

Name: Ms. Prarthi Hrishit Kothari

Class: M. Sc. Bioinformatics (Part II)

Roll Number: 115

Course: M. Sc. Bioinformatics

Department: Department of Bioinformatics

Paper: Mandatory Paper II

Paper Name and Code: Python Programming language and Machine Learning in Bioinformatics (GNKPSBIMJ2P503)

Academic Year: 2024-25



SGCP's

Guru Nanak Khalsa College
of Arts, Science & Commerce (Autonomous)

DEPARTMENT OF BIOINFORMATICS

CERTIFICATE

This is to certify that Ms. Prarthi Hrishit Kothari (Roll No: 115) of M. Sc. Bioinformatics (Part II) has satisfactorily completed the practical for Mandatory Paper II: Python Programming Language and Machine Learning in Bioinformatics (GNKPSBIMJ2P503) for Semester III course prescribed by the University of Mumbai during the academic year 2024-2025.

Teacher-in-Charge
(Mrs. Sermarani Nadar)
(Ms. Komalharini Tiwari)

Head of Department
(Dr. Gursimran Kaur Uppal)

External Examiner

INDEX

Practical No.	Title	Date	Page No.	Signature
1	PYTHON: Basic Programming	24/07/2024	1	
2	PYTHON: List	30/07/2024	16	
3	PYTHON: Tuples	28/08/2024	30	
4	PYTHON: Functions	20/09/2024	43	
5	PYTHON: Class and Object – Oriented Programming (OOPs)	26/09/2024	49	
6	PYTHON: Regular Expression	27/09/2024	64	
7	PYTHON: Biopython	03/10/2024	73	
8	Machine Learning	26/09/2024	83	

Date: 24/07/2024

Practical 1

PYTHON: Basic Programming

AIM:

To demonstrate the basic operations and functionalities of Python programming language by writing Python programs and executing the programs in the Python IDLE.

INTRODUCTION:

Python Programming Language

Python is a popular, high-level programming language renowned for its ease of use, readability, and adaptability. It was created by Guido van Rossum and first released in 1991.

Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

One must first install a Python interpreter on the system in order to begin programming in Python. IDLE, PyCharm, and Visual Studio Code are popular IDEs for Python programming.

Category	Property	Description
Language Fundamentals	Interpreted	Code runs line by line without prior compilation.
	Object-Oriented	Supports object-based programming concepts like classes and objects.
	High-Level	Abstractions hide machine-level details, making code easier to read and write.
	Dynamic Semantics	Variable types are determined at runtime, offering flexibility.
Code Readability and Flexibility	Emphasizes Readability	Uses indentation to define code blocks, improving clarity.
	Supports Multiple Paradigms	Can be used for structured, object-oriented, and functional programming styles.
Versatile Standard Library	Batteries Included	Provides built-in modules for common tasks, reducing development time.
	Wide Functionality	Offers tools for file handling, data manipulation, networking, and more.
Dynamic Typing and Memory Management	Dynamically Typed	Variables don't require pre-defined types, offering flexibility.
	Automatic Memory Management	Handles memory allocation and deallocation, simplifying development.
	Large Third-Party Library Support	Numerous libraries and frameworks extend Python's functionality significantly.

Active Community and Ecosystem	Python Package Index (PyPI)	Central repository for finding and installing these extensions.
Advantages	<ol style="list-style-type: none"> 1. Vast Standard Extensive Libraries – which provide pre-built functionalities for various applications 2. Open Source – free to use and distribute 3. High Productivity – allows developers to accomplish tasks with fewer lines of code compared to languages like Java or C++. 4. Cross-Platform Compatibility – Python code can run on various operating systems without modification, enhancing its portability 5. Interpreted Language – simplifies debugging by executing code line-by-line and halts the execution when an error occurs. 	
Disadvantages	<ol style="list-style-type: none"> 1. High Memory Consumption – due to Python's dynamic nature which can lead to higher memory usage. 2. Runtime Errors – The dynamic typing feature can lead to runtime errors if variables change types unexpectedly 3. Database Access – Python's database access capabilities are considered less robust compared to other technologies, which may hinder its use in large enterprise applications. 	
Broad Range of Applications	Server-Side Web Development	Creates dynamic web content.
	Software Development	Used for building various software applications.
	Data Analysis	Powerful tool for data exploration and manipulation.
	Artificial Intelligence	Popular language for machine learning and AI development.
	System Scripting	Automates tasks within computer systems.

Python Data Types

Data Type	Description	Example
int	hold signed integers of non – limited length	a = 3
long	holds long integers exist in python 2.x ; deprecated in python 3.x	>>> print type(65536*65536) <type 'long'>
float	holds floating precision numbers accurate up to 15 decimal places	a = 5.6
complex	holds complex numbers	a = 2 + 4j

Taking Input in Python

input (prompt) → this function first takes the input from the user and then evaluates the expression
Python automatically identifies whether user entered a string or a number or a list

CODE	OUTPUT
val = input ("Enter your value: ") print (val)	Enter your value: <u>123</u> 123

Basic Operators in Python

Types	Operator	Symbol	Description	Syntax and Example
Arithmetic Operators	Addition	+	Adds two operands	$x + y$ $7 + 2 = 9$
	Subtraction	-	Subtracts two operands	$x - y$ $7 - 2 = 5$
	Multiplication	*	Multiplies two operands	$x * y$ $7 * 2 = 14$
	Division	/	Divides the first operand by the second	x / y $7 / 2 = 3.5$
	Floor Division	//	Rounds off the answer to the nearest whole number	$x // y$ $7 // 2 = 3$
	Modulus	%	Returns the remainder when the first operand is divided by the second	$x \% y$ $7 \% 2 = 1$
	Exponent	**	Returns first raised to power second	$x ** y$ $7 ** 2 = 49$
Relational Operators (compares the values → returns either True or False according to the condition)	Less than	<	True if the left operand is less than the right	$a < b$
	Greater than	>	True if the left operand is greater than the right	$a > b$
	Less than or equal to	<=	True if left operand is less than or equal to the right	$a <= b$
	Greater than or equal to	>=	True if left operand is greater than or equal to the right	$a >= b$
	Equal to	==	True if both operands are equal	$a == b$
	Not equal to	!=	True if operands are not equal	$a != b$
Logical Operators	Logical AND	and	True → if both the operands are true	x and y True and True → True True and False → False False and True → False False and False → False
	Logical OR	or	True → if either of the operands is true	x or y True or True → True True or False → True False or True → True False or False → False
	Logical NOT	not	True → if operand is false	not x not True → False not False → True

Membership Operators (Tests for membership in a sequence such as strings, lists or tuples)	In	in	Evaluates to True if it finds a variable in the specified sequence and false otherwise	$x \text{ in } y$ \downarrow 'in' results in $\rightarrow 1 \rightarrow$ if x is a member of the sequence y \downarrow else $\rightarrow 0$
	Not in	not in	Evaluates to True if it does not find a variable in the specified sequence and false otherwise	$x \text{ not in } y$ \downarrow 'not in' results in $\rightarrow 1 \rightarrow$ if x is not a member of the sequence y \downarrow else $\rightarrow 0$
CODE <pre> a = 10 b = 20 list1 = [1, 2, 3, 4, 5] if (a in list1): print ("a is available") else: print ("a is not available") if (b not in list1): print ("b is not available") else: print ("b is available") </pre>			OUTPUT <pre> a is not available b is not available </pre>	
Assignment Operator	Assignment Operator	=	Used to assign values to the variable	$x = 10$ $\gg \text{print}(x)$ 10
	Addition Assignment Operator	+=	Add right side operand with left side operand and then assign the result to left operand	$x += y$
	Subtraction Assignment Operator	-=	Subtract right side operand from left side operand and then assign the result to left operand	$x -= y$
	Multiplication Assignment Operator	*=	Multiply right operand with left operand and then assign the result to the left operand	$x *= y$
	Division Assignment Operator	/=	Divide left operand with right operand and then assign the result to the left operand	$x /= y$
	Modulus Assignment Operator	%=	Divides the left operand with the right operand and then assign the	$x //= y$

			remainder to the left operand	
	Floor Division Assignment Operator	//=	Divide left operand with right operand and then assign the value(floor) to left operand	x % = y
	Exponent Assignment Operator	**=	Calculate exponent (raise power) value using operands and then assign the result to left operand	x ** = y

Python Strings

A String is a data structure in Python Programming that represents a sequence of characters. They are versatile and come with a rich set of features and methods that make them easy to manipulate.

Characteristics

- **Immutable:** Strings in Python are immutable, meaning that once a string is created, it cannot be changed. Any operation that modifies a string will return a new string.
- **Ordered:** Strings maintain the order of characters, and each character can be accessed using indexing.

Symbol	String Operators	Description	Example (a = "Hello" b = "Python")	Output
Concatenation	+	Adds values on either side of the operator	a + b	Hello Python
Repetition	*	Creates new strings concatenating multiple copies of the same string	a * 2	HelloHello
Slice	[]	Gives the character from the given index	a [1]	e
Range Slice	[:]	Gives the character from the given range	a [1:4]	ell
Membership	in	Returns True (1) if a character exists in the given string else False (0)	H in a	1
Membership	not in	Returns True (1) if a character does not exist in the given string else False (0)	M not in a	1

Conditional Statements

Conditional statements in Python are used to make decisions based on certain conditions. They allow you to execute different blocks of code depending on whether a condition is true or false.

1. if Statement

The if statement is used to execute a block of code if a certain condition is true.

Syntax – if condition:

```
# code to execute if condition is true
```

The condition can be any expression that evaluates to a Boolean value (True or False).

If the condition is True, the code block indented below the if statement will be executed.

If the condition is False, the code block will be skipped.

2. if-else Statement

The if-else statement is used to execute one block of code if a condition is true and another block if the condition is false.

Syntax – if condition:

```
# code to execute if condition is true
else:
    # code to execute if condition is false
```

If the condition is True, the code block indented below the if statement will be executed, and the code block indented below the else statement will be skipped. If the condition is False, the code block indented below the else statement will be executed, and the code block indented below the if statement will be skipped.

