# Module 2

## 1. Accessing List

- **Understanding how to create and access elements in a list:** A list in Python is one of the most versatile and commonly used data structures. It is an ordered collection of items, which means the items are stored in a specific sequence. Lists are mutable, meaning you can change, add, or remove items after the list has been created. To create a list, you place comma-separated values inside square brackets. For example, the syntax my_list = [1, 2, 3] creates a list of integers. Lists can hold different data types together, such as integers, strings, and floats, all in the same list. For example, mixed_list = [1, "Hello", 3.14]. Accessing an element is done by referring to its position number, or index.
- **Indexing in lists (positive and negative indexing):** Indexing is the method used to fetch a specific item from the list. Python uses zero-based indexing, which means the first item in the list is at index 0, the second item is at index 1, and so on.
    - **Positive Indexing:** This counts from the beginning of the list.
      fruits = ["apple", "banana", "cherry"]
      print(fruits[0]) # Output: apple

    - **Negative Indexing:** Python also supports negative indexing, which allows you to access items starting from the end of the list. The index -1 refers to the last item, -2 refers to the second-to-last item, etc. This is very useful when you do not know the exact length of the list but need the last element.
      print(fruits[-1]) # Output: cherry

- **Slicing a list:** Slicing is a technique used to retrieve a specific range of items from a list rather than just a single item. The syntax for slicing is list_name[start:stop:step]. The start index is inclusive (the slice starts here), but the stop index is exclusive (the slice stops right before this index). If you omit the start index, it defaults to 0. If you omit the stop index, it goes to the end of the list.
  numbers = [0, 1, 2, 3, 4, 5]
  print(numbers[1:4]) # Output: [1, 2, 3]

  This feature is incredibly powerful for data manipulation, allowing you to easily extract subsets of data for analysis or processing without writing complex loops.

## 2. List Operations

- **Common list operations:** Lists support several operations that allow you to combine or check their contents easily.
  - **Concatenation:** You can join two lists together using the + operator. This creates a new list containing elements from both.
    ```
    list1 = [1, 2]
    list2 = [3, 4]
    combined = list1 + list2 # Result: [1, 2, 3, 4]
    ```
  - **Repetition:** You can repeat the elements of a list multiple times using the * operator.
    ```
    zeros = [0] * 5 # Result: [0, 0, 0, 0, 0]
    ```
  - **Membership:** You can check if a specific item exists in a list using the in keyword. This returns a Boolean value (True or False).
    ```
    if "apple" in fruits:
       print("Found it!")
    ```
- **Understanding list methods:** Python provides built-in methods to modify lists efficiently.
  - **append():** Adds a single item to the end of the list.
    ```
    fruits.append("orange")
    ```
  - **insert():** Adds an item at a specific index, shifting other items to the right.
    ```
    fruits.insert(1, "mango") # Inserts mango at index 1
    ```
  - **remove():** Removes the first occurrence of a specific value. If the value is not found, it raises an error.
    ```
    fruits.remove("banana")
    ```
  - **pop():** Removes and returns the item at a specific index. If no index is specified, it removes the last item. This is often used to process items in a "Last In, First Out" manner.
    ```
    last_fruit = fruits.pop()
    ```

These methods modify the original list directly rather than creating a new one, which is important for memory management when working with large collections.

## 3. Working with Lists

- **Iterating over a list using loops:** Iteration is the process of going through each item in a list one by one. The for loop is the most common way to do this in Python. It is designed to work directly with sequences.
  ```
  colors = ["red", "green", "blue"]
  for color in colors:
      print(color)
  ```

  This loop assigns the value of the next item in the list to the variable color in each iteration until the end of the list is reached. You can also iterate using index numbers with range() and len() if you need to modify items based on their position:
  ```
  for i in range(len(colors)):
      print(f"Index {i} is {colors[i]}")
  ```

- **Sorting and reversing a list:** Python makes ordering data very simple.
  - **sort():** This method sorts the list in place, meaning the original list is changed.
    ```
    numbers = [3, 1, 2]
    numbers.sort() # numbers is now [1, 2, 3]
    ```

  - **sorted():** This function returns a *new* sorted list and leaves the original list unchanged.
    ```
    new_numbers = sorted(numbers)
    ```

  - **reverse():** This method reverses the order of elements in the list in place.
    ```
    numbers.reverse()
    ```

- **Basic list manipulations:** Beyond sorting, you perform many manipulations.
  - **Addition:** Using append() or extend() (which adds elements from another list).
  - **Deletion:** Using del statement to remove an item by index, or clear() to empty the entire list.
    ```
    del numbers[0] # Removes first item
    ```

  - **Updating:** You can change the value of a specific item by accessing its index and assigning a new value.
    ```
    numbers[1] = 99 # Changes the second item to 99
    ```

- **Slicing:** You can create a new list containing a subset of items from the original list using the slice syntax [start:stop].

  subset = numbers[1:3] # Creates a new list with items from index 1 to 2

These manipulations form the backbone of data processing tasks, allowing you to clean, organize, and restructure data dynamically.

