**Module 14) Python – Collections, functions and Modules**

**Accessing List:**

1. **Understanding how to create and access elements in a list**

We can create a Python list by placing elements within square brackets [ ]. Unlike sets, there is no need for built-in functions to create a list. Also, it can contain mutable items. Make sure that the items are separated by a comma (,) in square brackets.

**Example:**

x = [2, 3, -7, 6.7, ‘a’]

Each element stored in a list is related to unique integer number called an index. In a list, the first element is indexed as 0, the second as 1, and so on. This means that if a list contains seven items, it will have an index ranging from 0 to 6. We can access these elements in the list by mentioning the index with index operator [ ] preceded by the name of the list. The index must be an integer. Also, we can access nested lists using nested indexing.

**Example:**

x = [2, 3, -7, 6.7, ‘a’]

print(x[0])

print(x[1])

**Output:**

2

3

1. **Indexing in lists (positive and negative indexing).**

In Python, lists are zero-indexed, meaning that the first element is at index 0, the second element at index 1, and so on. Negative indexes, shows the beginning from the end. So, -1 means the last time, -2 means the second last item, and so on. It represents positions from the end of an array. We don’t have to compute the offset as in List, but we can simply write List.

**Example:**

x = [2, 3, -7, 6.7, ‘a’]

print(x[-1])

print(x[-2])

**Output:**

a

6.7

1. **Slicing a list: accessing a range of elements.**

To access a range of elements in a list, you must slice it. One method is to utilize the simple slicing operator, i.e. colon (:) With this operator, one can define where to begin slicing, and where to terminate slicing, and the step. List slicing creates a new list from an old one.

**Syntax:**

List[start: stop: stop]

where,

* **start –** Index position from where the slicing will start in a list
* **stop** – index position till which the slicing will end in a list
* **step –** number of steps, i.e. the start index is changed after every n step, and list slicing is performed on that index

**Note -**  In Python, indexing starts from 0, and not 1.

**Get all the Items**

my\_list = [1,3,5,7,9]

print(“Item in List are:”,my\_list[:])

**Output:**

Items in List are: [1,3,5,7,9]

In above program, to get all the elements of the list we use ‘:’. This is similar to the statement print(my\_list)

**Get all the Items After a Specific Position**

my\_list = [1,3,5,7,9,11,13,15]

print(“Items in List after 3rd position are:”,my\_list[3:])

**Output:**

Items in List after 3rd position are: [7,9,11,13,15]

Int this program, we have used the start parameter by specifying an integer value that indicates the starting position of the slicing of the list. The elements at index 3 and all the elements after index 3 will be displayed.

**Get all the Items Before a Specific Position**

my\_list = [1,3,5,7,9,11,13,15]

print(“Items in List till 4th index are:”,my\_list[:4])

**Output:**

Items in List till 4th index are: [1,3,5,7]

In this program, we have used the stop parameter by specifying an integer value that indicates the ending position of the slicing of a list. The items before index 4 are sliced in the example. Element on index position 4 is not included.

**Get all the Items from One Position to Another Position**

my\_list = [1,3,5,7,9,11,13,15]

print(“Items in List from 1st to 6th index are:”,my\_list[1:6])

**Output:**

Items in List from 1st to 6th index are: [3,5,7,9,11]

If you wish to display all the elements between two specific indices, put them before and after the ‘:’ symbol. In the preceding example, my list[1:6] returns the elements between the first and sixth positions. The beginning position (i.e. 1) is included, but the finishing position (i.e. 6) is not.

**Get the Items at Specified Intervals**

my\_list = [1,3,5,7,9,11,15,17,19]

print(“Items in List at step of 2 are:”,my\_list[::2])

**Output:**

Items in List at step of 2 are: [1,5,9,13,17]

In this type, we have made use of the step parameter for slicing. The step parameter is an integer value that prints the list elements after specific intervals. For example, in the above program, we have declared a step of 2. So the elements in the position of 0,2,4,6, and so on will be printed.

