**Name –** Gathram Kesava Murthy

**Registered mail id-** kesavagathram.murthy@outlook.com

**Course Name –** Master Generative AI: Data Science Course

**Assignment Name –** Functions Assignment

**Submission Date –** 25th October 2024

**Drive Link –**

[**https://colab.research.google.com/drive/1vfdWmVfpuwTW27n1aVVVVfzXwd7Iq2gv?usp=sharing**](https://colab.research.google.com/drive/1vfdWmVfpuwTW27n1aVVVVfzXwd7Iq2gv?usp=sharing)

**Github Link –**

**1. What is the difference between a function and a method in Python**

We have only methods in Java and only functions in C but in Python, we have both functions and methods.

**Functions in Python:**

A function is a collection of lines of code that accomplishes a certain task. Functions have:

* Name
* Parameters
* Return statement

Return statement and parameters are optional. A function can either have them or not.

**Creating a function in Python:**

* We can create a function using the keyword def.

**Syntax:**

**def** function\_name(parameters):

# Statements...

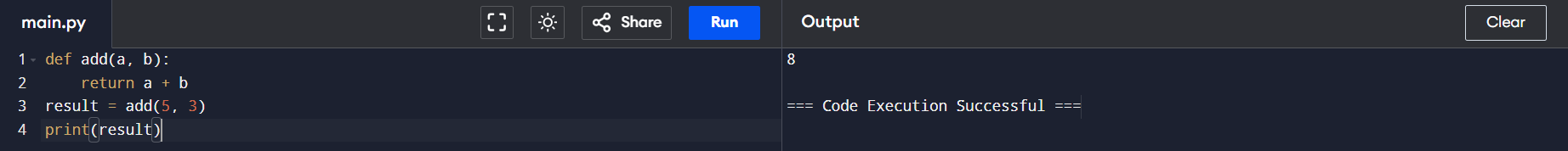
**Calling a function in Python:**

* Without calling, a function will never run. To call a function we use the following syntax

**Syntax:**

function\_name(arguments)

**Example:**

****

**Methods in Python:**

Functions inside a class are called methods. Methods are associated with a class/object.

**Creating a method in Python:**

* We use the same syntax as function but this time, it should be inside a class.

**Syntax:**

**class** ClassName:

**def** method\_name(parameters):

# Statements…

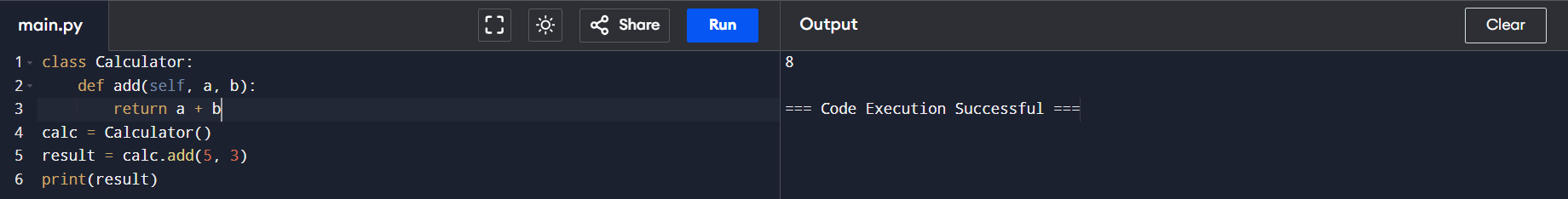
**Calling a Method in python :**

* To use a method, we need to call it. We call the method just like a function but since methods are associated with class/object, we need to use a class/object name and a dot operator to call it.

**Syntax:**

object\_name.method\_name(arguments)

**Example:**



**Functions Vs Methods**

|  |  |
| --- | --- |
| **Functions** | Methods |
| Functions are outside a class | Methods are created inside a class |
| Functions are not linked to anything | Methods are linked with the classes they are created in |
| Functions can be executed just by calling with its nameFunctions can be executed just by calling with its name | To execute methods, we need to use either an object name or class name and a dot operator. |
| Functions can have zero parameters. | Methods should have a default parameter either self or cls to get the object’s or class’s address. |
| Functions can not access or modify class attributes | Methods can access and modify class attributes |
| Functions are independent of classes | Methods are dependent on classes |

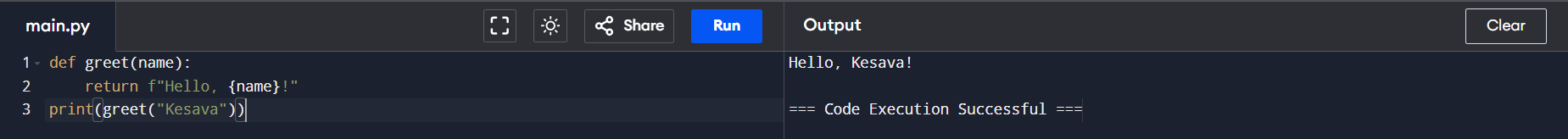
**2. Explain the concept of function arguments and parameters in Python.**

parameters and arguments are essential concepts in functions that allow you to pass data to a function when it’s called. Though sometimes used interchangeably, they actually refer to different things in a function’s lifecycle.

**1. Parameters**

* **Definition**: Parameters are the variables listed inside the parentheses in the function definition. They act as placeholders for the values that the function will use when it's executed.
* **Purpose**: They define the input data that the function expects. Parameters allow the function to perform operations on different values each time it is called.
* **Scope**: Parameters exist only within the function’s body, and they’re local to that function.

**Example of Parameters:**



**2. Arguments**

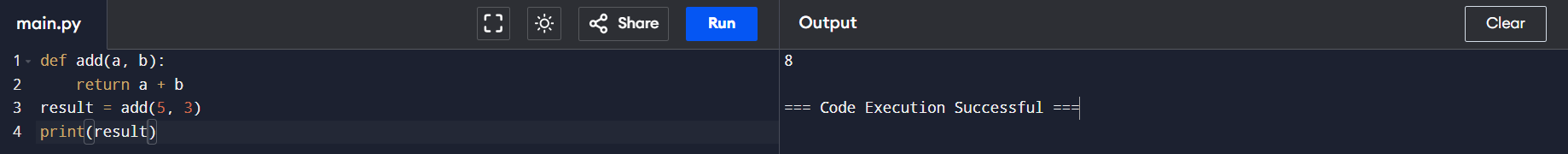
* **Definition**: Arguments are the actual values that you pass into the function when you call it. They correspond to the parameters defined in the function and provide the function with the data it needs to perform its task.
* **Purpose**: Arguments supply the real data that the function will work on. When a function is called, arguments are assigned to the function’s parameters in the order they are listed.
* **Types**: Arguments can be passed in various ways, such as positional, keyword, default, and variable-length arguments (explained below).
* In the above example greet(Kesava) is an argument where as name is parameter.

