**UDAY RAJPUT**

**COLLECTION-FUNCTION-MODULES**

**• What is List? How will you reverse a list?**

**Ans :-**

In Python, a list is a data structure that holds a collection of items. It's mutable, which means you can change its content after it's created. Lists are defined by enclosing a comma-separated sequence of elements in square brackets []. For example:

python

my\_list = [1, 2, 3, 4, 5]

To reverse a list in Python, you can use the reverse() method, which reverses the elements of the list in place:

python

my\_list = [1, 2, 3, 4, 5]

my\_list.reverse()

print(my\_list) Output: [5, 4, 3, 2, 1]

Another way to reverse a list is by using slicing. You can use the slicing syntax [::-1] to create a reversed copy of the original list:

python

my\_list = [1, 2, 3, 4, 5]

reversed\_list = my\_list[::-1]

print(reversed\_list) Output: [5, 4, 3, 2, 1]

This method does not modify the original list but creates a new one with the reversed order of elements.

**• How will you remove last object from a list? Suppose list1 is [2, 33, 222, 14, and 25], what is list1 [-1]?**

**Ans:-**

To remove the last object from a list in Python, you can use the pop() method with the index -1. This will remove and return the last element from the list. Here's how you can do it:

python

list1 = [2, 33, 222, 14, 25]

last\_item = list1.pop()

print(last\_item) Output: 25

print(list1) Output: [2, 33, 222, 14]

Regarding the expression list1[-1], it accesses the last element of the list list1. In this case, it returns 25, which is the last element of the list [2, 33, 222, 14, 25]. However, it's important to note that this doesn't remove the last element from the list, it just accesses it. If you want to remove the last element, you should use the pop() method as shown above.

**• Differentiate between append () and extend () methods?**

**Ans:-**

The append() and extend() methods in Python are both used to add elements to a list, but they differ in how they add those elements:

1. append(): This method adds its argument as a single element to the end of the list. If the argument is a list itself, it will be appended as a single element, making the list a nested list.

Example:

python

list1 = [1, 2, 3]

list1.append(4)

print(list1) Output: [1, 2, 3, 4]

list2 = [5, 6, 7]

list1.append(list2)

print(list1) Output: [1, 2, 3, 4, [5, 6, 7]]

2. extend(): This method iterates over its argument, which should be an iterable such as a list, and adds each element of the iterable to the end of the list. It essentially appends each individual element from the iterable.

Example:

python

list1 = [1, 2, 3]

list2 = [4, 5, 6]

list1.extend(list2)

print(list1) Output: [1, 2, 3, 4, 5, 6]

In summary, append() adds its argument as a single element, potentially creating a nested list if the argument itself is a list, while extend() adds individual elements from an iterable to the end of the list, effectively flattening the list.

**• How will you compare two lists?**

**Ans :-**

Comparing two lists involves checking whether they have the same elements, potentially in the same order or regardless of order, depending on the comparison criteria. Here are a few common ways to compare lists:

1. Element-wise Comparison:

- Check if each element in one list matches the corresponding element in the other list.

- This method is sensitive to both element values and their positions in the list.

2. Order-insensitive Comparison:

- Sort both lists and then compare them element by element.

- This method doesn't care about the order of elements within the lists, only whether they contain the same elements.

3. Set Comparison:

- Treat lists as sets and compare their set representations.

- This method checks if the two lists contain the same unique elements, regardless of order or duplicates.

4. Using Libraries:

- Many programming languages provide built-in functions or libraries for list comparison.

- These functions often offer various options for comparison criteria, such as strict equality, ignoring order, or handling duplicates differently.

5. Performance Consideration:

- Depending on the size of the lists and the language being used, the performance of different comparison methods may vary.

- For large lists, efficient algorithms or data structures can be employed to improve performance.

The choice of comparison method depends on the specific requirements of your application, such as whether order matters, whether duplicates should be considered, and the performance constraints.

**• What is tuple? Difference between list and tuple.**

**Ans :-**

A tuple in Python is a collection of ordered, immutable elements. It is similar to a list but with the key difference that tuples are immutable, meaning once they are created, their elements cannot be changed, added, or removed. Tuples are defined using parentheses ().