If you want the indexing to begin with the final item, we can use the negative sign ‘-‘. In the below example, Items at interval 2 beginning with the last index are sliced.

my\_list = [1,3,5,7,9,11,13,15,17,19]

print(“Items in List are:”,my\_list[::-2])

**Output:**

Items in List are: [19,15,11,7,3]

**List Operations:**

1. **Common list operations: concatenation, repetition, membership**

It is the computation or actions applied to the variable containing the list of data types in and expression.

List manipulation in Python can be done using various operators like concatenation (+), repetition (\*), slicing of the list, and membership operators (in/not in). So, Let’s understand each operation in brief.

* Concatenation operator (+)
* Repetition operator (\*)
* Membership Operator (in, not in)
* **Concatenation operator (+)**

The (+) operator is used to add to two lists.

The syntax of the given operation is*: List1 + List2*

>>> lst1 = [12,43, 53]

>>> lst2 = [78,90]

>>> print(lst1 + lst2)

**Output:**

[12,43,53,78,90]

* **Repetition operator (\*)**

Like string, (\*) operator replicates the string number of specified times.

The syntax of the given operation: *List \* n*

>>> lst1 = [12,43,56]

>>> print( lst1 \* 3 )

**Output:**

[12,43,56,12,43,56,12,43,56]

* **Membership Operators (in, not in)**

The membership operator checks whether an element exists in the given list.

* **In:** return true if an element exists in the given list; False otherwise
* **Not in:** return true if an element does not exist in the given list; False otherwise.

>>> lst1 = [12,43,56,78,90]

>>> 56 in lst1

>>> 12 not in lst1

**Output:**

True

False

1. **Understanding list methods like append (), insert (), remove (), pop ().**

* **append():** Add its argument as a single item to the end of a list.
* **insert():** Insert an element at a specified position.
* **pop():** Removes and returns the element at the specified position. (or the last element if no index is specified).
* **remove():** Remove the first occurrence of a specified element.
* **append():**

List have several useful built-in methods, one of which is the append methods. When calling append on a list, we append an object to the end of the list:

>>> my\_list = [1,2]

>>> my\_list.append(‘a’)

>>> print(my\_list)

**Output:**

[a,2,’a’]

* **pop():**

The pop() method removes and returns the last item by default unless you give it an index argument.

Here are a couple of examples that demonstrate both the default behavior and the behavior when given an index:

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

>>> my\_list.pop()

>>>print(my\_list)

**Output:**

[1,2,3,4]

* **remove():**

If you want to remove a specific value from the list, use the remove() method. This method will remove the first occurrence of the given object in a list. Let’s demonstrate this by remove the number two from my\_list.

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

>>> my\_list.remove(2)

>>> print(my\_list)

**Output:**

[1,3,4]

* **insert();**

The insert() method inserts an element at a specified position within a list.

>>> fruits = [‘apple’, ‘banana’, ‘cherry’]

>>> fruits.insert(1, ‘orange’)

>>> print(fruits)

**Output:**

[‘apple’, ‘orange’, ‘banana’, ‘cherry’]

**Working with Lists:**

1. **Iterating over a list using loops**

The simplest and the most common way to iterate over a list is to use a for loop. This method allows us to access each element in the list directly.

**Example:** Print all elements in the list one by one using for loop.

a = [1,3,5,7,9]

for val in a:

print(val)

**Output:**

1

3

5

7

9

1. **Sorting and reversing a list using sort(), sorted(), and reverse().**

* **sort() method:**

The sort() method in Python is a built-in function that allows us to sort the elements of a list in ascending or descending order and it modifies the list in place which means there is no new list created. This method is useful when working with lists where we need to arranged the elements in a specific order, whether numerically of alphabetically.

Below is a simple example that use sort() method to arrange a list of integer values in ascending order.