## 4. Tuple

- **Introduction to tuples, immutability:** A tuple is a collection data type that is similar to a list but has one critical difference: it is immutable. Immutability means that once a tuple is created, you cannot change, add, or remove items from it. This makes tuples faster and safer than lists for storing data that should not change, such as geographical coordinates or configuration settings. Tuples are defined using parentheses () instead of square brackets.
  my_tuple = (1, 2, 3)

  If you try to assign my_tuple[0] = 5, Python will raise a TypeError. This protection ensures data integrity throughout the life of the program.
- **Creating and accessing elements in a tuple:** Creating a tuple is straightforward. You can create an empty tuple t = () or one with items. Interestingly, a tuple with a single item requires a trailing comma, like t = (1,), to distinguish it from a mathematical expression in parentheses. Accessing elements works exactly the same way as lists, using zero-based indexing.
  coordinates = (10.5, 20.8)
  x = coordinates[0]

- **Basic operations with tuples:** Although tuples are immutable, you can still perform operations that do not modify the original tuple but create new ones.
  - **Concatenation:** You can join two tuples using +.
    t1 = (1, 2)
    t2 = (3, 4)
    t3 = t1 + t2 # Result: (1, 2, 3, 4)

  - **Repetition:** You can repeat tuple elements using *.
    t_rep = ("Hi",) * 3 # Result: ("Hi", "Hi", "Hi")

  - **Membership:** You can check for existence using in.
    if 1 in t1:
      print("Exists")

Because they are hashable (unlike lists), tuples can also be used as keys in dictionaries, which is a unique advantage over lists.

## 5. Accessing Tuples

- **Accessing tuple elements using positive and negative indexing:** Just like strings and lists, tuples support precise indexing. Positive indexing starts from 0 at the beginning.
  data = ("Apple", "Banana", "Cherry")
  print(data[1]) # Output: Banana

  Negative indexing allows you to access items relative to the end of the tuple. This is particularly useful for getting the last element without calculating the length of the tuple.
  print(data[-1]) # Output: Cherry (last item)
  print(data[-2]) # Output: Banana (second to last)

  Attempting to access an index that does not exist (e.g., index 10 in a tuple of size 3) will result in an IndexError.
- **Slicing a tuple:** Slicing allows you to create a new tuple containing a subset of elements from the original tuple. The syntax follows the standard [start:stop:step] format.
  - **Range:** data[0:2] returns ("Apple", "Banana"). It includes index 0 but excludes index 2.
  - **Omission:** data[1:] starts at index 1 and goes to the very end. data[:2] starts at the beginning and stops before index 2.
  - **Step:** You can skip elements. data[::2] would return every second element.

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
even_subset = numbers[::2] # Returns (0, 2, 4, 6, 8)
Importantly, slicing a tuple returns a new tuple. It does not modify the original because tuples cannot be changed. This ensures that the original data remains safe while you work with the extracted portion.

## 6. Dictionaries

- **Introduction to dictionaries:** A dictionary is a built-in Python data structure that stores data in key-value pairs. Unlike lists which are indexed by numbers, dictionaries are indexed by keys, which can be any immutable type (like strings or numbers). This makes dictionaries ideal for storing structured data representing real-world objects, like a user profile where "name" maps to "John" and "age" maps to 30.
  student = {"name": "Alice", "grade": "A"}

  Dictionaries are mutable, unordered (historically, though ordered by insertion in modern Python), and do not allow duplicate keys.
- **Accessing, adding, updating, and deleting dictionary elements:**
  - **Accessing:** You access values by referring to the key inside square brackets.
    print(student["name"]) # Output: Alice

    If the key doesn't exist, this raises an error. A safer way is using .get(), which returns None instead of crashing.
  - **Adding/Updating:** You assign a value to a key. If the key exists, the value is updated. If not, the key-value pair is added.
    student["age"] = 20 # Adds new key 'age'
    student["grade"] = "A+" # Updates existing key 'grade'

  - **Deleting:** You can use the del keyword or the pop() method.
    del student["grade"] # Removes the key 'grade'

- **Dictionary methods:**
  - **keys():** Returns a view object containing all the keys in the dictionary. Useful for iterating over just the labels.
  - **values():** Returns a view object containing all the values. Useful if you don't care about the keys.
  - **items():** Returns a view object containing tuples of (key, value) pairs. This is the most common way to loop through a dictionary.

```
for key, value in student.items():
    print(f"{key}: {value}")
```

These methods provide powerful ways to inspect and manipulate the contents of a dictionary efficiently.