**Types of Arguments in Python**

Python provides flexibility in how arguments are passed to functions. Here’s an overview of different types:

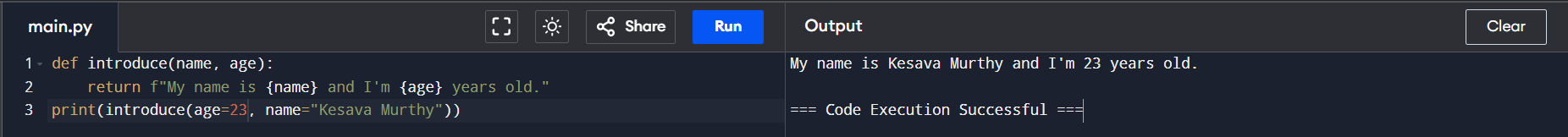
1. **Positional Arguments**

* **Definition**: Positional arguments are passed in the same order as the parameters are defined. Each argument is matched to its corresponding parameter by position.
* **Example :**

****

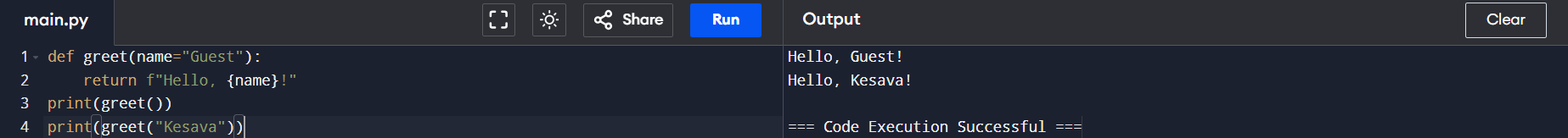
1. **Keyword Arguments**

* **Definition**: Keyword arguments allow you to specify the parameter name with its value when calling the function. This makes the code more readable and allows arguments to be passed out of order.
* **Example**:



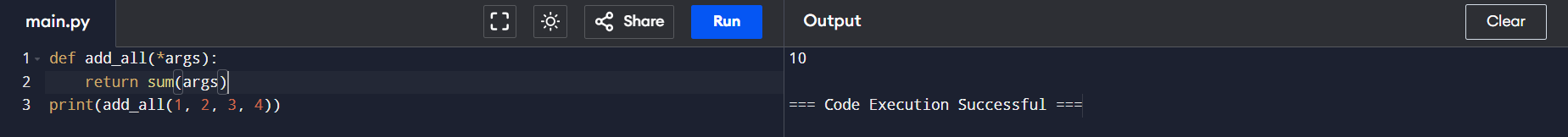
1. **Default Arguments**

* **Definition**: Default arguments are parameters that assume a default value if no argument is provided for them in the function call. They are defined by assigning a value to the parameter in the function definition.
* **Example:**

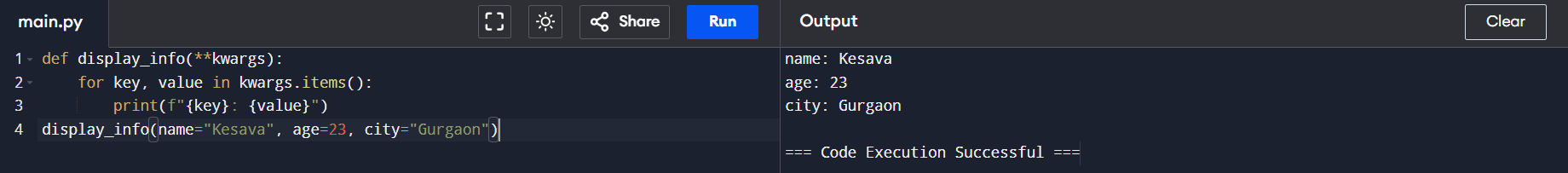


1. **Variable-Length Arguments**

* **Definition**: Variable-length arguments allow you to pass a variable number of arguments to a function. There are two types:
  + **\*args**: Allows a function to accept any number of positional arguments as a tuple.
  + **\*\*kwargs**: Allows a function to accept any number of keyword arguments as a dictionary.
* **Example with \*args**:



* **Example with \*\*kwargs**:



**Parameter vs. Argument Summary**

| **Aspect** | **Parameters** | **Arguments** |
| --- | --- | --- |
| **Definition** | Variables in function definition | Values passed to function at call time |
| **Purpose** | Define expected inputs | Provide actual data to function |
| **Scope** | Local to the function | Used during function call |

**3. What are the different ways to define and call a function in Python**

**1. Standard Function Definition:**

* The most common way to define a function is by using the def keyword, followed by the function name and any parameters in parentheses.

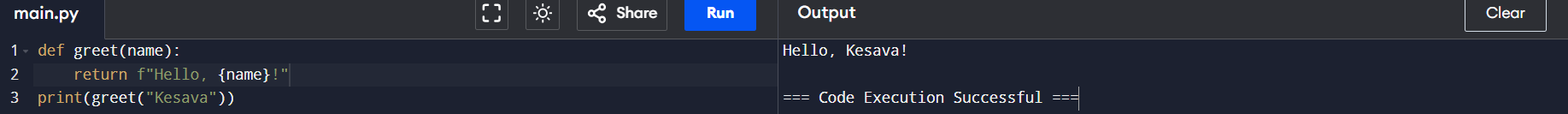
**Syntax:**

def function\_name(parameters):

# Code block

return result

**Example:**



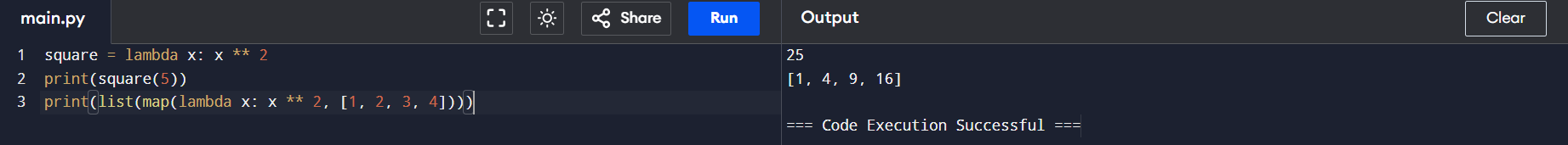
**2. Lambda (Anonymous) Functions**

* Lambda functions are small, anonymous functions defined using the lambda keyword. They are typically used for short, simple operations and are useful when you need a function for a quick, one-time task.

**Syntax:**

lambda parameters: expression

**Example:**

****

**3. Recursive Functions**

A recursive function is a function that calls itself. Recursive functions are helpful for solving problems that can be broken down into smaller subproblems, such as factorial calculations or traversing tree-like data structures.