**Example:**

a = [5,2,9,1,5,6]

a.sort()

print(a)

**Output:**

[1,2,5,5,6,9]

* **sorted() function:**

sorted() function returns a new sorted list from the elements of any iterable like (e.g., list, tuples, strings). It creates and returns a new sorted list and leaves the original iterable unchanged.

Let’s start with a basic example of sorting a list of numbers using the sorted() function.

**Example:**

a = [4,1,3,2]

b = sorted(a)

print(b)

**Output:**

[1,2,3,4]

1. **Basic list manipulations: addition, deletion, updating, and slicing.**

Python list methods are built-in function that allow us to perform various operations on lists, such as adding, removing, or modifying elements. In this article, we’ll explore all Python list methods with a simple example.

* **Addition (Appending and Inserting elements)**

You can add elements to a list using two common methods: append() and insert().

* + **append():** Adds an element to the end of the list.
  + **insert():** Adds an element at a specified position.

**Example:**

fruits = [‘apple’, ‘banana’, ‘orange’]

fruits.append(‘grapes’)

print(“After append:”,fruits)

fruits.insert(1, ‘mango’)

print(“After insert:”,fruits)

**Output:**

After append: [‘apple’, ‘banana’, ‘orange’, ‘grapes’]

After insert: [‘apple’, ‘mango’, ‘banana’, ‘orange’, ‘orange’]

* **Deletion (Removing elements)**

You can remove elements from a list using remove() or pop()

* + **remove():** Removes the first occurrence of a specific value.
  + **pop():** Removes and returns an element at a specified index.

**Example:**

fruits = [‘apple’, ‘mango’, ‘banana’, ‘orange’, ‘grapes’]

fruits.remove(‘banana’)

print(“After remove:”,fruits)

last\_fruit = fruits.pop()

print(“After pop:”,fruits)

print(“Popped element:”,last\_fruit)

**Output:**

After remove: [‘apple’, ‘mango’, ‘orange’, ‘grapes’]

After pop: [‘apple’, ‘mango’, ‘orange’]

Popped element: grapes

* **Updating (Modifying elements)**

You can update an element in the list by accessing its index and assigning a new value.

**Example:**

fruits = [‘apple’, ‘mango’, ‘orange’]

fruits[1] = ‘pineapple’

print(“After update:”, fruits)

**Output:**

After update: [‘apple’, ‘pineapple’, ‘orange’]

* **Slicing (Extracting parts of a list)**

You can extract a portion of a list using slicing by specifying a range of indices.

**Example:**

fruits = [‘apple’, ‘banana’, ‘orange’, ‘grapes’, ‘pineapple’]

sliced\_fruits = fruits[1:3]

print(“Sliced list (1:3):”, sliced\_fruits)

sliced\_fruits2 = fruits[:2]

print(“Sliced list (:2):”, sliced\_fruits2)

sliced\_fruits3 = fruits[2:]

print(“Sliced list (2:):”, sliced\_fruits3)

**Output:**

Sliced list (1:3): [‘banana’, ‘orange’]

Sliced list (:2): [‘apple’, ‘banana’]

Sliced list(2:): [‘orange’, ‘grapes’, ‘pineapple’]

**Tuple:**

1. **Introduction to tuples, immutability**

Python Tuple is a collection of objects separated by commas. A tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python tuple is immutable, unlike the Python list which is mutable.

**Example:**

t = (10,20,30)

print(t)

print(type(t))

**Output:**

(10,20,30)

<class ‘tuple’>

Unlike Python lists, tuples are immutable. Some characteristics of Tuples in Python.

* Like Lists, tuples are ordered and we can access their elements using their index values
* We cannot update items to a tuple once it is created.
* Tuples cannot be appended or extended.
* We cannot remove items from a tuple once it is created.

