Python and Data Science Handbook

1 Basics of Python and Variables & Data Types

1.1 Data Types in Python

Integer (int)
Description: Integers represent whole numbers, both positive and negative, without decimals.
Example:

```
age = 25
temperature = -5
```

Explanation: In this example, `age` and `temperature` are integers because they don't have any decimal parts.

Float (float)
Description: Floats represent real numbers that have decimal points.
Example:

```
price = 9.99
weight = 70.5
```

Explanation: Both `price` and `weight` have decimal parts, which makes them float values.

String (str)
Description: Strings are sequences of characters enclosed within quotes. They are used for text.
Example:

```
name = "Alice"
greeting = "Hello, world!"
```

Explanation: Here, `name` and `greeting` are strings because they contain characters enclosed in quotes.

Boolean (bool)
Description: Booleans represent one of two values: `True` or `False`. They are often used in

conditional statements.
Example:

```
is_active = True
has_permission = False
```

Explanation: `is_active` and `has_permission` are Boolean values, representing `True` or `False`.

List (list)
Description: Lists are ordered collections of items, which can be of any data type. Lists are mutable, meaning their contents can be changed.
Example:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
```

Explanation: Here, `fruits` is a list of strings, and `numbers` is a list of integers. Lists are written within square brackets `[]`.

Tuple (tuple)
Description: Tuples are ordered collections of items, similar to lists, but they are immutable, meaning they cannot be changed once created.
Example:

```
coordinates = (10.5, 20.8)
colors = ("red", "green", "blue")
```

Explanation: `coordinates` and `colors` are tuples. They are enclosed within parentheses `()` and cannot be modified.

Dictionary (dict)
Description: Dictionaries are collections of key-value pairs. They are unordered and accessed by unique keys rather than by index.
Example:

```
person = {"name": "Alice", "age": 25, "city": "New York"}
```

Explanation: `person` is a dictionary where each piece of data (like "name" and "age") is associated with a unique key. Dictionaries are written within curly braces {}.

Set (set)
Description: Sets are collections of unique items. They are unordered, meaning there's no guaranteed order for items. Sets automatically eliminate duplicate values.
Example:

```
unique_numbers = {1, 2, 3, 4, 4, 5}
```

Explanation: `unique_numbers` is a set. Even though 4 is added twice, the set stores only unique values {`1, 2, 3, 4, 5`}.

None Type (None)
Description: `None` represents the absence of a value or a null value. It is often used as a placeholder or to indicate "no value."
Example:

```
result = None
```

Explanation: `result` is set to `None`, which means it currently has no value.

Complex (complex)
Description: Complex numbers consist of a real and an imaginary part, represented as `a + bj` where `a` is the real part and `b` is the imaginary part.
Example:

```
z = 3 + 4j
```

Explanation: `z` is a complex number where 3 is the real part and `4j` is the imaginary part.

2 Loops and Conditions

2.1 Logical Operators Overview

- **and**: Returns `True` if both conditions are `True`. *Example:*

```
if has_lantern and has_map:
    print("You are ready!")
```

- **or**: Returns `True` if at least one condition is `True`. *Example:*

```
if has_lantern or has_map:
    print("You can proceed carefully.")
```

- **not**: Reverses the Boolean value. *Example:*

```
if not has_key:
    print("You need the key.")
```

2.2 Comparison Operators

Operator	Description	Example	Output
`<`	Less than	`5 < 10`	`True`
`>`	Greater than	`10 > 5`	`True`
`<=`	Less than or equal	`5 <= 5`	`True`
`>=`	Greater than or equal	`5 >= 10`	`False`
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 10`	`True`

Table 1: Comparison Operators in Python

2.3 Syntax of `if-elif-else` Statements

```
if condition1:
    # Code block for condition1
elif condition2:
    # Code block for condition2
else:
    # Code block for when none of the above conditions are true
```

2.4 For Loop

Used to iterate over sequences like lists, tuples, strings, or ranges. **Syntax:**

```
for variable in sequence:
    # Code to execute
```

2.5 While Loop

Used to repeatedly execute a block of code as long as a condition is `True`. **Syntax:**

```
while condition:
    # Code to execute
```

2.6 `break` Statement

The `break` statement is used to exit the loop early, regardless of the loop's condition. It terminates the loop immediately and control is transferred to the next statement after the loop. **Syntax:**

```
for i in range(10):
    if i == 5:
        break  # exit the loop when i is 5
```

2.7 `continue` Statement

The `continue` statement skips the current iteration of the loop and continues to the next iteration, without executing the remaining code for the current loop iteration. **Syntax:**

```
for i in range(10):
    if i == 5:
        continue  # skip the rest of the loop when i is 5
    print(i)
```

2.8 `pass` Statement

The `pass` statement is a placeholder. It is used when no action is required and is usually found in places where code is syntactically required but no action is needed. **Syntax:**

```
for i in range(10):
    if i == 5:
        pass  # do nothing
    else:
        print(i)
```

3 Inspiration of Functions

3.1 General Functions for Data Types

Common Functions

- `len()`: Returns the number of items in an object.
- `max()`: Returns the largest item in an object.
- `min()`: Returns the smallest item in an object.
- `sum()`: Returns the sum of items (only for numeric elements).
- `sorted()`: Returns a sorted list of elements from an object.

3.2 Lambda Functions

Purpose: A small, anonymous function defined with the `lambda` keyword.
Syntax: `lambda arguments: expression`
Commonly used in short operations, especially with `sorted()` and `map()`.
Example:

```
square = lambda x: x**2
print(square(5))  # Output: 25
```

3.3 `sorted()` Function

Purpose: Sorts elements in an iterable (e.g., a list) based on a key.
Syntax:

```
sorted(iterable, key=function, reverse=False)
```

3.4 Variable-Length Arguments (`*args`) and Keyword Arguments (`**kwargs`)

Allows a function to accept any number of positional arguments.
Syntax: `*args` collects extra arguments into a tuple.
Keyword Arguments (`**kwargs`):
Allows a function to accept any number of keyword arguments.
Syntax: `**kwargs` collects extra named arguments into a dictionary.

Combining `*args` and `**kwargs`
You can use both `*args` and `**kwargs` in the same function to handle both positional and keyword arguments.

```
def describe(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)
describe("artifact1", "artifact2", rarity="rare", year=1985)
# Output:
# Positional arguments: ('artifact1', 'artifact2')
# Keyword arguments: {'rarity': 'rare', 'year': 1985}
```

Example: Using with Functions:
`*args` and `**kwargs` make your functions flexible and reusable by allowing them to accept varying numbers and types of inputs.

3.5 Recursion

What is Recursion?
Recursion is a programming technique in which a function calls itself in order to solve smaller instances of the same problem. A recursive function typically has two parts:

1. **Base Case:** The condition under which the function stops calling itself and returns a result. Without a base case, the function would call itself indefinitely.

2. **Recursive Case:** The part where the function calls itself with modified arguments, progressively solving smaller parts of the problem.

How does Recursion work?
A recursive function reduces the problem into a simpler or smaller version of the same problem.
The function keeps calling itself with simpler inputs until it hits the base case.
The function then starts returning the results and building up the solution step by step.
Example: Fibonacci Sequence using Recursion
The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones. The sequence starts with 0 and 1, and the subsequent numbers are:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
The recursive formula for Fibonacci is:

$$fib(n) = fib(n-1) + fib(n-2) \quad \text{for } n > 1$$

Base cases: $fib(0) = 0$, $fib(1) = 1$

4 Demonstration of File Handling & Basics of File Handling

4.1 File Handling in Python

Key Concepts

1. **Opening and Closing Files:** Use `open()` to open a file and `close()` to close it.
2. **Modes:**
   - `'r'`: Read (default mode).
   - `'w'`: Write (overwrites file if it exists).
   - `'a'`: Append (adds content to the end of the file).
   - `'r+'`: Read and write.
3. **Reading Files:** Use `read()`, `readline()`, or `readlines()` to read file content.
4. **Writing Files:** Use `write()` to write data to a file.
5. **Context Manager:** Use `with` to handle files automatically (no need to call `close()`).

Examples
Opening a File and Reading Content:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # Prints the entire file content
```