**Syntax:**

def recursive\_function(parameters):

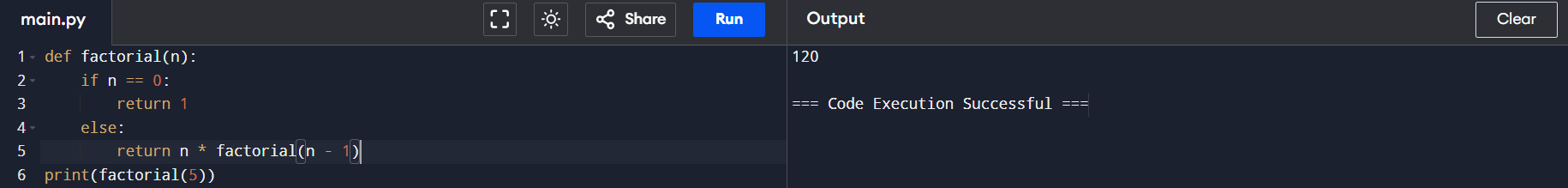
# Base case to prevent infinite recursion

if condition: return result

else: # Recursive call

return recursive\_function(modified\_parameters)

**Example:**

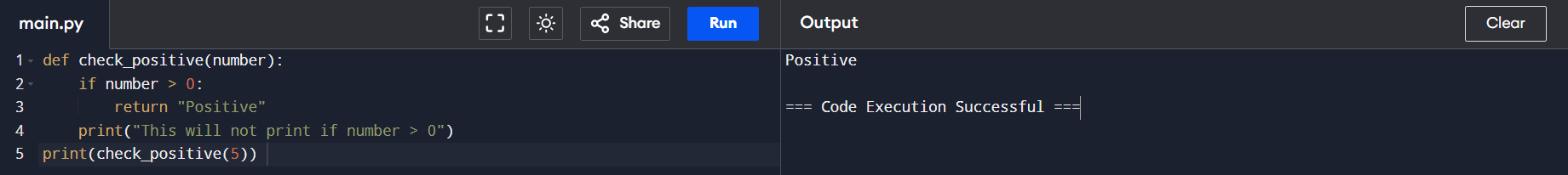
****

**4. What is the purpose of the `return` statement in a Python function?**

The return statement serves as a way for a function to send back a value to the caller and terminate the function’s execution. It plays a crucial role in controlling the function’s output and determining the behaviour of a function’s interaction with other parts of a program.

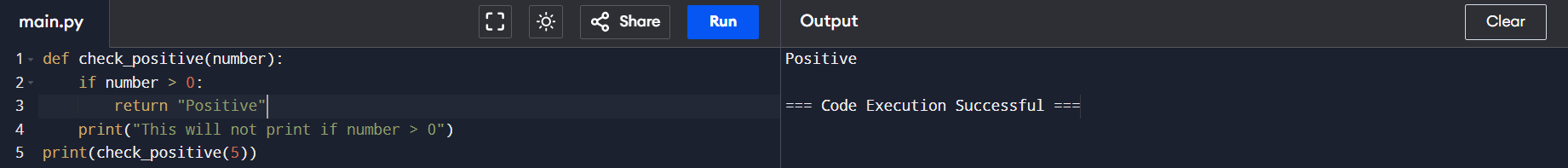
**1. Returning a Value to the Caller:**

The primary purpose of the return statement is to allow a function to output a result back to the caller. When a function completes, the return statement specifies the value that should be provided as the output of that function.



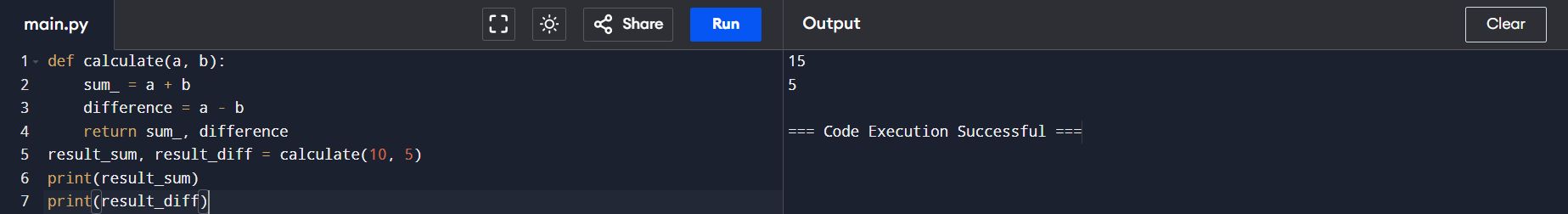
**2. Ending Function Execution**

When a return statement is encountered, it immediately terminates the function’s execution. Any code within the function that appears after return will not be executed.



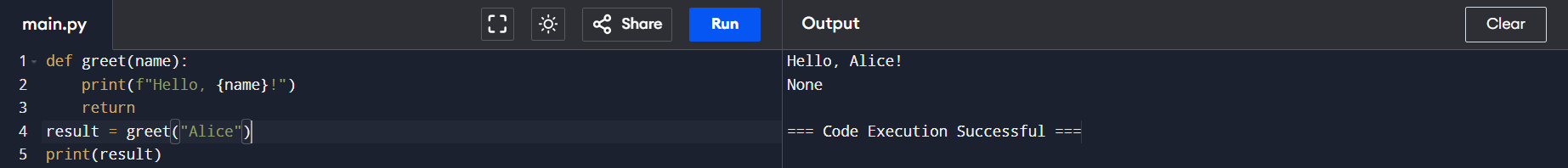
**3. Returning Multiple Values**

Python allows a function to return multiple values as a tuple. This can be useful when a function needs to output several related values.



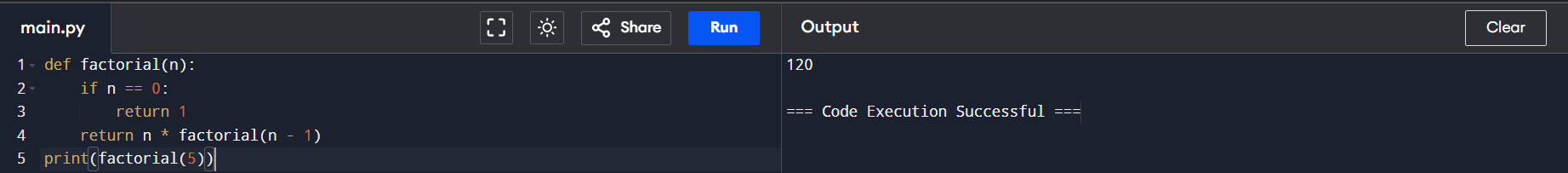
**4. Returning None**

If no return statement is present in a function, or if return is used without specifying a value, the function returns None by default. This is useful when a function performs an action but doesn’t need to produce a result.



**5. Using return in Recursive Functions**

In recursive functions (functions that call themselves), return is used to return the final result after a series of recursive calls.

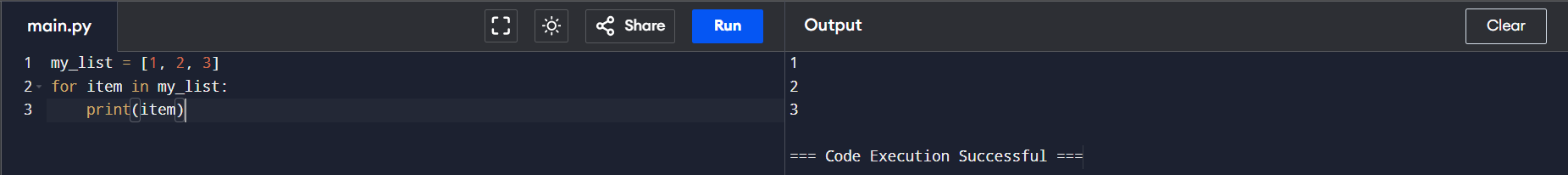


**5. What are iterators in Python and how do they differ from iterables?**

**1. Iterables:**

An iterable is any Python object that can return an iterator, allowing it to be looped over. In other words, an iterable is an object that has an \_\_iter\_\_() method, which returns an iterator, or it has a \_\_getitem\_\_() method that allows elements to be accessed by index (e.g., lists, tuples, strings).