**Example:**

t = (1,2,3,4,5)

print(t[1])

print(t[4])

t = (1,2,3,4,2,3)

t[1] = 100

print(t)

**Output:**

2

5

(1,2,3,4,2,3)

Traceback (most recent call last):

File “Solution.py”, line 12, in <module>

T[1] = 100

TypeError: ‘tuple’ object does not support item assignment

1. **Creating and accessing elements in a tuple**

We can access elements in a tuple in the same way as we access elements in a list. Python follows 0-based indexing, so a tuple with n elements has indices from 0 through n-1. To access an index in a tuple, we use the index operator [ ].

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| ‘Welcome’ | ‘to’ | ‘Python’ | ‘Have’ | ‘a’ | ‘great’ | ‘day!’ |
| -7 | -6 | -5 | -4 | -3 | -2 | -1 |

**Example:**

x = (‘welcome’, ‘to’, ‘Python’)

print(x[0])

print(x[1])

print(x[2])

print(x[-1])

print(x[-2])

print(x[-3])

**Output:**

Welcome

to

Python

Python

to

Welcome

1. **Basic operations with tuples: concatenation, repetition, membership**

* **Concatenation:**

Concatenating tuples means joining two or more tuples together. You can concatenate tuples using the + operator.

**Example:**

tuple1 = (1,3,2)

tuple2 = (4,6,5)

result = tuple1 + tuple2

print(result)

**Output:**

**(1,3,2,4,6,5)**

* **Repetition**

Repetition involves repeating the elements of a tuple multiple times. You can use the \* operator to repeat a tuple a specific number of times.

**Example:**

tuple1 = (1,2,3)

result = tuple \* 3

print(result)

**Output:**

(1,2,3,1,2,3,1,2,3)

* **Membership**

Membership checks whether a particular element exists in a tuple. You can use the in or not in operators for this.

**Example:**

tuple1 = (1,2,3,4,5)

print(3 in tuple1)

print(6 not in tuple1)

**Output:**

True

True

**Accessing Tuples:**

1. **Accessing tuple elements using positive and negative indexing**

* **Positive Indexing**

As we have seen that tuple items have index, as such we can access items using these indexes.

**Example:**

country = (‘Spain’, ‘Italy’, ‘India’, ‘England’, ‘Germany’)

print(country[1])

print(country[3])

print(country[0])

**Output:**

Italy

England

Spain

* **Negative Indexing**

Similar to positive indexing, negative indexing is also used to access items, but from the end of the tuple. The last item has index [-1], second last item has index [-2], third last item has index [-3] and so on.

**Example:**

country = (‘Spain’, ‘Italy’, ‘India’, ‘England’, ‘Germany’)

print(country[-1])

print(country[-2])

print(country[-3])

**Output:**

Germany

India

Italy

1. **Slicing a tuple to access ranges of elements.**

You can also use slicing to access a range of items in a tuple. The syntax for slicing is tuple [start:stop:step].

* Start is the index at which the slice starts (inclusive).
* Stop is the index at which the slice ends (exclusive).
* Step defines the increment between elements in the slice(optional).

**Example:**

my\_tuple = (‘apple’, ‘banana’, ‘cherry’, ‘date’, ‘elderberry’)

print(“Sliced tuple (1:4):”, my\_tuple[1:4])

print(“Slicedtuple (:3):”, my\_tuple[:3])

print(“Sliced tuple (2:):”, my\_tuple[2:])

print(“Sliced tuple (::2):”, my\_tuple[::2])

**Output:**

Sliced tuple (1:4): (‘banana’, ‘cherry’, ‘date’

Sliced tuple (:3): (‘apple’, ‘banana’, ‘cherry’)

Sliced tuple (2:): (‘cherry’, ‘date’, ‘elderberry’)

Sliced tuple (::2): (‘apple’, ‘cherry’, ‘elderberry’)

**Dictionaries:**

1. **Introduction to dictionaries: key-value pairs.**

The task of adding a key-value pair to a dictionary in Python involves inserting new pairs or updating existing ones. This operation allows us to expand the dictionary by adding new entries or modify the value of an existing key.

