Chapter IV

Data Structures

1. Lists and Tuples:

In programming, a data structure is a way of organizing and sorting data so that it can be accessed and manipulated efficiently.

For mathematics students, you can think of data structures as computational equivalents of:

- Sequences.
- Sets.
- Mapping (Applications).
- Collections of values.

1.1 Lists

A list is a **mutable ordered collection** of elements, so it's similar to a finite sequence in mathematics. Lists are perfect for mathematical sequences, matrices, or datasets that need modification.

1.1.1 Creation:

To create a list, just use a square-brackets:

```
numbers = [1, 2, 3, 4, 5]
print(type(numbers))
print(numbers)
```

it gives: A one-dimensional array.

1	2	3	4	5

```
matrix = [[1,2],
     [3, 4]]
print(type(matrix))
print(matrix)
```

It gives: A two-dimensional array (Matrix).

1	2
3	4

1.1.2 Indexing:

Each element in a list has a position called an **index.**

Indexing starts from 0 in python.

```
numbers = [10, 20, 30, 40]    #A list
print(numbers[0])   #10
print(numbers[2])   #30
```

Negative indexing counts from the end:

```
numbers = [10, 20, 30, 40]    #A list
print(numbers[-1])   #40
print(numbers[-2])   #30
```

1.1.3 Slicing:

Slicing allows us to extract a portion (a subsequence) of a list:

[start:end:step]

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    #A list
print(numbers)
print(numbers[2:5])   #[2, 3, 4]
print(numbers[:3])   #[0, 1, 2] (From start)
print(numbers[::2])   #[0, 2, 4, 6, 8] (Every second)
```

1.1.4 Concatenation:

Concatenation combines two lists using + operator, and create a new list without modifying the originals:

```
list1 = [0, 1, 2, 3]        #A list
list2=[4, 5, 6, 7, 8, 9]    #A list
list3 = list1 + list2    #A combined list
print(list1)
print(list2)
print(list3)   #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.2Tuples

A tuple is an **immutable ordered collection** of elements, once created, their contents cannot change. Tuples are ideal for fixed coordinate pairs or mathematical constants.

1.2.1 Creation:

To create a tuple, just use parentheses:

```
coordinates = (3, 4)    #Point  (3,4)
print(type(coordinates))
print(coordinates)
pi_digits = (3, 1, 4, 1, 5)   #Immutable constant
print(type(pi_digits))
print(pi_digits)
```

1.2.2 Indexing:

As lists, each element in a tuple has a position called an **index.**

Indexing starts from 0 in python.

```
coordinates = (3, 4)    #Point  (3,4)
print(coordinates[0])   #3
print(coordinates[1])   #4
```

Negative indexing counts from the end:

```
coordinates = (3, 4)    #Point  (3,4)
print(coordinates[-1])   #4
print(coordinates[-2])   #3
```

1.2.3 Slicing:

As lists, slicing allows us to extract a portion (a subsequence) of a tuples:

[start:end:step]

```
coordinates = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)    #A tuple
print(type(coordinates))
print(coordinates [2:5])   #(2, 3, 4)
print(coordinates [:3])   #(0, 1, 2) (From start)
print(coordinates [::2])   #(0, 2, 4, 6, 8) (Every second)
```

1.2.4 Concatenation:

As lists, concatenation combines two tuples using + operator, and create a new list without modifying the originals:

```
tuple1 = (0, 1, 2, 3)          #A tuple
tuple2=(4, 5, 6, 7, 8, 9)      #A tuple
tuple3 = tuple1 + tuple2     #A combined tuple
print(tuple3)   #(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

2. Dictionaries and Sets: Creation, Manipulation, Use cases

Dictionaries and sets are powerful data structures in Python that allow us to organize information in more meaningful ways than simple lists.

A **dictionary** stores data as ***key–value pairs***. You can think of it as a computational version of a mathematical function, where each input is associated with a specific output. This makes dictionaries especially useful when working with mappings or relationships between quantities.

A **set**, on the other hand, stores unique elements without any particular order. It closely reflects the mathematical notion of a set: no duplicates, just distinct elements. Sets are particularly useful when dealing with distinct solutions, collections of values, or operations such as union and intersection.

2.1. Dictionaries:

A dictionary is a data structure that associates each key with a value:

- Each key must be unique and immutable (such as a number, string, or tuple).
- The associated value can be of any type: Number, string, list or even another dictionary.

Dictionaries are particularly useful when we want to represent relationships or mappings between elements.

✓ ***Creation:***