* **Examples of Iterables**: Lists, tuples, strings, dictionaries, sets, and any object that implements the \_\_iter\_\_() or \_\_getitem\_\_() methods.
* **Behavior**: Iterables can be passed into a for loop to be iterated over. When iterated, they do not change state, and they can be iterated over multiple times.

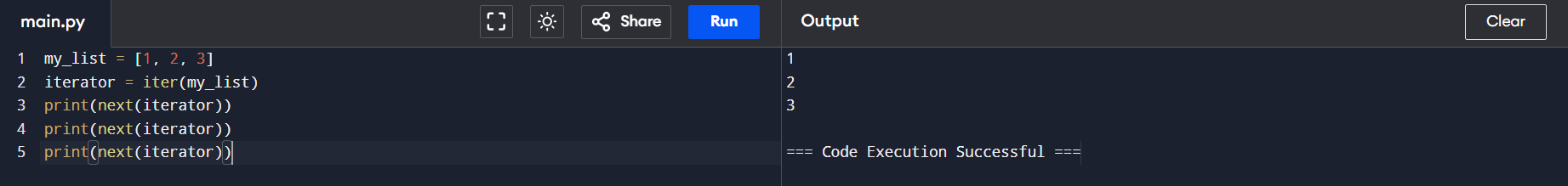


**2. Iterators:**

An *iterator* is an object that represents a stream of data; it produces items one at a time on demand rather than all at once. An iterator object in Python implements two special methods:

* **\_\_iter\_\_()**: Returns the iterator object itself. This allows it to be used in for loops and other places where an iterable is needed.
* **\_\_next\_\_()**: Returns the next item in the sequence. When there are no more items, it raises a StopIteration exception to signal the end of the iteration.

Iterators are often created from iterables using the iter() function. Once created, an iterator maintains an internal state, keeping track of where it is in the sequence. This state allows iterators to generate items one by one, making them suitable for processing large data sets without requiring all elements to be in memory at once.



**Differences Between Iterables and Iterators:**

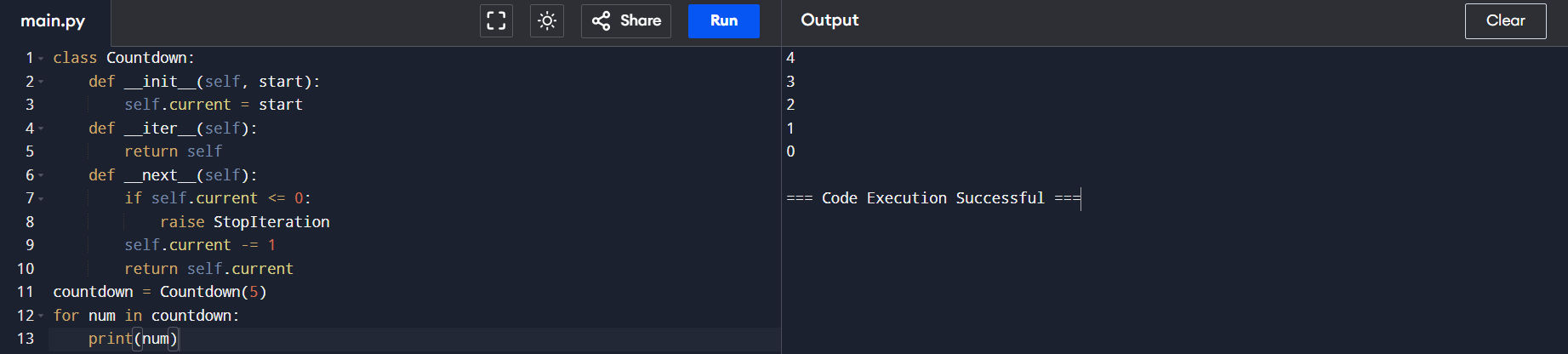
|  |  |
| --- | --- |
| **Iterable** | **Iterator** |
| An object to iterate over is Iterable. | Iterators are defined as an object that counts iteration via an internal state variable. |
| Every iterator is iterable. | Not every iterable is an iterator. |
| An object which would generate an iterator when passed to in-built method iter(). | The next() is used for iterating. |
| A List is iterable. | A List is not an iterator. |

* **Use Iterables**:

When you have a collection of items that can be iterated over multiple times (like a list or dictionary).

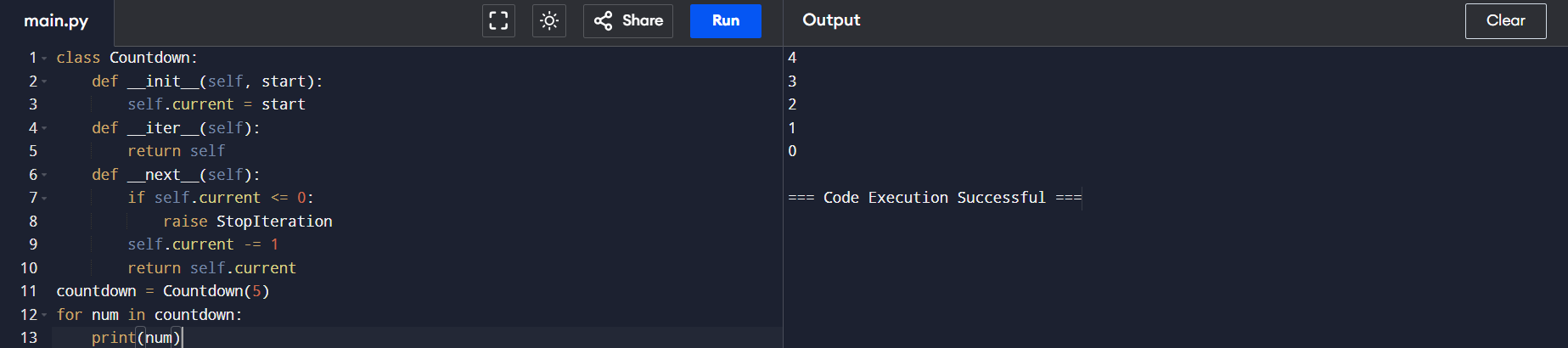
* **Use Iterators**:

When you want to retrieve items one at a time, often in situations where the entire collection would be too large to fit into memory at once. Iterators are particularly useful in lazy evaluation, such as reading large files line-by-line.



**3. Creating Custom Iterators**

You can create custom iterator objects by defining a class with \_\_iter\_\_() and \_\_next\_\_() methods. This can be useful if you want to define custom iteration behavior.



**6. Explain the concept of generators in Python and how they are defined**

**Defining a Generator**

Generators are defined similarly to functions, but instead of using return to send a value back to the caller, they use the yield keyword. Each yield statement in the generator function pauses the function’s execution and sends a value to the caller. When the generator is resumed, it continues from where it left off.

**Key Characteristics of Generators**

1. **Lazy Evaluation**: Generators produce items only when needed, which is efficient for large datasets or infinite sequences.
2. **Single Iteration**: Generators maintain their internal state and can only be iterated through once. After completion, they cannot be restarted unless redefined.
3. **Memory Efficiency**: Since values are produced on demand rather than stored in memory, generators are ideal for memory-constrained tasks.