Here are the main differences between lists and tuples:

1. Mutability:

- Lists are mutable, meaning you can change, add, or remove elements after the list is created.

- Tuples are immutable, meaning once they are created, their elements cannot be changed.

2. Syntax:

- Lists are defined using square brackets [].

- Tuples are defined using parentheses ().

3. Performance:

- Tuples are generally faster than lists because of their immutability. Once a tuple is created, Python does not need to allocate additional memory for it or perform extra checks to ensure immutability.

- Lists, being mutable, require more overhead in terms of memory allocation and manipulation.

4. Use Cases:

- Lists are used when you need a collection of elements that can change over time. For example, when you're working with data that needs to be modified or updated.

- Tuples are used when you want to create a collection of elements that should not be changed, such as a set of coordinates or configuration settings.

Here's an example to illustrate the difference:

python

List example

my\_list = [1, 2, 3]

my\_list[0] = 10 Valid operation, changes the first element to 10

Tuple example

my\_tuple = (1, 2, 3)

my\_tuple[0] = 10 Invalid operation, raises TypeError: 'tuple' object does not support item assignment

In this example, changing the first element of the list my\_list to 10 is a valid operation because lists are mutable. However, attempting to change the first element of the tuple my\_tuple to 10 results in a TypeError because tuples are immutable.

**• How will you create a dictionary using tuples in python**

**Ans :-**

Sure, here’s a simple explanation of how to create a dictionary from a list of tuples in Python, using two methods:

Method 1: Using dict()

You can use the dict() function to convert a list of tuples directly into a dictionary.

python

List of tuples

tuples\_list = [(1, 'a'), (2, 'b'), (3, 'c')]

Convert list of tuples to dictionary

dictionary = dict(tuples\_list)

Print the result

print("Dictionary:", dictionary)

Method 2: Using a Loop

You can also use a loop to add each tuple’s key-value pair to the dictionary.

python

List of tuples

tuples\_list = [(1, 'a'), (2, 'b'), (3, 'c')]

Create an empty dictionary

dictionary = {}

Loop through each tuple in the list

for key, value in tuples\_list:

dictionary[key] = value

Print the result

print("Dictionary:", dictionary)

Explanation

- Initial List of Tuples: You start with a list of tuples, where each tuple contains a key and a value.

- Using dict(): The dict() function converts the list of tuples directly into a dictionary.

- Using a Loop: You create an empty dictionary and then loop through the list of tuples, adding each key-value pair to the dictionary.

Output

For both methods, the output will be:

python

Dictionary: {1: 'a', 2: 'b', 3: 'c'}

This means that the list of tuples has been successfully converted into a dictionary where the first element of each tuple is the key and the second element is the value.

* **How Do You Traverse Through A Dictionary Object In Python?**

**Ans :-**

In Python, dictionaries are collections of key-value pairs, and you can traverse through them using several methods. Here are the common ways to iterate over a dictionary:

1. Iterate over keys:

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

for key in my\_dict:

print(key, my\_dict[key])

In this method, the loop iterates over the keys of the dictionary, and you can access the corresponding value using the key.

2. Iterate over values:

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

for value in my\_dict.values():

print(value)

Here, you use the .values() method to get an iterable view of the values in the dictionary.

3. Iterate over key-value pairs:

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

for key, value in my\_dict.items():

print(key, value)

The .items() method returns an iterable view of the key-value pairs in the dictionary, allowing you to unpack each pair directly in the loop.

4. Iterate using dictionary comprehension:

You can also use dictionary comprehensions to create new dictionaries or lists based on the original dictionary:

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

Create a list of keys

keys = [key for key in my\_dict]

print(keys)

Create a list of values

values = [value for value in my\_dict.values()]

print(values)

Create a list of key-value pairs

items = [(key, value) for key, value in my\_dict.items()]

print(items)

5. Using enumerate with dictionaries:

While enumerate is typically used with lists, it can also be useful with dictionaries when you need the index along with the key-value pairs:

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

for index, (key, value) in enumerate(my\_dict.items()):

print(index, key, value)

**• How Do You Check The Presence Of A Key In A Dictionary?**

**Ans :-**

In Python, you can check for the presence of a key in a dictionary using several methods. Here are the most common and recommended ways:

1. Using the in operator:

The in operator is the most straightforward and idiomatic way to check for a key in a dictionary.