Writing to a File:

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

Appending to a File:

```
with open("example.txt", "a") as file:
    file.write("\nThis is a new line.")
```

Reading File Line by Line:

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # Removes leading/trailing whitespace
```

Notes:

- Always use `with` to handle files, as it ensures proper resource management.
- Be cautious with `'w'` mode, as it will overwrite existing files.

Seek and Tell:
Manage the file pointer using `seek()` and find its position with `tell()`.

```
with open("example.txt", "r") as file:
    print(file.tell())  # Outputs the current file pointer position
    file.seek(5)
    # Moves the pointer to the 5th byte
    print(file.read())  # Reads from the 5th byte onwards
```

5 Tuple, List and Dictionaries

5.1 List (list)

Description: Lists are ordered, mutable collections of items. They are one of the most versatile data types in Python, capable of holding items of different data types. Lists are defined by enclosing comma-separated items in square brackets `[]`.

Key Characteristics of Lists:

- **Ordered:** Items in a list have a defined order, and that order is maintained.
- **Mutable:** You can change lists after they are created. Items can be added, removed, or modified.
- **Allow Duplicates:** Lists can contain duplicate values.
- **Heterogeneous Data Types:** A single list can contain items of different types (integers, strings, other lists, etc.).

Common List Operations:

- **Indexing:** Access elements by their position (index), starting from 0.
- **Slicing:** Extract a portion of a list to create a new sublist.
- **Adding Items:** Use methods like `append()` to add to the end, `insert()` to add at a specific position, and `extend()` to add multiple items from another list.
- **Removing Items:** Use methods like `remove()` to remove a specific value, `pop()` to remove an item at a given index (and return it), and `del` to remove items by index or slice.
- **Searching and Counting:** Use `in` operator to check for existence, `index()` to find the index of a value, and `count()` to count occurrences.
- **Sorting:** Use `sort()` to sort the list in-place or `sorted()` function to get a new sorted list.
- **List Comprehensions:** A concise way to create lists using expressions and loops.

Example:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed_list = [1, "hello", 3.4, True]

# Accessing elements
print(fruits[0])  # Output: apple

# Slicing
print(numbers[1:4]) # Output: [2, 3, 4]

```

```
# Modifying list
fruits.append("orange")
print(fruits) # Output: ['apple', 'banana', 'cherry', 'orange']

del fruits[1]
print(fruits) # Output: ['apple', 'cherry', 'orange']
```

5.2 Tuple (tuple)

Description: Tuples are ordered, immutable collections of items, similar to lists. Once a tuple is created, its contents cannot be changed (items cannot be added, removed, or modified). Tuples are defined by enclosing comma-separated items in parentheses `()`.

Key Characteristics of Tuples:

- **Ordered:** Items in a tuple have a defined order.
- **Immutable:** Tuples cannot be changed after creation. This is the main difference from lists.
- **Allow Duplicates:** Tuples can contain duplicate values.
- **Heterogeneous Data Types:** Tuples can hold items of different types.

Common Tuple Operations:

- **Indexing:** Access elements by index, like lists.
- **Slicing:** Extract portions to create new tuples.
- **Concatenation:** Combine tuples using the `+` operator to create a new tuple.
- **Repetition:** Repeat tuple elements using the `*` operator to create a new tuple.
- **Count and Index:** Methods like `count()` and `index()` are available, similar to lists.

Example:

```
coordinates = (10, 20)
colors = ("red", "green", "blue")
mixed_tuple = (1, "world", False)

# Accessing elements
print(coordinates[0]) # Output: 10

# Slicing
print(colors[1:]) # Output: ('green', 'blue')

# Concatenation
combined_tuple = coordinates + colors
print(combined_tuple) # Output: (10, 20, 'red', 'green', 'blue')
```

When to Use Tuples vs. Lists: Use tuples when you need to ensure data integrity and prevent accidental modification. Tuples are often used for:

- Representing fixed collections of items that should not change.
- Function return values when you want to return multiple items as a single, immutable unit.
- As keys in dictionaries (since keys must be immutable).
- Situations where performance is critical and the slight performance advantage of tuples over lists (in creation and iteration) becomes relevant.

5.3 Dictionary (dict)

Description: Dictionaries are unordered (from Python 3.7+, insertion order is preserved, but conceptually still unordered in older versions), mutable collections of key-value pairs. Each key in a dictionary must be unique and immutable (e.g., string, number, tuple), while values can be of any data type and can be duplicates. Dictionaries are defined by enclosing comma-separated key-value pairs in curly braces {}, with each pair in the format `key: value`.

Key Characteristics of Dictionaries:

- **Unordered (Historically):** Dictionaries are generally considered unordered collections. (Note: In Python 3.7 and later, dictionaries maintain insertion order as an implementation detail, but reliance on order is generally discouraged for conceptual clarity and compatibility with older versions).
- **Mutable:** Dictionaries can be changed. You can add, remove, or modify key-value pairs.
- **Unique Keys:** Each key within a dictionary must be unique. If you try to use a key that already exists, it will overwrite the existing value associated with that key.
- **Keys Must Be Immutable:** Keys must be of an immutable type (like strings, numbers, or tuples). Lists cannot be used as keys because they are mutable. Values can be of any type.

Common Dictionary Operations:

- **Accessing Values:** Retrieve values using their keys, either with square brackets `[]`, or using the `get()` method (which allows for a default value if the key is not found).
- **Adding/Updating Pairs:** Add new key-value pairs or update existing ones by assigning a value to a key using square brackets.
- **Removing Pairs:** Use `del` statement to remove a pair by key, or `pop()` method to remove a key and return its value, or `popitem()` to remove and return the last inserted key-value pair (in Python 3.7+).
- **Iterating:** Iterate over keys, values, or key-value pairs (items) using loops and dictionary methods like `keys()`, `values()`, and `items()`.

- **Checking Key Existence:** Use the `in` operator to check if a key exists in the dictionary.
- **Dictionary Comprehensions:** A concise way to create dictionaries using expressions and loops.

Example:

```
person = {"name": "Alice", "age": 25, "city": "New York"}
empty_dict = {}

# Accessing values
print(person["name"]) # Output: Alice
print(person.get("age")) # Output: 25

# Adding/Updating pairs
person["occupation"] = "Engineer"
person["age"] = 26 # Update age
print(person) # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', '
    occupation': 'Engineer'}

# Removing pairs
del person["city"]
print(person) # Output: {'name': 'Alice', 'age': 26, 'occupation': '
    Engineer'}
```

When to Use Dictionaries: Use dictionaries when you need to store and retrieve data based on unique keys. Dictionaries are ideal for:

- Representing mappings between pieces of data.
- Storing configuration settings, where each setting can be accessed by its name (key).
- Implementing associative arrays or hash maps.
- Building data structures where fast lookup of values by key is essential.

