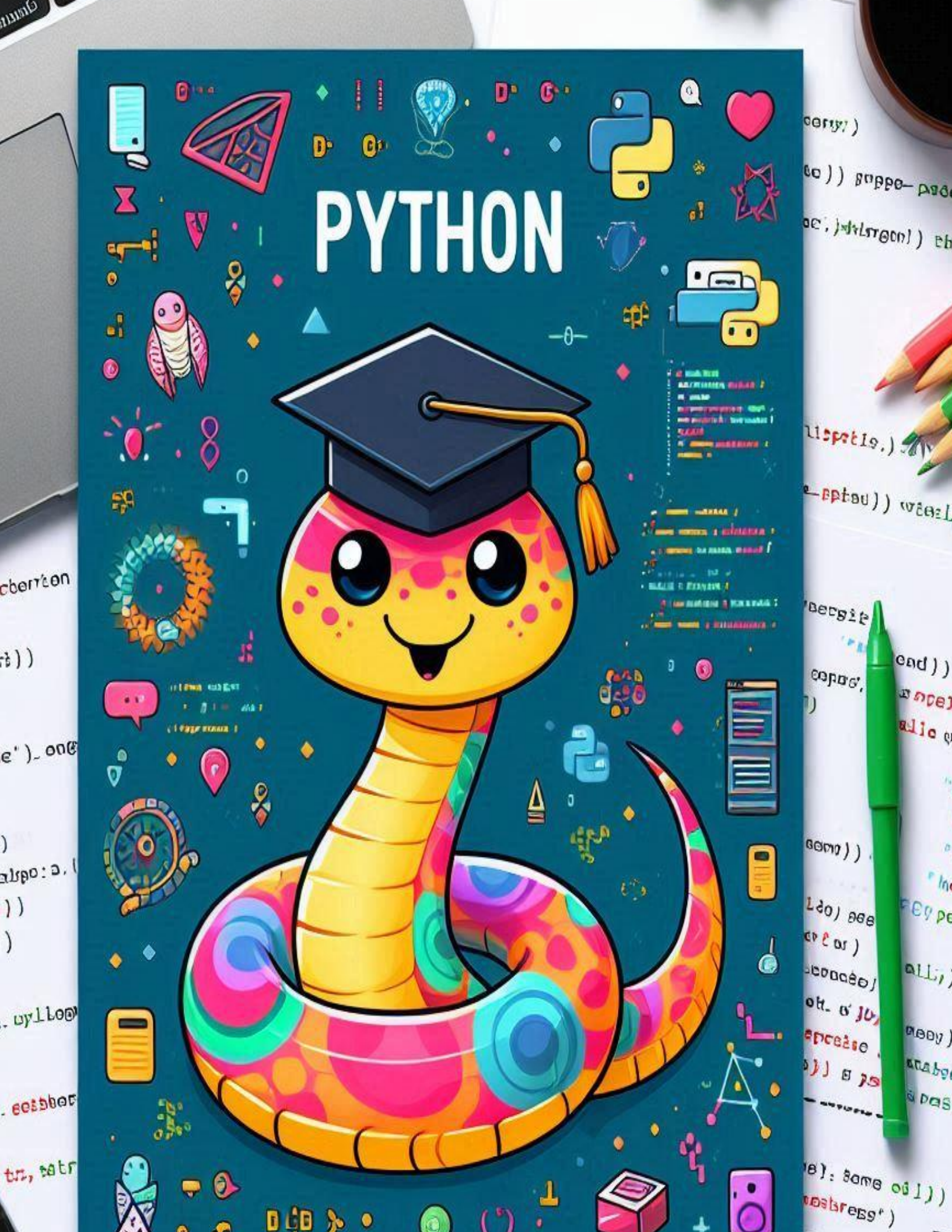
PYTHON

Advanced Python Notes

INSTRUCTOR NAME	MUHAMMAD RIZWAN KHAN
STUDENT NAME	SYED MANSOOR UL HASSAN BUKHARI
COURSE	ARTIFICIAL INTELLIGENCE
INSTITUTE	CORVIT MUZAFFARABAD AZAD KASHMIR
NOTEBOOK FILES	GITHUB
PRACTICE FILES	GITHUB

Overview:

These notes summarize what we've learned in our Advanced Python sessions with Sir Muhammad Rizwan. Our learning journey has covered many important topics and practical skills, including:

Data Structures: We explored various ways to store and organize data, such as lists, dictionaries, sets, and tuples. Understanding these structures helps us manage and manipulate data efficiently.

Object-Oriented Programming (OOP): We learned how to use classes and objects to model real-world scenarios. This approach helps us write clean, modular, and reusable code.

Exception Handling: We covered techniques to handle errors and exceptions in our programs. This ensures that our code runs smoothly and can recover gracefully from unexpected issues.

These notes act as a guide for mastering Python. I also keep our GitHub repository up-to-date with lecture notes, practice exercises, and other helpful materials we've covered in class or that I've found useful during our studies.

AI Day 01 Notes

Syed Mansoor ul Hassan Bukhari

Installation

1. **Install Anaconda**:
   - Download and install the latest version of Anaconda, which includes Python, Jupyter Notebook, and other commonly used packages for scientific computing and data science.
2. **Install Jupyter Notebook using pip**:

```
python3 -m pip install --upgrade pip
python3 -m pip install jupyter
```

Starting Jupyter Notebook

- To start Jupyter Notebook, type the following command in the terminal:

```
jupyter notebook
```

- This will open the Notebook Dashboard in your default web browser, showing a list of notebooks, files, and subdirectories in the directory where the notebook server was started.

Creating a Notebook

- Click on the **New** button at the top right corner and select **Python 3** to create a new notebook.

Cells in Jupyter Notebook

- **Code Cells**: Used to write and execute code.
- **Markdown Cells**: Used to write text, including explanations, formulas, and charts.
- **Raw Cells**: Used to write raw text that is not meant to be executed.

Running Code

- **Run a cell**: Press **Shift + Enter** after selecting the cell you want to execute.
- **Run all cells**: Select **Cell** > **Run All** from the menu.

Keyboard Shortcuts

- **Running Code (F3)**: Press **F3** (or **Ctrl+Enter**) to execute the selected cell(s) of code.
- **Insert Cell**: Press **A** to insert a cell above the current cell, and **B** to insert a cell below the current cell.

- **Delete Cell**: Press **D** twice to delete the current cell.
- **Save Notebook**: Press **Ctrl+S** to save the notebook.
- **Change Cell Type**: Press **Y** to change the cell to code, and **M** to change the cell to markdown.

Theme Selection (Monokai)

- Access the **"Kernel"** menu (usually top-right) and navigate to **"Change theme"** for customization options.
- The **Monokai theme** is a popular choice for its clear and concise visual style.

Example: Hello World Program

```
print("Hello World!")
```

- This classic program outputs **"Hello World!"** to the console.
- The **print** function displays values passed to it.

Multiple Values in Output (print Function)

```
print("Hello", 5, 6.78, sep=", ")
```

- Use the **sep** argument in **print** to specify a separator between elements. Here, **",** " separates values with commas and spaces.

New Line and Comments

- **\n** (newline) creates a new line in the output.
- **Single line comments**: `# Your comment here`
- **Multi-line comments**: Use triple quotes (`'''` or `"""`).

Variables and Data Types

- **Variables** store data with a specific name for reference.
- **Data types** define the kind of data a variable holds (e.g., numbers, text).

Variable Creation and Data Types

```
a = 1  # Integer (int)
b = True  # Boolean
c = "Hello"  # String
d = None  # None (special data type)
e = 3.14  # Float (decimal number)
f = complex(2, 3)  # Complex number (real + imaginary parts)
g = [1, 2.5, "Cat"]  # List (ordered collection of elements)
h = ("Dog", "Bird")  # Tuple (immutable ordered collection)
i = {"name": "Alice", "age": 30}  # Dictionary (unordered key-value pairs)
```

Type Function

```
print(type(a))  # Output: <class 'int'>
```

Built-in Data Types

- **Numbers**:
  - **int**: Integers (whole numbers)
  - **float**: Floating-point numbers (decimals)
  - **complex**: Complex numbers (real + imaginary parts)
- **Collections**:
  - **list**: Ordered, changeable collection of elements (can hold mixed data types)
  - **tuple**: Ordered, immutable collection of elements (can hold mixed data types)
  - **set**: Unordered collection of unique elements (removes duplicates)
  - **dictionary**: Unordered collection of key-value pairs (flexible data storage)

AI Day 02 Notes
Syed Mansoor ul Hassan Bukhari

Type Casting and String Manipulation in Python

1. Type Casting:

- **Type casting** involves converting one data type into another.
- Python provides various built-in functions for type casting, such as `int()`, `float()`, `str()`, `ord()`, `tuple()`, `list()`, `set()`, `dict()`, `hex()`, and `oct()`.
- **Explicit type casting** is done manually by the developer using these functions.
- **Implicit type casting** occurs automatically by the Python interpreter based on data type hierarchy.

2. Example:

```
string = "13"
number = 7
str_num = int(string)  # Convert string to integer
sum = number + str_num
print("The sum of two numbers is:", sum)
```

- In this example, the string `"13"` is explicitly converted to an integer using the `int()` function. The sum of `number` and `str_num` is then calculated and printed.

3. User Input:

- Use the `input()` function to take user input (returns a string).
- Convert input to the desired data type (e.g., `int(input())`, `float(input())`).

4. Example:

```
name = input("Enter Your Name: ")
print("My Name is:", name)
```

- This example takes a user's name as input and prints it.