3. if-elif-else Statement

The if-elif-else statement is used to check multiple conditions in sequence. It allows you to execute different code blocks depending on which condition is true.

Syntax – if condition1:

```
# code to execute if condition1 is true
elif condition2:
    # code to execute if condition1 is false and condition2 is true
elif condition3:
    # code to execute if condition1 and condition2 are false, and
    # condition3 is true
else:
    # code to execute if all conditions are false
```

4. Nested if Statements

Nested if statements are if statements inside another if statement. They are used to check for multiple conditions within a single if block.

Syntax – if condition1:

code to execute if condition1 is true

if condition2:

code to execute if both condition1 and condition2 are true

Loops

Loops enable the repeated execution of a block of code until a predetermined condition is met. They allow for efficient iteration over sequences (like lists, tuples, or strings) and other iterable objects.

1. for loop

The ‘for’ loop is used to iterate over a sequence (like a list, tuple, dictionary, set, or string). It executes a block of code for each item in the sequence.

Syntax	Example	Output
for variable in sequence: #code block	fruits = ["apple", "banana", "cherry"] for fruit in fruits: print(fruit)	apple banana cherry
	for i in range(5): print(i)	0 1 2 3 4

2. while loop

The ‘while’ loop repeatedly executes a block of code as long as a specified condition is true. It is useful when the number of iterations is not known beforehand.

Syntax	Example	Output
while condition: # code block	count = 0 while count < 5: print (count) count += 1 # Increment the count to avoid an infinite loop	0 1 2 3 4

PROGRAMS –

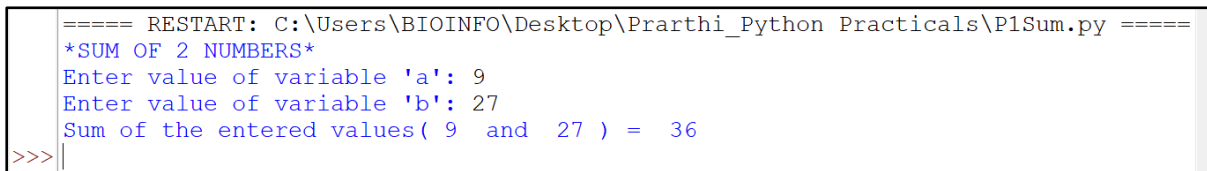
Question 1 –

Write a Python program to add two numbers and print the sum of the two numbers.

CODE –

```
print("*SUM OF 2 NUMBERS*")
a = int(input("Enter value of variable 'a': "))
b = int(input("Enter value of variable 'b': "))
sum = a + b
print("Sum of the entered values(", a, " and ", b, ") = ", sum)
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P1Sum.py =====
*SUM OF 2 NUMBERS*
Enter value of variable 'a': 9
Enter value of variable 'b': 27
Sum of the entered values( 9 and 27 ) = 36
>>>
```

Figure 1: Calculating and displaying the sum of two numbers entered by the user

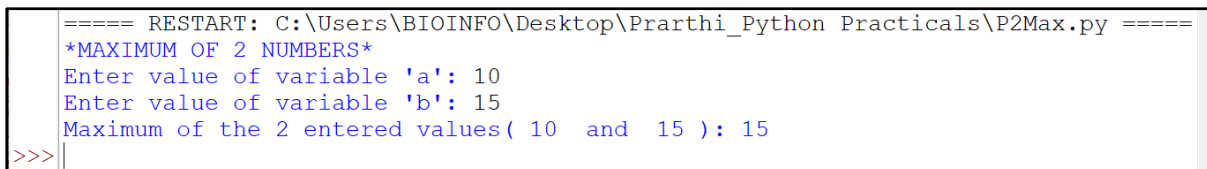
Question 2 –

Write a Python program to display the maximum of two numbers.

CODE –

```
print("*MAXIMUM OF 2 NUMBERS*")
a = int(input("Enter value of variable 'a': "))
b = int(input("Enter value of variable 'b': "))
if(a > b):
    print("Maximum of the 2 entered values(", a, " and ", b, "):", a)
else:
    print("Maximum of the 2 entered values(", a, " and ", b, "):", b)
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P2Max.py =====
*MAXIMUM OF 2 NUMBERS*
Enter value of variable 'a': 10
Enter value of variable 'b': 15
Maximum of the 2 entered values( 10 and 15 ): 15
>>>
```

Figure 2: Displaying the maximum value of the two user – entered numbers

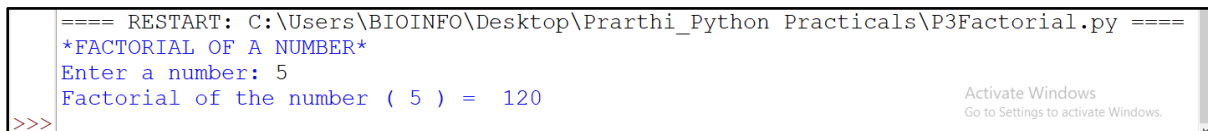
Question 3 –

Write a Python program to calculate the factorial of a user – entered number.

CODE –

```
print("*FACTORIAL OF A NUMBER*")
num = int(input("Enter a number: "))
fact = 1
num1 = num #to store the original value of the number
while num > 0:
    fact *= num
    num -= 1
print("Factorial of the number (", num1, ") = ", fact)
```

OUTPUT –



```
==== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P3Factorial.py ====
*FACTORIAL OF A NUMBER*
Enter a number: 5
Factorial of the number ( 5 ) = 120
```

Figure 3: Calculating and displaying the factorial of a user – entered number

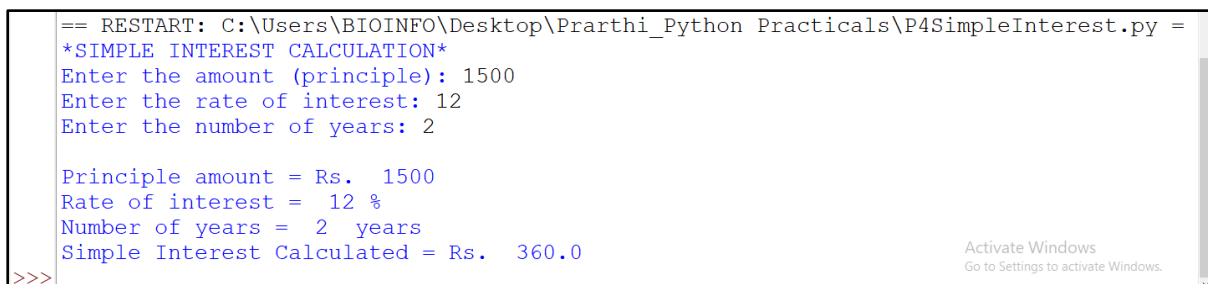
Question 4 –

Write a Python program to calculate the simple interest.

CODE –

```
print("*SIMPLE INTEREST CALCULATION*")
p = int(input("Enter the amount (principle): "))
r = int(input("Enter the rate of interest: "))
n = int(input("Enter the number of years: "))
si = (p * n * r)/100
print("\nPrinciple amount = Rs. ", p)
print("Rate of interest = ", r, "%")
print("Number of years = ", n, " years")
print("Simple Interest Calculated = Rs. ", si)
```

OUTPUT –



```
== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P4SimpleInterest.py ==
*SIMPLE INTEREST CALCULATION*
Enter the amount (principle): 1500
Enter the rate of interest: 12
Enter the number of years: 2

Principle amount = Rs. 1500
Rate of interest = 12 %
Number of years = 2 years
Simple Interest Calculated = Rs. 360.0
```

Figure 4: Displaying the amount of Simple Interest calculated on the basis of the principle amount, rate of interest and the number of years entered by the user

Question 5 –

Write a Python program to sort the user – entered words in an alphabetic order.

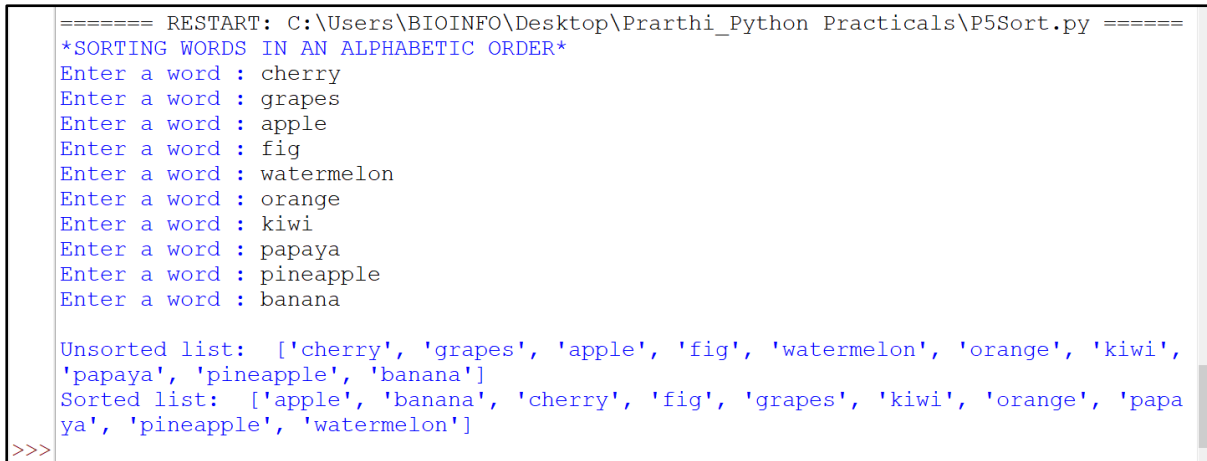
CODE –

```
print("*SORTING WORDS IN AN ALPHABETIC ORDER*")

l1 = []
for i in range(10):
    l1.append(input("Enter a word : "))

print("\nUnsorted list: ", l1)
l1.sort()
print("Sorted list: ", l1)
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P5Sort.py =====
*SORTING WORDS IN AN ALPHABETIC ORDER*
Enter a word : cherry
Enter a word : grapes
Enter a word : apple
Enter a word : fig
Enter a word : watermelon
Enter a word : orange
Enter a word : kiwi
Enter a word : papaya
Enter a word : pineapple
Enter a word : banana

Unsorted list: ['cherry', 'grapes', 'apple', 'fig', 'watermelon', 'orange', 'kiwi', 'papaya', 'pineapple', 'banana']
Sorted list: ['apple', 'banana', 'cherry', 'fig', 'grapes', 'kiwi', 'orange', 'papaya', 'pineapple', 'watermelon']
>>>
```

Figure 5: Displaying the list of user – entered word before and after sorting in an alphabetic order

Question 6 –

Write a Python program to check if a user – entered number is an Armstrong number or not.

