1. **Accessing List**

• Understanding how to create and access elements in a list.

To create a list, you use square brackets and separate the elements with commas. Lists are ordered collections, meaning the elements are stored in a specific sequence. Accessing elements in a list is done using an index, which starts from 0 for the first element, 1 for the second, and so on. You can also use negative indices to access elements from the end of the list, where -1 refers to the last element. Lists are mutable, meaning you can change an element by assigning a new value to a specific index. Additionally, you can extract parts of a list using slicing, which allows you to specify a range of indices to create a sublist. Iterating through a list is possible with loops, which enable you to access each element sequentially. The len() function can be used to determine the number of elements in the list.

• Indexing in lists (positive and negative indexing).

In lists, **indexing** is the way we access individual elements based on their position in the list. Python supports both **positive** and **negative indexing** to allow flexible access to list elements.

**1. Positive Indexing**

Positive indexing starts at **0** for the first element, **1** for the second element, and so on. This is the most common way to access list elements. The index increases by 1 as you move from left to right through the list.

* **Index 0** refers to the first element.
* **Index 1** refers to the second element.
* **Index 2** refers to the third element, and so on.

**2. Negative Indexing**

Negative indexing, on the other hand, allows you to access list elements starting from the end. The last element has an index of **-1**, the second-to-last element has an index of **-2**, and so on. Negative indexing is useful when you want to access elements at the end of the list without needing to know the exact length of the list.

* **Index -1** refers to the last element.
* **Index -2** refers to the second-to-last element.
* **Index -3** refers to the third-to-last element, and so on.

**Example:**

For a list like my\_list = [10, 20, 30, 40, 50], you can access elements using both positive and negative indices.

* **Positive indexing**:
  + my\_list[0] will return 10 (first element)
  + my\_list[2] will return 30 (third element)
  + my\_list[4] will return 50 (fifth element)
* **Negative indexing**:
  + my\_list[-1] will return 50 (last element)
  + my\_list[-3] will return 30 (third-to-last element)
  + my\_list[-5] will return 10 (first element)

**Key Points:**

* Positive indices begin from 0 and move from left to right.
* Negative indices begin from -1 and move from right to left.
* Both types of indexing can be used interchangeably, and their use depends on your needs, such as whether you're working with elements at the end of the list.

• Slicing a list: accessing a range of elements.

**Slicing a List** allows you to access a range of elements from a list. The syntax is:

python

Copy code

list[start:end]

* **start**: Index where the slice begins (inclusive).
* **end**: Index where the slice ends (exclusive).
* **Omitting** start defaults to the beginning, and omitting end defaults to the end.

**2. List Operations**

• Common list operations: concatenation, repetition, membership.

1. **Concatenation (+ operator)**  
Combines two or more lists into a single new list.

2. **Repetition (\* operator)**  
Creates a new list by repeating the elements of an existing list a specified number of times.

3. **Membership (in and not in operators)**  
Checks whether a specific element is present in a list, returning a boolean result (True or False).

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

1. **append()**  
Adds a single element to the end of the list. Modifies the original list.

2. **insert()**  
Inserts an element at a specified index. Moves existing elements to accommodate the new one.

3. **remove()**  
Removes the first occurrence of a specified element from the list. Raises an error if the element is not found.

4. **pop()**  
Removes and returns the element at a specified index (default is the last element). Modifies the original list.

**4. Tuple**

• Introduction to tuples, immutability.

 **Definition:**  
A tuple is an ordered, immutable collection of elements. Tuples are similar to lists but use parentheses () instead of square brackets [].

 **Immutability:**  
Once a tuple is created, its elements cannot be modified, added, or removed. This makes tuples suitable for situations where data integrity and consistency are important.

 **Key Features:**

Ordered: Elements maintain their sequence.

Heterogeneous: Can store different data types.

Hashable: Can be used as dictionary keys if they contain only immutable elements.

• Creating and accessing elements in a tuple.

**Creating a Tuple:**

* **Basic Tuple:**  
  Use parentheses () to define a tuple.

python

Copy code

my\_tuple = (1, 2, 3)

* **Single-element Tuple:**  
  Include a trailing comma to differentiate it from a regular variable.

python

Copy code

single\_tuple = (1,)

* **Without Parentheses:**  
  Parentheses are optional in many cases, and a tuple can be created using just commas.

python

Copy code

implicit\_tuple = 1, 2, 3

**Accessing Tuple Elements:**

* **By Index:**  
  Use square brackets with an index (starting from 0) to access elements.

python

Copy code

my\_tuple[0] # Accesses the first element

* **Negative Indexing:**  
  Use negative indices to access elements from the end of the tuple.

python

Copy code

my\_tuple[-1] # Accesses the last element

* **Slicing:**  
  Retrieve a subset of elements using slicing syntax (start:stop:step).

python

Copy code

my\_tuple[1:3] # Returns elements from index 1 to 2

Tuples provide a straightforward way to store ordered, immutable collections and can be easily accessed using indexing and slicing.