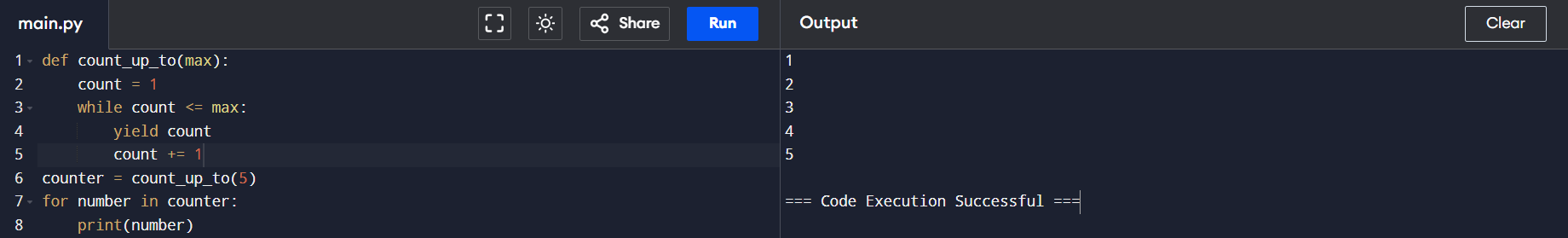
**Basic Syntax:**

def generator\_name(parameters):

# Code that produces items

yield value

Example:



**Differences Between return and yield:**

* **return**: Exits the function and sends a value back to the caller. Once a function returns, it cannot resume.
* **yield**: Pauses the function and sends a value back to the caller without terminating the function. The function can later resume from where it left off.

**Advantages of Generators:**

1. **Memory Efficiency**: Generators don’t store the entire data sequence in memory. Instead, they produce items on demand, making them useful for large data processing.
2. **Improved Performance**: By avoiding the need to generate and store large data structures all at once, generators often reduce the time complexity.
3. **Infinite Sequences**: Generators can be used to produce infinite sequences (like Fibonacci numbers or endless streams of data), which would otherwise be impossible with lists.

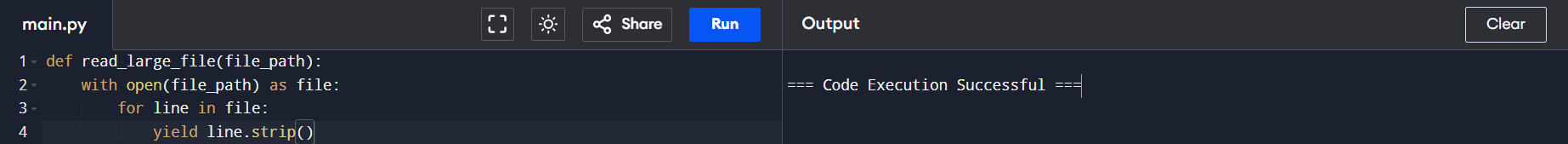
**Limitations**: Once exhausted, generators cannot be restarted; they need to be redefined to start from the beginning.

**7. What are the advantages of using generators over regular functions?**

**1. Memory Efficiency**

Generators produce items one at a time, on demand, rather than storing an entire collection of items in memory. This makes them ideal for working with large datasets or streams of data where it’s impractical to load everything into memory at once.

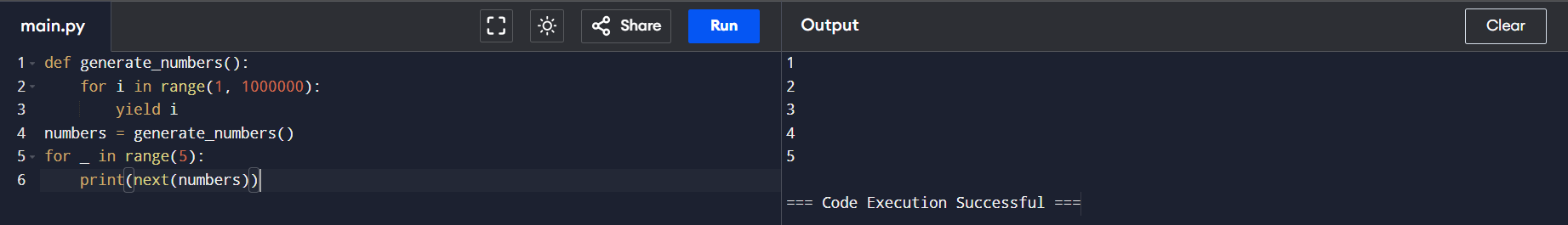
**Example**:



**2. Improved Performance**

**By generating values only when requested, generators avoid the time and memory cost of creating and storing a large data structure all at once. This results in faster initial execution, especially in scenarios where only a portion of the dataset is required or where the data needs to be processed iteratively.**

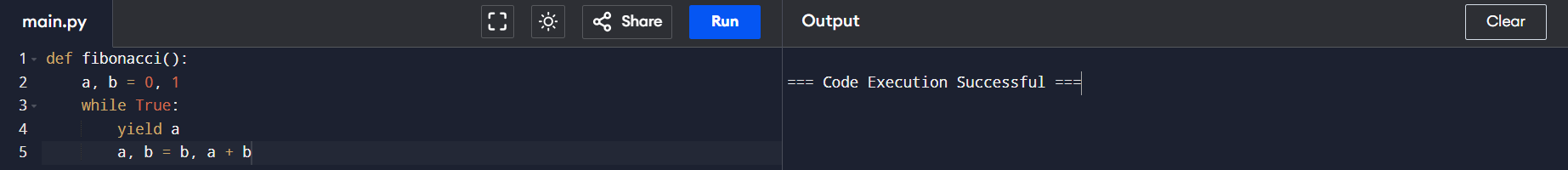
**Example**:



**3. Ability to Handle Infinite Sequences**

**Generators can be used to represent potentially infinite sequences, such as a stream of data, a countdown, or the Fibonacci series. Since they generate items on the fly, generators can be used to model endless series that would be impossible to store in a list or other data structure.**

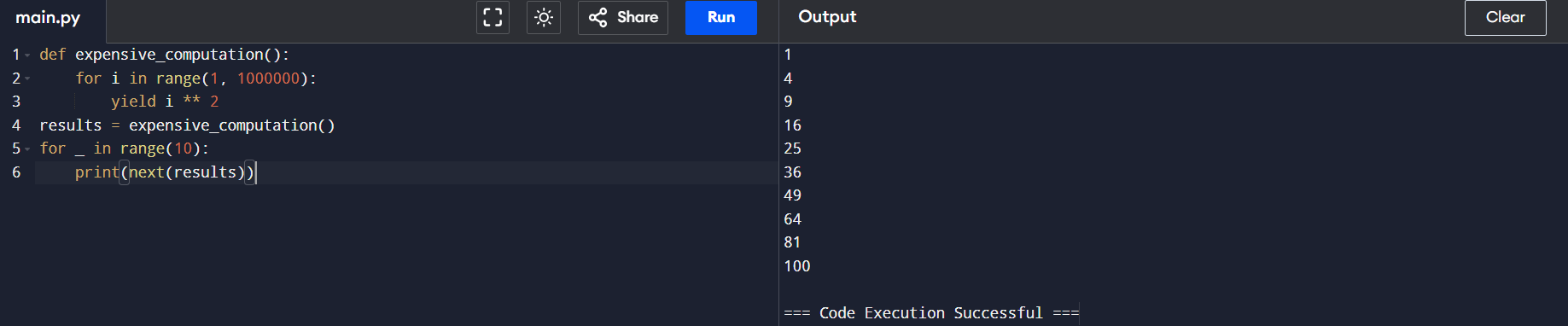
**Example**:



**4. Lazy Evaluation**

**Generators support lazy evaluation, meaning they calculate values only when needed. This behavior is beneficial for performance, as it allows a generator to start producing results before the full computation is completed. Lazy evaluation is particularly useful in scenarios where intermediate values might be useful before the final result is fully computed.**

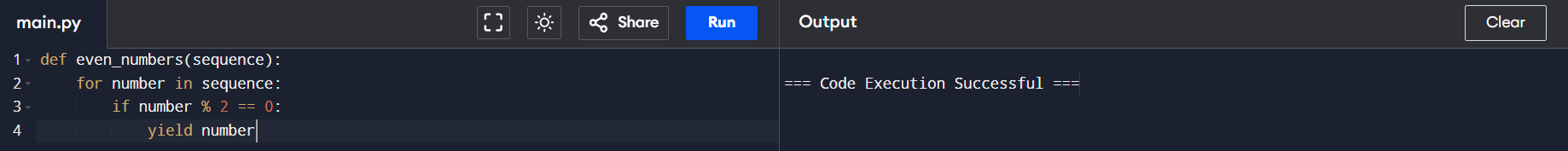
**Example**:



**5. Cleaner and More Readable Code**

Generators allow for cleaner and more readable code by abstracting the concept of iteration. Instead of managing an internal state or index variable (as with a while loop), generators handle state automatically, simplifying code.

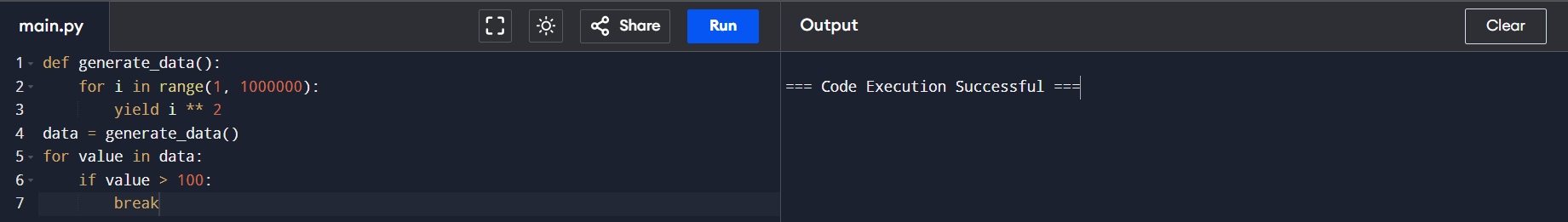
**Example**:



**6. Reduced Overhead in Function Calls**

With regular functions that return large datasets, the function call itself can be costly in terms of time and memory usage. Generators avoid this overhead by yielding one value at a time, rather than returning a large result all at once.

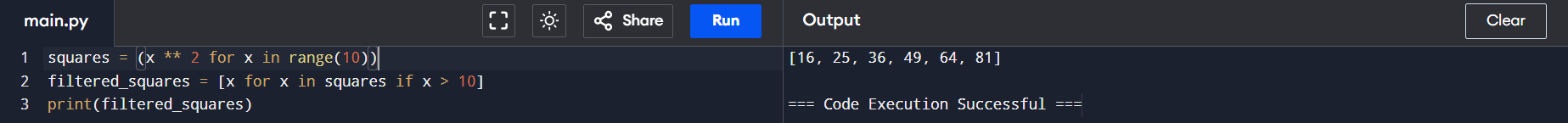
**Example**:



**7. Natural Support for Iteration Protocol**

Generators automatically support Python’s iterator protocol, so they can be used directly in for loops, comprehensions, and other iterator-based structures. This compatibility makes them more flexible and allows seamless integration with Python’s iteration tools.

**Example**:



**8. What is a lambda function in Python and when is it typically used?**

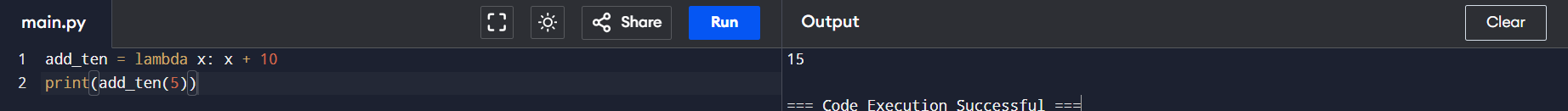
A lambda function in Python is a small, anonymous function defined using the lambda keyword. Unlike regular functions defined with def, lambda functions are single-line expressions without a formal name or complex statements. They’re commonly used for short, throwaway functions where defining a full function with def would be unnecessary.

**Lambda functions are defined with the following syntax:**

lambda arguments: expression

* **Arguments**: Lambda functions can take multiple arguments, separated by commas.
* **Expression**: The expression is evaluated and returned as the function’s result.

**Example :**

****

**Typical Use Cases for Lambda Functions:**

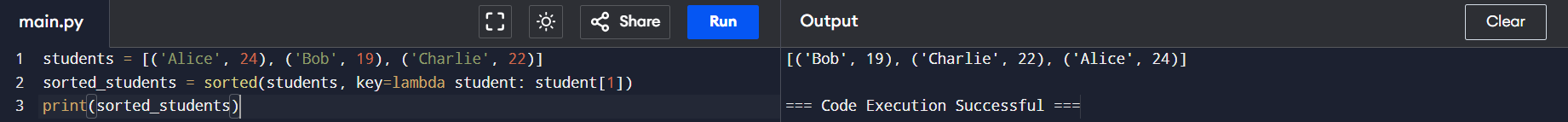
Lambda functions are often used in contexts where a simple function is required temporarily or where defining a full function would be excessive. Here are some common use cases:

**1. Sorting and Filtering Data :**

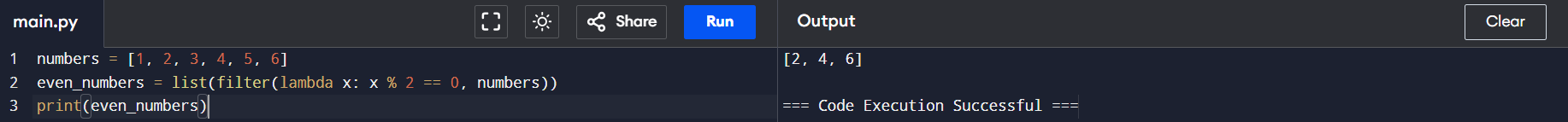
Lambda functions are frequently used with built-in functions like sorted(), filter(), and map() where you need to pass a simple function to define a sorting key, filtering condition, or mapping transformation.