5. Strings:

- Strings are enclosed in double or single quotes.
- Escape double quotes within a string using `\"`.
- Access individual characters using indexing (e.g., `name[0]`).
- Iterate through characters using loops.
- Find string length using `len()`.

6. Example:

```
fruit = "Mango"
print(len(fruit))  # Shows length including spaces
```

- This example prints the length of the string `"Mango"`.

7. String Slicing:

- Treat strings like arrays.
- **Positive slicing**: `name[start:end]` (inclusive start, exclusive end).
- **Negative slicing**: `name[:-n]` (exclude last n characters).

8. Example:

```
name = "Mango"
print(name[0:4])  # First four letters
```

- This example prints the first four letters of the string `"Mango"`.

9. Loop Through String:

- Strings are iterable; loop through characters.

10. Example:

```
alpha = "ABCD"
for char in alpha:
    print(char)  # Output: A B C D
```

- This example loops through each character in the string `"ABCD"` and prints them.

11. Task:

Given a sentence, perform the following:

- Count characters (including spaces).
- Count words.
- Convert to uppercase.
- Convert to lowercase.
- Print in reverse order.

Detailed Explanation:

1. **Count Characters**:

   ```
   sentence = "Hello World"
   char_count = len(sentence)
   print("Character count:", char_count)
   ```

   This counts the total number of characters in the sentence, including spaces.

2. **Count Words**:

   ```
   word_count = len(sentence.split())
   print("Word count:", word_count)
   ```

This splits the sentence into words and counts them.

3. **Convert to Uppercase**:
```
upper_case = sentence.upper()
print("Uppercase:", upper_case)
```

This converts the entire sentence to uppercase.

4. **Convert to Lowercase**:
```
lower_case = sentence.lower()
print("Lowercase:", lower_case)
```

This converts the entire sentence to lowercase.

5. **Print in Reverse Order**:
```
reverse_sentence = sentence[::-1]
print("Reversed:", reverse_sentence)
```

This reverses the entire sentence.

AI Day 03 Notes
Syed Mansoor ul Hassan Bukhari

Type Casting

- **Type Casting:** The conversion of one data type into another data type.
- Python supports various functions/methods for type casting:
    - **int():** Converts to an integer.
    - **float():** Converts to a floating-point number.
    - **str():** Converts to a string.
    - **ord():** Converts a character to its Unicode code point.
    - **tuple(), list(), set(), dict():** Convert to respective data structures.
    - **hex(), oct():** Convert to hexadecimal and octal representations.

Array of Characters

- **Example: name = "MRK"**
    - Access individual characters using indexing: name[0], name[1], name[2]
    - Note: Accessing an out-of-range index (e.g., name[3]) results in an error.
    - Iterating through characters in a string using a loop:

```
for character in name:
  print(character)
```

String Length

- Use len() function to find the length of a string.
- Example: names = "HARIS, Rizwan", len(names) gives the total character count (including spaces).

String Slicing

- Positive Slicing (left to right):
    - name[0:4] prints the first four letters.
    - name[:4] automatically starts from index 0.
    - name[2:5] includes characters from index 2 to 4.
    - name[:] prints the entire string.
    - name[1:] starts from index 1 and goes to the end.

- **Negative Slicing (right to left):**
    - name[:-4] gives only the first character (“M”).
    - name[:len(name)-4] is equivalent.

Looping Through Strings

- Strings are iterable (like arrays), so we can loop through them.
- Example: alpha = "ABCD"

```
for i in alpha:
  print(i)  # Output: A B C D
```

Quick Task

- Output of name = "Muhammad" and print(name[-6:-1]): “hamma”

String Methods

- **Common methods:**
    - **upper():** Converts to uppercase.
    - **lower():** Converts to lowercase.
    - **strip():** Removes leading and trailing whitespaces.
    - **rstrip("!"):** Removes trailing exclamation marks.
    - **replace("Rizwan", "Muhammad"):** Replaces a word.
    - **split(" "):** Splits the string into a list based on spaces.
    - **capitalize():** Capitalizes the first letter.
    - **endswith("d"):** Checks if the string ends with a specific character.

Example Programs

```
# Example of string slicing
name = "Mango"
print(name[1:4])  # Output: "ang"

# Example of loop through string
fruit = "Apple"
for char in fruit:
   print(char)  # Output: Apple
```

AI Day 04 Notes
Syed Mansoor ul Hassan Bukhari

Notes on Conditional Statements in Python

- **Conditional Operators:**
  - Conditional operators allow you to compare values and make decisions based on the comparison results.
- **Common conditional operators include:**
  - **< (less than)**
  - **> (greater than)**
  - **<= (less than or equal to)**
  - **>= (greater than or equal to)**
  - **== (equal to)**
  - **!= (not equal to)**

- **if and else Statements:**
  - Used for decision-making based on a condition.
- **Syntax:**

```
if condition:
    # Code to execute if condition is True
else:
    # Code to execute if condition is False
```

- **Example:**

```
Age = int(input("Enter Your age: "))
if Age > 18:
    print("You are eligible to drive.")
else:
    print("You are not eligible to drive.")
```

- **elif Statement:**
  - Allows multiple conditions to be checked sequentially.
- **Syntax:**

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if no conditions are True
```

- **Example:**

```
N = int(input("Enter the value: "))
if N < 0:
    print("The Number is Negative")
elif N == 0:
    print("The Number is Zero")
```

```
else:
    print("The Number is greater or equal to zero")
```

- **Nested if Statements:**
  - **Allows conditions to be nested inside each other.**

- **Example:**

```
num = float(input("Enter a number: "))
if num < 0:
    print("Number is Negative")
elif num > 0:
    if num <= 10:
        print("The number is between 1 and 10")
    elif 10 < num <= 11:
        print("The number is between 10 and 11")
    elif 10 < num <= 20:
        print("The Number is between 11 and 20")
    else:
        print("Number is greater than 20")
else:
    print("Number is zero")
```

- **Nested if and else Statements:**
  - **Conditions can be nested within both if and else blocks.**
- **Example:**

```
num = int(input("Enter a value: "))
if num < 0:
    print("The value is less than zero")
elif num > 0:
    print("The Number is Greater than zero")
    if num <= 10:
        print("The Number is less or equal to 10")
    elif 10 < num <= 11:
        print("The Number is in between 10 and 11")
    elif 10 < num <= 20:
        print("The number is in between 10 and 21")
    else:
        print("The number is greater than 20")
else:
    print("The Number is equal to zero")
```

AI Day 05 Notes
Syed Mansoor ul Hassan Bukhari

Detailed Notes on Python's `match` Statement

The `match` statement in Python, introduced in version 3.10, provides a more powerful and flexible alternative to traditional `if`, `elif`, `else` statements for pattern matching. It allows you to compare a given variable's value against different patterns, executing the corresponding code block for the first pattern that matches.

Main Entities of `match` Statement

- **The `match` Keyword:** Initiates the pattern matching process.
- **Case Clauses:** One or more `case` clauses follow the `match` keyword, each containing a pattern to compare against the variable.
- **Expressions for Each Case:** The code block executed when a pattern matches.

Important Notes

- Python 3.10 or later is required to use the `match` statement.
- The `match` statement eliminates the need for `switch` statements used in languages like C++.

```
match variable_name:
   case 'pattern1':
      statement1
   case 'pattern2':
      statement2
   case 'pattern3':
      ...
   case 'patternN':
      statement
```

Example Usage

Basic Match Statement Example

```
x = int(input("Enter the value of X: "))
match x:
   case 0:
      print("case is zero")
   case 4:
      print("case is 4")
   case _:
      print(x)
```

Pattern Matching with Conditions

```
x = int(input("Enter the value of X: "))
match x:
   case 0:
      print("case is zero")
   case 4:
```

```
        print("case is 4")
    case _ if x != 90:
        print(x, "is not 90")
    case _ if x != 80:
        print(x, "is not 80")
    case _ if x != 70:
        print(x, "is not 70")
    case _:  # Working like an else statement here
        print(x)
```

Combining Patterns with Conditions

```
x = 4

match x:
    case 0:
        print("x is zero")
    case 4 if x % 2 == 0:
        print("x % 2 == 0 and case is 4")
    case _ if x < 10:
        print("x is less than 10")
    case _:
        print(x)
```

Example with Even and Odd Number Check

```
x = int(input("Enter the Number: "))
match x:
    case _ if x % 2 == 0:
        print(x, "the number is even")
    case _:
        print(x, "The number is odd")
```

Implementing a Function to Get the Day of the Week

Below is an implementation of a function called `get_day_of_week` that takes an integer representing the day of the week (1 for Monday, 2 for Tuesday, ..., 7 for Sunday) and returns the corresponding name of the day using the `match` statement.

