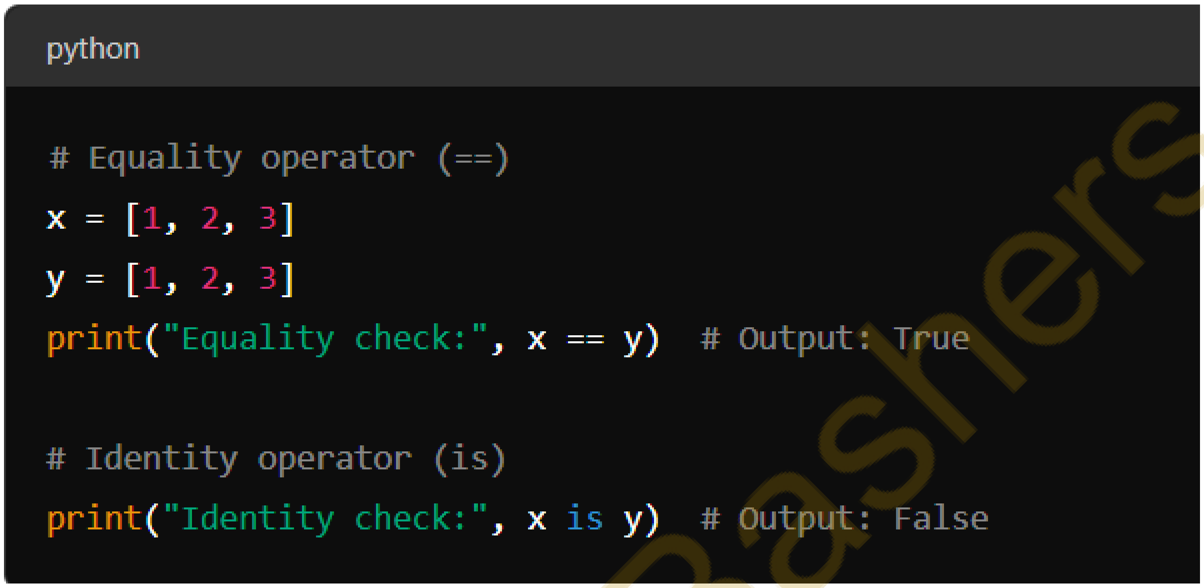
18. What is the difference between / and // in Python?



python

```python
# Regular division (/)
result_regular = 7 / 2
print("Regular division:", result_regular)  # Output: 3.5

# Floor division (//)
result_floor = 7 // 2
print("Floor division:", result_floor)  # Output: 3
```

19. What is the difference between '==' and 'is' in Python?

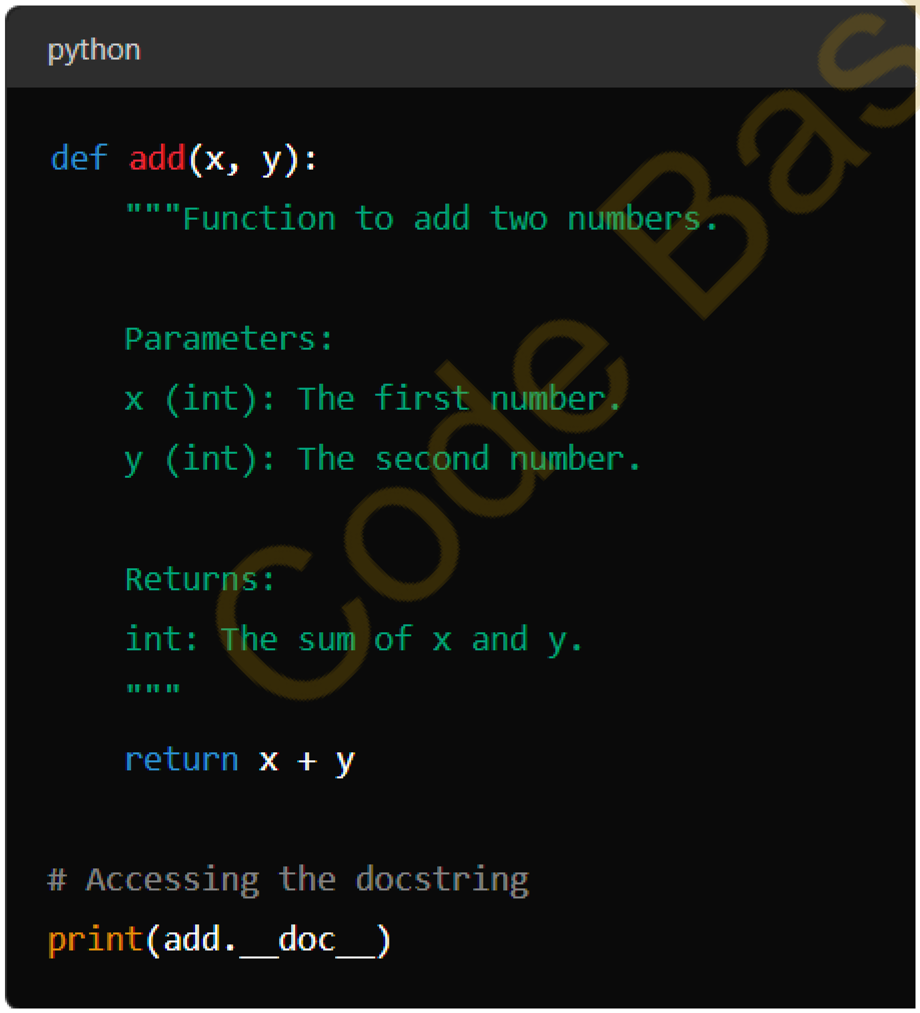


python

```python
# Equality operator (==)
x = [1, 2, 3]
y = [1, 2, 3]
print("Equality check:", x == y)  # Output: True

# Identity operator (is)
print("Identity check:", x is y)  # Output: False
```

20. What is Docstring in Python

docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Its purpose is to provide documentation about the object it belongs to, such as its purpose, usage, parameters, and return values. Docstrings are accessible through the `__doc__` attribute of the object.



```python
def add(x, y):
    """Function to add two numbers.

    Parameters:
    x (int): The first number.
    y (int): The second number.

    Returns:
    int: The sum of x and y.
    """
    return x + y

# Accessing the docstring
print(add.__doc__)
```