**Example:**

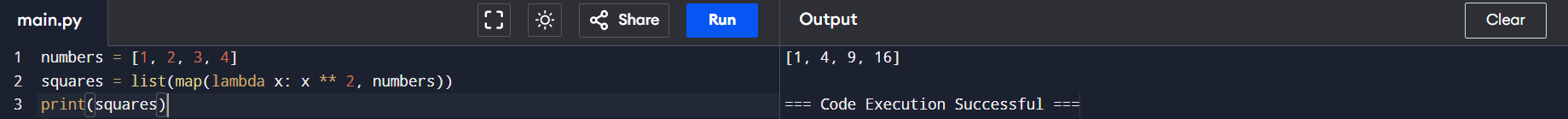
1. **Sorting**

****

1. **Filtering**

****

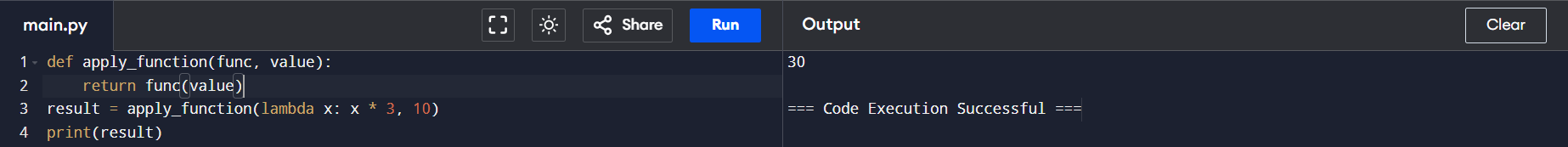
1. **Mapping**

****

**2. Using Lambda Functions in Higher-Order Functions**

Lambda functions are often passed as arguments to higher-order functions—functions that accept other functions as parameters. This is useful when you want to specify custom operations without defining standalone functions.

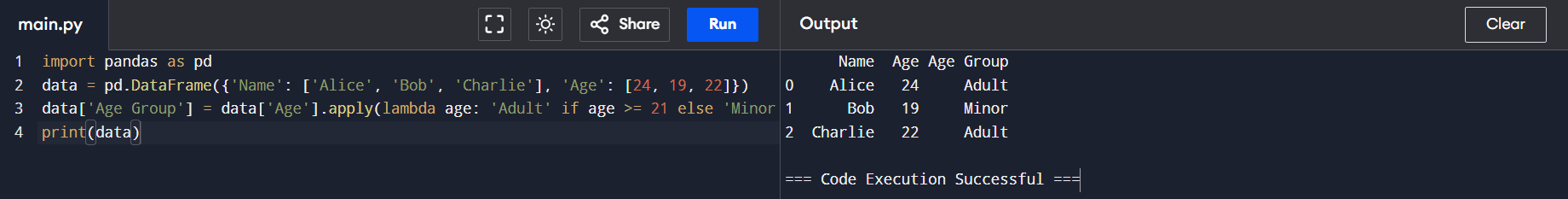
**Example:**



**3. Using Lambdas in Data Processing (e.g., Pandas)**

In data processing and analysis with libraries like Pandas, lambda functions are often used for transforming data in DataFrame columns, as they provide a compact syntax for applying custom operations.

**Example:**



Benefits and Limitations of Lambda Functions:

**Benefits**

* **Conciseness**: Lambda functions are quick to write, making code more compact.
* **Inline Usage**: Ideal for use in functions where temporary, one-off functionality is needed.
* **Functional Programming**: Lambda functions fit naturally in functional programming styles, especially with built-in functions like map(), filter(), and sorted().

**Limitations**

* **Single Expression Only**: Lambdas are limited to a single expression and cannot include statements like if, for, or while. This limits their complexity.
* **No Multi-line Capability**: Lambda functions cannot span multiple lines, so they are not suitable for tasks that require more extensive processing.
* **Lacks Readability for Complex Logic**: Using lambdas for anything beyond simple expressions can reduce code readability.

**9. Explain the purpose and usage of the map() function in Python.**

The map() function in Python is a built-in higher-order function that applies a given function to each item of an iterable (such as a list, tuple, or string) and returns a new iterable (specifically a map object) containing the results. It’s useful for transforming data by applying a function to each element in a sequence.

**Syntax of map():**

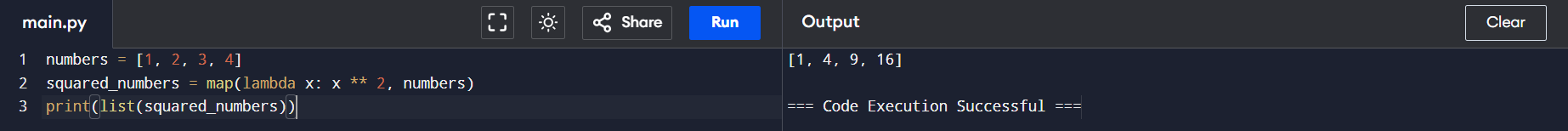
map(function, iterable, ...)

* **function**: The function to apply to each item in the iterable. It can be a built-in function, a user-defined function, or a lambda function.
* **iterable**: The iterable(s) (e.g., list, tuple) to which the function is applied. Multiple iterables can be passed as arguments to map(), and the function should accept as many arguments as there are iterables.

**Basic Usage of map():**

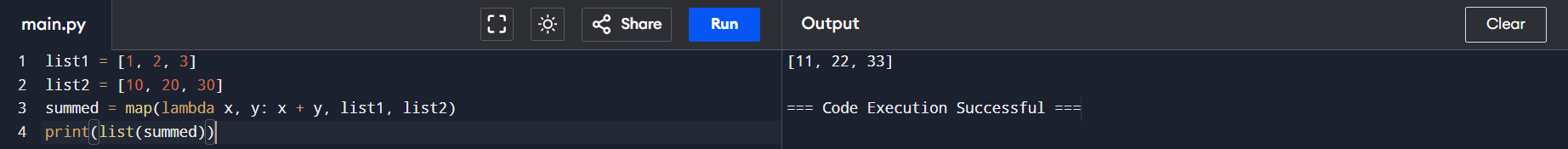
* The most common use of map() is to apply a function to all elements of a single iterable, transforming each element based on the logic in the function.

**Example Using map() with a Built-In Function :**

****

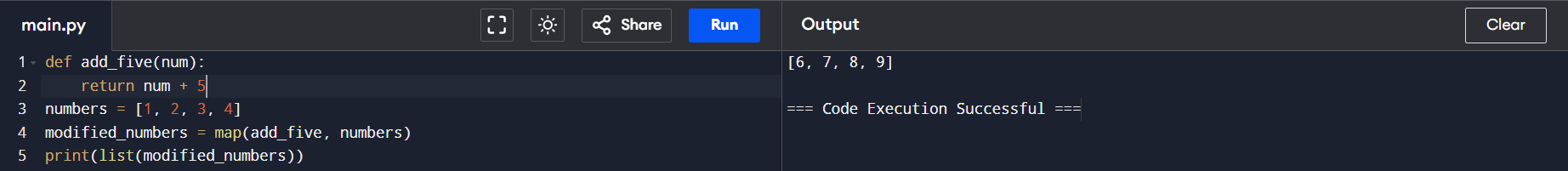
**Example Using map() with Multiple Iterables:**

* If multiple iterables are passed, map() applies the function to corresponding elements from each iterable. All iterables should have the same length; otherwise, map() stops when the shortest iterable is exhausted.