CODE –

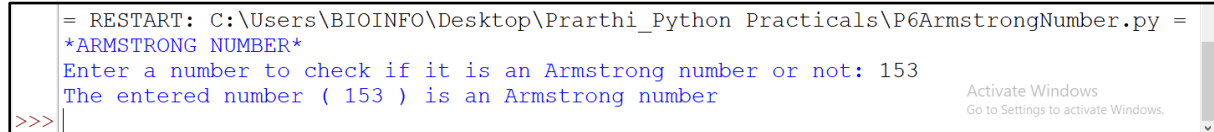
```
print("*ARMSTRONG NUMBER*")
num = int(input("Enter a number to check if it is an Armstrong number or not: "))

sum = 0
num3 = num #to store original number entered

while num != 0:
    unitdigit = num % 10
    sum += unitdigit ** len(str(num3))
    num = num // 10

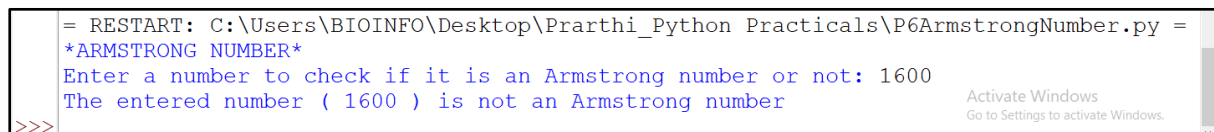
if(sum == num3):
    print("The entered number (", num3, ") is an Armstrong number")
else:
    print("The entered number (", num3, ") is not an Armstrong number")
```

OUTPUT –



```
= RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P6ArmstrongNumber.py =
*ARMSTRONG NUMBER*
Enter a number to check if it is an Armstrong number or not: 153
The entered number ( 153 ) is an Armstrong number
>>>
```

Figure 6: Displaying the output when the user – entered number is an Armstrong number



```
= RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P6ArmstrongNumber.py =
*ARMSTRONG NUMBER*
Enter a number to check if it is an Armstrong number or not: 1600
The entered number ( 1600 ) is not an Armstrong number
>>>
```

Figure 7: Displaying the output when the user – entered number is not an Armstrong number

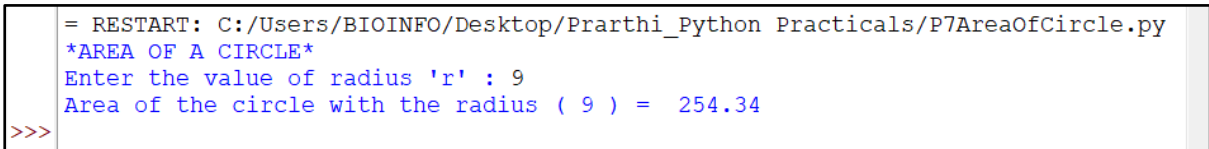
Question 7 –

Write a Python program to find the area of a circle.

CODE –

```
print("*AREA OF A CIRCLE*")
radius = int(input("Enter the value of radius 'r' : "))
pi = 3.14
area = pi * radius * radius
print("Area of the circle with the radius (", radius, ") = ", area)
```

OUTPUT –



```
= RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/P7AreaOfCircle.py
*AREA OF A CIRCLE*
Enter the value of radius 'r' : 9
Area of the circle with the radius ( 9 ) =  254.34
>>>
```

Figure 8: Calculating and displaying the area of a circle with the radius entered by the user

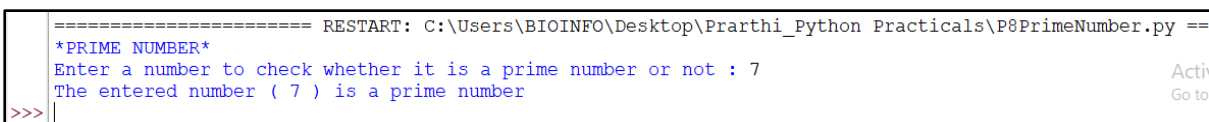
Question 8 –

Write a Python program to check whether a number is prime or not.

CODE –

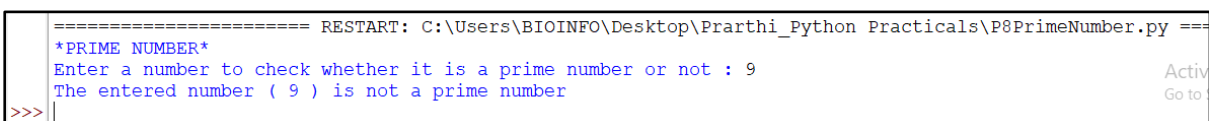
```
print("*PRIME NUMBER*")
num = int(input("Enter a number to check whether it is a prime number or not : "))
count = 0
for i in range(2, num):
    if(num != i):
        if(num % i == 0):
            count += 1
            break
if(count != 0):
    print("The entered number (", num, ") is not a prime number")
else:
    print("The entered number (", num, ") is a prime number")
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P8PrimeNumber.py ==
*PRIME NUMBER*
Enter a number to check whether it is a prime number or not : 7
The entered number ( 7 ) is a prime number
>>>
```

Figure 9: Displaying the output when the user – entered number is a Prime number



```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\P8PrimeNumber.py ==
*PRIME NUMBER*
Enter a number to check whether it is a prime number or not : 9
The entered number ( 9 ) is not a prime number
>>>
```

Figure 10: Displaying the output when the user – entered number is not a Prime number

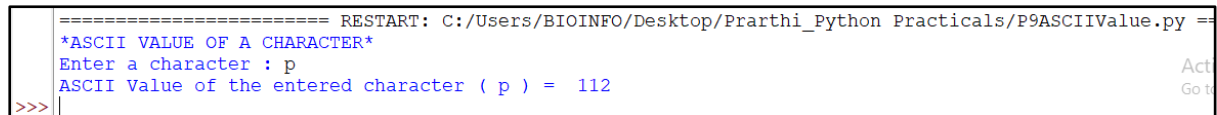
Question 9 –

Write a Python program to print the ASCII value of a character.

CODE –

```
print("*ASCII VALUE OF A CHARACTER*")
character = input("Enter a character : ")
print("ASCII Value of the entered character (", character, ") = ", ord(character))
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/P9ASCIIValue.py =====
*ASCII VALUE OF A CHARACTER*
Enter a character : p
ASCII Value of the entered character ( p ) = 112
>>>
```

Figure 11: Displaying the ASCII Value of a character entered by the user

Question 10 –

Write a Python program to calculate and display the sum of the squares of first ‘n’ natural numbers.

CODE –

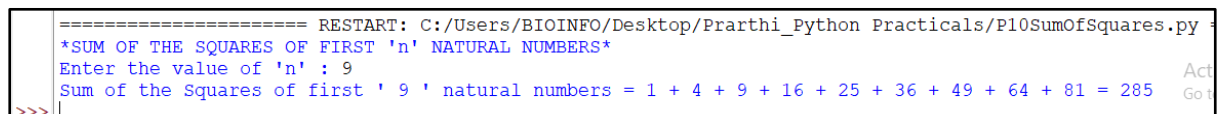
```
print("*SUM OF THE SQUARES OF FIRST 'n' NATURAL NUMBERS*")
n = int(input("Enter the value of 'n' : "))

sum = 1

print("Sum of the Squares of first '", n, "' natural numbers = ", end = "1 + ")
for i in range(2, n+1):
    sum += (i**2)
    if(i != n):
        print(i**2, end = " + ")
    else:
        print(i**2, end = " = ")

print(sum)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/P10SumOfSquares.py =====
*SUM OF THE SQUARES OF FIRST 'n' NATURAL NUMBERS*
Enter the value of 'n' : 9
Sum of the Squares of first ' 9 ' natural numbers = 1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 = 285
>>>
```

Figure 12: Calculating and displaying the sum of the squares of first ‘n’ natural numbers

Question 11 –

Write a Python program to check whether the user – entered sequence is an RNA sequence or not.

CODE –

```
print("*TO CHECK WHETHER A SEQUENCE IS AN RNA OR NOT*")

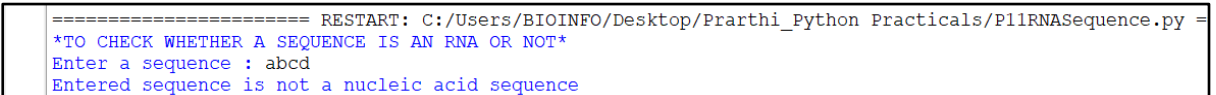
seq = input("Enter a sequence : ")
flag1 = True #to check if it is a nucleic acid sequence or not
flag2 = True #to check if it is an RNA sequence or not

for i in range(len(seq)):
    if(seq[i] not in ('a', 't', 'u', 'g', 'c')):
        flag1 = False
        break

    if(seq[i] not in ('a', 'u', 'g', 'c')):
        flag2 = False


if(flag1 == False):
    print("Entered sequence is not a nucleic acid sequence")
elif(flag2 == False):
    print("Entered sequence is not an RNA sequence")
else:
    print("Entered sequence is an RNA sequence")
```

OUTPUT –



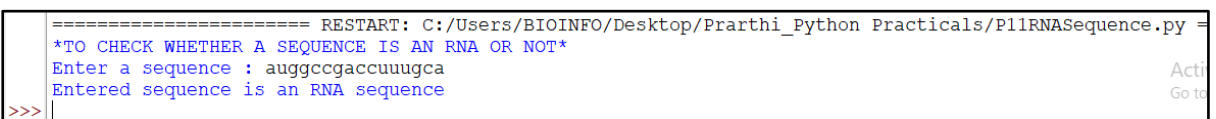
```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/P11RNASequence.py =
*TO CHECK WHETHER A SEQUENCE IS AN RNA OR NOT*
Enter a sequence : abcd
Entered sequence is not a nucleic acid sequence
```

Figure 13: Displaying the output when the user – entered sequence is not a nucleic acid sequence



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/P11RNASequence.py =
*TO CHECK WHETHER A SEQUENCE IS AN RNA OR NOT*
Enter a sequence : atgccgtcag
Entered sequence is not an RNA sequence
```

Figure 14: Displaying the output when the user – entered sequence is not an RNA sequence



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/P11RNASequence.py =
*TO CHECK WHETHER A SEQUENCE IS AN RNA OR NOT*
Enter a sequence : augccgaccuuugca
Entered sequence is an RNA sequence
>>>
```

Figure 15: Displaying the output when the user – entered sequence is an RNA sequence

RESULTS:

The basic operations and functionalities of the Python programming language were demonstrated by writing python programs and executing the programs in the Python IDLE to display the sum of two user – entered numbers, to display the maximum value of two user – entered numbers, to calculate and display the factorial of a user – entered number, to calculate the simple interest, to sort the user – entered words in an alphabetic order, to check if a user – entered number is an Armstrong number or not, to find the area of a circle with the value of radius entered by the user, to check whether a user – entered number is a Prime number or not, to display the ASCII value of a user – entered character and to check whether a user – entered sequence is an RNA sequence or not.