```
#Empty dictionary
grades = {}

#with initial values
student_scores = {"Ahmed": 16, "Mohamed": 18, "Karim":12}

#Mathematical function mapping
F_values = {0: 0, 1: 1, 2: 4, 3: 9}      #f(x)=x2

print(type(student_scores))
print(type(F_values))

#to print the dictionarie content
print(student_scores)
print(F_values)
```

✓ ***Manipulation:***

```
#Access value by key
print(student_scores["Ahmed"])    #16

#Add/Update
#if the key does not exist, Python adds it, otherwise, it will be updated
student_scores["Souad"] = 13    #New entry
student_scores["Karim"] = 14    #Update existing

#Remove
del student_scores [ "Mohamed"]    #Delete entry

#Check existence
if  "Ali" in student_scores:
    print("Ali has a score")
else:
    print("There is no student named Ali")
```

✓ ***Use cases***

1. Function lookup table

Instead of recalculating a function repeatedly, we can pre-compute and store its values:

```python
def quadratic(x):
      return x**2 + 2*x + 1

#Pre-computed values (faster)
quad_table = {i: quadratic(i) for i in range(10)}

print(quad_table)   #{0: 1, 1: 4, 2: 9, 3: 16, 4: 25, 5: 36, 6: 49, 7: 64, 8: 81, 9: 100}
```

2. Matrix coefficient storage

A dictionary can store matrix elements:

```python
A_matrix = {
          "a11": 2, "a12": 1,
          "a21": 3, "a22": 4
    }
print(type(A_matrix))
print(A_matrix)
```

3. Solution mapping

```python
Equation_solutions = {
          "x2 +2x + 1":  "x = -1 (Double root)",
          "2x + 3 = 7":  "x = 2"
    }
print(type(Equation_solutions))
print(Equation_solutions)
```

2.2. Sets:

A set is an ***unordered*** collection of ***unique*** elements, it corresponds directly to the mathematical concept of a set:

- No duplicates.
- No specific order.

Sets are particularly useful when we need to:

- Eliminate duplicate values.
- Perform union, intersection, or difference operations.

✓ ***Creation:*** Be careful, {} creates a dictionary, not a set.

```
#Empty set
prime_numbers = set()
solutions = set()

#From other structures
numbers = [1, 2, 2, 3, 3, 4]
unique_numbers = set(numbers)   #it automatically remove duplicates

print(type(prime_numbers))
print(prime_numbers)
print(type(solutions))
print(solutions)
print(type(numbers))
print(numbers)
print(type(unique_numbers))
#to print the set content
print(unique_numbers)
```

✓ ***Manipulation:***

```
#Add / remove
primes = {2, 3, 5}
print(primes)
primes.add(7)    #{2, 3, 5, 7}
print(primes)
primes.remove(3)  #{2, 5, 7}
print(primes)
#Membership test (very fast)
if 2 in primes:
   print("2 is a prime numer")

evens = {0, 2, 4, 6, 8}
print(evens)
odds = {1, 3, 5, 7, 9}
print(odds)
# Union (U)
all_numbers = evens | odds # {0,1,2,3,4,5,6,7,8,9}
print(all_numbers)
# Intersection (n)
common = evens & odds # set() - empty
print(common)
# Difference (\)
evens_not_odds = evens - odds # {0, 2, 4, 6, 8}
print(evens_not_odds)
# Symmetric difference
exclusive = evens ^ odds # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(exclusive)
```

✓ *Use cases*

```python
# 1. Distinct roots from quadratic equations
roots = {1.5, -1.5, 0}  # No duplicates

# 2. Prime number testing
def is_prime(n):
    if n < 2: return False
    primes = {2, 3, 5, 7, 11, 13}
    return n in primes

# 3. Set operations in geometry
points_A = {(0,0), (1,1), (2,2)}
points_B = {(1,1), (2,2), (3,3)}
intersection = points_A & points_B  # {(1,1), (2,2)}
```

3. Operations on Data Structures:

In Python, data structures such as lists, tuples, dictionaries, and sets provide flexible ways to store and manipulate collections of data. Once data are organized in these structures, it becomes necessary to perform various operations to analyze, transform, or extract useful information from them. Among the most common and important operations are sorting, filtering, and list comprehensions. These techniques allow programmers to process data efficiently while maintaining readable and concise code. In scientific programming, where datasets may represent numerical results, experimental observations, or mathematical objects, mastering these operations is essential.

3.1. Sorting

Sorting is the process of arranging the elements of a data structure according to a specific order, typically ascending or descending. In Python, lists can be sorted using built-in methods such as sort() or functions like sorted(). Sorting is particularly useful when organizing numerical data, preparing datasets for analysis, or identifying minimum and maximum values. For instance, in mathematical or statistical applications, sorting a dataset can help reveal patterns, facilitate median

calculation, or prepare data for graphical representation. Python's sorting mechanisms are efficient and allow customization through parameters such as sorting order or sorting according to a specific key function.

3.1.1. List sorting