****

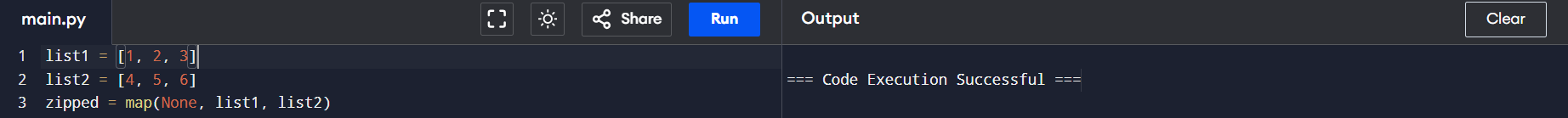
**Example Using map() with a Named Function:**

* You can also use map() with a regular function instead of a lambda. This can be useful for applying more complex transformations.

****

**Example Using map() with None:**

* When map() is called with None as the function argument, it returns the items of the iterable(s) directly. This feature is rarely used, but it effectively “zips” the elements when used with multiple iterables.



**10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?**

map(), reduce(), and filter() are higher-order functions in Python that allow functional-style data processing. They each serve different purposes:

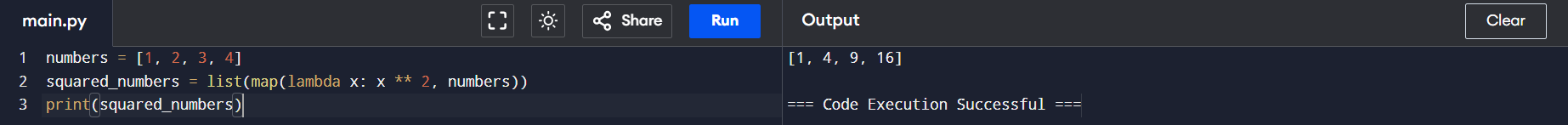
1. **map()**: Transforms each item in an iterable by applying a given function.
2. **reduce()**: Reduces an iterable to a single cumulative value by repeatedly applying a function to pairs of elements.
3. **filter()**: Filters items in an iterable based on a condition, retaining only items that meet the criteria.
4. **map():**

**Purpose**: To apply a function to each item in an iterable, creating a new iterable with the transformed items.

**Syntax**:

map(function, iterable, ...)

**Example:**

****

**When to Use:** When you need to apply the same transformation or operation to each item in a list, tuple, or other iterable.

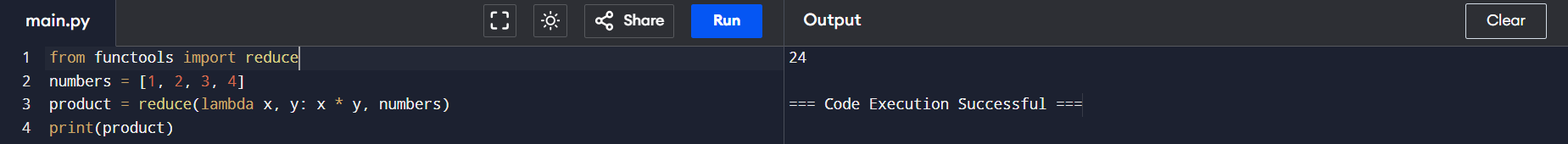
**2. reduce()**

**Purpose:** To iteratively apply a function to the items in an iterable, reducing it to a single cumulative value.

**Syntax:**

reduce(function, iterable, initializer=None)

**Example:**

****

**When to Use:** When you need to combine elements in a list to produce a single result, such as summing all elements, finding a product, or calculating an aggregate.

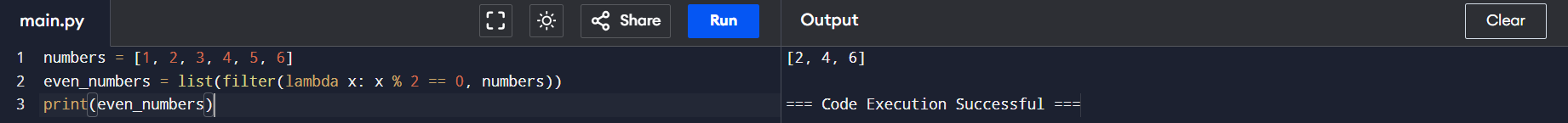
**3. filter()**

**Purpose:** To filter items from an iterable based on a condition specified by a function, creating a new iterable containing only the items that satisfy the condition.

**Syntax:**

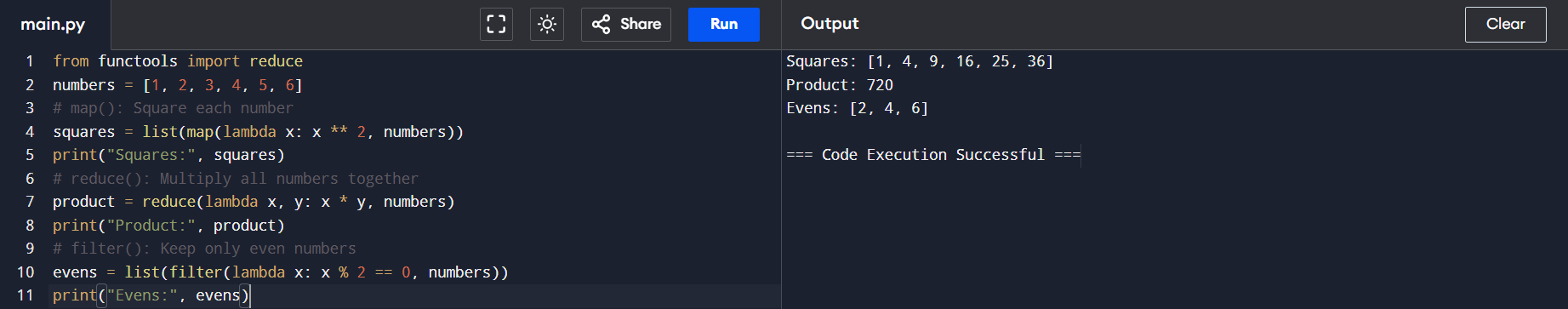
filter(function, iterable)

**Example**:



**When to Use**: When you want to select items from a list or other iterable based on specific criteria, like keeping only even numbers or non-empty strings.

**Example of Comparison:**

****

**11. Using pen & Paper write the internal mechanism for sum operation using  reduce function on this given list:[47,11,42,13];**

**List: [47, 11, 42, 13]**

1. **First Step:**
   * Initial values: x = 47, y = 11
   * Operation: 47 + 11 = 58
   * Result after the first step: 58
2. **Second Step:**
   * Now x becomes the result of the previous step, x = 58, and y is the next element in the list, y = 42.
   * Operation: 58 + 42 = 100
   * Result after the second step: 100
3. **Third Step:**
   * Now x becomes the result of the previous step, x = 100, and y is the next element in the list, y = 13.
   * Operation: 100 + 13 = 113
   * Result after the third step: 113

At this point, all elements have been processed, and reduce() returns the final cumulative result, 113

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Step** | **x** | **y** | **Operation** | **Result** |
| **1** | **47** | **11** | **47+11** | **58** |
| **2** | **58** | **42** | **58+42** | **100** |
| **3** | **100** | **13** | **100+13** | **113** |