6 Algorithm Efficiency

6.1 Algorithm Efficiency

Overview:
Understanding algorithm efficiency helps in designing programs that run effectively. Efficiency is measured in terms of time and space requirements, usually expressed using Big-O notation.

6.1.1 Time Complexity

Definition:
Measures how the runtime of an algorithm increases with the size of the input.
Big-O Notation:

Common classes include O(1), O(n), O(log n), O(n^2), etc.
Examples:

- Linear Search: O(n)
- Binary Search: O(log n) (on a sorted list)

6.1.2 Space Complexity

Definition:
Indicates the amount of memory an algorithm uses relative to the input size.
Example:
An algorithm that creates a new list from an existing list of n elements generally has O(n) space complexity.

6.1.3 Best, Average, and Worst Cases

Explanation:
Efficiency can vary based on the input; the best-case scenario often differs significantly from the worst-case.
Example:
In a sorted list, a linear search might find an element in the first position (best case O(1)), while in an unsorted list, it may scan all elements (worst case O(n)).

6.1.4 Practical Relevance

Explanation:
Choosing the right algorithm affects performance, especially for large datasets, making this analysis essential for efficient programming.

7 Basics of Sorting (Bubble, Quick and Insertion)

7.1 Sorting Techniques and Analysis

7.1.1 Bubble Sort

Concept
Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.
Time Complexity

- Best case: O(n) – This occurs when the list is already sorted. The algorithm only needs one pass through the list to confirm that no swaps are needed.

- Worst case: $O(n^2)$ – This occurs when the list is sorted in reverse order, meaning the algorithm has to compare and swap each element in every pass.

Space Complexity
O(1) (in-place sorting).
How It Works

1. Compare each pair of adjacent elements.
2. If the first element is larger than the second, swap them.
3. Repeat the process for every element until no swaps are needed.
4. The largest element "bubbles" up to the correct position after each pass.

Example
Initial list: `[5, 2, 9, 1, 5, 6]`
After 1st pass: `[2, 5, 1, 5, 6, 9]`
After 2nd pass: `[2, 1, 5, 5, 6, 9]`
After 3rd pass: `[1, 2, 5, 5, 6, 9]` (sorted)

7.1.2 Quick Sort

Concept
Quick Sort is a divide-and-conquer algorithm. It picks a "pivot" element and partitions the list into two sublists: elements smaller than the pivot and elements greater than the pivot. Then, it recursively sorts the sublists.
Time Complexity

- Best case: O(n log n) – This occurs when the pivot divides the list into two nearly equal halves. This results in balanced partitioning at each step, leading to efficient sorting.
- Worst case: $O(n^2)$ – This occurs when the pivot is the smallest or largest element in the list, causing the partitioning to be unbalanced (one side has almost all the elements, and the other side has very few), leading to poor performance.

Space Complexity
O(log n) (in-place sorting).
How It Works

1. Choose a pivot element from the list (commonly the last element).
2. Partition the list into two sublists: one with elements smaller than the pivot and one with elements greater than the pivot.
3. Recursively apply the same process to the sublists.

4. Once the sublists have been sorted, the entire list is sorted.

Example
Initial list: `[10, 80, 30, 90, 40, 50, 70]`
Pivot: `70`
Partitioned: `[10, 30, 40, 50, 70, 90, 80]`
Now recursively sort the sublists: `[10, 30, 40, 50]` and `[90, 80]`
Final sorted list: `[10, 30, 40, 50, 70, 80, 90]`

7.1.3 Merge Sort

Concept
Merge Sort is another divide-and-conquer algorithm. It divides the list into two halves, recursively sorts each half, and then merges the two sorted halves into a single sorted list.
Time Complexity

- Best case: O(n log n) – The time complexity is always O(n log n) because the list is always divided in half, and merging still takes linear time even in the best case.
- Worst case: O(n log n) – Merge Sort performs the same regardless of the initial order of elements because the list is always recursively divided and merged in a consistent manner.

Space Complexity
O(n) (requires extra space for merging).
How It Works

1. Divide the list into two halves.
2. Recursively sort each half.
3. Merge the sorted halves back together by comparing elements from both halves and adding them to the sorted list.

Example
Initial list: `[38, 27, 43, 3, 9, 82, 10]`
Split: `[38, 27, 43]`, `[3, 9, 82, 10]`
Recursively sort: `[27, 38, 43]`, `[3, 9, 10, 82]`
Merge: `[3, 9, 10, 27, 38, 43, 82]`

7.1.4 Insertion Sort

Concept:
Insertion Sort is a simple, comparison-based sorting algorithm that builds the final sorted array one item at a time. It is analogous to the way people sort playing cards in their hands.
Time Complexity:

- Best case: O(n) – Occurs when the list is already sorted. Only one comparison is needed per element.
- Worst case: $O(n^2)$ – Occurs when the list is sorted in reverse order. Each element is compared with all previous elements.

Space Complexity:
O(1) (in-place sorting).
How It Works:

1. Divide the list into a "sorted" and "unsorted" section. Initially, the first element is considered sorted.
2. Take the first element from the unsorted section and insert it into its correct position in the sorted section.
3. Repeat the process for all elements in the unsorted section until the entire list is sorted.

Example:
Initial list: 12, 11, 13, 5, 6

1. Start with the second element (11): Insert it into the correct position in `[12]`. Result: `[11, 12, 13, 5, 6]`
2. Take 13: It is already in the correct position. Result: `[11, 12, 13, 5, 6]`
3. Take 5: Insert it before 11. Result: `[5, 11, 12, 13, 6]`
4. Take 6: Insert it after 5. Result: `[5, 6, 11, 12, 13]`

Final sorted list: `[5, 6, 11, 12, 13]`

7.1.5 Selection Sort

Concept:
Selection Sort is a comparison-based sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted section and moves it to the sorted section.
Time Complexity:

- Best case: $O(n^2)$ – Even in the best case, the algorithm performs the same number of comparisons as it does in the worst case.
- Worst case: $O(n^2)$ – Each element is compared to all others to find the minimum.

Space Complexity:
O(1) (in-place sorting).
How It Works:

1. Divide the list into a "sorted" and "unsorted" section. Initially, the entire list is unsorted.
2. Find the smallest element in the unsorted section and swap it with the first element of the unsorted section.
3. Move the boundary between the sorted and unsorted sections one element to the right.
4. Repeat the process until the entire list is sorted.

Example:
Initial list: 64, 25, 12, 22, 11

1. Find the smallest element (11) and swap it with 64. Result: `[11, 25, 12, 22, 64]`
2. Find the next smallest element (12) and swap it with 25. Result: `[11, 12, 25, 22, 64]`
3. Find the next smallest element (22) and swap it with 25. Result: `[11, 12, 22, 25, 64]`
4. 64 is already the largest, no need to swap.

Final sorted list: `[11, 12, 22, 25, 64]`

8 Searching and its relevance

8.1 Searching and Its Relevance

Overview:
Searching is a fundamental operation in programming used to locate specific data within a structure. Different algorithms offer trade-offs in speed and complexity.