**Example:**

d = {‘key1’: ‘Tops’, ‘key2’: ‘for’}, we add multiple key-value pairs at once: ‘key3’: ‘Tops’, ‘key4’: ‘is’, ‘key5’: ‘portal’, and ‘key6’: ‘Computer’. After the update, the dictionary becomes

{‘key1’: ‘Tops’, ‘key2’: ‘for’, ‘key3’: ‘Tops’, ‘key4’: ‘is’, ‘key5’: ‘portal’, ‘key6’: ‘Computer’}

1. **Accessing, adding, updating, and deleting dictionary elements.**

* **Accessing**

A dictionary in Python is a useful way to store data in pairs, where each key is connected to a value. To access an item in the dictionary, refer to its key name inside square brackets.

**Example:**

a = {‘Tops’: 3, ‘for’: 2, ‘tops’: 1}

x = a[‘tops’]

print(x)

**Output:**

1

* **Adding**

In this article, we will explore various methods to add new keys to a dictionary in Python. The simplest way to add a new key is by using Assignment operator (=).

Using = Assignment Operator

**Example:**

d = {‘a’: 1, ‘b’: 2}

d[‘c’] = 3

print(d)

**Output:**

{‘a’: 1, ‘b’: 2, ‘c’: 3}

* **Updating**

Python dictionary update() method updates the dictionary with the elements from another dictionary object or from an iterable of key/value paris.

**Example:**

d1 = {‘A’: ‘Tops’, ‘B’: ‘For’}

d2 = {‘C’: ‘Tops’, ‘D’: ‘Python’}

d1.update(d2)

d1.update(A=’Hello’)

print(d1)

**Output:**

{‘A’: ‘Hello’, ‘B’: ‘Tops’, ‘C’: ‘Python’}

* **Deleting**

We are given a dictionary and our task is to remove a specific key from it.

* + Using pop()

**Example:**

a = {‘name’: ‘ABC’, ‘age’: 18, ‘city’: ‘New York’}

rv = a.pop(‘age’)

print(a)

print(rv)

**Output:**

{‘name’: ‘ABC’, ‘city’: ‘New York’}

18

**Explanation:** pop() method removes the specified key (“age”) from the dictionary and also returns the value associated with the key allowing us to use it elsewhere if needed.

* + Using del()

**Example:**

a = {‘name’: ‘ABC’, ‘age’: 18, ‘city’: ‘New York’}

del a[‘city’]

print(a)

**Output:**

{‘name’: ‘ABC’, ‘age’: 18}

**Explanation:** del() statement directly removes the key-value pair for ‘city’ and does not return the value making it ideal when the value is not needed.

1. **Dictionary methods like keys(), values(), and items().**

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| Key | None | Returns a view of the keys in the dictionary |
| Values | None | Returns a view of the values in the dictionary |
| Items | None | Returns a view of the key-value pairs in the dictionary |

**Working with Dictionaries:**

1. **Iterating over a dictionary using loops.**

In this article, we will cover How to Iterate Through a Dictionary in Python. To Loop through values in a dictionary you can use built-in methods like values(), items() or even directly iterate over the dictionary to access values with keys.

To iterate through all values of a dictionary in Python using .values(), you can employee a for loop, accessing each value sequentially. This method allows you to process or display each individual value in the dictionary without explicitly referencing the corresponding keys.

**Example:**

**d =** {‘name’: ‘Tops’, ‘topic’: ‘dict’, ‘task’: ‘iterate’}

for val in d.values():

print(val)

1. **Merging two lists into a dictionary using loops or zip().**

We are given two lists, we need to convert both of the list into dictionary. We need to convert two list into a form of dictionary. We can do this using method like zip, dictionary comprehension, itertools.starmap.