```
def get_day_of_week(day_number):
    match day_number:
        case 1:
            return "Monday"
        case 2:
            return "Tuesday"
        case 3:
```

```
    return "Wednesday"
case 4:
    return "Thursday"
case 5:
    return "Friday"
case 6:
    return "Saturday"
case 7:
    return "Sunday"
case _:
    return "Invalid day number"
```

Test the function

```
print(get_day_of_week(1))  # Output: Monday
print(get_day_of_week(3))  # Output: Wednesday
print(get_day_of_week(8))  # Output: Invalid day number
```

In this implementation:

- The function `get_day_of_week` uses a `match` statement to check the value of `day_number`.
- Each `case` corresponds to a day of the week, returning the name of the day.
- The default case (`case _`) handles invalid day numbers, returning "Invalid day number".

AI Day 06 Notes
Syed Mansoor ul Hassan Bukhari

Detailed Notes on Loops in Python

Introduction to Loops

In programming, loops are used to execute a block of statements repeatedly based on a condition. Python supports several types of loops:

- **For Loop**
- **While Loop**
- **Nested Loops**

1. For Loop

The `for` loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string). This loop is more efficient and readable when you need to perform an action on each item in a sequence.

Syntax

```
for item in sequence:
    # Execute some statements
```

Example: String Iteration

```
name = "Muhammad"
for i in name:
    print(i, end=", ")
    if i == "u":
        print("This is Amazing!")
```

Example: List Iteration

```
colors = ["Red", "Green", "Blue", "Yellow", "Magenta", "Violet", "Indigo"]
for color in colors:
    print(color)
    for char in color:
        print(char)
```

Using `range()` with For Loop

The `range()` function generates a sequence of numbers, which is particularly useful when you want to execute a loop a specific number of times.

Syntax

```
range(start, stop, step)
```

Examples

```
for k in range(5):
    print(k)  # Prints 0 to 4
for k in range(1, 5):
    print(k)  # Prints 1 to 4
for k in range(-10, 10):
```

```
    print(k)  # Prints from -10 to 9
for k in range(0, -10, -1):
    print(k)  # Prints from 0 to -9
for k in range(1, 15, 2):
    print(k)  # Prints odd numbers up to 15
```

2. While Loop

The `while` loop in Python repeatedly executes a block of code as long as a given condition is `True`.

Syntax

```
while condition:
    # Execute some statements
```

Example

```
x = 0
while x < 5:
    print(x)
    x += 1
print("Done with the Loop")
```

Taking Input from User

```
x = int(input("Enter the Value: "))
while x <= 38:
    x = int(input("Enter the Value: "))
    print(x)
print("Limit exceeds")
```

Example with `else`

```
x = 5
while x > 0:
    print(x)
    x -= 1
else:
    print("I am Inside Else Block")
```

3. Nested Loops

A loop inside another loop is called a nested loop. This is useful for iterating over multi-dimensional data structures.

Example

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

Break and Continue Statements

Break Statement

The `break` statement terminates the loop and transfers control to the statement immediately following the loop.

Example

```
for i in range(12):
   print("5 X", i + 1, "=", 5 * (i + 1))
   if i == 9:
      break
print("Left the loop and get out of the loop")
```

Continue Statement

The `continue` statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

Example

```
for i in range(12):
   if i == 9:
      print("Skip the Iteration")
      continue
   print("5 X", i + 1, "=", 5 * (i + 1))
```

Task: Sum of Even Numbers from 1 to 100

```
total = 0
for i in range(1, 101):
   if i % 2 == 0:
      total += i
print("Sum of even numbers from 1 to 100 is:", total)
```

Examples for Practice

```
# Print Numbers from 1 to 100
for i in range(1, 101):
   print(i, end=" ")
   if i == 50:
      break
   print("Mission Passed")
print("Thank You")
```

Skip Even Numbers

```
for i in [2, 3, 9, 4, 5, 6, 8, 0]:
   if i % 2 == 0:
      continue
   print(i)
```

 # Infinite Loop with Break Condition

```
i = 0
while True:
```

```
    print(i)
    i += 1
    if i % 100 == 21:
        break
```

Conclusion

Loops are a fundamental concept in programming that allow you to execute a block of code repeatedly. Understanding how to use `for` and `while` loops, along with control statements like `break` and `continue`, can greatly enhance your ability to write efficient and readable code.

AI Day 07 Notes

Syed Mansoor ul Hassan Bukhari

Python Functions

Why Use Functions?

Functions make the code less error-prone and more organized. A function is a block of code that performs a specific task whenever it is called. In larger programs, it is beneficial to create and use existing functions to keep the program flow organized and neat.

Types of Functions

1. **User-Defined Functions**: Created by the user as per their needs. It usually starts with def your_function(argument).
2. **Built-in Functions**: Defined and pre-coded. Examples include min(), max(), len(), sum(), type(), range(), dict(), list(), tuple(), set(), print(), etc.

Function Arguments & Return Statement

1. **Default Arguments**: We can provide a default value while creating a function. This way, the function assumes a default value even if a value is not provided in the function call for that argument.
2. **Keyword Arguments**: If we wish the order does not matter, then we use the keyword argument. We can provide arguments with key = value, this way the interpreter recognizes the arguments by parameter name. Hence, the order in which the arguments are passed does not matter.
3. **Required Arguments**: In this case, we must provide the value of arguments. If we don't pass the arguments with a key = value syntax, passing should be in the exact positional order with the actual function definition.
4. **Arbitrary Arguments**: While creating a function, pass a * before the parameter name while defining the function. The function accesses the arguments by processing them in the form of a tuple.

Decorators

Decorators in Python are a way to modify or extend the behavior of functions or methods without changing their code. They allow you to wrap another function or method in order to execute code before and/or after the wrapped function runs, or to modify its arguments or return value.

Examples

1. **User-Defined Function**: These are functions that users define themselves to perform a specific task.

```
def greet(name):
    print("Hello, " + name + "!")
greet("Mansoor Bukhari")
```

In this example, `greet` is a user-defined function that takes one argument, `name`, and prints a greeting message.

2. **Built-in Function**: Python provides several built-in functions like `print()`, `len()`, etc.

```
my_string = "Hello, world!"
print(len(my_string))
```

Here, `len` is a built-in function that returns the length of the string.

3. **Function with Default Arguments**: These functions assume a default value if a value is not provided in the function call for that argument.

```
def greet(name="World"):
    print("Hello, " + name + "!")
greet()
```

In this example, if no argument is passed while calling the `greet` function, it uses "World" as the default value for `name`.

4. **Function with Keyword Arguments**: In these functions, arguments are identified by the parameter name, so the order of arguments doesn't matter.

```
def full_name(first, last):
    print(first + " " + last)
full_name(last="Salfi", first="Hacker")
```

Here, we're calling `full_name` function with keyword arguments. The order of arguments doesn't matter because we're specifying the values of `first` and `last` parameters by their names.

5. **Function with Arbitrary Arguments**: These functions can take any number of arguments.

```
def add(*numbers):
    return sum(numbers)
print(add(1, 2, 3, 4, 5))
```

In this example, `add` function takes any number of arguments and returns their sum. The `*numbers` parameter in the function definition is used to collect all the extra arguments into a tuple.

6. **Decorators**: Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

In this example, `my_decorator` is a decorator that wraps the `say_hello` function and modifies its behavior. The `@my_decorator` line is a decorator syntax, which is equivalent to `say_hello = my_decorator(say_hello)`.

AI Day 08 Notes

Syed Mansoor ul Hassan Bukhari

Function Arguments & Return Statement

Default Arguments

Default arguments are those that take a default value if no argument value is passed during the function call. You can assign this default value while defining the function.

```
def name(fname, mname = "Muhammad", lname = "Rizwan"):
        print("Welcome", fname, mname, lname)
name("Khan")
```

Keyword Arguments

Keyword arguments allow you to pass arguments in any order. You just have to mention the argument name with its value.

```
def name(fname,mname,lname):
        print("Hello!", fname, mname, lname)
name(mname = "Ali", lname = "Khan", fname = "Muhammad")
```

Required Arguments

Required arguments are those that must be passed during the function call. The argument passing should follow the exact positional order as defined in the function.

```
def average(a,b):
        print("Average of two numbers: ", (a+b)/2)
average(a = int(input("Enter 1st value: ")), b = int(input("Enter 2nd value:
")))
```

Arbitrary Arguments

If you do not know how many arguments will be passed into your function, add a * before the parameter name in the function definition. This way the function will receive a tuple of arguments.

```
def avg(*num):
    sum = 0
    for i in num:
     sum = sum+i
     print("Average is: ", sum/len(num))
avg(10,10,10)
```

Lists

Lists are ordered collections of data items. They can store multiple items in a single variable. List items are separated by commas and enclosed within square brackets. Lists are mutable, meaning we can alter them after creation.

```
l = [3, 5, 6, "MRK", True, 9.8]
print(l)
print(type(l))
print(l[0])     #Accessing the index
```

List Index

Each item in a list has its own unique index. This index can be used to access any particular item from the list.

```
l = [3, 5, 6]
print(l[0])     #Accessing the index
print(l[1])
print(l[2])
```

Accessing List Items

You can access list items by using the index with the square bracket syntax. You can use both positive and negative indexing.

```
colors = ["Red", "blue", "Green"]
print(colors[-1])
print(colors[-2])
print(colors[-3])
```

Checking Item in the List

You can check if a given item is present in the list using the “in” keyword.

```
colors = ["Red", "G", "B","y"]
if "y" in colors:
        print("Y is present")
else:
        print("Y is not present in the list")
```

Range of Index

You can print a range of list items by specifying where you want to start, where do you want to end and if you want to skips elements in between the range.

```
animals = ["cat", "dog", "bat", "mouse", "horse", "elephant"]
print(animals[1:6:3])
```

List Comprehension

List comprehensions are used for creating new lists from other iterables like lists, tuples, dictionaries, sets, and even in arrays and strings.

```
lst = [i*i for i in range(20) if i%2==0]
print(lst)
```

Updating Lists

There are several methods for updating list:

- `list.append(value)`: Appends a value to the end of the list.
- `list.extend(iterable)`: Appends a series of values to the list.
- `list.insert(index, value)`: Inserts a value at a specific index.