8.1.1 Linear Search

Concept:
Traverse each element until the target is found.
Example:

```
def linear_search(lst, target):
    for i, value in enumerate(lst):
        if value == target:
            return i
    return -1
```

8.1.2 Binary Search

Concept:
Works on sorted lists by repeatedly dividing the search interval in half.
Example:

```
def binary_search(lst, target):
    low, high = 0, len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] == target:
            return mid
        elif lst[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

8.1.3 Relevance of Searching

Explanation:
Searching algorithms are used in various applications like database queries, inventory systems, and user interfaces.
They demonstrate the importance of data structure organization in efficient data retrieval.

9 Baker's problem - Linear Algebra

9.1 Baker's Problem – Linear Algebra

Overview:
The Baker's Problem is a practical scenario used to demonstrate how linear algebra can optimize resource allocation. It involves formulating and solving systems of linear equations.

9.1.1 Problem Formulation

Definition:
A baker must determine the optimal mix of ingredients (e.g., flour, sugar, eggs) to maximize production while minimizing waste.
Setting Up Equations:
Each ingredient constraint is modeled as a linear equation.

9.1.2 Matrix Representation

Explanation:
The system of equations can be expressed in the matrix form $Ax = b$, where:

- A represents the coefficients of the variables,
- x is the vector of unknowns (e.g., quantities of ingredients),

- b is the vector of constants (e.g., total available resources).

9.1.3 Solving Techniques

- **Gaussian Elimination:**
  Systematically eliminate variables to solve for the unknowns.
- **Matrix Inversion:**
  When A is invertible, compute $x = A^{-1}b$.

9.1.4 Practical Application

Explanation:
Helps in decision-making by providing a clear method to distribute resources efficiently.

10 Vector and Matrices (Operations)

10.1 Vector and Matrices

Overview:
Vectors and matrices are core elements of linear algebra that represent data, perform transformations, and solve systems of equations.

10.1.1 Vectors

Definition:
An ordered collection of numbers representing magnitude and direction.
Basic Operations:

- **Addition/Subtraction:** Combine or compare vectors element-wise.
- **Scalar Multiplication:** Multiply each element by a constant.
- **Dot Product:** Compute the sum of the products of corresponding entries.

Example:

```
v1 = [1, 2, 3]
v2 = [4, 5, 6]
dot_product = sum(a * b for a, b in zip(v1, v2))
```

10.1.2 Matrices

Definition:
A rectangular array of numbers arranged in rows and columns.
Basic Operations:

- **Addition/Subtraction:** Performed element-wise on matrices of the same dimensions.
- **Multiplication:** Combining rows of the first matrix with columns of the second.
- **Transpose:** Flipping a matrix over its diagonal.
- **Determinant and Inverse:** Important for solving linear systems.

Example (Matrix Multiplication):

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.dot(A, B)
```

10.1.3 Applications

Explanation:
Vectors and matrices are used in computer graphics, data analysis, physics simulations, and more.

11 Matrix Operations

11.1 Arithmetic Operators in Python

Python provides several operators for performing arithmetic operations.

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	8
`-`	Subtraction	`5 - 3`	2
`*`	Multiplication	`5 * 3`	15
`/`	Division (float result)	`5 / 3`	1.666...
`//`	Floor Division	`5 // 3`	1
`%`	Modulus (Remainder)	`5 % 3`	2
`**`	Exponentiation	`5 ** 3`	125

Table 2: Arithmetic Operators in Python

11.1.1 Examples of Usage

Basic Arithmetic

```
a = 10
b = 3
print(a + b)  # Output: 13
print(a - b)  # Output: 7
```

Advanced Operations

```
x = 7
y = 2
print(x ** y)  # 7 to the power of 2 = 49
print(x // y)  # Floor division = 3
print(x % y)   # Remainder = 1
```

11.2 Matrix Operations with NumPy

11.2.1 Matrix Multiplication

```
m1 = np.array([[1, 2], [3, 4]])
m2 = np.array([[5, 6], [7, 8]])
# Element-wise multiplication
m_mul = np.multiply(m1, m2)
# Matrix multiplication
m_dot = np.dot(m1, m2) # Alternatively, use m1 @ m2
```

11.2.2 Transpose

```
matrix = np.array([[1, 2], [3, 4]])
print(matrix.T)  # Transpose of the matrix
matrix = np.array([[1, 2, 3], [4, 5, 6]])
m_transpose = np.transpose(matrix)
```

11.2.3 Determinant

```
from numpy.linalg import det
print(det(matrix))
```

12 Probability

12.1 Probability

Overview:
Probability quantifies uncertainty and is a cornerstone for decision-making in both everyday life and scientific research.

12.1.1 Basic Definitions

- **Experiment:** An action or process that leads to outcomes.
- **Sample Space:** The set of all possible outcomes.
- **Event:** A subset of the sample space.

12.1.2 Rules of Probability

- **Addition Rule:**
  For mutually exclusive events, $P(A \cup B) = P(A) + P(B)$
- **Multiplication Rule:**
  For independent events, $P(A \cap B) = P(A) \times P(B)$

12.1.3 Conditional Probability

Definition:
The probability of event A given that event B has occurred is given by:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Example:
Calculating the probability of drawing an ace from a deck after removing a card.

12.1.4 Real-World Applications

Explanation:
Used in risk assessment, decision making, statistical inference, and in fields such as finance and engineering.

13 Descriptive statistics - Mean, Median, Mode, Std dev, variance, range

13.1 Statistical Analysis with NumPy

13.1.1 Mean

The mean (average) is calculated using `numpy.mean`.

```
arr = np.array([1,2,3,4,5])
mean_val = np.mean(arr)
```

13.1.2 Median and Mode

The median is the middle value, while the mode is the most frequent value.

```
arr = np.array([1,2,3,4,5])
median_val = np.median(arr)
from scipy.stats import mode
mode_val = mode(arr)
```

13.1.3 Standard deviation and Variance

Standard deviation measures spread, while variance measures the squared spread.

```
arr = np.array([1,2,3,4,5])
std_val = np.std(arr)
var_val = np.var(arr)
```

14 Numpy

14.1 NumPy

NumPy (Numerical Python) is a foundational library for numerical and scientific computing in Python. It provides robust support for large multi-dimensional arrays and matrices, along with an extensive collection of high-level mathematical functions to manipulate these arrays. NumPy is highly optimized for performance and is a cornerstone in fields like data science, machine learning, and scientific research.

14.1.1 Array

Basic array creation
An array is a central data structure in NumPy, enabling efficient storage and manipulation of homogeneous data types.

```
import numpy as np
# Creating a 1D array
arr = np.array([1, 2, 3, 4])
# Creating a 2D array
arr_2d = np.array([[1, 2], [3, 4]])
# Creating a 3D array
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

Difference between NumPy arrays and lists
NumPy arrays offer better performance, more functionality, and memory efficiency compared to Python lists. Unlike lists, NumPy arrays are homogeneous, allowing for optimized operations.
Higher dimension arrays
Higher-dimensional arrays are useful for applications like image processing or deep learning.

```
# Creating a three-dimensional array
a5 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4,5,6]]])
print(a5)
# Creating a higher-dimensional array (4D)
high_dim_array = np.ones((2, 3, 4, 5))
```

14.1.2 Array Properties/Attributes

NumPy arrays have several attributes that provide information about their structure and data.

```
arr = np.array([1, 2, 3, 4])
print(arr.ndim)  # Number of dimensions
print(arr.shape) # Shape of the array (dimensions per axis)
print(arr.size)  # Total number of elements
print(arr.dtype) # Data type of elements
print(arr.itemsize) # Size of each element in bytes
```

14.1.3 Accessing Array Elements

Accessing elements in a NumPy array uses zero-based indexing, similar to Python lists.