4o

• Basic operations with tuples: concatenation, repetition, membership.

1. **Concatenation (+ operator):**  
Combines two or more tuples to create a new tuple.

* Tuples are not modified; a new tuple is formed.

2. **Repetition (\* operator):**  
Creates a new tuple by repeating the elements of an existing tuple a specified number of times.

* The original tuple remains unchanged.

3. **Membership (in and not in operators):**  
Checks if an element is present in a tuple, returning a boolean value (True or False).

* Useful for verifying the existence of an element within a tuple.

**5. Accessing Tuples**

• Accessing tuple elements using positive and negative indexing.

**Positive Indexing:**

* Accesses elements from the beginning of the tuple.
* Indexing starts at **0** for the first element.
  + Example: In the tuple (10, 20, 30), index 0 refers to 10.

**Negative Indexing:**

* Accesses elements from the end of the tuple.
* Indexing starts at **-1** for the last element.
  + Example: In the tuple (10, 20, 30), index -1 refers to 30.

• Slicing a tuple to access ranges of elements.

**Definition:**  
Slicing allows you to extract a subset of elements from a tuple using the syntax:

python

Copy code

tuple[start:stop:step]

**Components of Slicing:**

1. **start:** The index where the slice begins (inclusive). Defaults to 0 if omitted.
2. **stop:** The index where the slice ends (exclusive). Defaults to the length of the tuple if omitted.
3. **step:** The interval between elements. Defaults to 1 if omitted.

**Behavior of Slicing:**

* **Omitting start:** The slice starts from the beginning of the tuple.
* **Omitting stop:** The slice goes up to the end of the tuple.
* **Negative step:** Reverses the tuple while slicing.

**6. Dictionaries**

• Introduction to dictionaries: key-value pairs.

**Definition:**  
A dictionary is a collection in Python that stores data as key-value pairs. It allows you to associate unique keys with corresponding values, enabling fast data retrieval.

**Key Concepts:**

* **Keys:**  
  Must be unique, immutable objects (e.g., strings, numbers, tuples).
* **Values:**  
  Can be any data type (e.g., strings, numbers, lists, other dictionaries).
* **Syntax:**  
  Dictionaries are defined using curly braces {} with key-value pairs separated by colons:

python

Copy code

my\_dict = {"name": "Alice", "age": 25}

**Key Features:**

* **Unordered (before Python 3.7):** Elements do not maintain a specific order.
* **Ordered (from Python 3.7+):** Elements retain their insertion order.
* **Mutable:** You can add, update, or remove key-value pairs.

Dictionaries are widely used for data storage and quick

• Accessing, adding, updating, and deleting dictionary elements.

**1. Accessing Elements:**

1. You can access dictionary values using their corresponding keys either by using square brackets ([]) or with the .get() method.

**2. Adding Elements:**

* To add a new key-value pair, simply assign a value to a new key using square brackets.

**3. Updating Elements:**

* To update an existing key-value pair, assign a new value to the key. If the key does not exist, it will be added.

**4. Deleting Elements:**

* You can remove elements using:
  + del to delete a specific key-value pair.
  + .pop() to remove a key-value pair and return the value.
  + .popitem() to remove and return the last inserted key-value pair.

• Dictionary methods like keys(), values(), and items().

**1. keys() Method:**

* Returns a view object that displays all the keys in the dictionary.
* You can iterate over the keys using this view.

**2. values() Method:**

* Returns a view object that displays all the values in the dictionary.
* You can iterate over the values using this view.

**3. items() Method:**

* Returns a view object that displays all the key-value pairs in the dictionary as tuples.
* This allows you to iterate over both keys and values simultaneously.

**7. Working with Dictionaries**

• Iterating over a dictionary using loops.

1. **Iterating Over Keys:**

* You can loop through a dictionary to access its keys by default, or by explicitly using the .keys() method.

2. **Iterating Over Values:**

* You can loop through the values of a dictionary using the .values() method.

3. **Iterating Over Key-Value Pairs:**

* To access both keys and their corresponding values simultaneously, you can use the .items() method

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

 **Using a Loop:**

* You can iterate over two lists simultaneously, creating key-value pairs from corresponding elements of each list to merge them into a dictionary.

 **Using zip():**

* The zip() function combines two lists into pairs of elements, which can then be converted into a dictionary.

• Counting occurrences of characters in a string using dictionaries.

To count the occurrences of each character in a string, you can use a dictionary where the keys are characters, and the values are their respective counts.

* **Approach:**
  + Iterate over each character in the string.
  + For each character, check if it exists in the dictionary.
    - If it does, increment its count.
    - If it doesn't, add it to the dictionary with an initial count of 1.

**8. functions**

• Defining functions in Python.