```
lst = [1, 2, 3]
lst.append(4)  # lst is now [1, 2, 3, 4]
lst.extend([5, 6])  # lst is now [1, 2, 3, 4, 5, 6]
lst.insert(0, 0)  # lst is now [0, 1, 2, 3, 4, 5, 6]
```

Click here to get more detail about list methods

Updating Function Parameters Based on Input

You can update function parameters based on input. This can be done by creating a global variable and updating it inside the function.

```
def test(b):
    global a
    a = a + b

a = 0
test(1)  # Now, a is 1
```

Click on this Link to get more detail about function and return statement

AI Day 09 Notes

Syed Mansoor ul Hassan Bukhari

Python Lists:

1. **Definition**: Lists in Python are used to store multiple items in a single variable. They are ordered, changeable, and allow duplicate values. Lists are written with square brackets.

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

2. **List Items**: List items are ordered, changeable, and allow duplicate values. They are indexed, with the first item having an index of 0.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])  # Output: banana
```

3. **Order**: Lists are ordered, meaning the items have a defined order that will not change. You can access items by referring to their index.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[0])  # Output: apple
```

4. **Changeability**: Lists are mutable, meaning you can change, add, or remove items after the list has been created.

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)  # Output: ['apple', 'blackcurrant', 'cherry']
```

5. **Duplication**: Lists can have items with the same value.

```
thislist = ["apple", "banana", "cherry", "apple"]
print(thislist)  # Output: ['apple', 'banana', 'cherry', 'apple']
```

6. **List Length**: The `len()` function is used to determine how many items a list has.

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))  # Output: 3
```

7. **Data Types**: The values in list items can be of any data type.

```
thislist = ["apple", 1, True, 3.14]
print(thislist)
```

8. **The `list()` Constructor**: It is also possible to use the `list()` constructor to make a list.

```
thislist = list(("apple", "banana", "cherry"))
print(thislist)
```

9. **Accessing Items**: Items of a list can be accessed by referring to its index number, inside square brackets.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])  # Output: banana
```

10. **Checking if Item Exists**: To determine if a specified item is present in a list, use the `in` keyword.

```
thislist = ["apple", "banana", "cherry"]
if "banana" in thislist:
  print("Yes, 'banana' is in the list")
```

11. **Updating List**: You can change the value of a specific item by referring to its index number.

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)  # Output: ['apple', 'blackcurrant', 'cherry']
```

12. **Adding Items**: Adding an item to the list is done using the `append()` method.

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)  # Output: ['apple', 'banana', 'cherry', 'orange']
```

13. **Removing Items**: The `remove()` method removes the specified item.

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)  # Output: ['apple', 'cherry']
```

14. **Looping Through a List**: You can loop through a list by using a `for` loop.

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

15. **Nested Lists**: A list can contain other lists, this is called nested lists.

```
mylist = [["apple", "banana"], ["cherry", "date"]]
print(mylist)
```

16. **Clearing List**: The `clear()` method empties the list.

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
```

```
print(thislist)  # Output: []
```

17. **Deleting List**: The `del` keyword removes the list completely.

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Python Sets:

1. **Definition**: Sets in Python are used to store multiple items in a single variable. They are unordered, unchangeable (but you can add or remove items), and do not allow duplicate values. Sets are written with curly brackets.

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

2. **Set Items**: Set items are unordered, unchangeable, and do not allow duplicate values. They are not indexed.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

3. **Order**: Sets are unordered, meaning the items do not have a defined order and you cannot refer to an item by using an index.

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

4. **Changeability**: Sets are unchangeable, meaning you cannot change the items after the set has been created, but you can add or remove items.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)  # Output: {'apple', 'banana', 'cherry', 'orange'}
```

5. **Duplication**: Sets cannot have two items with the same value.

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)  # Output: {'apple', 'banana', 'cherry'}
```

6. **Set Length**: The `len()` function is used to determine how many items a set has.

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))  # Output: 3
```

7. **Data Types**: The values in set items can be of any data type.

```
thisset = {"apple", 1, True, 3.14}
print(thisset)
```

8. **The `set()` Constructor**: It is also possible to use the `set()` constructor to make a set.

```
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

9. **Accessing Items**: You cannot access items in a set by referring to an index or a key, but you can loop through the set items using a `for` loop.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

10. **Checking if Item Exists**: To determine if a specified item is present in a set, use the `in` keyword.

```
thisset = {"apple", "banana", "cherry"}
if "banana" in thisset:
  print("Yes, 'banana' is in the set")
```

11. **Updating Set**: You cannot change the items in a set, but you can add new items.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)  # Output: {'apple', 'banana', 'cherry', 'orange'}
```

12. **Adding Items**: Adding an item to the set is done using the `add()` method.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)  # Output: {'apple', 'banana', 'cherry', 'orange'}
```

13. **Removing Items**: The `remove()` method removes the specified item.

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)  # Output: {'apple', 'cherry'}
```

14. **Looping Through a Set**: You can loop through a set by using a `for` loop.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

15. **Nested Sets**: Sets cannot contain other sets, but they can contain other iterable objects like tuples.

```
thisset = {"apple", "banana", ("cherry", "date")}
```

```
print(thisset)
```

16. **Clearing Set**: The `clear()` method empties the set.

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)  # Output: set()
```

17. **Deleting Set**: The `del` keyword removes the set completely.

```
thisset = {"apple", "banana", "cherry"}
del thisset
```

AI Day 10 Notes
Syed Mansoor ul Hassan Bukhari

Python Dictionaries:

1. **Definition**: Dictionaries in Python are used to store data values in key:value pairs. They are collections that are ordered (as of Python 3.7), changeable, and do not allow duplicates. They are written with curly brackets, and have keys and values.

```
thisdict = {
  "brand": "Suzuki",
  "model": "Skoda",
  "year": 2021
}
print(thisdict)
```

2. **Dictionary Items**: Dictionary items are ordered, changeable, and do not allow duplicates. They are presented in key:value pairs and can be referred to by using the key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

3. **Order**: As of Python version 3.7, dictionaries are ordered, meaning the items have a defined order that will not change. In Python 3.6 and earlier, dictionaries are unordered, meaning the items do not have a defined order and you cannot refer to an item by using an index.
4. **Changeability**: Dictionaries are mutable, meaning that we can change, add, or remove items after the dictionary has been created.
5. **Duplication**: Dictionaries cannot have two items with the same key. Duplicate values will overwrite existing values.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

6. **Dictionary Length**: The `len()` function is used to determine how many items a dictionary has.

```
print(len(thisdict))
```

7. **Data Types**: The values in dictionary items can be of any data type.

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
```

```
  "colors": ["red", "white", "blue"]
}
print(thisdict)
```

8. **The dict() Constructor**: It is also possible to use the `dict()` constructor to make a dictionary.

```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

9. **Accessing Items**: Items of a dictionary can be accessed by referring to its key name, inside square brackets, or by using the `get()` method.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
x = thisdict.get("model")
```

10. **Checking if Key Exists**: To determine if a specified key is present in a dictionary, use the in keyword.

```
thisdict = {

 "brand": "Ford",

 "model": "Mustang",

 "year": 1964

}

if "model" in thisdict:

 print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

11. **Updating Dictionary**: You can change the value of a specific item by referring to its key name.

```
thisdict = {

 "brand": "Ford",

 "model": "Mustang",

 "year": 1964

}

thisdict["year"] = 2020
```

12. **Adding Items**: Adding an item to the dictionary is done by using a new index key and assigning a value to it.

```
thisdict = {

  "brand": "Ford",

  "model": "Mustang",

  "year": 1964

}

thisdict["color"] = "red"
```

13. **Removing Items**: The pop() method removes the item with the specified key name.

```
thisdict = {

  "brand": "Ford",

  "model": "Mustang",

  "year": 1964

}

thisdict.pop("model")
```

14. **Looping Through a Dictionary**: You can loop through a dictionary by using a for loop. The for loop returns the keys of the dictionary, but there are methods to return the values as well.

```
for x in thisdict:

  print(x)  # print keys

  print(thisdict[x])  # print values
```

15. **Nested Dictionaries**: A dictionary can contain dictionaries, this is called nested dictionaries.

```
myfamily = {

  "child1" : {

    "name" : "Emil",

    "year" : 2004
```

```
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

16. **Clearing Dictionary**: The clear() keyword empties the dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
```

17. **Deleting Dictionary**: The del keyword removes the dictionary completely.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
```

AI Day 11 Notes

Syed Mansoor ul Hassan Bukhari

Python Classes and Objects:

1. **Definition**: Classes in Python are blueprints for creating objects. An object is an instance of a class. Classes encapsulate data and functions that operate on the data.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

my_car = Car("Suzuki", "Skoda", 2021)
print(my_car.brand)
```

2. **Class Attributes**: Class attributes are variables that belong to the class itself and are shared among all instances of the class.

```
class Car:
    wheels = 4  # Class attribute

    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

print(Car.wheels)
```

3. **Instance Attributes**: Instance attributes are variables that belong to an instance of the class. Each instance can have different values for these attributes.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

my_car = Car("Ford", "Mustang", 1964)
print(my_car.brand)
```

4. **Methods**: Methods are functions defined inside a class that describe the behaviors of the objects created from the class.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
```

```
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Brand: {self.brand}, Model: {self.model}, Year: {self.year}")

my_car = Car("Ford", "Mustang", 1964)
my_car.display_info()
```

5. **The `__init__` Method**: The `__init__` method is a special method that is called when an object is instantiated. It initializes the object's attributes.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

my_car = Car("Ford", "Mustang", 1964)
print(my_car.year)
```

6. **Creating Objects**: Objects are instances of a class. You can create multiple objects from the same class.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

car1 = Car("Ford", "Mustang", 1964)
car2 = Car("Toyota", "Corolla", 2020)
print(car1.model)
print(car2.model)
```

7. **Modifying Object Attributes**: You can change the value of an object's attributes after it has been created.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

my_car = Car("Ford", "Mustang", 1964)
my_car.year = 2020
print(my_car.year)
```

8. **Deleting Object Attributes**: You can delete an object's attributes using the `del` keyword.