```
# Accessing elements in a 1D array
print(arr[0])
# Accessing elements in a 2D array
print(arr_2d[0, 1])
# Accessing elements in a 3D array
print(arr_3d[0, 1, 1])
```

14.1.4 Conditional Indexing

Conditional indexing allows for filtering elements based on a condition.

```
arr = np.array([10,20,30,40,50])
print(arr[arr > 30]) # Retrieves elements greater than 30
arr[arr > 30] = -1  # Updates elements greater than 30 to -1
print(arr)
```

14.1.5 Mutability

NumPy arrays are mutable, meaning you can modify their contents.

```
arr[0] = 10  # Modifying an element
print(arr)
```

14.1.6 Referencing

Assigning an array to another variable creates a reference, not a copy. Changes in the new variable affect the original.

```
ref = arr
ref[1] = 20
print(arr)  # Original array is also modified
```

14.1.7 Reshaping

You can reshape arrays to change their structure without altering the data.

```
arr = np.ones((2,3,4), dtype=np.int16)
print("Before reshape:")
print(arr)
arr = arr.reshape(6,4)
print("After reshape:")
print(arr)
```

14.1.8 Data Types

Changing data types
Some common data types are You can explicitly change the data type of array elements using the `astype` method.

```
arr_float = arr.astype(float)
print(arr_float.dtype)
print(arr_float)
```

14.1.9 Initialization

Zeros and Ones
You can initialize arrays filled with zeros or ones, useful for default values.

```
zeros = np.zeros((2, 3))
ones = np.ones((2, 3))
arr_eye = np.eye(4) # Generates a square identity matrix
```

Range
NumPy offers functions to create arrays with numerical ranges.

```
# Creating a range of numbers
arr_range = np.arange(0, 10, 2)
# Creating equally spaced numbers
linspace = np.linspace(0, 1, 5)  # Five numbers between 0 and 1
```

14.1.10 Operations

Arithmetic
Arithmetic operations in NumPy are performed elementwise.

```
sum_result = arr + 2  # Add 2 to each element
mul_result = arr * 3  # Multiply each element by 3
arr_cumsum = np.cumsum(arr) # Finds cumulative sum
arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])
print(arr1 + arr2) # Addition
print(arr1 - arr2) # Subtraction
print(arr1 * arr2) # Multiplication
print(arr1 / arr2) # Division
```

14.1.11 Universal Functions

Universal functions (ufuncs) are vectorized operations applied element-wise.

```
sqrt_result = np.sqrt(arr)  # Square root of each element
sin_result = np.sin(arr)    # Sine of each element
```

14.1.12 Random Module

NumPy's random module provides functions to generate random numbers.

```
# Generating random numbers between 0 and 1
random_array = np.random.rand(2, 3)
# Generating random integers within a range
random_integers = np.random.randint(0, 10, size=(3, 3))
```

Distribution

`numpy.random.rand(d0, d1, ..., dn)` *Uniform Distribution:* Generates random numbers to form an array of dimension (`d0, d1, ..., dn`) from a uniform distribution over `[0, 1)`.

```
arr = np.random.rand(10000) # This generates 10,000 numbers between 0 and
    1
```

`numpy.random.randn(d0, d1, ..., dn)` Generates numbers from a standard normal distribution (mean = 0, std dev = 1).

```
arr = np.random.randn(10000) # This generates 10,000 numbers from the
    normal distribution
```

`numpy.random.exponential(scale, size=None)` Here scale is the inverse of the rate parameter, size determines the output shape.

```
arr = np.random.exponential(1, 10000)
```

Seed

Setting a seed ensures that the random numbers generated are reproducible.

```
numpy.random.seed(seed) # seed is an integer value
np.random.seed(182)
print(np.random.rand(10))
```

15 Pandas

15.1 Pandas DataFrame

Introduction to Pandas DataFrames

Pandas provides a powerful data structure called DataFrame, similar to a table or spreadsheet.

It allows for easy manipulation and analysis of structured data.

Creating a DataFrame:

```
import pandas as pd
import numpy as np
student_data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': np.random.randint(18, 22, size=5),
    'Major': ['Computer Science', 'Physics', 'Mathematics', 'Engineering',
        'Biology'],
    'GPA': np.random.uniform(2.5, 4.0, size=5).round(2)
}
df = pd.DataFrame(student_data)
print(type(df))
print(df)
```

Basic Information About the DataFrame:

```
df.info()
```

Displays:

- Number of rows and columns.
- Data types of each column.
- Memory usage.

15.1.1 Accessing Data

Accessing Columns:

```
print(df['Name'])  # Access using column name as key
print(df.Age)  # Access using dot notation
```

Accessing Multiple Columns
Concept
Pandas allows users to access multiple columns by passing a list of column names to the DataFrame. This technique is particularly useful for isolating specific subsets of data for further analysis or visualization.
Example

```
# Assuming df is your DataFrame
multiple_columns_df = df[['Name', 'Major', 'GPA']]
print(multiple_columns_df)
```

Accessing Rows:

```
print(df.iloc[0])  # Access the first row by index
```

`iloc`: Access rows by their numerical index.
`loc`: Access rows by labels (if available).

15.1.2 Summary Statistics

Pandas provides functions to compute basic statistics for numerical columns:

```
print(df.describe())  # Summary statistics for numerical columns
```

15.1.3 Additional Pandas Topics

Adding New Columns:

```
df['Graduation Year'] = 2025
print(df)
```

Filtering Data:

```
print(df[df['GPA'] > 3.5])  # Students with GPA greater than 3.5
```

Handling Missing Data:
Fill missing values:

```
df['GPA'].fillna(df['GPA'].mean(), inplace=True)
```

Drop rows with missing values:

```
df.dropna(inplace=True)
```

15.1.4 Slicing Rows

Concept
Row slicing allows you to extract specific rows from a DataFrame based on their index positions. This operation is essential for exploring data subsets or preparing training and testing datasets.
Syntax
`df.iloc[start:stop]`
`start`: The starting index (inclusive).
`stop`: The ending index (exclusive).
Example

```
print(df.iloc[1:4])  # Retrieves rows with index 1, 2, and 3.
```

This operation returns a slice of the DataFrame containing the specified range of rows.

15.1.5 Accessing First and Last Few Rows

Accessing First Rows
Concept
The `head()` function retrieves the first few rows of the DataFrame. It is useful for previewing the dataset or verifying its structure.
Syntax
`df.head(n)`
`n`: The number of rows to retrieve (default is 5).
Example

```
print(df.head(2))  # Retrieves the first 2 rows.
```

Accessing Last Rows
Concept
The `tail()` function retrieves the last few rows of the DataFrame. It is commonly used to check the end of the dataset.
Syntax
`df.tail(n)`
`n`: The number of rows to retrieve (default is 5).
Example

```
print(df.tail(2))  # Retrieves the last 2 rows.
```

15.1.6 Statistical Analysis on Data

Concept
Pandas offers built-in methods for performing statistical computations and summarizing data. These methods are crucial for understanding data distributions and trends.
Common Functions

1. `describe()`: Provides a statistical summary of numerical columns, including count, mean, standard deviation, minimum, and quartiles.

```
df.describe()
```

2. `mean()`: Calculates the average value of a column.

```
df['column_name'].mean()
```

3. `min()`: Identifies the smallest value in a column.

```
df['column_name'].min()
```

4. `max()`: Identifies the largest value in a column.