CONCLUSION:

The basic operations and functionalities of Python programming language were demonstrated by writing Python programs and executing the programs in the Python IDLE.

Date: 30/07/2024

Practical 2

PYTHON: List

AIM:

To demonstrate the functions and manipulations of lists using Python programming language.

INTRODUCTION:

Python is a popular, high-level programming language renowned for its ease of use, readability, and adaptability. It was created by Guido van Rossum and first released in 1991. Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

Python Lists

In Python, lists are fundamental data structures that allow you to store and manipulate collections of data.

A list is an ordered, mutable (changeable) collection of items. Lists can hold a variety of data types, including numbers, strings, and even other lists. Lists are ideal for collections of items where order matters, such as maintaining a sequence of user inputs or storing multiple values of the same type.

Characteristics

- **Ordered** – The items in a list maintain their order.
- **Mutable** – You can change, add, or remove items after the list is created.
- **Dynamic** – Lists can grow and shrink in size.

Syntax – `list1 = [1, 2, 3, 4, 5, 'abc', 'def']`

Add and remove items

- To add item → `append()` function
Example – `userAge.append(100)` → `userAge = [15, 25, 30, 45, 50, 65, 55, 100]`
- To remove any value from the list → `del listName [index of the item to be deleted]`
Example – `del userAge [2]` → `userAge = [15, 25, 45, 50, 65, 55, 100]` #‘30’ was removed

Updating lists

Can update single or multiple elements of list by giving the slice on the LHS of the assignment operator

CODE	OUTPUT
<pre>list1 = ['Physics', 'Chemistry', 1997, 2000] print ("Value available at index 2: ", list1 [2]) list [2] = 2001 print ("New value available at index 2: ", list [2])</pre>	<pre>Value available at index 2: 1997 New value available at index 2: 2001</pre>

Built – in Functions of list

Function / Method	Description	Example	Output
cmp (list1, list2)	Compares elements of both list	<pre>list1 = [1, 2, 3] list2 = [1, 2, 4] # Comparison using operators if list1 < list2: print("list1 is less than list2") elif list1 > list2: print("list1 is greater than list2") else: print("list1 is equal to list2")</pre>	list1 is less than list2
len (list)	Gives the total length of the list	<pre>my_list = [1, 2, 3, 4, 5] length = len(my_list) print(length)</pre>	5
max (list)	Returns item from the list with maximum value	<pre>my_list = [1, 2, 3, 4, 5] maximum = max(my_list) print(maximum)</pre>	5
min (list)	Returns item from the list with minimum value	<pre>my_list = [1, 2, 3, 4, 5] minimum = min(my_list) print(minimum)</pre>	1
list.append (obj)	Appends object obj to list returns	<pre>my_list = [1, 2, 3] my_list.append(4) print(my_list)</pre>	[1, 2, 3, 4]
list.count (obj)	Count of how many times obj occurs in the list	<pre>my_list = [1, 2, 2, 3, 4] count = my_list.count(2) print(count)</pre>	2
list.extend (seq)	Appends the contents of sequence to list	<pre>my_list = [1, 2, 3] my_list.extend([4, 5, 6]) print(my_list)</pre>	[1, 2, 3, 4, 5, 6]
list.index (obj)	Returns the lowest index in list that obj appears	<pre>my_list = [1, 2, 3, 2, 4] index = my_list.index(2) print(index)</pre>	1
list.insert (index, obj)	Inserts object obj into list at offset particular index	<pre>my_list = [1, 2, 3] my_list.insert(1, 4) print(my_list)</pre>	[1, 4, 2, 3]

list.pop (obj = list [-1])	Removes and returns the last object or obj from list	my_list = [1, 2, 3] popped_item = my_list.pop() print(popped_item) print(my_list)	3 [1, 2]
list.remove (obj)	Removes object obj from list	my_list = [1, 2, 3, 2] my_list.remove(2) print(my_list)	[1, 3, 2]
list.reverse ()	Reverses object of list in place	my_list = [1, 2, 3] my_list.reverse() print(my_list)	[3, 2, 1]
list.sort ([func])	Sorts objects of list use compare function if given	my_list = [3, 1, 2] my_list.sort() print(my_list)	[1, 2, 3]

PROGRAMS –

Question 1 –

Write a Python program to create an array of 5 integers and display the array items. Access individual elements through indexes.

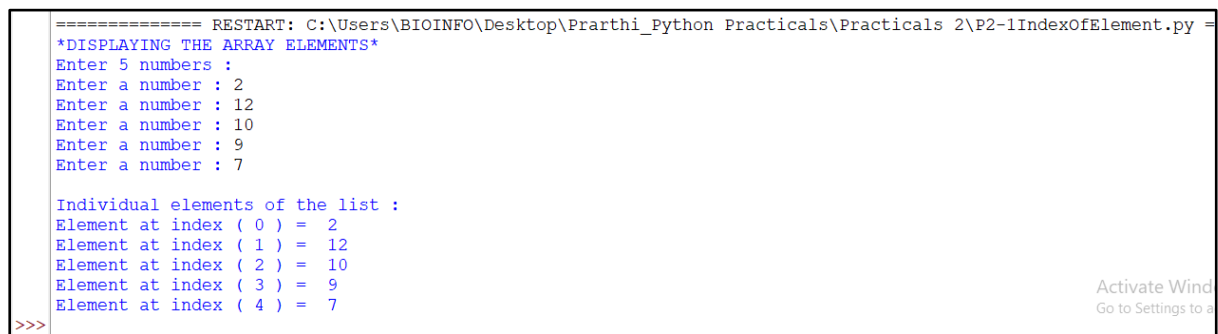
CODE –

```
print("*DISPLAYING THE ARRAY ELEMENTS*")

list1 = [ ]
print("Enter 5 numbers : ")
for i in range(5):
    num = int(input("Enter a number : "))
    list1.append(num)

print("\nIndividual elements of the list : ")
for i in range(5):
    print("Element at index (", i , ") = ", list1[i])
```

OUTPUT –

A screenshot of a Python IDE window. The title bar reads "RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\Practicals 2\P2-1IndexOfElement.py". The editor contains the following code:

```
***** DISPLAYING THE ARRAY ELEMENTS*****
Enter 5 numbers :
Enter a number : 2
Enter a number : 12
Enter a number : 10
Enter a number : 9
Enter a number : 7

Individual elements of the list :
Element at index ( 0 ) = 2
Element at index ( 1 ) = 12
Element at index ( 2 ) = 10
Element at index ( 3 ) = 9
Element at index ( 4 ) = 7
>>>
```

The output shows the user entering five numbers (2, 12, 10, 9, 7) and then the program displaying each element of the list along with its index from 0 to 4. On the right side of the IDE, there is a small text that says "Activate Windows Go to Settings to activate Windows".

Figure 1: Displaying the individual elements of an array by accessing them through their indices

Question 2 –

Write a Python program to append a new item at the end of the array.

CODE –

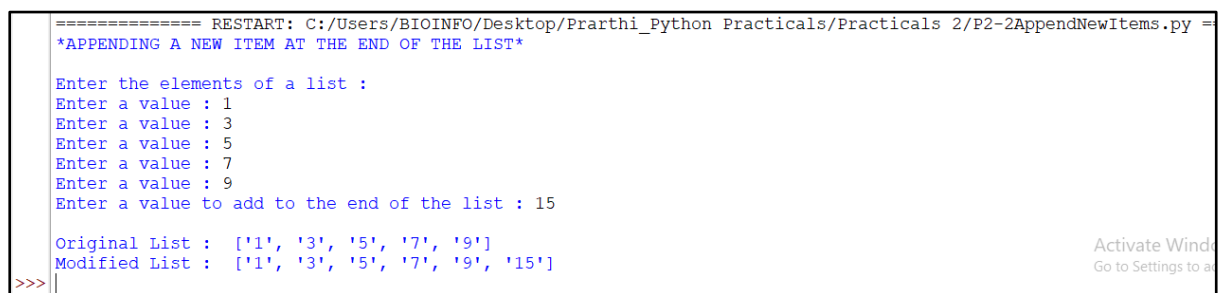
```
print("*APPENDING A NEW ITEM AT THE END OF THE LIST*")

list1 = [ ]
print("\nEnter the elements of a list : ")

for i in range(5):
    val1 = input("Enter a value : ")
    list1.append(val1)

val2 = input("Enter a value to add to the end of the list : ")
print("\nOriginal List : ", list1)
list1.append(val2)
print("Modified List : ", list1)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-2AppendNewItems.py =====
*APPENDING A NEW ITEM AT THE END OF THE LIST*

Enter the elements of a list :
Enter a value : 1
Enter a value : 3
Enter a value : 5
Enter a value : 7
Enter a value : 9
Enter a value to add to the end of the list : 15

Original List : ['1', '3', '5', '7', '9']
Modified List : ['1', '3', '5', '7', '9', '15']
>>>
```

Figure 2: Displaying the original list and the modified list after appending a new item at the end of the array

Question 3 –

Write a Python program to reverse the order of the items of an array.