```
class Car:
    def __init__(self, brand, model, year):
```

```
        self.brand = brand
        self.model = model
        self.year = year

my_car = Car("Ford", "Mustang", 1964)
del my_car.year
```

9. **Class Methods**: Class methods are methods that are bound to the class and not the instance of the class. They can modify class state that applies across all instances of the class.

```
class Car:
    wheels = 4

    @classmethod
    def change_wheels(cls, num_wheels):
        cls.wheels = num_wheels

Car.change_wheels(6)
print(Car.wheels)
```

10. **Static Methods**: Static methods are methods that do not modify class or instance state. They are defined using the `@staticmethod` decorator.

```
class Car:
    @staticmethod
    def honk():
        print("Beep beep!")

Car.honk()
```

11. **Inheritance**: Inheritance allows a class to inherit attributes and methods from another class. The class that inherits is called the child class, and the class being inherited from is called the parent class.

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand, model)
        self.year = year

my_car = Car("Ford", "Mustang", 1964)
print(my_car.brand)
```

12. **Polymorphism**: Polymorphism allows methods to do different things based on the object it is acting upon.

```
class Vehicle:
    def start(self):
```

```
        print("Starting vehicle...")

class Car(Vehicle):
    def start(self):
        print("Starting car...")

my_vehicle = Vehicle()
my_car = Car()
my_vehicle.start()
my_car.start()
```

13. **Encapsulation**: Encapsulation is the concept of wrapping data and methods that work on the data within one unit. This prevents data from being modified directly.

```
class Car:
    def __init__(self, brand, model, year):
        self.__brand = brand  # Private attribute
        self.model = model
        self.year = year

    def get_brand(self):
        return self.__brand

my_car = Car("Ford", "Mustang", 1964)
print(my_car.get_brand())
```

14. **Abstraction**: Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

class Car(Vehicle):
    def start(self):
        print("Starting car...")

my_car = Car()
my_car.start()
```

15. **Multiple Inheritance**: Multiple inheritance allows a class to inherit from more than one class.

```
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

class Wheels:
    def __init__(self, size):
        self.size = size
```

```
class Car(Engine, Wheels):
    def __init__(self, brand, model, year, horsepower, size):
        Engine.__init__(self, horsepower)
        Wheels.__init__(self, size)
        self.brand = brand
        self.model = model
        self.year = year

my_car = Car("Ford", "Mustang", 1964, 300, 18)
print(my_car.horsepower)
print(my_car.size)
```

16. **Method Overriding**: Method overriding allows a child class to provide a specific implementation of a method that is already defined in its parent class.

```
class Vehicle:
    def start(self):
        print("Starting vehicle...")

class Car(Vehicle):
    def start(self):
        print("Starting car...")

my_car = Car()
my_car.start()
```

17. **Operator Overloading**: Operator overloading allows you to define how operators behave for user-defined types.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2
print(v3.x, v3.y)
```

AI Day 12 Notes

Syed Mansoor ul Hassan Bukhari

Python File Handling:

1. **Definition**: File handling in Python is used to read and write files. It allows us to create, read, update, and delete files.

```
# Open a file
file = open("demo.txt", "w")
file.write("Hello, world!")
file.close()
```

2. **Opening a File**: The `open()` function is used to open a file. The mode in which the file is opened is specified as a second argument.

```
file = open("demo.txt", "r")  # Read mode
file = open("demo.txt", "w")  # Write mode
file = open("demo.txt", "a")  # Append mode
file = open("demo.txt", "b")  # Binary mode
```

3. **Reading a File**: The `read()` method is used to read the content of a file.

```
file = open("demo.txt", "r")
content = file.read()
print(content)
file.close()
```

4. **Writing to a File**: The `write()` method is used to write content to a file.

```
file = open("demo.txt", "w")
file.write("Hello, world!")
file.close()
```

5. **Appending to a File**: The `append()` method is used to add content to the end of a file.

```
file = open("demo.txt", "a")
file.write("Appending text.")
file.close()
```

6. **File Modes**: Different modes can be used to open a file:

   - `"r"`: Read mode (default)
   - `"w"`: Write mode
   - `"a"`: Append mode
   - `"b"`: Binary mode
   - `"x"`: Create mode (creates a new file, returns an error if the file exists)

```
file = open("demo.txt", "x")
```

7. **Reading Lines**: The `readline()` method reads a single line from the file.

```
file = open("demo.txt", "r")
line = file.readline()
print(line)
file.close()
```

8. **Reading All Lines**: The `readlines()` method reads all the lines from the file and returns them as a list.

```
file = open("demo.txt", "r")
lines = file.readlines()
for line in lines:
    print(line)
file.close()
```

9. **Using `with` Statement**: The `with` statement is used to wrap the execution of a block of code. It ensures that the file is properly closed after its suite finishes.

```
with open("demo.txt", "r") as file:
    content = file.read()
    print(content)
```

10. **Checking if File Exists**: The `os` module can be used to check if a file exists.

```
import os
if os.path.exists("demo.txt"):
    print("File exists")
else:
    print("File does not exist")
```

11. **Deleting a File**: The `os` module can also be used to delete a file.

```
import os
if os.path.exists("demo.txt"):
    os.remove("demo.txt")
else:
    print("File does not exist")
```

12. **Creating a Directory**: The `os` module can be used to create a directory.

```
import os
os.mkdir("new_directory")
```

13. **Removing a Directory**: The `os` module can also be used to remove a directory.

```
import os
os.rmdir("new_directory")
```

14. **File Position**: The `tell()` method returns the current file position.

```
file = open("demo.txt", "r")
print(file.tell())
file.close()
```

15. **Changing File Position**: The `seek()` method changes the file position.

```
file = open("demo.txt", "r")
file.seek(0)  # Move to the beginning of the file
print(file.read())
file.close()
```

16. **Truncating a File**: The `truncate()` method is used to resize the file to a specified size.

```
file = open("demo.txt", "w")
file.write("Hello, world!")
file.truncate(5)
file.close()
```

17. **File Attributes**: The `os` module can be used to get file attributes.

```
import os
file_info = os.stat("demo.txt")
print(file_info.st_size)  # Size of the file
print(file_info.st_mtime)  # Last modification time
```

AI Day 13 Notes

Syed Mansoor ul Hassan Bukhari

Python Exception Handling:

1. **Definition**: Exception handling in Python is used to manage errors that occur during the execution of a program. It allows the program to continue running or gracefully terminate.

```
try:
    # Code that may raise an exception
    x = 10 / 0
except ZeroDivisionError:
    # Code to handle the exception
    print("Cannot divide by zero!")
```

2. **Try and Except**: The `try` block lets you test a block of code for errors. The `except` block lets you handle the error.

```
try:
    x = int("hello")
except ValueError:
    print("ValueError: Invalid literal for int()")
```

3. **Multiple Exceptions**: You can handle multiple exceptions by specifying multiple `except` blocks.

```
try:
    x = int("hello")
except ValueError:
    print("ValueError: Invalid literal for int()")
except TypeError:
    print("TypeError: Unsupported operation")
```

4. **Else Clause**: The `else` block lets you execute code if no exceptions were raised.

```
try:
    x = int("10")
except ValueError:
    print("ValueError: Invalid literal for int()")
else:
    print("No exceptions occurred")
```

5. **Finally Clause**: The `finally` block lets you execute code, regardless of whether an exception was raised or not.

```
try:
```

```
    x = int("10")
except ValueError:
    print("ValueError: Invalid literal for int()")
finally:
    print("This will always execute")
```

6. **Raising Exceptions**: You can raise an exception using the `raise` keyword.

```
x = -1
if x < 0:
    raise ValueError("Negative values are not allowed")
```

7. **Custom Exceptions**: You can define your own exceptions by creating a new class that inherits from the built-in `Exception` class.

```
class MyCustomError(Exception):
    pass

try:
    raise MyCustomError("This is a custom error")
except MyCustomError as e:
    print(e)
```

8. **Exception Hierarchy**: Python has a built-in hierarchy of exceptions. The `BaseException` class is the base class for all built-in exceptions.

```
try:
    x = 10 / 0
except BaseException as e:
    print(f"An error occurred: {e}")
```

9. **Handling Multiple Exceptions in One Block**: You can handle multiple exceptions in a single `except` block by specifying a tuple of exception types.

```
try:
    x = int("hello")
except (ValueError, TypeError) as e:
    print(f"An error occurred: {e}")
```

10. **Assertions**: The `assert` statement is used to test if a condition is true. If the condition is false, an `AssertionError` is raised.

```
x = 10
assert x > 0, "x should be greater than 0"
```

11. **Logging Exceptions**: The `logging` module can be used to log exceptions.

```
import logging

try:
    x = 10 / 0
```

```
except ZeroDivisionError as e:
    logging.error("An error occurred", exc_info=True)
```

12. **Context Managers**: The `with` statement can be used to handle exceptions in a context manager.

```
class MyContextManager:
    def __enter__(self):
        print("Entering context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")
        if exc_type:
            print(f"An error occurred: {exc_value}")
        return True

with MyContextManager():
    x = 10 / 0
```

13. **Ignoring Exceptions**: You can ignore exceptions by using a bare `except` block, but this is not recommended.

```
try:
    x = 10 / 0
except:
    pass
```

14. **Re-raising Exceptions**: You can re-raise an exception using the `raise` keyword without specifying the exception.