```
df['column_name'].max()
```

15.1.7 Querying Data

Concept
Querying enables you to filter rows in a DataFrame based on specific conditions. This technique is pivotal for isolating relevant data.
Syntax
`df[condition]`
`condition`: A logical condition used to filter rows.
Examples

1. **Student with Highest GPA:**

```
df[df['GPA'] == df['GPA'].max()]
```

   Retrieves the row where the "GPA" column has the maximum value.

2. **Details of a Specific Student (e.g., Bob):**

```
df[df['Name'] == 'Bob']
```

   Filters rows where the "Name" column matches "Bob".

3. **Specific Columns for a Condition:**

```
df[df['Name'] == 'Bob'][['Major', 'GPA']]
```

Retrieves the "Major" and "GPA" columns for the student named "Bob".

4. **Students with a Specific Age:**

```
df[df['Age'] == 18]
```

Filters rows where the "Age" column equals 18.

15.1.8 Reading Data from CSV

Concept
The `read_csv()` function reads data from a CSV file and loads it into a DataFrame. This is a primary method for importing datasets.
Syntax
`pd.read_csv(filepath)`
`filepath`: The file path to the CSV file (string).

15.1.9 Getting DataFrame Information

Concept
info(): Displays metadata about the DataFrame, such as column names, data types, and non-null values.

```
df.info()
```

This is useful for assessing the dataset's structure and identifying missing data.
describe(): Provides a statistical summary of numerical columns.

```
df.describe()
```

This function is essential for gaining insights into numerical data distributions.

15.1.10 Finding Specific Values in DataFrame

Concept
Conditional indexing enables you to locate rows that meet specific criteria.
Example

1. **Day with Maximum Temperature:**

```
# Assuming df is a DataFrame with a 'Temperature' column
# df[df['Temperature'] == df['Temperature'].max()]
```

Retrieves the row where the "Temperature" column has the maximum value.

16 Visualisation - Matplotlib and Seaborn

16.1 Visualization with Matplotlib

Matplotlib is a fundamental plotting library in Python, providing control over every aspect of a plot. It's excellent for creating static, interactive, and animated visualizations in Python.

16.1.1 Basic Plotting with Matplotlib

Example: Image Display (from previous version)
NumPy arrays can represent image data, which can be visualized using Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np

# Generate random image data
image = np.random.random((256, 256, 3))
plt.imshow(image)
plt.title('Random Image')
plt.show()
```

Example: Line Plot
Line plots are used to display trends over time or continuous ranges.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100) # Create 100 points between 0 and 10
y = np.sin(x)

plt.plot(x, y)
plt.title('Sine Wave')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```

Example: Scatter Plot
Scatter plots are useful for showing the relationship between two variables.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.rand(50)
y = np.random.rand(50)
colors = np.random.rand(50)
sizes = (30 * np.random.rand(50))**2  # Square to vary area

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5)
plt.title('Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.colorbar(label='Color Intensity')
plt.show()
```

Example: Bar Chart
Bar charts are used to compare categorical data.

```
import matplotlib.pyplot as plt

categories = ['A', 'B', 'C', 'D']
```

```
values = [25, 40, 30, 35]

plt.bar(categories, values)
plt.title('Bar Chart')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

16.2 Visualization with Seaborn

Seaborn is built on top of Matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn aims to make visualization a central part of exploring and understanding data.

16.2.1 Basic Plotting with Seaborn

Example: Scatter Plot with Seaborn
Seaborn simplifies creating aesthetically pleasing scatter plots.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Create a simple DataFrame for Seaborn
data = pd.DataFrame({'X': np.random.rand(50), 'Y': np.random.rand(50), '
    Category': ['Cat1'] * 25 + ['Cat2'] * 25})

sns.scatterplot(x='X', y='Y', hue='Category', data=data)
plt.title('Seaborn Scatter Plot')
plt.show()
```

Example: Histogram/Distribution Plot
Seaborn's `histplot` (or `displot`) is excellent for visualizing distributions.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

data = np.random.randn(1000) # Generate 1000 random numbers from a normal
    distribution

sns.histplot(data, kde=True) # kde=True adds a kernel density estimate
    line
plt.title('Histogram with Seaborn')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

Example: Box Plot
Box plots are useful for displaying the distribution of data across different categories.

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Sample data for boxplot
data = pd.DataFrame({
    'Group': ['Group A'] * 50 + ['Group B'] * 50,
    'Values': np.concatenate([np.random.normal(0, 1, 50), np.random.normal
        (2, 1.5, 50)])
})

sns.boxplot(x='Group', y='Values', data=data)
plt.title('Box Plot with Seaborn')
plt.show()
```

Note:
Seaborn often works best with data in a Pandas DataFrame format, making it very convenient for data analysis workflows. It simplifies the creation of complex statistical visualizations with minimal code.

17 OOP

17.1 Object-Oriented Programming (OOP)

17.1.1 Classes and Objects

Description:

- **Class:** A blueprint that defines attributes (data) and methods (functions) to represent real-world entities.
- **Object:** A concrete instance of a class with actual values. Think of a class as a template and an object as a specific example built from that template.

Example:

```
# Define a class named 'Car'
class Car:
    # The __init__ method initializes new objects with attributes.
    def __init__(self, brand, model):
        self.brand = brand  # Attribute to store the brand of the car
        self.model = model  # Attribute to store the model of the car

    # A method that returns a string containing the car information.
    def get_info(self):
        return f"{self.brand} {self.model}"

# Create objects (instances) of the Car class
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")
```

```
15
16 print(car1.get_info())
17 print(car2.get_info())
```

Explanation:

- The `Car` class serves as a blueprint for creating car objects.
- The `__init__` method is called automatically when a new object is created and sets up its brand and model attributes.
- `car1` and `car2` are objects created from the `Car` class.
- The `get_info()` method, when called, returns the car's details in a formatted string.

17.1.2 Encapsulation

Description:
Encapsulation is the process of wrapping data (attributes) and methods (functions) into a single unit (a class) and controlling access to that data. This helps protect the data from being modified directly from outside the class.
Example:

```
1  class BankAccount:
2      def __init__(self, account_number, balance):
3          self.account_number = account_number
4          self.__balance = balance  # Private attribute (by convention)
5
6      def deposit(self, amount):
7          if amount > 0:
8              self.__balance += amount
9          return self.__balance
10
11     def withdraw(self, amount):
12         if 0 < amount <= self.__balance:
13             self.__balance -= amount
14             return self.__balance
15         else:
16             return "Insufficient funds"
17
18 # Create a BankAccount object
19 account = BankAccount("12345", 1000)
20 print(account.deposit(500))
21 print(account.withdraw(300))
```

Explanation:

- The `BankAccount` class encapsulates the account data and operations.
- The attribute `__balance` is prefixed with two underscores to indicate it is private and should not be directly accessed from outside the class.

- Methods `deposit()` and `withdraw()` provide controlled ways to modify the balance, ensuring that invalid operations (like withdrawing too much) are handled safely.