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

if 'a' in my\_dict:

print("Key 'a' is present in the dictionary")

else:

print("Key 'a' is not present in the dictionary")

2. Using the get method:

The get method of a dictionary can also be used to check for a key. It returns None (or a specified default value) if the key is not found, without raising an exception.

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

if my\_dict.get('a') is not None:

print("Key 'a' is present in the dictionary")

else:

print("Key 'a' is not present in the dictionary")

3. Using the keys method:

While less common and not as efficient as the in operator, you can use the keys method to check for the presence of a key.

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

if 'a' in my\_dict.keys():

print("Key 'a' is present in the dictionary")

else:

print("Key 'a' is not present in the dictionary")

4. Handling with exceptions (not recommended for simple presence check):

You can also use exception handling to check for a key, though this method is generally not recommended for a simple presence check as it is less efficient and less readable.

python

my\_dict = {'a': 1, 'b': 2, 'c': 3}

try:

value = my\_dict['a']

print("Key 'a' is present in the dictionary")

except KeyError:

print("Key 'a' is not present in the dictionary")

**• Why Do You Use the Zip () Method in Python?**

**Ans:-**

The zip() function in Python is used to combine multiple iterables (e.g., lists, tuples) element-wise into a single iterable of tuples. It is particularly useful for iterating over multiple sequences in parallel. Here are some common use cases and explanations for why you might use the zip() method:

1. Parallel Iteration:

- When you have two or more sequences and you want to iterate over them simultaneously, zip() allows you to do this easily.

python

names = ['Alice', 'Bob', 'Charlie']

scores = [85, 92, 78]

for name, score in zip(names, scores):

print(f"{name}: {score}")

Output:

Alice: 85

Bob: 92

Charlie: 78

2. Creating Dictionaries:

- You can use zip() to create dictionaries by combining two lists: one of keys and one of values.

python

keys = ['name', 'age', 'city']

values = ['Alice', 30, 'New York']

my\_dict = dict(zip(keys, values))

print(my\_dict)

Output:

{'name': 'Alice', 'age': 30, 'city': 'New York'}

3. Transposing Matrices:

- zip() can be used to transpose a matrix, i.e., to swap rows with columns.

python

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

transposed\_matrix = list(zip(matrix))

print(transposed\_matrix)

Output:

[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

4. Aggregating Data:

- When you need to aggregate related data points from multiple lists.

python

dates = ['2024-06-19', '2024-06-20', '2024-06-21']

temperatures = [72, 75, 78]

humidity = [60, 65, 70]

for date, temp, hum in zip(dates, temperatures, humidity):

print(f"Date: {date}, Temp: {temp}, Humidity: {hum}")

Output:

Date: 2024-06-19, Temp: 72, Humidity: 60

Date: 2024-06-20, Temp: 75, Humidity: 65

Date: 2024-06-21, Temp: 78, Humidity: 70

5. Unpacking and Repacking:

- Sometimes you might need to unpack a list of tuples and repack them into different structures.

python

pairs = [('a', 1), ('b', 2), ('c', 3)]

letters, numbers = zip(pairs)

print(letters) Output: ('a', 'b', 'c')

print(numbers) Output: (1, 2, 3)

Benefits of Using zip():

- Readability: It makes the code more readable and concise.

- Convenience: It simplifies the task of iterating over multiple iterables in parallel.

- Efficiency: It is efficient in terms of both time and space, as it returns an iterator instead of a list.

Important Notes:

- If the input iterables have different lengths, zip() stops creating tuples when the shortest input iterable is exhausted.