CODE –

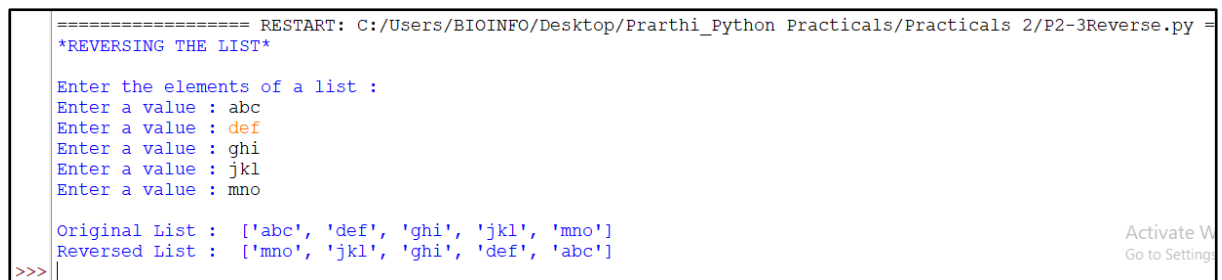
```
print("*REVERSING THE LIST*")

list1 = [ ]
print("\nEnter the elements of a list : ")

for i in range(5):
    val1 = input("Enter a value : ")
    list1.append(val1)

print("\nOriginal List : ", list1)
list1.reverse()
print("Reversed List : ", list1)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-3Reverse.py =
*REVERSING THE LIST*
Enter the elements of a list :
Enter a value : abc
Enter a value : def
Enter a value : ghi
Enter a value : jkl
Enter a value : mno

Original List : ['abc', 'def', 'ghi', 'jkl', 'mno']
Reversed List : ['mno', 'jkl', 'ghi', 'def', 'abc']
>>>
```

Figure 3: Displaying the reverse of a user – entered list

Question 4 –

Write a Python program to get the number of occurrences of a specified element in an array.

CODE –

```
print("*TO GET THE NUMBER OF OCCURRENCES OF AN ELEMENT IN AN  
ARRAY*")

count = 0
list1 = []
print("\nEnter the elements of a list : ")

for i in range(10):
    num = int(input("Enter a value : "))
    list1.append(num)

num2 = int(input("\nEnter a number to get its number of occurrences in the list : "))

for i in range(10):
    if(num2 == list1[i]):
        count += 1

print("\nList : ", list1)
print("Number of occurrences of the number (", num2, ") in the list = ", count)
```

OUTPUT –

```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-4NumberOfOccurrences.py =
*TO GET THE NUMBER OF OCCURRENCES OF AN ELEMENT IN AN ARRAY*

Enter the elements of a list :
Enter a value : 1
Enter a value : 3
Enter a value : 18
Enter a value : 9
Enter a value : 18
Enter a value : 27
Enter a value : 5
Enter a value : 63
Enter a value : 18
Enter a value : 48

Enter a number to get its number of occurrences in the list : 18

List : [1, 3, 18, 9, 18, 27, 5, 63, 18, 48]
Number of occurrences of the number ( 18 ) in the list = 3
>>>
```

Activate Windows
Go to Settings to activate Windows

Figure 4: Displaying the number of occurrences of an element in a list

Question 5 –

Write a Python program to append items from a specified list.

CODE –

```
print("*APPENDING THE ELEMENTS OF ANOTHER LIST*")

list1 = [ ]
list2 = [ ]

print("Enter the values for the first list : ")
for i in range(5):
    val = input("Enter a value : ")
    list1.append(val)

print("\nEnter the values for the second list : ")
for i in range(5):
    val2 = input("Enter a value : ")
    list2.append(val2)

print("\nFirst list : ", list1)
print("Second list : ", list2)

list1.extend(list2)
print("Appended (Merged) list : ", list1)
```

OUTPUT –

```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python Practicals\Practicals 2\P2-5AppendingElementsOfLists.py =
*APPENDING THE ELEMENTS OF ANOTHER LIST*
Enter the values for the first list :
Enter a value : 1
Enter a value : ABC
Enter a value : 99
Enter a value : DEF
Enter a value : GHI

Enter the values for the second list :
Enter a value : 4
Enter a value : 5
Enter a value : PQR
Enter a value : MNO
Enter a value : 9

First list : ['1', 'ABC', '99', 'DEF', 'GHI']
Second list : ['4', '5', 'PQR', 'MNO', '9']
Appended (Merged) list : ['1', 'ABC', '99', 'DEF', 'GHI', '4', '5', 'PQR', 'MNO', '9']
```

Activate Windows
Go to Settings to activate V

Figure 5: Displaying the final list after appending the items of a specified list to another list

Question 6 –

Write a Python program to insert a new item before the second element in an existing array.

CODE –

```
print("*INSERTING AN ELEMENT IN AN EXISTING ARRAY AT A SPECIFIC POSITION*")

list1 = [ ]
print("\nEnter the elements of a list : ")

for i in range(5):
    num = int(input("Enter a value : "))
    list1.append(num)

val = input("\nEnter a value to insert in the list before the second element : ")

print("\nOriginal List : ", list1)
list1.insert(1, val)
print("Modified List : ", list1)
```

OUTPUT –

```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-6InsertingElement.py =
*INSERTING AN ELEMENT IN AN EXISTING ARRAY AT A SPECIFIC POSITION*

Enter the elements of a list :
Enter a value : 1
Enter a value : 2
Enter a value : 3
Enter a value : 4
Enter a value : 5

Enter a value to insert in the list before the second element : abc

Original List : [1, 2, 3, 4, 5]
Modified List : [1, 'abc', 2, 3, 4, 5]
```

Figure 6: Displaying the modified list after inserting a new element before the second element in an existing array (list)

Question 7 –

Write a Python program to remove the first occurrence of a specified element from an array.

CODE –

```
print("*REMOVING THE FIRST OCCURRENCE OF A SPECIFIED ELEMENT  
FROM AN ARRAY*")

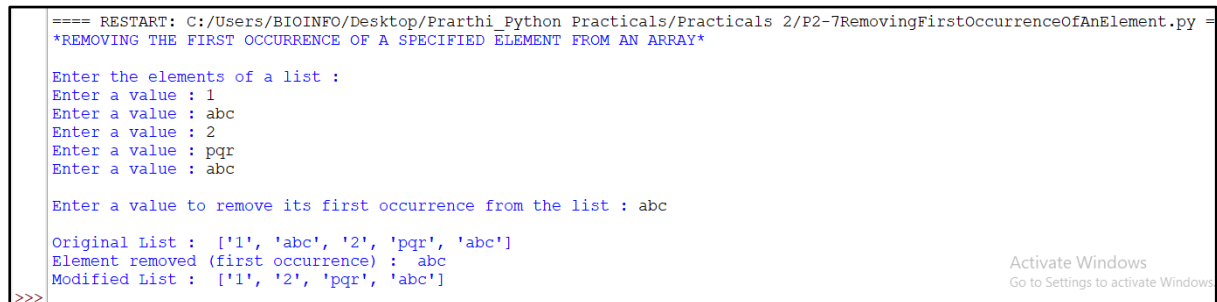
list1 = [ ]
print("\nEnter the elements of a list : ")

for i in range(5):
    val = input("Enter a value : ")
    list1.append(val)

val = input("\nEnter a value to remove its first occurrence from the list : ")

print("\nOriginal List : ", list1)
list1.remove(val)
print("Element removed (first occurrence) : ", val)
print("Modified List : ", list1)
```

OUTPUT –

The screenshot shows a Windows command prompt window with a Python script being executed. The script prompts the user to enter five elements for a list. The user enters '1', 'abc', '2', 'pqr', and 'abc'. Then, it prompts for a value to remove, and the user enters 'abc'. The output shows the original list ['1', 'abc', '2', 'pqr', 'abc'], the element removed ('abc'), and the modified list ['1', '2', 'pqr', 'abc'].

```
==== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-7RemovingFirstOccurrenceOfAnElement.py =
*REMOVING THE FIRST OCCURRENCE OF A SPECIFIED ELEMENT FROM AN ARRAY*

Enter the elements of a list :
Enter a value : 1
Enter a value : abc
Enter a value : 2
Enter a value : pqr
Enter a value : abc

Enter a value to remove its first occurrence from the list : abc

Original List : ['1', 'abc', '2', 'pqr', 'abc']
Element removed (first occurrence) : abc
Modified List : ['1', '2', 'pqr', 'abc']
>>>
```

Figure 7: Displaying the modified list after removing the first occurrence of a specified element from an array (list)

Question 8 –

Write a Python program to find the first duplicate element in a given array of integers.
Return -1 if there are no such integers.

CODE –

```
print("*FINDING DUPLICATES OF AN ELEMENT IN A LIST*")
list1 = [ ]
flag = False

for i in range(10):
    val = input("Enter an element : ")
    list1.append(val)
    print("\nList : ", list1)

for val in list1:
    count = list1.count(val)
    if count > 1:
        flag = True
        break
    else:
        count = -1
        continue

if(flag == True):
    print("\nDuplicate element found in the list : ", val)
    print("Number of times the element '", val, "' was found : ", count)
else:
    print("\nNo duplicate elements were found")
    print("Count = " , count)
```

OUTPUT –

```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python
*FINDING DUPLICATES OF AN ELEMENT IN A LIST*
Enter an element : 1
Enter an element : 2
Enter an element : 3
Enter an element : 4
Enter an element : 5
Enter an element : 6
Enter an element : 5
Enter an element : 7
Enter an element : 5
Enter an element : 9

List : ['1', '2', '3', '4', '5', '6', '5', '7', '5', '9']

Duplicate element found in the list : 5
Number of times the element ' 5 ' was found : 3
```

Figure 8: Displaying the duplicate element present in the list and the count of its occurrence

```
===== RESTART: C:\Users\BIOINFO\Desktop\Prarthi_Python
*FINDING DUPLICATES OF AN ELEMENT IN A LIST*
Enter an element : 1
Enter an element : 2
Enter an element : 3
Enter an element : 4
Enter an element : 5
Enter an element : 6
Enter an element : 7
Enter an element : 8
Enter an element : 9
Enter an element : 10

List : ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']

No duplicate elements were found
Count = -1
```

Figure 9: Displaying the output when no duplicate element is present in the list and returning the value of count as -1

Question 9 –

Write a Python program to create 2 lists with even numbers and odd numbers from a list.