17.1.3 Inheritance

Description:
Inheritance allows a new class (known as the child class) to acquire the properties and behaviours (attributes and methods) of an existing class (the parent class). This promotes code reusability and a hierarchical organization of classes.
Example:

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Animal makes a sound"

# Child class that inherits from Animal
class Dog(Animal):
    # Override the speak method to provide specific behaviour
    def speak(self):
        return f"{self.name} barks"

dog = Dog("Buddy")
print(dog.speak())
```

Explanation:

- The `Animal` class defines a basic template for all animals.
- The `Dog` class inherits from `Animal`, meaning it automatically gets the `name` attribute and any methods from `Animal`.
- The `Dog` class then overrides the `speak()` method to return a message specific to dogs.

17.1.4 Polymorphism

Description:
Polymorphism means "many shapes" and allows methods to have different behaviours in different classes. With polymorphism, a method defined in a parent class can be redefined (overridden) in a child class, enabling the same method name to work differently depending on the object that calls it.
Example:

```
class Bird:
    def move(self):
        return "Flies in the sky"

class Penguin(Bird):
```

```
    # Overriding the move method for a different behaviour
    def move(self):
        return "Swims in the water"

def show_movement(animal):
    print(animal.move())

sparrow = Bird()
pingu = Penguin()

show_movement(sparrow)
show_movement(pingu)
```

Explanation:

- Both `Bird` and `Penguin` have a method called `move()`, but their implementations differ.
- The `show_movement()` function demonstrates polymorphism by calling the `move()` method on different objects.
- This flexibility allows you to write code that can work with objects of different types while using a common interface.

17.1.5 Abstraction

Description:
Abstraction is about hiding the complex implementation details and showing only the necessary features of an object. In Python, this is typically achieved through abstract classes and methods, using the `abc` (Abstract Base Class) module.
Example:

```
from abc import ABC, abstractmethod

# Abstract class that cannot be instantiated directly
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass  # Implemented by subclasses

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    # Concrete implementation of the abstract method
    def area(self):
        return self.width * self.height

rect = Rectangle(5, 4)
print(rect.area())
```

Explanation:

- The `Shape` class is declared as an abstract class using `ABC` and contains an abstract method `area()`.
- Abstract methods do not have an implementation in the abstract class; they must be defined in any subclass.
- The `Rectangle` class inherits from `Shape` and provides a specific implementation for `area()`, thus allowing objects of `Rectangle` to compute their area.
- Abstraction simplifies complex systems by focusing on high-level mechanisms.

17.1.6 Method Overloading vs. Overriding

Description:

- **Method Overloading:** Involves having multiple methods with the same name but different parameters. Although Python does not support traditional overloading, you can simulate it with default parameters or variable-length arguments.
- **Method Overriding:** Occurs when a child class provides a new implementation for a method already defined in its parent class.

Example:

Method Overloading (using default arguments)

```
class MathOperations:
    def add(self, a, b, c=0):  # c is optional
        return a + b + c

math_obj = MathOperations()
print(math_obj.add(5, 10))
print(math_obj.add(5, 10, 20))
```

Method Overriding

```
class Parent:
    def show(self):
        return "Parent class method"

class Child(Parent):
    # Overriding the show method of the Parent class
    def show(self):
        return "Child class method"

child_obj = Child()
print(child_obj.show())
```

Explanation:

- In overloading, the `add()` method can handle both two and three arguments by using a default value for the third parameter.

- In overriding, the `Child` class redefines the `show()` method from `Parent`, so when called on a `Child` object, it produces a different output.

17.1.7 Access Modifiers

Description:
Access modifiers control the visibility of class attributes and methods:

- **Public:** No special symbol; accessible from anywhere.
- **Protected:** Prefixed with a single underscore (_), suggesting that it should not be accessed directly outside the class.
- **Private:** Prefixed with two underscores (__); these attributes are not easily accessible from outside the class due to name mangling.

Example:

```
class Example:
    def __init__(self):
        self.public = "Public Attribute"
        self._protected = "Protected Attribute"
        self.__private = "Private Attribute"

obj = Example()
print(obj.public)
print(obj._protected)
# print(obj.__private)   # This would raise an error because __private is
   not accessible directly
```

Explanation:

- Public attributes are intended to be accessible from any part of the program.
- Protected attributes are a hint to programmers that they are internal to the class (or its subclasses) and should be accessed carefully.
- Private attributes are hidden from outside the class to protect sensitive data and enforce encapsulation.

17.1.8 Summary Table

18 Markov chains & Transition Matrix

18.1 Markov Chain

18.1.1 Introduction

Markov Chains: How Probabilities Shape Our Future Decisions
Markov Chains are mathematical models that help predict the likelihood of moving between different states based on probabilities. They are widely used in machine learning and artificial intelligence to model uncertainty and dynamic systems. For instance, they are applied in:

Concept	Description	Key Example
Class & Object	Blueprint for objects; an instance created from the class	`Car("Toyota", "Corolla")`
Encapsulation	Bundling data and methods, restricting direct access to sensitive data	`BankAccount` with `__balance`
Inheritance	A child class reusing and extending features of a parent class	`Dog` inheriting from `Animal`
Polymorphism	Same method name acting differently for different classes	`move()` in `Bird` vs. `Penguin`
Abstraction	Hiding complex details to expose only necessary parts	Abstract `Shape` class
Overloading & Overriding	Overloading: Same name, different parameters; Overriding: Redefining method	`add()` method & `Child.show()`
Access Modifiers	Defines visibility: public, protected, and private attributes	`public`, `_protected`, `__private`

Table 3: Summary of OOP Concepts

- Reinforcement learning
- Sequence modeling
- Text and speech recognition

Real-Life Example: Mall Advertisement Problem
Imagine you are in a mall with four regions:

- Entrance (E)
- Clothing (C)
- Electronics (X)
- Food Court (F)

Advertisers want to know the best location to place their advertisements to maximize customer engagement. How can probabilities help solve this problem?

18.1.2 Independent Events

Independent events are those where the outcome of one does not affect the outcome of another. A simple example is tossing a fair coin.
Law of Large Numbers (LLN)
The LLN states that as the number of trials increases, the sample average converges to the expected value. For instance, in a coin toss simulation:

- Tossing the coin 10 times may not produce exactly 50% heads.
- Tossing it a very high number of times leads the proportion of heads to 0.5.

Task 1
Simulate 1,000 coin tosses and plot the proportion of heads over time to visualize the LLN. Use Python or any programming language of your choice.

18.1.3 Dependent Events: Transition and States

Dependent events rely on previous outcomes. Markov Chains model such dependencies through states and transitions.
Example: Study and Social Media Automata
Imagine two states:

- Study
- Scroll Social Media

Initially, 500 students are studying, and 500 are scrolling social media. The transitions are as follows:

- A student studying continues studying with a probability of 0.6 and switches to social media with a probability of 0.4.
- A student on social media switches to studying with a probability of 0.2 and continues scrolling with a probability of 0.8.

Transition Calculation
At time T1:

- Students studying $= 500 \times 0.6 + 500 \times 0.2 = 400$
- Students scrolling $= 500 \times 0.4 + 500 \times 0.8 = 600$

Task 2
Calculate and tabulate the number of students studying and scrolling at times T2, T3, ..., T10. Use software like Excel or Python to generate the table and observe convergence.
As time progresses, the number of students converges to approximately:

- Studying: 333
- Scrolling: 667

18.1.4 The Mall Problem Revisited

The mall has four regions:

- Entrance (E)
- Clothing (C)
- Electronics (X)
- Food Court (F)

The transition probabilities are given in the matrix below:

$$\begin{bmatrix} 0.1 & 0.5 & 0.3 & 0.1 \\ 0.3 & 0.2 & 0.4 & 0.1 \\ 0.2 & 0.2 & 0.1 & 0.5 \\ 0.2 & 0.1 & 0.1 & 0.6 \end{bmatrix}$$

Each row represents the probabilities of transitioning from one region to another. For example, the second element in the first row (0.5) indicates that a customer moves from the Entrance to Clothing with a probability of 0.5.

Simulation

Start with 1,000 customers at the entrance and 0 in all other regions. Calculate the distribution of customers over multiple time steps (similar to the study/social media example).

Task 3

Simulate the transitions and plot the customer distribution over time. Verify that the steady-state distribution converges to (approximately):

- Entrance: 194
- Clothing: 332
- Electronics: 175
- Food Court: 199

18.1.5 Transition Matrix and Markov Chain Model

The transition matrix is a square matrix that represents the probabilities of transition between states. For instance, the transition matrix for the study/social media example is:

$$\begin{bmatrix} 0.6 & 0.4 \\ 0.2 & 0.8 \end{bmatrix}$$

Markov Chains exhibit the memoryless property: the future state depends only on the current state, not on the sequence of past states.

18.2 Markov Chain: Key Concepts

18.2.1 Key Concepts

1. **States:** The possible conditions or positions the system can be in. Represented as a finite or countable set (e.g., {S1, S2, S3}).
2. **Transition Probabilities:** Probabilities associated with moving from one state to another. Represented as a matrix (Transition Matrix). *Example:*

$$P = \begin{bmatrix} P(S1 \to S1) & P(S1 \to S2) & P(S1 \to S3) \\ P(S2 \to S1) & P(S2 \to S2) & P(S2 \to S3) \\ P(S3 \to S1) & P(S3 \to S2) & P(S3 \to S3) \end{bmatrix}$$

3. **Initial State Distribution:** A vector representing the probabilities of starting in each state. *Example:*

$$\pi = [P(S1 \text{ at } t = 0), P(S2 \text{ at } t = 0), P(S3 \text{ at } t = 0)]$$

4. **Stationary Distribution:** A probability distribution that remains unchanged as the system evolves. Satisfies: $\pi \times P = \pi$

18.2.2 Properties

1. **Memorylessness:** Future states depend only on the current state.
2. **Irreducibility:** Every state can be reached from every other state.
3. **Aperiodicity:** The system does not oscillate in fixed cycles.
4. **Reversibility:** The Markov chain satisfies detailed balance.

18.2.3 Steps to Solve Markov Chain Problems

1. **Identify States:** Enumerate all possible states of the system.
2. **Define Transition Matrix:** Construct the matrix of probabilities.
3. **Analyze Transitions:** Compute probabilities for specific sequences or steady-state conditions.
4. **Compute Stationary Distribution:** Solve for π using the stationary condition equation.

19 Taking inputs from users

19.1 Taking Inputs from Users

Overview:
Interacting with users by receiving data during program execution is fundamental. This topic covers methods to prompt, capture, and validate user input.

19.1.1 The Input Function

Definition:
The `input()` function pauses program execution and waits for the user to enter data from the keyboard.
Syntax and Example:

```
user_input = input("Enter your data: ")
print("You entered: " + user_input)
```

19.1.2 Type Conversion

Explanation:
Data obtained via `input()` is always a string. To use it in numerical or logical operations, conversion to types such as `int`, `float`, or `bool` is necessary.
Example:

```
age = int(input("Enter your age: "))
temperature = float(input("Enter the temperature: "))
```

19.1.3 Input Validation

Purpose:
Ensuring the input meets expected criteria (e.g., numeric, non-empty) is crucial to avoid errors during runtime.
Techniques:
Use conditionals and exception handling to verify input and prompt the user again if necessary.
Example:

```
try:
    value = int(input("Enter a positive number: "))
    if value <= 0:
        raise ValueError("Not positive")
except ValueError:
    print("Please enter a valid positive number.")
```

20 Hypothesis Testing

20.1 Hypothesis Testing

20.1.1 The Activation Threshold Paradox

The Silent Café Dilemma

Scenario: Four friend groups with different action thresholds:

- Threshold 0: Immediate action (Radhika: "I'll go now!")
- Threshold 1: Needs 1 follower (Aryan: "If 1 friend joins...")
- Threshold 2: Needs 2 followers (Neha: "I'll wait for 2")
- Threshold 3: Needs critical mass (Raj: "Only if 3 commit")

Surprise Outcome:

- Group 2 (Mean=1.8): No one acts - missing initiators
- Group 4 (Mean=2): Partial activation - critical mass not reached

20.1.2 Key Statistical Insight

- Traditional means/medians fail to predict group behavior
- Hypothesis testing reveals hidden activation dynamics
- Requires analysis of extreme values, not just central tendency

20.1.3 Core Concepts

Foundations

- **Null Hypothesis (H_0)**: $\mu_A = \mu_B$ (No difference)
- **Alternative Hypothesis (H_1)**: $\mu_A \neq \mu_B$ (Significant difference)
- **p-value**: Probability of observing data if H_0 is true
- **Significance Level (α)**: 0.05 (5% risk tolerance)

20.1.4 Z-Test Formula

$$Z = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}}$$

Where:

- $\bar{X}$ = Sample mean

- μ = Population mean
- σ = Standard deviation
- n = Sample size

20.1.5 Hands-On Testing Framework

Variance Checking with Levene's Test

```
from scipy.stats import levene

salaries_A = [55,60,62,58,63,59,61,60,57,64]
salaries_B = [52,55,57,53,58,54,56,55,51,59]

stat, p = levene(salaries_A, salaries_B)
print(f"Variance Equality: p={p:.4f}")  # p > 0.05 $->$ equal variances
```

Independent T-Test

```
from scipy.stats import ttest_ind

t_stat, p_value = ttest_ind(salaries_A, salaries_B,
                            equal_var=True)
print(f"T-Test: t={t_stat:.2f}, p={p_value:.4f}")
```

Welch's T-Test for Unequal Variances

```
t_stat, p_value = ttest_ind(salaries_A, salaries_B,
                            equal_var=False)
print(f"Welch's Test: t={t_stat:.2f}, p={p_value:.4f}")
```

20.1.6 Real-World Applications

Chocolate Factory Quality Control

Problem: Claimed weight=50g, Sample (n=10): $\bar{X}$=52g, σ=2g

```
from scipy.stats import norm

def z_test(pop_mean, sample_mean, sample_std, n, alpha=0.05):
    se = sample_std / (n ** 0.5)
    z = (sample_mean - pop_mean) / se
    critical_z = norm.ppf(1 - alpha/2)
    return z, critical_z

z_score, critical_z = z_test(50, 52, 2, 10)
print(f"Z={z_score:.2f}, Critical={critical_z:.2f}")  # Reject  H
```

Chi-Square Brand Preference Analysis

```
from scipy.stats import chi2_contingency

observed = [[30,40], [50,30], [20,30]]  # [Apple, Samsung, OnePlus]
```

```
chi2, p, dof, expected = chi2_contingency(observed)
print(f"X2={chi2:.2f}, p={p:.4f}")  # Check association
```

20.1.7 Key Insights

Decision Framework

- **p < 0.05**: Reject H_0 (Statistical significance)
- **Type I Error**: 5% false positive risk
- **Type II Error**: Missed true effects
- **Test Selection**: Match method to data characteristics

When to Use Which Test

Test	Use Case	Assumptions
Z-Test	Large samples (n $\geq$ 30)	Known variance
T-Test	Small samples	Normal distribution
Welch's	Unequal variances	Independent groups
Chi-Square	Categorical data	Minimum cell counts