- If you need to handle iterables of different lengths and you want to avoid truncation, you can use itertools.zip\_longest() from the itertools module.

python

from itertools import zip\_longest

a = [1, 2, 3]

b = ['x', 'y']

result = list(zip\_longest(a, b, fillvalue='N/A'))

print(result)

Output:

[(1, 'x'), (2, 'y'), (3, 'N/A')]

Overall, zip() is a powerful tool in Python for combining and iterating over multiple sequences in a clean and efficient manner.

**• How Many Basic Types Of Functions Are Available In Python?**

**Ans :-**

In Python, functions can be broadly classified into three basic types:

1. Built-in Functions:

- These are functions that are provided by Python and are always available. Examples include print(), len(), type(), int(), and str(). They are part of the Python standard library and provide essential functionality.

python

print("Hello, World!") Built-in function

2. User-Defined Functions:

- These are functions that you define yourself using the def keyword. They allow you to encapsulate reusable blocks of code.

python

def greet(name):

return f"Hello, {name}!"

print(greet("Alice")) User-defined function

3. Anonymous (Lambda) Functions:

- These are small, unnamed functions defined using the lambda keyword. They are typically used for short, simple operations and are often used as arguments to higher-order functions like map(), filter(), and sorted().

python

add = lambda x, y: x + y Anonymous (lambda) function

print(add(3, 5))

Examples and Uses:

1. Built-in Functions:

- Python provides numerous built-in functions for various tasks. Here are a few examples:

python

numbers = [1, 2, 3, 4, 5]

print(len(numbers)) Output: 5

print(sum(numbers)) Output: 15

print(max(numbers)) Output: 5

2. User-Defined Functions:

- These functions are created to perform specific tasks and can take arguments and return values.

python

def multiply(a, b):

return a b

result = multiply(3, 4)

print(result) Output: 12

3. Anonymous (Lambda) Functions:

- Lambda functions are useful for small, one-time operations, often used in conjunction with functions like map(), filter(), and sorted().

python

numbers = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x2, numbers))

print(squared) Output: [1, 4, 9, 16, 25]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers) Output: [2, 4]

Summary:

- Built-in Functions: Predefined functions that are part of Python's standard library.

- User-Defined Functions: Functions created by the user to perform specific tasks, using the def keyword.

- Anonymous (Lambda) Functions: Small, unnamed functions defined using the lambda keyword, typically used for short operations.

These basic types of functions allow Python to be a versatile and powerful programming language, enabling both simple and complex operations to be performed efficiently.

**• How can you pick a random item from a list or tuple?**

**Ans :-**

To pick a random item from a list or tuple in Python, you can use the random.choice() function from the random module. Here are the steps and examples to demonstrate how to do this:

1. Import the random module:

- Before using any functions from the random module, you need to import it.

2. Use random.choice():

- The random.choice() function takes a sequence (such as a list or tuple) as an argument and returns a randomly selected item from that sequence.

Example with a List

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Pick a random item from the list

random\_item = random.choice(my\_list)

print(random\_item)

Example with a Tuple

python

import random

Define a tuple

my\_tuple = (10, 20, 30, 40, 50)

Pick a random item from the tuple

random\_item = random.choice(my\_tuple)

print(random\_item)

Additional Examples

Here are a few more examples to illustrate different scenarios:

Random Choice from a List of Strings

python

import random

Define a list of strings

colors = ['red', 'blue', 'green', 'yellow', 'purple']

Pick a random color

random\_color = random.choice(colors)

print(random\_color)

Random Choice from a List of Tuples

python

import random

Define a list of tuples

coordinates = [(1, 2), (3, 4), (5, 6), (7, 8)]

Pick a random coordinate

random\_coordinate = random.choice(coordinates)

print(random\_coordinate)

Notes

- The random.choice() function works with any sequence (lists, tuples, strings, etc.).

- If you need to select multiple random items without replacement, you can use random.sample():

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Pick 2 random items from the list without replacement

random\_items = random.sample(my\_list, 2)

print(random\_items)

- If you need to select multiple random items with replacement, you can use random.choices():

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Pick 2 random items from the list with replacement

random\_items\_with\_replacement = random.choices(my\_list, k=2)

print(random\_items\_with\_replacement)

By using these functions from the random module, you can effectively and easily pick random items from lists, tuples, and other sequences in Python.