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Handling ZeroDivisionError")
    raise
```

15. **Exception Chaining**: You can chain exceptions using the `from` keyword.

```
try:
    x = int("hello")
except ValueError as e:
    raise RuntimeError("A runtime error occurred") from e
```

16. **SystemExit Exception**: The `SystemExit` exception is raised when the `sys.exit()` function is called.

```
import sys
try:
    sys.exit()
except SystemExit:
    print("SystemExit exception caught")
```

17. **KeyboardInterrupt Exception**: The `KeyboardInterrupt` exception is raised when the user interrupts the program (usually by pressing Ctrl+C).

```
try:
    while True:
        pass
except KeyboardInterrupt:
    print("KeyboardInterrupt exception caught")
```

AI Day 14 Notes

Syed Mansoor ul Hassan Bukhari

Inheritance in Python:

1. **Definition**: Inheritance allows a class to inherit attributes and methods from another class. The class that inherits is called the child class, and the class being inherited from is called the parent class.

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

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

child = Child("Saad", 12)
print(child.name, child.age)
```

2. **Parent Class**: The parent class is the class being inherited from, also called the base class.

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

parent = Parent("Saad")
print(parent.name)
```

3. **Child Class**: The child class is the class that inherits from another class, also called the derived class.

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

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

child = Child("Saad", 12)
print(child.name, child.age)
```

4. **super() Function**: The `super()` function allows us to call methods of the parent class in the child class.

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

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

child = Child("Saad", 12)
print(child.name, child.age)
```

5. **Method Overriding**: Child classes can override methods from the parent class.

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

class Child(Parent):
    def show(self):
        print("Child method")

child = Child()
child.show()
```

6. **Multiple Inheritance**: A class can inherit from multiple classes.

```
class Parent1:
    def __init__(self, name):
        self.name = name

class Parent2:
    def __init__(self, age):
        self.age = age

class Child(Parent1, Parent2):
    def __init__(self, name, age):
        Parent1.__init__(self, name)
        Parent2.__init__(self, age)

child = Child("Saad", 12)
print(child.name, child.age)
```

7. **Multilevel Inheritance**: A class can inherit from another class, which in turn inherits from another class.

```
class Grandparent:
    def __init__(self, name):
        self.name = name

class Parent(Grandparent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age
```

```
class Child(Parent):
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

child = Child("Saad", 12, "7th")
print(child.name, child.age, child.grade)
```

8. **Hierarchical Inheritance**: Multiple classes inherit from the same parent class.

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

class Child1(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

class Child2(Parent):
    def __init__(self, name, grade):
        super().__init__(name)
        self.grade = grade

child1 = Child1("Saad", 12)
child2 = Child2("Asad", "7th")
print(child1.name, child1.age)
print(child2.name, child2.grade)
```

9. **Hybrid Inheritance**: A combination of two or more types of inheritance.

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

class Child1(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

class Child2(Parent):
    def __init__(self, name, grade):
        super().__init__(name)
        self.grade = grade

class Grandchild(Child1, Child2):
    def __init__(self, name, age, grade):
        Child1.__init__(self, name, age)
        Child2.__init__(self, name, grade)

grandchild = Grandchild("Saad", 12, "7th")
print(grandchild.name, grandchild.age, grandchild.grade)
```

10. **Accessing Parent Class Attributes**: Attributes of the parent class can be accessed using the `super()` function.

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

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

child = Child("Saad", 12)
print(child.name, child.age)
```

11. **Accessing Parent Class Methods**: Methods of the parent class can be accessed using the `super()` function.

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

class Child(Parent):
    def show(self):
        super().show()
        print("Child method")

child = Child()
child.show()
```

12. **Using `isinstance()`**: The `isinstance()` function checks if an object is an instance of a class or a subclass thereof.

```
class Parent:
    pass

class Child(Parent):
    pass

child = Child()
print(isinstance(child, Child))  # True
print(isinstance(child, Parent))  # True
```

13. **Using `issubclass()`**: The `issubclass()` function checks if a class is a subclass of another class.

```
class Parent:
    pass

class Child(Parent):
    pass

print(issubclass(Child, Parent))  # True
```

```
print(issubclass(Parent, Child))  # False
```

14. **Private Members**: Private members of the parent class are not accessible directly in the child class.

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

class Child(Parent):
    def __init__(name, age):
        super().__init__(name)
        self.age = age

child = Child("Saad", 12)
# print(child.__name)  # AttributeError
```

15. **Protected Members**: Protected members of the parent class can be accessed in the child class.

```
class Parent:
    def __init__(name):
        self._name = name

class Child(Parent):
    def __init__(name, age):
        super().__init__(name)
        self.age = age

child = Child("Saad", 12)
print(child._name)
```

16. **Method Resolution Order (MRO)**: The order in which methods are resolved in the presence of multiple inheritance.

```
class A:
    def show(self):
        print("A method")

class B(A):
    def show(self):
        print("B method")

class C(A):
    def show(self):
        print("C method")

class D(B, C):
    pass

d = D()
d.show()  # B method
print(D.mro())  # [D, B, C, A, object]
```

17. **Diamond Problem**: The diamond problem occurs when a class inherits from two classes that have a common base class.

```
class A:
    def show(self):
        print("A method")

class B(A):
    def show(self):
        print("B method")

class C(A):
    def show(self):
        print("C method")

class D(B, C):
    pass

d = D()
d.show()  # B method
print(D.mro())  # [D, B, C, A, object]
```

AI Day 15 Notes

Syed Mansoor ul Hassan Bukhari

Abstraction and Encapsulation in Python

Abstraction:

1. **Definition**: Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object.
2. **Purpose**: It helps in reducing programming complexity and effort by providing a simplified model of the real-world scenario.
3. **Implementation**: In Python, abstraction can be achieved using abstract classes and methods provided by the `abc` module.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

dog = Dog()
cat = Cat()
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

Encapsulation:

1. **Definition**: Encapsulation is the concept of wrapping data (variables) and methods (functions) that work on the data into a single unit, called a class.
2. **Purpose**: It restricts direct access to some of an object's components, which can prevent the accidental modification of data.
3. **Implementation**: In Python, encapsulation can be achieved using private and protected access modifiers.

```
class Car:
    def __init__(self, make, model):
        self._make = make  # Protected attribute
        self.__model = model  # Private attribute

    def get_model(self):
```

```
        return self.__model

    def set_model(self, model):
        self.__model = model

car = Car("Toyota", "Corolla")
print(car._make)  # Output: Toyota
print(car.get_model())  # Output: Corolla
car.set_model("Camry")
print(car.get_model())  # Output: Camry
```

Key Differences:

- **Abstraction** focuses on hiding the complexity of the system by providing a simplified interface.
- **Encapsulation** focuses on restricting access to the internal state and behavior of an object.

Real-World Example:

- **Abstraction**: When you use a TV remote, you only interact with the buttons (interface) without knowing the internal circuitry (implementation).
- **Encapsulation**: In a company, the finance department's data is hidden from the sales department. Only authorized personnel can access and modify the financial data.

AI Course
Syed Mansoor ul Hassan Bukhari

Python Match Case Statements:

1. **Definition**: The match case statement in Python, introduced in Python 3.10, is similar to the switch case statement in other languages like C/C++ or Java. It allows for more readable and efficient code by matching an expression against a series of patterns.
2. **Syntax**: The match case statement is initialized with the `match` keyword followed by the parameter to be matched. Various cases are defined using the `case` keyword and the pattern to match the parameter. The _ is the wildcard character that runs when all the cases fail to match the parameter value.

```
match parameter:
    case pattern1:
        # code for pattern 1
    case pattern2:
        # code for pattern 2
    ...
    case patternN:
        # code for pattern N
    case _:
        # default code block
```

3. **Simple Match Case Statement**: In a simple match case statement, the exact value is compared and matched with the case pattern value.

```
def runMatch():
    num = int(input("Enter a number between 1 and 3: "))
    match num:
        case 1:
            print("One")
        case 2:
            print("Two")
        case 3:
            print("Three")
        case _:
            print("Number not between 1 and 3")

runMatch()
```

4. **Match Case Statement with OR Operator**: The OR operator (`|`) can be used to match multiple patterns that result in the same output.

```
def runMatch():
    num = int(input("Enter a number between 1 and 6: "))
    match num:
        case 1 | 2:
            print("One or Two")
        case 3 | 4:
            print("Three or Four")
        case 5 | 6:
            print("Five or Six")
```

```
        case _:
            print("Number not between 1 and 6")

runMatch()
```

5. **Match Case with Sequence Pattern**: Match case statements can also be used with sequence patterns like lists or tuples.

```
def runMatch():
    sequence = [1, 2, 3]
    match sequence:
        case [1, 2, 3]:
            print("Matched [1, 2, 3]")
        case [4, 5, 6]:
            print("Matched [4, 5, 6]")
        case _:
            print("No match found")

runMatch()
```

6. **Match Case with Dictionary**: Match case statements can be used to match dictionary patterns.