CODE –

```
print("*SEPARATING A LIST INTO EVEN AND ODD NUMBERS*")

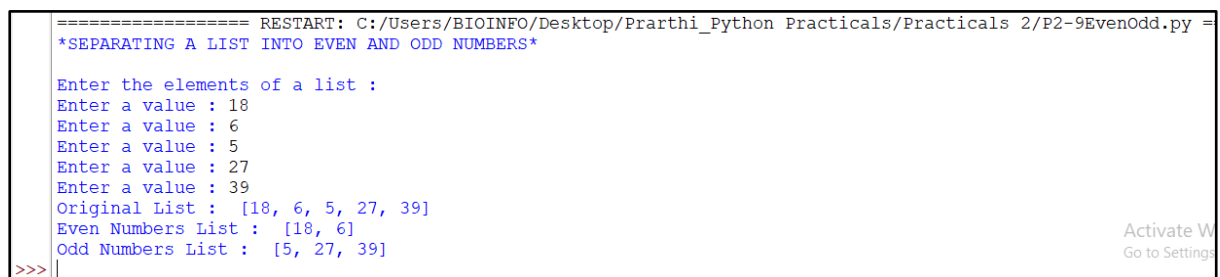
list1 = [ ]
list_even = [ ]
list_odd = [ ]
print("\nEnter the elements of a list : ")

for i in range(5):
    num = int(input("Enter a value : "))
    list1.append(num)

for i in range(len(list1)):
    if(list1[i] % 2 == 0):
        list_even.append(list1[i])
    else:
        list_odd.append(list1[i])

print("Original List : ", list1)
print("Even Numbers List : ", list_even)
print("Odd Numbers List : ", list_odd)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-9EvenOdd.py =
*SEPARATING A LIST INTO EVEN AND ODD NUMBERS*

Enter the elements of a list :
Enter a value : 18
Enter a value : 6
Enter a value : 5
Enter a value : 27
Enter a value : 39
Original List : [18, 6, 5, 27, 39]
Even Numbers List : [18, 6]
Odd Numbers List : [5, 27, 39]
>>>|
```

Figure 10: Displaying the two lists after separating a user – entered list into odd numbers and even numbers

Question 10 –

Write a Python program to extend a list using '+' Operator.

CODE –

```
print("*EXTENDING A LIST USING '+' OPERATOR*")
list1 = [ ]
list2 = [ ]

for i in range(5):
    val = input("Enter an element for list1 : ")
    list1.append(val)

print("\n")

for i in range(5):
    val = input("Enter an element for list2 : ")
    list2.append(val)

print("\nList 1 : ", list1)
print("List 2 : ", list2)
print("Extended List : " , (list1 + list2))
```

OUTPUT –

```
===== RESTART: C:/Users/BIOINFO/Desktop/Prarthi_Python Practicals/Practicals 2/P2-10Extend.py =
*EXTENDING A LIST USING '+' OPERATOR*
Enter an element for list1 : 1
Enter an element for list1 : 2
Enter an element for list1 : 3
Enter an element for list1 : 4
Enter an element for list1 : 5

Enter an element for list2 : ab
Enter an element for list2 : cd
Enter an element for list2 : ef
Enter an element for list2 : gh
Enter an element for list2 : ij

List 1 : ['1', '2', '3', '4', '5']
List 2 : ['ab', 'cd', 'ef', 'gh', 'ij']
Extended List : ['1', '2', '3', '4', '5', 'ab', 'cd', 'ef', 'gh', 'ij']
```

Activate W
Go to Setting

Figure 11: Displaying the final list after extending a list with another list using '+' operator

RESULTS:

The functions and manipulations of lists was demonstrated using Python programming language by creating programs to display the individual elements of an array by accessing them through their indices, to display the original list and the modified list after appending a new item at the end of the array, to display the reverse of a user – entered list, to display the number of occurrences of an element in a list, to display the final list after appending the items of a specified list to another list, to display the modified list after inserting a new element before the second element in an existing array (list), to display the modified list after removing the first occurrence of a specified element from an array (list), to display the duplicate element present in the list and the count of its occurrence, to display the two lists after separating a user – entered list into odd numbers and even numbers, and to display the final list after extending a list with another list using ‘+’ operator.

CONCLUSION:

The functions and manipulations of lists was demonstrated using Python programming language.

REFERENCES:

1. Python, R. (2023, October 21). Python’s list Data Type: A Deep Dive With Examples. <https://realpython.com/python-list/>
 2. GeeksforGeeks. (2024, June 19). Python Lists. GeeksforGeeks. <https://www.geeksforgeeks.org/python-lists/>
-

Practical 3 PYTHON: Tuples

AIM:

To demonstrate the functionality of tuples using Python programming language.

INTRODUCTION:

Python is a popular, high – level programming language renowned for its ease of use, reusability and adaptability. It was created by Guido van Rossum and first released in 1991. Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

Python Tuples

Python tuples are immutable sequences that have the capacity to store a group of elements. Once a tuple is generated, its elements cannot be modified. This is known as immutability. Tuples are useful for storing fixed sets of objects because of this capability. Tuples can be created by enclosing elements in parentheses, separated by commas. Following are the characteristics of a tuple:

1. **Immutable:** Once created, elements cannot be modified, added, or removed.
2. **Ordered:** The order of elements is preserved.
3. **Heterogeneous:** Tuples can contain elements of different data types.
4. **Allow Duplicates:** Since tuples are indexed, they can have items with the same value.

Syntax for creating a tuple:

```
tuple1 = (1, 2, 3, 'a', 'b')
```

Built – in Functions of Tuples

Function / Method	Syntax	Description	Example	Output
len()	len(tuple)	Returns the number of elements in a tuple.	<pre>tuple1 = (1, 2, 3, 4, 5) length = len(tuple1) print("The length of the tuple is", length)</pre>	The length of the tuple is 5
max()	max(tuple)	Returns the maximum value in a tuple.	<pre>tuple1 = (5, 6, 7, 8, 9) maximum = max(tuple1) print("The maximum value in the tuple is", maximum)</pre>	The maximum value in the tuple is 9

min()	min(tuple)	Returns the minimum value in a tuple.	tuple1 = (5, 6, 7, 8, 9) minimum = min(tuple1) print("The minimum value in the tuple is", minimum)	The minimum value in the tuple is 5
cmp()	cmp(tuple1, tuple2)	Used to compare two tuples (or other objects) Returns: <ul style="list-style-type: none"> • A negative integer if tuple1 is less than tuple2. • Zero if they are equal. • A positive integer if tuple1 is greater than tuple2. 	tuple1 = (1, 2, 3) tuple2 = (1, 2, 4) result = cmp(tuple1, tuple2) print("Comparison result:", result)	Comparison result: -1
count()	tuple.count (element)	Returns the number of occurrences of a specified element in the tuple.	tuple1 = (1, 2, 3, 2, 4, 2) count_of_2 = tuple1.count(2) print("The number 2 appears", count_of_2, "times in the tuple.")	The number 2 appears 3 times in the tuple.
index()	tuple.index (element)	Returns the index of the first occurrence of a specified element in the tuple.	tuple1 = (1, 2, 3, 4, 2) index_of_2 = tuple1.index(2) print("The index of the first occurrence of 2 is", index_of_2)	The index of the first occurrence of 2 is 1
tuple()	tuple(iterable _object)	Used to create a tuple from an iterable (like a list).	list1 = [1, 2, 3] tuple1 = tuple(list1) print("The tuple created from the list is", tuple1)	The tuple created from the list is (1, 2, 3)
sum()	sum(tuple)	Returns the sum of all elements in a tuple.	tuple1 = (2, 5, 7) total = sum(tuple1) print("The sum of the tuple elements is", total)	The sum of the tuple elements is 14

sorted()	sorted(tuple)	Returns a new sorted list from the elements of the tuple.	<pre>tuple1 = (5, 3, 1, 4, 2) sorted_tuple = sorted(tuple1) print("The sorted tuple is", sorted_tuple)</pre>	The sorted tuple is [1, 2, 3, 4, 5]
-----------	---------------	---	--	-------------------------------------

Adding items to a Tuple

Since tuples are immutable, they do not have a built-in append() method.

Example:	<pre>tuple1 = ("apple", "banana", "cherry") list1 = list(tuple1) list1.append("orange") tuple2 = tuple(list1) print("Original Tuple: ", tuple1) print("Modified Tuple: ", tuple2)</pre>
Output:	<pre>Original Tuple: ("apple", "banana", "cherry") Modified Tuple: ("apple", "banana", "cherry", "orange")</pre>

Concatenating Tuples

The '+' operator can be used to concatenate 2 tuples.

Example:	<pre>test_tup1 = (1, 3, 5) test_tup2 = (4, 6) print("The original tuple 1 : " + test_tup1) print("The original tuple 2 : " + test_tup2) # using + operator concatenated_tuple = test_tup1 + test_tup2 print("The tuple after concatenation is : " + concatenated_tuple)</pre>
Output:	<pre>The original tuple 1 : (1, 3, 5) The original tuple 2 : (4, 6) The tuple after concatenation is : (1, 3, 5, 4, 6)</pre>

Removing items from a Tuple

Example:	<pre>tuple1 = ("apple", "banana", "cherry") list1 = list(tuple1) list1.remove("apple") tuple2 = tuple(list1) print("Original Tuple: ", tuple1) print("Modified Tuple: ", tuple2)</pre>
Output:	<pre>Original Tuple: ("apple", "banana", "cherry") Modified Tuple: ("banana", "cherry")</pre>

Deleting an entire Tuple

The **'del'** keyword can delete the tuple completely.