**• How can you pick a random item from a range?**

**Ans:-**

To pick a random item from a range in Python, you can use the random.choice() function from the random module, or, for better performance with large ranges, use the random.randint() or random.randrange() functions. Here's how you can do it:

Using random.choice() with range()

You can use random.choice() directly on a range object. This is simple and works well for relatively small ranges.

python

import random

Define a range

my\_range = range(1, 11) This creates a range of numbers from 1 to 10

Pick a random item from the range

random\_item = random.choice(my\_range)

print(random\_item)

Using random.randint()

The random.randint() function can be used to generate a random integer within a specified range, inclusive of both endpoints.

python

import random

Define the start and end of the range

start = 1

end = 10

Pick a random item from the range

random\_item = random.randint(start, end)

print(random\_item)

Using random.randrange()

The random.randrange() function can also be used to generate a random integer within a specified range, similar to random.randint(). It provides more flexibility by allowing you to specify a step.

python

import random

Define the start, stop, and step of the range

start = 1

stop = 11 The stop value is exclusive

step = 1

Pick a random item from the range

random\_item = random.randrange(start, stop, step)

print(random\_item)

Summary

- For small ranges: random.choice(range(start, stop))

- For inclusive range: random.randint(start, end)

- For exclusive range with optional step: random.randrange(start, stop, step)

Examples

Example with random.choice()

python

import random

my\_range = range(1, 100)

random\_item = random.choice(my\_range)

print(random\_item) Outputs a random number between 1 and 99

Example with random.randint()

python

import random

random\_item = random.randint(1, 99)

print(random\_item) Outputs a random number between 1 and 99 (inclusive)

Example with random.randrange()

python

import random

random\_item = random.randrange(1, 100, 2)

print(random\_item) Outputs a random odd number between 1 and 99

These methods provide flexible and efficient ways to pick a random item from a range in Python.

**• How can you get a random number in python?**

**Ans:-**

In Python, you can generate random numbers using the random module, which provides various functions for different types of random number generation. Here are the common ways to get random numbers in Python:

1. Random Integer

To get a random integer within a specified range, use the random.randint() function.

python

import random

Get a random integer between 1 and 10 (inclusive)

random\_integer = random.randint(1, 10)

print(random\_integer)

2. Random Float

To get a random floating-point number between 0.0 (inclusive) and 1.0 (exclusive), use the random.random() function.

python

import random

Get a random float between 0.0 and 1.0

random\_float = random.random()

print(random\_float)

3. Random Float in a Range

To get a random floating-point number within a specified range, use the random.uniform() function.

python

import random

Get a random float between 1.5 and 10.5

random\_float\_in\_range = random.uniform(1.5, 10.5)

print(random\_float\_in\_range)

4. Random Number from a Range

To get a random integer from a specified range, use the random.randrange() function. This function is similar to range() and allows you to specify a step.

python

import random

Get a random integer from 0 to 9

random\_integer = random.randrange(10)

print(random\_integer)

Get a random integer from 1 to 10 (exclusive of 10)

random\_integer\_in\_range = random.randrange(1, 10)

print(random\_integer\_in\_range)

Get a random integer from 1 to 10 with step 2

random\_integer\_with\_step = random.randrange(1, 10, 2)

print(random\_integer\_with\_step)

5. Random Choice from a Sequence

To select a random item from a list or tuple, use the random.choice() function.

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Get a random item from the list

random\_item = random.choice(my\_list)

print(random\_item)

6. Random Sample

To get multiple random items from a sequence without replacement, use the random.sample() function.

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Get 3 random items from the list

random\_sample = random.sample(my\_list, 3)

print(random\_sample)

7. Random Choices with Replacement

To get multiple random items from a sequence with replacement, use the random.choices() function.

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Get 3 random items from the list with replacement

random\_choices = random.choices(my\_list, k=3)

print(random\_choices)

8. Shuffle a Sequence

To shuffle the elements of a list in place, use the random.shuffle() function.