**Using zip**

Use zip to pair elements from two lists, where the first list provides the keys and second provides the values after that we convert the zipped object into a dictionary using dict(). Which creates key-values pairs().

**Example:**

a = [‘name’, ‘age’, city’]

b = [‘ABC’, 18, ‘New York’]

res = dict(zip(a, b))

print(res)

**Output:**

{‘name’: ‘ABC’, ‘age’: 18, ‘city’: ‘New York’}

**Explanation:**

* Zip(a, b) pairs each element from list a with the corresponding element from list b, creating tuples of key-value pairs.
* dict() function is used to convert the zipped pairs into a dictionary where elements from a become the keys and elements from b become values

**Using Loop:**

a = [‘name’, ‘age’, ‘city’]

b = [‘ABC’, 18, ‘New York’]

res = {}

for key, value in zip(a, b):

res[key] = value

print(res)

**Output:**

{‘name’: ‘ABC’, ‘age’: 18, ‘city’: ‘New York’}

1. **Counting occurrences of characters in a string using dictionaries**

This code helps count the number of items each character appears in a given string. By using a Python dictionary, the code creates a key-value pair where the character is the key and the frequency of its appearance is the corresponding value. The output is displayed by printing each key-value pair.

**Example:**

st = input(“Enter a string:”)

dict = {}

for ch in st:

if ch in dict:

dict[ch] += 1

else:

dict[ch] = 1

for key in dict:

print(key, ‘:’, dict[key])

**Functions:**

1. **Defining functions in Python.**

Python Function is a block of statement that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Some Benefits of Using Functions

* Increase Code Readability
* Increase Code Reusability

Python Function Declaration

def function\_name(parameters):

# statement # Body of Statement

return expression

Function return

**Example:**

def fun():

print(“Welcome to GFG”)

fun()

**Output:**

Welcome to GFG

1. **Different types of function: with/without parameters, with/without return values.**

* Built-in library function: These are Standard functions in Python that are available to use.
* User-defined function: We can create our own functions based on our requirements.

**Creating a Function in Python**

We can define a function in Python, using the def keyword. We can add any type of functionalities and properties to it as we require. By the following example, we can understand how to write a function in Python. In this way we can create Python function definition by using def keyword.

**Example:**

def fun():

print(“Welcome to GFG”)

fun()

**Output:**

Welcome to GFG

**Python Function with Parameters**

If you have experience in C/C++ or Java then you must be thinking about the return type of the function and data type of arguments. That is possible in Python as well.

**Syntax:**

def function\_name(parameter: data\_type) -> return\_type

“””Docstring”””

# body of the function

Return expression

The following example uses arguments and parameters that you will learn later in this article so you can come back to it again if you understood.

**Example:**

def add(num1: int, num2: int) ->

“””Add two numbers”””

num3 = num1 +num2

return num3

num1, num2 = 5, 15

ans = add(num1, num2)

print(f“The addition of {num1} and {num2} results {ans}.”)

**Output:**

The addition of 5 and 15 results 20.

1. **Anonymous functions (lambda functions).**

Python Lambda Functions are anonymous functions means that the function is without a name. As we already know the def keyword is used to define a normal function in Pyhon. Similarly, the lambda keyword is used to define an anonymous function in Python.

In the example, we defined a lambda function(upper) to convert a string to its upper case using upper().

**Example:**

s1 = ‘Tops Technologies’

s2 = lambda func: func.upper()

print(s2(s1))

**Output:**

TOPS TECHNOLOGIES

**Modules:**

1. **Introduction to Python modules and importing modules**

Python Module is a file that contains built-in functions, classes, its and variables. There are many Python modules, each with its specific work.