```
numbers = [5, 2, 8, 1, 9]
print(numbers)

sorted_numbers = sorted(numbers)  # New list
print(sorted_numbers)

numbers.sort()           # In-place: [1, 2, 5, 8, 9]
print(numbers)

# Custom sorting (by length for strings)
words = ["cat", "elephant", "dog"]
print(words)

sorted_words = sorted(words, key=len)  # ['cat', 'dog', 'elephant']
print(sorted_words)
```

3.1.2. Tuples sorting

```
numbers = (5, 2, 8, 1, 9)  #A tuple
print(type(numbers))
print(numbers)

sorted_numbers = sorted(numbers)  # sort the tuple into a New list
print(type(sorted_numbers))
print(sorted_numbers)



# Custom sorting (by length for strings)
words = ("cat", "elephant", "dog")
print(type(words))
print(words)

sorted_words = sorted(words, key=len)  # sort the tuple into a New list
print(type(sorted_words))
print(sorted_words)
```

3.1.3. Dictionaries sorting

```
# Original dictionary
grades = {"Ali": 78, "Mohamed": 85, "Karim": 92,"Ahmed": 65}
print(type(grades))
# Sort by KEYS (alphabetical)
sorted_by_key = dict(sorted(grades.items()))
print(type(sorted_by_key))
print(sorted_by_key)  # {'Ahmed': 65, 'Ali': 78, 'Karim': 92, 'Mohamed': 85}

# Sort by VALUES (ascending)
sorted_by_value = dict(sorted(grades.items(), key=lambda x: x[1]))
print(sorted_by_value)  # {'Ahmed': 65, 'Ali': 78, 'Mohamed': 85, 'Karim': 92}
```

3.1.4. Sets sorting

```
numbers = {3, 1, 4, 1, 5, 9}
print(type(numbers))
print(numbers)
sorted_numbers = sorted(numbers)  # Returns LIST, not set!
print(type(sorted_numbers))
print(sorted(numbers))  # [1, 3, 4, 5, 9] (list)

# Convert back to set (order lost again)
sorted_set = set(sorted(numbers))  # {1, 3, 4, 5, 9} (unordered)
print(type(sorted_set))
print(sorted_set)
```

3.2. Filtering

Filtering refers to the process of selecting elements from a data structure that satisfy a given condition. This operation is fundamental in data analysis, where only a subset of data may be relevant for a particular computation. In Python, filtering can be achieved using conditional statements within loops, the filter() function, or more commonly through list comprehensions. For example, one might filter a list of numbers to retain only positive values, even numbers, or elements exceeding a certain threshold. In scientific contexts, filtering allows

researchers to isolate meaningful observations from larger datasets, thereby simplifying subsequent analysis.

3.2.1. Lists filtering: Filter even numbers from a sequence

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = [x for x in numbers if x % 2 == 0]
print(type(numbers))
print(numbers)
print(type(evens))
print(evens)  # [2, 4, 6, 8, 10]
```

3.2.2. Tuples filtering: Filter points inside unit circule($x^2+y^2<=1$)

```
points = [(0,0), (1,0), (0,1), (1,1), (-1,1)] #A list of tuples
print(type(points))     #A list
print(type(points[0]))  #A tuple
print(type(points[1]))  #A tuple
print(points)
inside_circle = [p for p in points if p[0]**2 + p[1]**2 <= 1]
print(type(inside_circle))
print(inside_circle)  # [(0, 0), (1, 0), (0, 1)]
```

3.2.3. Dictionaries filtering: Keep students with grades >=80:

```
grades = {"Ali": 78, "Mohamed": 85, "Karim": 92, "Ahmed": 65}
print(type(grades))   #<class 'dict'>
print(grades)    #{'Ali': 78, 'Mohamed': 85, 'Karim': 92, 'Ahmed': 65}
high_achievers = {k: v for k, v in grades.items() if v >= 80}
print(type(high_achievers))  #<class 'dict'>
print(high_achievers)  # {'Mohamed': 85, 'Karim': 92}
```

3.2.4. Sets filtering: Filter primes greater than 5:

```
primes = {2, 3, 5, 7, 11, 13, 17}
print(type(primes))
print(primes)
large_primes = {p for p in primes if p > 5}
print(type(large_primes))
print(large_primes)  # {7, 11, 13, 17}
```