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Shuffle the list

random.shuffle(my\_list)

print(my\_list)

These methods cover most common scenarios for generating random numbers and selecting random elements in Python. The random module provides a versatile set of tools for random number generation and manipulation.

**• How will you set the starting value in generating random numbers?**

**Ans:-**

In Python, you can set the starting value (seed) for generating random numbers using the random.seed() function from the random module. Setting a seed ensures that you get the same sequence of random numbers each time you run your program, which can be useful for debugging and testing.

Here’s how you can set the seed value and generate random numbers:

Setting the Seed

1. Import the random module:

- Before you can use the random functions, you need to import the module.

2. Set the seed using random.seed():

- Call random.seed() with an integer argument. This integer is the seed value.

Example

Setting the Seed and Generating Random Integers

python

import random

Set the seed

random.seed(42)

Generate random integers

print(random.randint(1, 10)) Output will be consistent

print(random.randint(1, 10))

print(random.randint(1, 10))

Setting the Seed and Generating Random Floats

python

import random

Set the seed

random.seed(42)

Generate random floats

print(random.random()) Output will be consistent

print(random.random())

print(random.random())

Setting the Seed and Generating Random Choices from a List

python

import random

Set the seed

random.seed(42)

Define a list

my\_list = [1, 2, 3, 4, 5]

Pick random items from the list

print(random.choice(my\_list)) Output will be consistent

print(random.choice(my\_list))

print(random.choice(my\_list))

Explanation

- random.seed(a=None, version=2): This initializes the random number generator. If you pass the same seed value, you will get the same sequence of numbers. If you don't pass any value to seed(), or if you pass None, it will use the current system time or some other default source of randomness.

Consistency in Different Runs

Using the same seed ensures that the sequence of random numbers is reproducible:

python

import random

First run

random.seed(42)

print(random.randint(1, 10)) Output: 2

print(random.randint(1, 10)) Output: 1

Second run

random.seed(42)

print(random.randint(1, 10)) Output: 2

print(random.randint(1, 10)) Output: 1

Use Cases for Seeding

- Testing and Debugging: Ensures that tests are reproducible and can be debugged reliably.

- Research and Analysis: Ensures that experimental results can be replicated.

- Games and Simulations: Ensures consistency in scenarios where predictable randomness is required (e.g., replaying game moves).

By using random.seed(), you can control the randomness in your program, making it deterministic and repeatable for the same seed value.

**• How will you randomizes the items of a list in place?**

**Ans :-**

To randomize (shuffle) the items of a list in place in Python, you can use the random.shuffle() function from the random module. This function modifies the list directly and does not return a new list.

Here is how you can use random.shuffle() to shuffle the items of a list in place:

Example

1. Import the random module:

- Before using the shuffle() function, you need to import the random module.

2. Use random.shuffle() to shuffle the list:

- Pass the list to random.shuffle() to shuffle it in place.

Code Example

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Print the list before shuffling

print("Before shuffling:", my\_list)

Shuffle the list in place

random.shuffle(my\_list)

Print the list after shuffling

print("After shuffling:", my\_list)

Output

The output will show the list before and after shuffling. The order of elements in the list will be randomized after the shuffle.

Before shuffling: [1, 2, 3, 4, 5]

After shuffling: [3, 5, 1, 4, 2] Example output, the order will vary

Notes

- In-place Operation: random.shuffle() shuffles the list in place, meaning the original list is modified and no new list is created.

- No Return Value: The function does not return any value. It returns None.

- Reproducibility: If you need reproducible shuffling for testing purposes, you can set a seed using random.seed() before calling random.shuffle().

Example with Seed for Reproducibility

python

import random

Define a list

my\_list = [1, 2, 3, 4, 5]

Set the seed

random.seed(42)

Shuffle the list in place

random.shuffle(my\_list)

Print the shuffled list

print("Shuffled list with seed 42:", my\_list)

Output with Seed

Shuffled list with seed 42: [4, 2, 3, 5, 1] Example output, will be consistent with the same seed

By using random.shuffle(), you can efficiently randomize the order of elements in a list directly, which is useful for scenarios like shuffling a deck of cards, randomizing a playlist, or mixing items in any list.