## 7. Working with Dictionaries

- **Iterating over a dictionary using loops:** While you can iterate over lists directly, iterating over dictionaries offers more options because you have both keys and values. By default, a standard for loop iterates over the keys.
- prices = {'apple': 1.0, 'banana': 0.5}
  for k in prices:
  print(k) # Prints 'apple', then 'banana'

- However, it is often more useful to iterate over both at once using the .items() method, which unpacks the key and value into two variables.
- for fruit, price in prices.items():
  print(f"The price of {fruit} is ${price}")

- **Merging two lists into a dictionary:** A common task is converting two separate lists (one of keys, one of values) into a single dictionary. You can do this using a loop or the zip() function.
    - **Using Loops:** You can iterate through the length of the lists and assign keys to values manually.
- keys = ['name', 'age']
  values = ['Bob', 25]
  user = {}
  for i in range(len(keys)):
  user[keys[i]] = values[i]

    - **Using zip():** The most Pythonic way is using the zip() function, which pairs elements efficiently.
- user_info = dict(zip(keys, values))

- This method handles lists of different lengths by stopping at the shortest one, making it safer and more concise than manual looping.
- **Counting occurrences of characters in a string:** Dictionaries are excellent for counting frequency because the key represents the item and the value represents the count.
- text = "hello"
  counts = {}
  for char in text:

```
    if char in counts:
        counts[char] += 1
    else:
        counts[char] = 1
```

- In this logic, we check if the character is already a key in the dictionary. If it is, we increment the count. If not, we initialize it with 1. The result would be {'h': 1, 'e': 1, 'l': 2, 'o': 1}. This pattern is fundamental in data analysis tasks like word frequency analysis.

## 8. Functions

- **Defining functions in Python:** Functions are the building blocks of organized code. A function is a block of code that only runs when it is called. You can pass data, known as parameters, into a function, and a function can return data as a result. You define a function using the def keyword, followed by the function name and parentheses.

  ```
  def greet_user():
      print("Hello!")
  ```

  Functions promote code reusability (writing code once and using it many times) and modularity (breaking complex problems into smaller, manageable pieces).
- **Different types of functions:**
  - **With Parameters:** These accept input values to work with.

    ```
    def add(a, b): # a and b are parameters
        print(a + b)
    ```

  - **Without Parameters:** These perform a static task.
  - **With Return Values:** These send a result back to the caller using the return keyword. The function stops executing immediately after the return statement.

    ```
    def square(x):
        return x * x
    result = square(5) # result becomes 25
    ```

  - **Without Return Values:** These perform an action (like printing) but implicitly return None.
- **Anonymous functions (lambda functions):** Python allows you to create small, unnamed functions called lambda functions. These are defined using the lambda keyword and are typically used for short operations that are passed as arguments to other functions (like sorting or filtering). A lambda function can take any number of arguments, but can only have one expression.

  ```
  # Syntax: lambda arguments : expression
  double = lambda x: x * 2
  print(double(5)) # Output: 10
  ```

  While regular functions defined with def are preferred for complex logic, lambdas are perfect for quick, throwaway calculations inside tools like map() or filter().

## 9. Modules

- **Introduction to Python modules and importing modules:** A module is simply a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Modules allow you to organize your Python code logically. Grouping related code into a module makes the code easier to understand and use. To use the code in a module, you use the import statement.
  import my_module
  my_module.my_function()

  You can also import specific parts of a module using from ... import ... or give a module an alias using as.
  import pandas as pd

- **Standard library modules:** Python comes with a "battery included" philosophy, meaning it has a rich standard library of modules available immediately after installation.
  - **math:** Provides access to mathematical functions defined by the C standard.
    import math
    print(math.sqrt(16)) # Output: 4.0
    print(math.pi)  # Output: 3.14159...

  - **random:** Implements pseudo-random number generators for various distributions.
    import random
    print(random.randint(1, 10)) # Random integer between 1 and 10
    print(random.choice(['win', 'lose'])) # Pick random item

- **Creating custom modules:** You can create your own modules to organize your code. If you save a file named calculator.py with function definitions for adding and subtracting, you can then import that file in another script in the same directory using import calculator.
  # In main.py
  import calculator
  calculator.add(5, 3)

  This is essential for large projects, allowing developers to split thousands of lines of code into separate files based on functionality (e.g., database logic, user

interface logic, utility functions).