Example:	<pre>tuple1 = ("apple", "banana", "cherry") del tuple1 print(tuple1) #this will raise an error because the tuple no longer exists</pre>
Output:	<pre>Traceback (most recent call last): File "demo_tuple_del.py", line 3, in <module> print(thistuple) #this will raise an error because the tuple no longer exists NameError: name 'thistuple' is not defined</pre>

PROGRAMS

Question 1 –

Write a Python program to create a tuple and display the tuple.

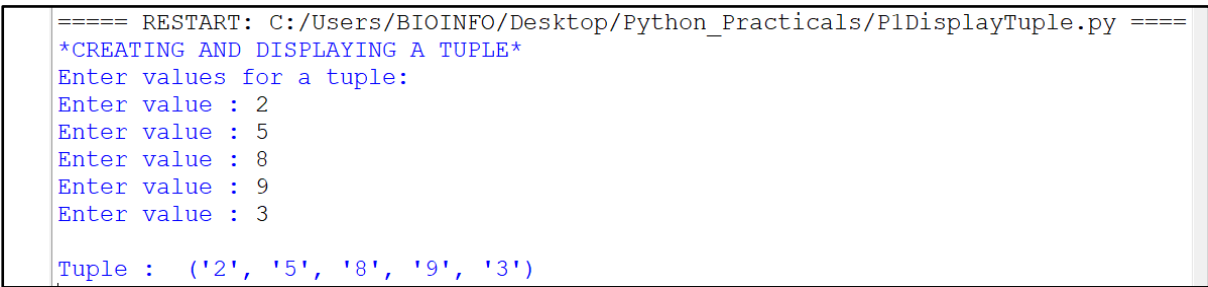
CODE –

```
print("*CREATING AND DISPLAYING A TUPLE*")
list1 = [ ]

print("Enter values for a tuple: ")
for i in range(5):
    x = input("Enter value : ")
    list1.append(x)

tuple1 = tuple(list1)
print("\nTuple : ", tuple1)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P1DisplayTuple.py =====
*CREATING AND DISPLAYING A TUPLE*
Enter values for a tuple:
Enter value : 2
Enter value : 5
Enter value : 8
Enter value : 9
Enter value : 3

Tuple :  ('2', '5', '8', '9', '3')
```

Figure 1: Creating and displaying a tuple with the values entered by the user

Question 2 –

Write a Python program to create a tuple with different data types.

CODE –

```
print("*CREATING AND DISPLAYING A TUPLE WITH DIFFERENT
DATATYPES*")

list1 = [ ]

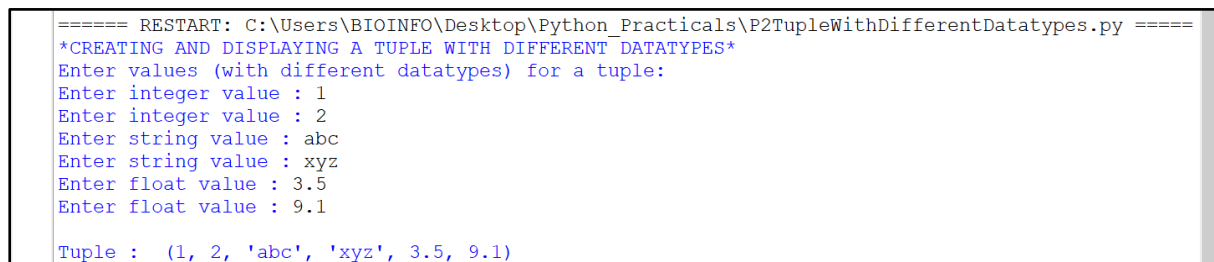
print("Enter values (with different datatypes) for a tuple: ")
for i in range(2):
    x = int(input("Enter integer value : "))
    list1.append(x)

for i in range(2):
    x = input("Enter string value : ")
    list1.append(x)

for i in range(2):
    x = float(input("Enter float value : "))
    list1.append(x)

tuple1 = tuple(list1)
print("\nTuple : ", tuple1)
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P2TupleWithDifferentDatatypes.py =====
*CREATING AND DISPLAYING A TUPLE WITH DIFFERENT DATATYPES*
Enter values (with different datatypes) for a tuple:
Enter integer value : 1
Enter integer value : 2
Enter string value : abc
Enter string value : xyz
Enter float value : 3.5
Enter float value : 9.1

Tuple : (1, 2, 'abc', 'xyz', 3.5, 9.1)
```

Figure 2: Creating and displaying a tuple with the values (with different data types) entered by the user

Question 3 –

Write a Python program to create a tuple with numbers and print one item.

CODE –

```
print("*CREATING A TUPLE WITH NUMBERS AND DISPLAYING ONE  
ITEM*")  
  
list1 = [ ]  
  
print("Enter values for a tuple: ")  
for i in range(5):  
    num = int(input("Enter value : "))  
    list1.append(num)  
  
tuple1 = tuple(list1)  
print("\nTuple : ", tuple1)  
  
i = int(input("Enter the index to access the element of a tuple: "))  
print("Number present at index '", i, "' in the tuple: ", tuple1[i])
```

OUTPUT –

```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P3NumberTuple.py =====  
*CREATING A TUPLE WITH NUMBERS AND DISPLAYING ONE ITEM*  
Enter values for a tuple:  
Enter value : 1  
Enter value : 5  
Enter value : 9  
Enter value : 3  
Enter value : 7  
  
Tuple : (1, 5, 9, 3, 7)  
Enter the index to access the element of a tuple: 2  
Number present at index ' 2 ' in the tuple: 9
```

Figure 3: Creating a tuple with user – entered numbers and displaying the element of a tuple using the index number entered by the user

Question 4 –

Write a Python program to convert a tuple to a string.

CODE –

```
print("*CONVERTING A TUPLE INTO A STRING*")

list1 = [ ]
str1 = ""

print("Enter values for a tuple: ")
for i in range(5):
    x = input("Enter value : ")
    list1.append(x)

tuple1 = tuple(list1)
print("\nTuple : ", tuple1)

for i in tuple1:
    str1 += i

print("Tuple converted to string : ", str1)
```

OUTPUT –

```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P4TupleToString.py =
*CONVERTING A TUPLE INTO A STRING*
Enter values for a tuple:
Enter value : H
Enter value : e
Enter value : l
Enter value : l
Enter value : o

Tuple : ('H', 'e', 'l', 'l', 'o')
Tuple converted to string : Hello
```

Figure 4: Converting a user – entered tuple into a string and displaying the string

Question 5 –

Write a Python program to add an item in a tuple.

CODE –

```
print("*ADDING AN ITEM TO A TUPLE*")

list1 = [ ]

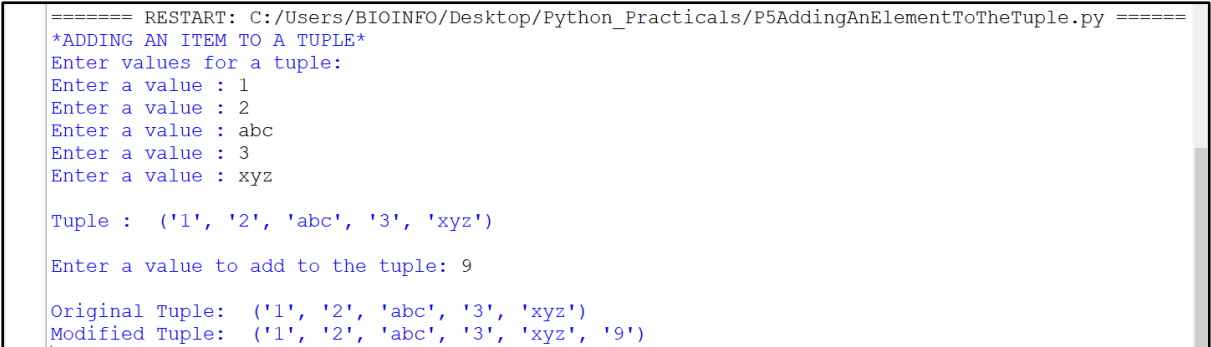
print("Enter values for a tuple: ")
for i in range(5):
    x = input("Enter a value : ")
    list1.append(x)

tuple1 = tuple(list1)
print("\nTuple : ", tuple1)

val = input("\nEnter a value to add to the tuple: ")
list1.append(val)
tuple2 = tuple(list1)

print("\nOriginal Tuple: ", tuple1)
print("Modified Tuple: ", tuple2)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P5AddingAnElementToTheTuple.py =====
*ADDING AN ITEM TO A TUPLE*
Enter values for a tuple:
Enter a value : 1
Enter a value : 2
Enter a value : abc
Enter a value : 3
Enter a value : xyz

Tuple : ('1', '2', 'abc', '3', 'xyz')

Enter a value to add to the tuple: 9

Original Tuple: ('1', '2', 'abc', '3', 'xyz')
Modified Tuple: ('1', '2', 'abc', '3', 'xyz', '9')
```

Figure 5: Adding an element to a user – entered tuple and displaying both, the original and the modified tuple

Question 6 –

Write a Python program to check whether an element exists within a tuple.

