Day 4: Learning About the random Module and Python Lists

1. Random Module

The random module provides functions to generate pseudo-random numbers and perform random operations. These functions are widely used in simulations, gaming, cryptography, and anywhere unpredictability is required. Here's what I learned:

Key Functions in the random Module:

- **random.randint(a, b)**
  Generates a random integer between a and b (inclusive).

```
import random
print(random.randint(1, 10))  # Output: Random integer between 1 and 10
```

- **random.random()**
  Generates a random float between 0.0 and 1.0.

```
print(random.random())  # Output: A decimal number like 0.485294018
```

- **random.uniform(a, b)**
  Generates a random float between a and b.

```
print(random.uniform(1.5, 10.5))  # Output: Random float between 1.5 and 10.5
```

- **random.choice(sequence)**
  Selects a random item from a non-empty sequence like a list, tuple, or string.

```
fruits = ['apple', 'banana', 'cherry']
print(random.choice(fruits))  # Output: A random fruit
```

- **random.shuffle(sequence)**
  Shuffles a list in place (modifies the original list).

```
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)  # Output: A shuffled version of the list
```

- **random.sample(sequence, k)**
  Returns a new list containing k random elements from the sequence without

replacement.

```
letters = ['a', 'b', 'c', 'd', 'e']
print(random.sample(letters, 3))  # Output: A random subset of 3 letters
```

Key Notes on the Random Module:

- The numbers generated are **pseudo-random**, meaning they're deterministic and rely on a seed value.
- You can set the seed manually for reproducibility using random.seed(seed_value).
- Avoid using random for cryptographic purposes. Instead, use the secrets module.

2. Python Lists

A **list** is a built-in Python data structure used to store collections of items. Lists are **mutable**, meaning their contents can be changed after they are created.

Basic Operations on Lists:

- **Creating a List:**
  Lists can hold multiple data types, including other lists (nested lists).

```
my_list = [10, 20, 30, 40]
mixed_list = [10, 'hello', True, [1, 2, 3]]
print(my_list)
print(mixed_list)
```

- **Accessing Elements:**
  Use indexing to access elements in a list. Indices start at 0, and negative indices access elements from the end.

```
print(my_list[0])  # First element
print(my_list[-1]) # Last element
```

- **Adding Elements:**
  - **append()** adds an element to the end of the list.
  - **insert(index, value)** adds an element at a specific index.

```
my_list.append(50)
my_list.insert(2, 25)  # Insert 25 at index 2
print(my_list)  # Output: [10, 20, 25, 30, 40, 50]
```

- **Removing Elements:**

- **remove(value)** removes the first occurrence of a value.
- **pop(index)** removes and returns an element at a given index.

```
my_list.remove(30)
popped_item = my_list.pop(1)  # Remove the element at index 1
print(my_list, popped_item)
```

- **Slicing a List:**
  Extract a sublist by specifying a start, stop, and step.

```
sublist = my_list[1:4]  # Extract elements at index 1 to 3
print(sublist)
```

- **Iterating Through a List:**
  Use a loop to process each element in a list.

```
for item in my_list:
    print(item)
```

- **Sorting and Reversing:**
  - **sort()** sorts the list in place (modifies the list).
  - **sorted(list)** returns a new sorted list without modifying the original.
  - **reverse()** reverses the list in place.

```
my_list.sort()
reversed_list = list(reversed(my_list))
print(my_list, reversed_list)
```

- **List Comprehension:**
  A compact way to create or modify lists.

```
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

Key Notes on Lists:

- Lists are **heterogeneous**: they can store elements of different data types.
- Python lists are **dynamic**, meaning their size can change at runtime.

Exercises

Exercise 1: Random Number Guessing

- Write a program that generates a random number between 1 and 20. The user has to guess the number, and the program gives hints like "Too high" or "Too low" until the correct number is guessed.

Exercise 2: Random List Picker

- Create a program that asks the user for five favorite foods and stores them in a list. Then, randomly pick one item from the list and display it.

Exercise 3: Word Shuffle

- Write a program that takes a sentence as input, splits it into words, and shuffles the words randomly. Print the shuffled sentence.

Exercise 4: Lottery Number Generator

- Write a program that generates a list of six random numbers between 1 and 49 for a lottery ticket. Make sure the numbers are unique.

Exercise 5: Even or Odd Game

- Create a program that generates a random number between 1 and 100. The user must guess if the number is even or odd. After their guess, tell them whether they were correct or not.

Exercise 6: To-Do List Randomizer

- Write a program that lets the user input a list of tasks (e.g., "Do homework," "Clean the room"). Then, randomly shuffle the tasks and print them in a random order.

Exercise 7: Password Strength Checker

- Write a program that generates a random password of a specified length using a mix of letters, numbers, and symbols. The program should ensure the password contains at least one uppercase letter, one number, and one special character.

Exercise 8: Favorite Color Selector

- Write a program that asks the user to enter a list of their favorite colors. Randomly select one and print it as the result.

Exercise 9: Random Quiz Question Selector

- Create a program that stores five questions in a list (e.g., “What is 2 + 2?”). The program should randomly select one question and display it for the user to answer.

Exercise 10: Sum of Random Numbers

- Generate a list of 10 random numbers between 1 and 100. Calculate and print the sum of the numbers.

Reflection

Learning the random module and Python lists gave me powerful tools for creating dynamic and flexible programs. The random module introduces unpredictability, while lists allow me to store, manipulate, and analyze data efficiently.