In Python, functions are defined using the def keyword followed by the function name and parentheses. The body of the function contains the code to be executed when the function is called.

**Syntax:**

python

Copy code

def function\_name(parameters):

# Function body

# Code to execute

return result # Optional

* **function\_name:** The name of the function, which should follow Python's naming rules.
* **parameters:** Optional inputs (arguments) that the function can accept. If no parameters are needed, you can leave the parentheses empty.
* **return:** The return statement is optional and is used to return a value from the function. If omitted, the function returns None by default.

• Different types of functions: with/without parameters, with/without return values.

1. **Functions with Parameters:**

* These functions accept inputs (arguments) when called, which are then used in the function body.
* **With parameters and with return value:** Functions that take inputs and return a value based on those inputs.
* **With parameters and without return value:** Functions that take inputs but do not return any value; they might perform an action instead.

2. **Functions without Parameters:**

* These functions do not take any arguments when called.
* **Without parameters and with return value:** Functions that do not take any inputs but return a value when executed.
* **Without parameters and without return value:** Functions that do not take any inputs and do not return any value; they usually perform an action or print something.

• Anonymous functions (lambda functions).

Anonymous functions, or **lambda functions**, are small, unnamed functions defined using the lambda keyword. They are typically used for short-term operations where defining a full function would be unnecessarily verbose.

**Syntax:**

python

Copy code

lambda arguments: expression

* **arguments:** The inputs to the lambda function, which can be zero or more.
* **expression:** A single expression that the lambda function computes and returns. Unlike regular functions, a lambda function can only contain one expression.

**Key Characteristics:**

* **Anonymous:** Lambda functions are not named and are often used in places where the function is only needed temporarily.
* **Concise:** They are a compact way to define simple functions in a single line.
* **Limited functionality:** Lambda functions can only execute a single expression and cannot include multiple statements or complex logic.

**9.** **Modules**

• Introduction to Python modules and importing modules.

Anonymous functions, or **lambda functions**, are small, unnamed functions defined using the lambda keyword. They are typically used for short-term operations where defining a full function would be unnecessarily verbose.

**Syntax:**

python

Copy code

lambda arguments: expression

* **arguments:** The inputs to the lambda function, which can be zero or more.
* **expression:** A single expression that the lambda function computes and returns. Unlike regular functions, a lambda function can only contain one expression.

**Key Characteristics:**

* **Anonymous:** Lambda functions are not named and are often used in places where the function is only needed temporarily.
* **Concise:** They are a compact way to define simple functions in a single line.
* **Limited functionality:** Lambda functions can only execute a single expression and cannot include multiple statements or complex logic.

**Example use cases:**

* **With map(), filter(), and sorted()**: Lambda functions are often used for quick operations like transformations, filtering, or sorting.

• Standard library modules: math, random.

Python's **standard library** comes with many built-in modules that provide various functionalities. Two commonly used modules are math and random.

**1. math Module:**

* The math module provides mathematical functions and constants.
* It includes functions for basic mathematical operations, trigonometry, logarithms, and more.

**Common Functions in math:**

* math.sqrt(x) – Returns the square root of x.
* math.pow(x, y) – Returns x raised to the power of y.
* math.sin(x), math.cos(x), math.tan(x) – Trigonometric functions (radians).
* math.log(x, base) – Returns the logarithm of x to the given base.
* math.factorial(x) – Returns the factorial of x.
* math.pi – Constant for the value of Pi.

**2. random Module:**

* The random module implements pseudo-random number generators and randomization tools.
* It provides functions for generating random numbers, choosing random items, and shuffling sequences.

**Common Functions in random:**

* random.random() – Returns a random float between 0 and 1.
* random.randint(a, b) – Returns a random integer between a and b (inclusive).
* random.choice(sequence) – Returns a random element from the sequence.
* random.shuffle(sequence) – Shuffles the elements of a list in place.
* random.sample(sequence, k) – Returns a random sample of k unique elements from a sequence.

Both math and random modules are part of Python’s standard library, meaning you don’t need to install them separately. These modules help simplify many common mathematical and randomization tasks.

• Creating custom modules.

A **custom module** is a Python file that contains your own functions, variables, or classes, which you can reuse across different programs.

**Steps to Create and Use a Custom Module:**

1. **Create a Python File (Module):**
   * Create a .py file containing the functions, variables, or classes you want to include in the module.
2. **Define Functions, Variables, or Classes:**
   * Inside the module file, define the necessary functions, variables, or classes.
3. **Import the Custom Module:**
   * In other Python scripts, use the import statement to access the contents of the custom module.

**Benefits of Custom Modules:**

* **Code Reusability:** You can use the same module across different programs.
* **Organization:** Group related functionality into separate files for better structure.
* **Maintainability:** Updating a module only requires changes in one place.

Custom modules allow you to organize and reuse your code efficiently.