```
def runMatch():
    data = {"name": "Asad", "age": 36}
    match data:
        case {"name": "Asad", "age": 36}:
            print("Matched Asad, 36")
        case {"name": "Umar", "age": 28}:
            print("Matched Umar, 28")
        case _:
            print("No match found")

runMatch()
```

7. **Match Case with Class**: Match case statements can be used to match class instances.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def runMatch():
    person = Person("Asad", 36)
    match person:
        case Person(name="Asad", age=36):
            print("Matched Asad, 36")
        case Person(name="Umar", age=28):
            print("Matched Umar, 28")
        case _:
            print("No match found")

runMatch()
```

OS Module in Python

OS Module in Python

The `os` module in Python provides a way of using operating system-dependent functionality like file and directory management, process management, and environment variables. Below are some key features and methods of the `os` module:

File and Directory Management

- **Creating Directories**: `os.mkdir(path)` creates a directory at the specified path.
- **Removing Directories**: `os.rmdir(path)` removes an empty directory.
- **Listing Directories**: `os.listdir(path)` returns a list of entries in the directory.
- **Changing Directory**: `os.chdir(path)` changes the current working directory to the specified path.
- **Getting Current Directory**: `os.getcwd()` returns the current working directory.
- **Removing Files**: `os.remove(path)` removes the specified file.
- **Renaming Files/Directories**: `os.rename(src, dst)` renames the file or directory from `src` to `dst`.

File Descriptors

- **Opening a File**: `os.open(path, flags)` opens the file specified by `path` and returns a file descriptor.
- **Closing a File**: `os.close(fd)` closes the file descriptor `fd`.
- **Reading from a File**: `os.read(fd, n)` reads `n` bytes from the file descriptor `fd`.
- **Writing to a File**: `os.write(fd, str)` writes the string `str` to the file descriptor `fd`.

Process Management

- **Forking a Process**: `os.fork()` forks a child process.
- **Executing a Program**: `os.execv(path, args)` executes the program specified by `path` with the arguments `args`.
- **Getting Process ID**: `os.getpid()` returns the current process ID.
- **Getting Parent Process ID**: `os.getppid()` returns the parent process ID.
- **Terminating a Process**: `os.kill(pid, sig)` sends the signal `sig` to the process with ID `pid`.

Environment Variables

- **Getting Environment Variables**: `os.getenv(key)` returns the value of the environment variable `key`.
- **Setting Environment Variables**: `os.putenv(key, value)` sets the environment variable `key` to `value`.

- **Deleting Environment Variables**: `os.unsetenv(key)` deletes the environment variable `key`.

Path Manipulation

- **Joining Paths**: `os.path.join(path, *paths)` joins one or more path components.
- **Splitting Paths**: `os.path.split(path)` splits the path into a pair `(head, tail)`.
- **Checking Path Existence**: `os.path.exists(path)` returns `True` if the path exists.
- **Checking File Type**: `os.path.isfile(path)` returns `True` if the path is a file.
- **Checking Directory Type**: `os.path.isdir(path)` returns `True` if the path is a directory.

Example Programs

```
# Example of creating and removing a directory
import os

os.mkdir("test_dir")
print("Directory created")
os.rmdir("test_dir")
print("Directory removed")

# Example of changing and getting the current directory
import os

os.chdir("C:/user/SalfiHacker/Desktop/AI")
print("Current Directory:", os.getcwd())

# Example of reading and writing to a file
import os

fd = os.open("test_file.txt", os.O_RDWR | os.O_CREAT)
os.write(fd, b"Hello, World!")
os.lseek(fd, 0, os.SEEK_SET)
print(os.read(fd, 12))  # Output: b'Hello, World!'
os.close(fd)
```

Collection Module

Collections Module in Python:

1. **Definition:** The `collections` module in Python provides specialized container datatypes that offer alternatives to Python's general-purpose built-in containers like `dict`, `list`, `set`, and `tuple`.
2. **namedtuple():** A factory function for creating tuple subclasses with named fields.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(11, 22)
print(p.x, p.y)  # Output: 11 22
```

3. **deque:** A list-like container with fast appends and pops on either end.

```
from collections import deque

d = deque(['a', 'b', 'c'])
d.append('d')
d.appendleft('z')
print(d)  # Output: deque(['z', 'a', 'b', 'c', 'd'])
```

4. **ChainMap:** A class for creating a single view of multiple mappings.

```
from collections import ChainMap

dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
chain = ChainMap(dict1, dict2)
print(chain)  # Output: ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4})
```

5. **Counter:** A dict subclass for counting hashable objects.

```
from collections import Counter

c = Counter('gallahad')
print(c)  # Output: Counter({'a': 3, 'l': 2, 'g': 1, 'h': 1, 'd': 1})
```

6. **OrderedDict:** A dict subclass that remembers the order entries were added.

```
from collections import OrderedDict

od = OrderedDict()
od['one'] = 1
od['two'] = 2
od['three'] = 3
print(od)  # Output: OrderedDict([('one', 1), ('two', 2), ('three', 3)])
```

7. **defaultdict:** A dict subclass that calls a factory function to supply missing values.

```
from collections import defaultdict

dd = defaultdict(int)
dd['key1'] += 1
print(dd)  # Output: defaultdict(<class 'int'>, {'key1': 1})
```

8. **UserDict, UserList, UserString:** Wrapper classes around dictionary, list, and string objects for easier subclassing.

```
from collections import UserDict, UserList, UserString

class MyDict(UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key, value*2)