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

**Create a Python Module**

To create a Python module, write the desired code and save that in a file with .py extension. Let’s understand it better with an example:

**Example:**

Let’s create a simple calc.py in which we define two functions, one add and another subtract.

def add(x, y):

return (x + y)

def subtract(x, y):

return (x-y)

**Import module in Python**

We can import the functions, and classes defined in a module to another module using the import statement in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

**Example:**

import cacl

print(calc.add(10,2))

**Output:**

12

**Python Import From Module**

Python’s from statement lets you import specific attributes from a module without importing the module as a whole.

**Import Specific Attributes from a Python module**

Here, we are importing specific sqrt and factorial attributes from the math module.

**Example:**

from math import sqrt, factorial

print(sqrt(16))

print(factorial(6))

**Output:**

4.0

720

**Import all Names**

The \* symbol used with the import statement is used to import all the names from a module to a current namespace.

**Syntax:**

from module\_name import \*

The use of \* has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use \*, else do so.

**Example:**

from math import \*

print(sqrt(16))

print(factorial(6))

**Output: 4.0 720**

1. **Standard library modules: math, random.**

The math module is Python provides access to mathematical functions, while the random module generates pseudo-random numbers for various distributions.

**Math Module:**

Purpose:

The math module offers a wide range of mathematical functions, including trigonometric functions (like sin, cos, tan), logarithmic functions (like log, log10), and other mathematical operations.

**Exmaple:**

* math.sqrt(x): Returns the square root of x.
* math.sin(x): Returns the sine of x (in radians)
* math.log(x, base): Returns the logarithm of x to the given base.
* math.pi: Returns the value of pi.
* math.e Returns the value of Euler’s number€.

**Use Cases:**

Scientific calculations, engineering applications, and any scenario requiring mathematical operations.

**Random Module:**

**Purpose:**

The random module provides functions for generating pseudo-random numbers, which are useful for simulations, games, random sampling, and other applications where unpredictability is needed.

**Example:**

* random.randint(a, b): Returns a random integer between a and b (inclusive).
* random.random(): Returns a random floating-point number between 0.0 and 1.0.
* random.choice(sequence): Returns a random element from a sequence.
* random.shuffle(sequence): Shuffles the elements of a sequence in place.
* random.sample(population, k): Returns a list of k unique elements chosen randomly from the population.

**Use Cases:**

Simulations, games, data sampling, and any application requiring random behavior.

1. **Creating custom modules.**

To create a custom module in Python, save your code in a .py file; the filename becomes the module name. You can then import and use this module in other Python scripts.

Here’s a more detailed explanation:

1. **Create a Module File:**

* Open a text editor or your preferred IDE and create a new file.
* Save the file a .py extension (e.g., my\_module.py)
* This file will contain your Python code(functions, classes, variables, etc.) that you want to organize as a module.

1. **Example Module(my\_module.py):**

# my\_module.py

def greet(name):

“””Greets a person.”””

print(f”Hello, {fname}!”)

def add(a, b):

“””Adds two numbers.”””

Return a + b

1. **Import and Use the Module:**

* In another Python script or the Python interpreter, import the module using import.
* Use the module name followed by dot(.) to access its functions or variables.

# main.py

import my\_module

my\_module.greet(“Alice”) *# Output: Hello, Alice!*

Result = my\_module.add(5, 3)

Print(result) *# Output: 8*

1. **Importing Specific Elements:**

* You can import specific functions or variables from a module using from module import item.
* This allows you to use the imported item directly without the module name prefix.

# main.py

from my\_module import greet, add

greet(“Bob”) *# Output: Hello, Bob!*

result = add(10, 2)

print(result) *# Output: 12I*

1. **Aliasing Modules:**

* You can assign an alias to a module using import module as alias.
* This can be useful for brevity or to avoid naming conflicts.

# main.py

Import my\_module as mm

mm.greet(“Charlie”) *# Output: Hello, Charlie!*

1. **Packages (Organizing Modules):**

* If you have a group of related modules, you can organize them into a package.
* A package is a directory containing Python modules and \_\_init\_\_.py file(Which can be empty).
* This helps is structuring larger projects.