d = MyDict()
d['a'] = 3
print(d)  # Output: {'a': 6}
```

Click here to get more info on collection module

100 Exam Preparation Questions for Python

Syed Mansoor ul Hassan Bukhari

Advanced Python Exam Preparation Questions

i. What is the difference between list and tuples in Python?
ii. What are the key features of Python?
iii. What type of language is Python?
iv. How is Python an interpreted language?
v. What is PEP 8?
vi. How is memory managed in Python?
vii. What is a namespace in Python?
viii. What is PYTHONPATH?
ix. What are Python modules?
x. What are local variables and global variables in Python?
xi. What are Python iterators?
xii. What is `self` in Python?
xiii. What is a lambda function in Python?
xiv. What are decorators in Python?
xv. What is the difference between `deepcopy` and `copy`?
xvi. How do you manage packages in Python?
xvii. What is the use of the `with` statement in Python?
xviii. What is the difference between `__init__` and `__new__`?
xix. What is the purpose of the `pass` statement in Python?
xx. How do you handle exceptions in Python?
xxi. What is the difference between `range` and `xrange`?
xxii. What are Python's built-in data types?
xxiii. How do you create a virtual environment in Python?
xxiv. What is the difference between `==` and `is`?
xxv. What is the Global Interpreter Lock (GIL)?
xxvi. How do you perform static type checking in Python?
xxvii. What is the difference between `@staticmethod` and `@classmethod`?
xxviii. What is the purpose of the `yield` keyword?
xxix. How do you use the `map()` function in Python?
xxx. What is the difference between `filter()` and `reduce()`?
xxxi. How do you use list comprehensions in Python?
xxxii. What is the purpose of the `zip()` function?
xxxiii. How do you handle file operations in Python?
xxxiv. What is the difference between `os` and `sys` modules?
xxxv. How do you use the `argparse` module?
xxxvi. What is the purpose of the `collections` module?
xxxvii. How do you use the `itertools` module?
xxxviii. What is the difference between `__str__` and `__repr__`?
xxxix. How do you use the `functools` module?

xl. What is the purpose of the `dataclasses` module?
xli. How do you use the `enum` module?
xlii. What is the purpose of the `abc` module?
xliii. How do you use the `contextlib` module?
xliv. What is the difference between `@property` and `@staticmethod`?
xlv. How do you use the `logging` module?
xlvi. What is the purpose of the `unittest` module?
xlvii. How do you use the `pytest` framework?
xlviii. What is the difference between `mock` and `patch`?
xlix. How do you use the `subprocess` module?
l. What is the purpose of the `multiprocessing` module?
li. How do you use the `threading` module?
lii. What is the difference between `threading` and `multiprocessing`?
liii. How do you use the `asyncio` module?
liv. What is the purpose of the `aiohttp` library?
lv. How do you use the `requests` library?
lvi. What is the difference between `urllib` and `requests`?
lvii. How do you use the `BeautifulSoup` library?
lviii. What is the purpose of the `Scrapy` framework?
lix. How do you use the `pandas` library?
lx. What is the difference between `pandas` and `numpy`?
lxi. How do you use the `matplotlib` library?
lxii. What is the purpose of the `seaborn` library?
lxiii. How do you use the `scikit-learn` library?
lxiv. What is the difference between `TensorFlow` and `PyTorch`?
lxv. How do you use the `Keras` library?
lxvi. What is the purpose of the `NLTK` library?
lxvii. How do you use the `spaCy` library?
lxviii. What is the difference between `Flask` and `Django`?
lxix. How do you use the `FastAPI` framework?
lxx. What is the purpose of the `SQLAlchemy` library?
lxxi. How do you use the `Alembic` library?
lxxii. What is the difference between `SQLite` and `PostgreSQL`?
lxxiii. How do you use the `MongoDB` library?
lxxiv. What is the purpose of the `Redis` library?
lxxv. How do you use the `Celery` library?
lxxvi. What is the difference between `RabbitMQ` and `Kafka`?
lxxvii. How do you use the `Docker` library?
lxxviii. What is the purpose of the `Kubernetes` library?
lxxix. How do you use the `Ansible` library?
lxxx. What is the difference between `Terraform` and `CloudFormation`?
lxxxi. How do you use the `Jupyter` notebook?
lxxxii. What is the purpose of the `Google Colab`?
lxxxiii. How do you use the `Streamlit` library?
lxxxiv. What is the difference between `Dash` and `Plotly`?

lxxxv. How do you use the `Bokeh` library?
lxxxvi. What is the purpose of the `Altair` library?
lxxxvii. How do you use the `Pytest` framework?
lxxxviii. What is the difference between `UnitTest` and `Pytest`?
lxxxix. How do you use the `Mock` library?
xc. What is the purpose of the `Hypothesis` library?
xci. How do you use the `Selenium` library?
xcii. What is the difference between `Selenium` and `BeautifulSoup`?
xciii. How do you use the `Appium` library?
xciv. What is the purpose of the `Robot Framework`?
xcv. How do you use the `Tox` library?
xcvi. What is the difference between `Tox` and `Nox`?
xcvii. How do you use the `Black` library?
xcviii. What is the purpose of the `Flake8` library?
xcix. How do you use the `Mypy` library?
c. What is the difference between `Pylint` and `Pyflakes`?

These questions cover a wide range of advanced Python topics and will help you prepare thoroughly for your exam. Good luck with your studies!

85 Exam Preparation Problems for Python

Syed Mansoor ul Hassan Bukhari

Advanced Python Exam Preparation Coding Problems

I. Write a Python program to find the factorial of a number using recursion.
II. Implement a Python function to check if a given string is a palindrome.
III. Write a Python program to find the GCD (Greatest Common Divisor) of two numbers.
IV. Implement a Python function to perform binary search on a sorted list.
V. Write a Python program to find the nth Fibonacci number using dynamic programming.
VI. Implement a Python function to sort a list using the merge sort algorithm.
VII. Write a Python program to solve the Tower of Hanoi problem.
VIII. Implement a Python function to find the longest common subsequence of two strings.
IX. Write a Python program to find the shortest path in a graph using Dijkstra's algorithm.
X. Implement a Python function to check if a given binary tree is a valid binary search tree.
XI. Write a Python program to find the maximum subarray sum using Kadane's algorithm.
XII. Implement a Python function to perform matrix multiplication.
XIII. Write a Python program to find the number of islands in a given grid.
XIV. Implement a Python function to find the minimum spanning tree of a graph using Kruskal's algorithm.
XV. Write a Python program to solve the N-Queens problem.
XVI. Implement a Python function to find the longest increasing subsequence in a list.
XVII. Write a Python program to find the median of two sorted arrays.
XVIII. Implement a Python function to perform depth-first search (DFS) on a graph.
XIX. Write a Python program to find the number of ways to climb a staircase with n steps.
XX. Implement a Python function to find the maximum product subarray.
XXI. Write a Python program to find the minimum number of coins needed to make a given amount.
XXII. Implement a Python function to find the longest palindromic substring in a given string.
XXIII. Write a Python program to find the maximum profit from buying and selling stocks.
XXIV. Implement a Python function to perform breadth-first search (BFS) on a graph.
XXV. Write a Python program to find the number of distinct subsequences of a given string.
XXVI. Implement a Python function to find the minimum path sum in a grid.
XXVII. Write a Python program to find the number of ways to partition a set into k subsets.
XXVIII. Implement a Python function to find the maximum sum of non-adjacent elements in a list.
XXIX. Write a Python program to find the number of ways to decode a given string of digits.
XXX. Implement a Python function to find the longest substring without repeating characters.
XXXI. Write a Python program to find the number of ways to make change for a given amount.
XXXII. Implement a Python function to find the maximum length of a subarray with sum equal to k.
XXXIII. Write a Python program to find the number of ways to arrange n objects in a circle.
XXXIV. Implement a Python function to find the maximum sum of a subarray with at most k elements.
XXXV. Write a Python program to find the number of ways to paint a fence with n posts.

XXXVI. Implement a Python function to find the maximum sum of a subarray with at least k elements.
XXXVII. Write a Python program to find the number of ways to distribute n candies to k children.
XXXVIII. Implement a Python function to find the maximum sum of a subarray with exactly k elements.
XXXIX. Write a Python program to find the number of ways to place n queens on an n x n chessboard.
XL. Implement a Python function to find the maximum sum of a subarray with at most k distinct elements.
XLI. Write a Python program to find the number of ways to partition a string into palindromes.
XLII. Implement a Python function to find the maximum sum of a subarray with at least k distinct elements.
XLIII. Write a Python program to find the number of ways to arrange n objects in a line.
XLIV. Implement a Python function to find the maximum sum of a subarray with exactly k distinct elements.
XLV. Write a Python program to find the number of ways to partition a set into k non-empty subsets.
XLVI. Implement a Python function to find the maximum sum of a subarray with at most k non-adjacent elements.
XLVII. Write a Python program to find the number of ways to partition a string into k palindromes.
XLVIII. Implement a Python function to find the maximum sum of a subarray with at least k non-adjacent elements.
XLIX. Write a Python program to find the number of ways to arrange n objects in a circle with k distinct elements.
L. Implement a Python function to find the maximum sum of a subarray with exactly k non-adjacent elements.
LI. Write a Python program to find the number of ways to partition a set into k subsets with at least one element.
LII. Implement a Python function to find the maximum sum of a subarray with at most k adjacent elements.
LIII. Write a Python program to find the number of ways to partition a string into k distinct palindromes.
LIV. Implement a Python function to find the maximum sum of a subarray with at least k adjacent elements.
LV. Write a Python program to find the number of ways to arrange n objects in a line with k distinct elements.
LVI. Implement a Python function to find the maximum sum of a subarray with exactly k adjacent elements.
LVII. Write a Python program to find the number of ways to partition a set into k non-empty subsets with at least one element.
LVIII. Implement a Python function to find the maximum sum of a subarray with at most k non-adjacent elements.
LIX. Write a Python program to find the number of ways to partition a string into k distinct palindromes with at least one element.

LX. Implement a Python function to find the maximum sum of a subarray with at least k non-adjacent elements.

LXI. Write a Python program to find the number of ways to arrange n objects in a circle with k distinct elements and at least one element.

LXII. Implement a Python function to find the maximum sum of a subarray with exactly k non-adjacent elements.

LXIII. Write a Python program to find the number of ways to partition a set into k subsets with at least one element and at least one distinct element.

LXIV. Implement a Python function to find the maximum sum of a subarray with at most k adjacent elements.

LXV. Write a Python program to find the number of ways to partition a string into k distinct palindromes with at least one element and at least one distinct element.

LXVI. Implement a Python function to find the maximum sum of a subarray with at least k adjacent elements.

LXVII. Write a Python program to find the number of ways to arrange n objects in a line with k distinct elements and at least one element.

LXVIII. Implement a Python function to find the maximum sum of a subarray with exactly k adjacent elements.

LXIX. Write a Python program to find the number of ways to partition a set into k non-empty subsets with at least one element and at least one distinct element.

LXX. Implement a Python function to find the maximum sum of a subarray with at most k non-adjacent elements.

LXXI. Write a Python program to find the number of ways to partition a string into k distinct palindromes with at least one element and at least one distinct element.

LXXII. Implement a Python function to find the maximum sum of a subarray with at least k non-adjacent elements.

LXXIII. Write a Python program to find the number of ways to arrange n objects in a circle with k distinct elements and at least one element and at least one distinct element.

LXXIV. Implement a Python function to find the maximum sum of a subarray with exactly k non-adjacent elements.

LXXV. Write a Python program to find the number of ways to partition a set into k subsets with at least one element and at least one distinct element and at least one non-empty subset.

LXXVI. Implement a Python function to find the maximum sum of a subarray with at most k adjacent elements.

LXXVII. Write a Python program to find the number of ways to partition a string into k distinct palindromes with at least one element and at least one distinct element and at least one non-empty subset.

LXXVIII. Implement a Python function to find the maximum sum of a subarray with at least k adjacent elements.

LXXIX. Write a Python program to find the number of ways to arrange n objects in a line with k distinct elements and at least one element and at least one distinct element and at least one non-empty subset.

LXXX. Implement a Python function to find the maximum sum of a subarray with exactly k adjacent elements.

LXXXI. Write a Python program to find the number of ways to partition a set into k non-empty subsets with at least one element and at least one distinct element and at least one non-empty subset and at least one distinct subset.

LXXXII. Implement a Python function to find the maximum sum of a subarray with at most k non-adjacent elements.

LXXXIII. Write a Python program to find the number of ways to partition a string into k distinct palindromes with at least one element and at least one distinct element and at least one non-empty subset and at least one distinct subset.

LXXXIV. Implement a Python function to find the maximum sum of a subarray with at least k non-adjacent elements.

LXXXV. Write a Python program to find the number of ways to arrange n objects in a circle with k distinct elements and at least one element and at least one distinct element and at least one non-empty subset and at least one distinct subset.

These questions cover a wide range of advanced Python topics and will help you prepare thoroughly for your exam. Good luck with your studies!

I would like to express my sincere gratitude to **Sir Muhammad Rizwan** for his outstanding guidance and support throughout our Advanced Python course. Your expertise and dedication have been instrumental in our learning journey, and your insightful instruction continues to inspire us.

A big thank you to my **fellow classmates** for your active participation and collaborative spirit. Our shared enthusiasm and teamwork have made this course both engaging and rewarding.

As we transition into the AI section of our course, I am excited about the new challenges and opportunities this area presents. I look forward to deepening our understanding of artificial intelligence and applying these concepts in practical scenarios.

As a fun note, I used my repository named merged-pdf-files to combine all my notes from different classes into a single PDF document. You're seeing the final result of this effort right here!

Here's to our continued exploration and learning in the world of AI!

Warm regards,

Mansoor Bukhari