CODE –

```
print("*CHECKING THE EXISTENCE OF AN ELEMENT WITHIN A TUPLE*")

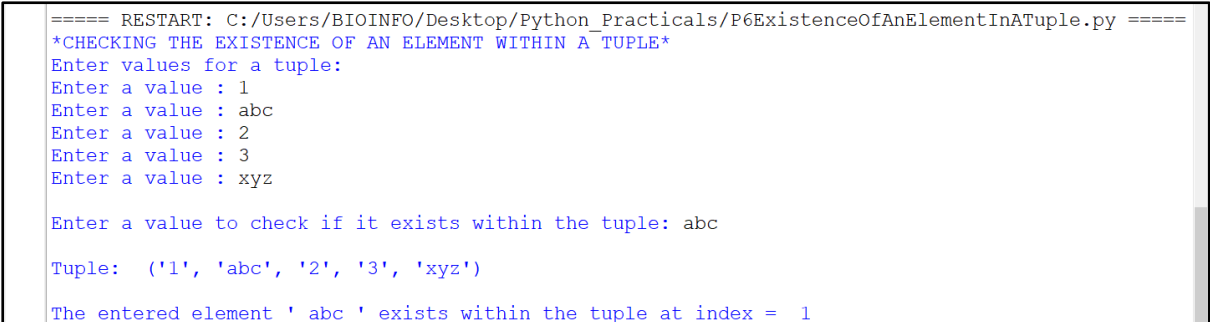
list1 = [ ]

print("Enter values for a tuple: ")
for i in range(5):
    x = input("Enter a value : ")
    list1.append(x)

tuple1 = tuple(list1)

element1 = input("\nEnter a value to check if it exists within the tuple: ")
print("\nTuple: ", tuple1)
if(element1 in tuple1):
    print("\nThe entered element '", element1, "' exists within the tuple at index = ",
    tuple1.index(element1))
else:
    print("The entered element '", element1, "' does not exist within the tuple")
```

OUTPUT –



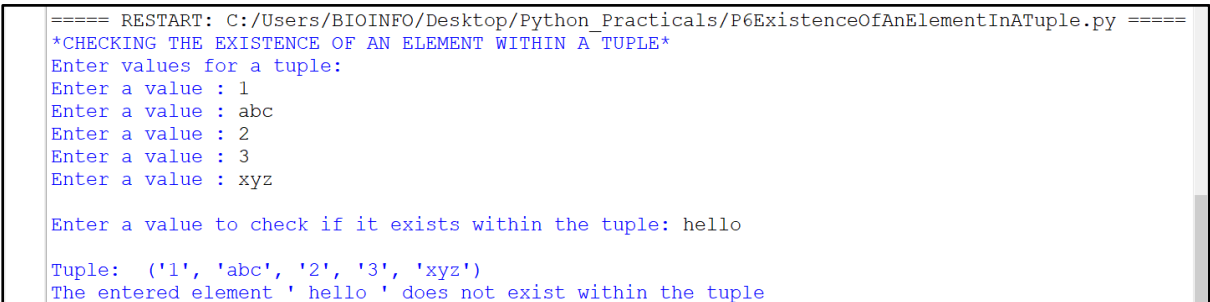
```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P6ExistenceOfAnElementInATuple.py =====
*CHECKING THE EXISTENCE OF AN ELEMENT WITHIN A TUPLE*
Enter values for a tuple:
Enter a value : 1
Enter a value : abc
Enter a value : 2
Enter a value : 3
Enter a value : xyz

Enter a value to check if it exists within the tuple: abc

Tuple:  ('1', 'abc', '2', '3', 'xyz')

The entered element ' abc ' exists within the tuple at index = 1
```

Figure 6: Displaying the output when an element exists in the user – entered tuple along with the index position where it is present



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P6ExistenceOfAnElementInATuple.py =====
*CHECKING THE EXISTENCE OF AN ELEMENT WITHIN A TUPLE*
Enter values for a tuple:
Enter a value : 1
Enter a value : abc
Enter a value : 2
Enter a value : 3
Enter a value : xyz

Enter a value to check if it exists within the tuple: hello

Tuple:  ('1', 'abc', '2', '3', 'xyz')

The entered element ' hello ' does not exist within the tuple
```

Figure 7: Displaying the output when an element does not exist in the user – entered tuple

Question 7 –

Write a Python program to slice a tuple.

CODE –

```
print("*SLICING A TUPLE*")

list1 = [ ]

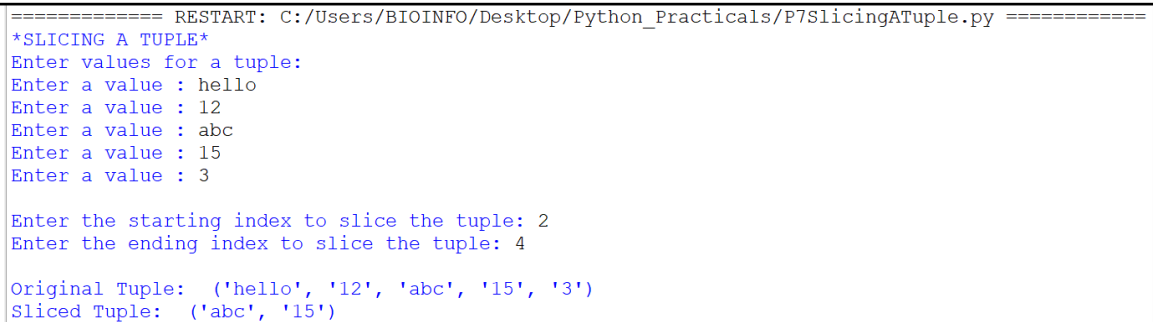
print("Enter values for a tuple: ")
for i in range(5):
    x = input("Enter a value : ")
    list1.append(x)

tuple1 = tuple(list1)

startpos = int(input("\nEnter the starting index to slice the tuple: "))
endpos = int(input("Enter the ending index to slice the tuple: "))
s_tuple1 = tuple1[startpos : endpos]

print("\nOriginal Tuple: ", tuple1)
print("Sliced Tuple: ", s_tuple1)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P7SlicingATuple.py =====
*SLICING A TUPLE*
Enter values for a tuple:
Enter a value : hello
Enter a value : 12
Enter a value : abc
Enter a value : 15
Enter a value : 3

Enter the starting index to slice the tuple: 2
Enter the ending index to slice the tuple: 4

Original Tuple: ('hello', '12', 'abc', '15', '3')
Sliced Tuple: ('abc', '15')
```

Figure 8: Displaying the original tuple and the sliced tuple post splicing using the start position and the end position entered by the user

Question 8 –

Write a Python script to store roll number of 5 students in tuple and perform the following:

1. Find length of a tuple.
2. Display the smallest element from a tuple.
3. Display the largest element from a tuple.

CODE –

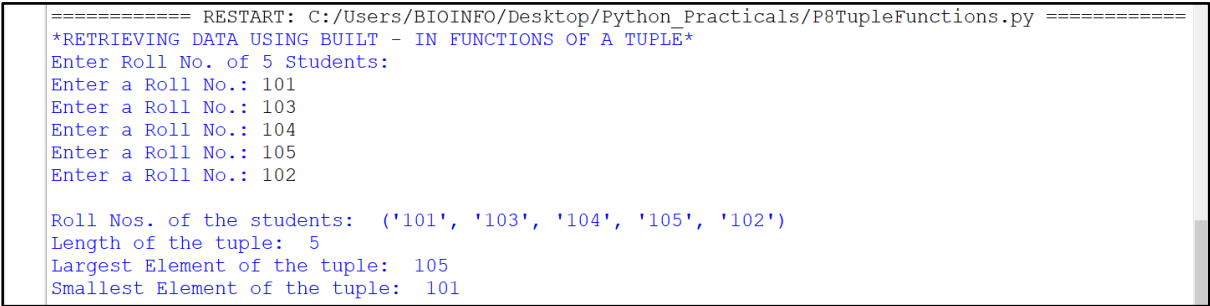
```
print("*RETRIEVING DATA USING BUILT - IN FUNCTIONS OF A TUPLE*")

list1 = [ ]

print("Enter Roll No. of 5 Students: ")
for i in range(5):
    x = input("Enter a Roll No.: ")
    list1.append(x)

tuple1 = tuple(list1)
print("\nRoll Nos. of the students: ", tuple1)
print("Length of the tuple: ", len(tuple1))
print("Largest Element of the tuple: ", max(tuple1))
print("Smallest Element of the tuple: ", min(tuple1))
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P8TupleFunctions.py =====
*RETRIEVING DATA USING BUILT - IN FUNCTIONS OF A TUPLE*
Enter Roll No. of 5 Students:
Enter a Roll No.: 101
Enter a Roll No.: 103
Enter a Roll No.: 104
Enter a Roll No.: 105
Enter a Roll No.: 102

Roll Nos. of the students: ('101', '103', '104', '105', '102')
Length of the tuple: 5
Largest Element of the tuple: 105
Smallest Element of the tuple: 101
```

Figure 9: Displaying the length, the largest element and the smallest element of a user – entered tuple consisting the roll numbers of 5 students

RESULTS:

The functionality of tuples was demonstrated using Python programming language by creating and executing python programs in the Python IDLE to create and display a tuple with the values entered by the user, to create and display a tuple with the values (with different data types) entered by the user, to create a tuple with user – entered numbers and display the element of a tuple using the index number entered by the user, to convert a user – entered tuple into a string and display the string, to add an element to a user – entered tuple and display both, the original and the modified tuple, to display the output whether an element exists in the user – entered tuple or not along with its index position (if it exists), to display the original tuple and the sliced tuple post splicing using the start position and the end position entered by the user and to display the length, the largest element and the smallest element of a user – entered tuple consisting the roll numbers of 5 students.

CONCLUSION:

The functionality of tuples was demonstrated using Python programming language.

REFERENCES:

1. GeeksforGeeks. (2023, September 6). Tuples in Python. GeeksforGeeks. <https://www.geeksforgeeks.org/tuples-in-python/>
 2. W3Schools.com. (n.d.). https://www.w3schools.com/Python/python_tuples.asp
 3. Python Tuples - javatpoint. (n.d.). [www.javatpoint.com. https://www.javatpoint.com/python-tuples](https://www.javatpoint.com/python-tuples)
-

Practical 4

PYTHON: Functions

AIM:

To demonstrate the utility of functions / methods using Python programming language.

INTRODUCTION:

Python is a popular, high – level programming language renowned for its ease of use, reusability and adaptability. It was created by Guido van Rossum and first released in 1991. Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

Python Functions

A function in Python is a reusable block of code that executes a specific task when called. Functions help in organizing code into manageable sections, making it modular and easier to maintain. They can take inputs (known as parameters or arguments) and may return outputs. Understanding both built-in and user-defined functions allows developers to leverage Python's capabilities effectively while also creating tailored solutions for their specific programming needs. Functions in Python are essential for writing clean and efficient code.

To define a function in Python, you use the **def** keyword, followed by the function name and parentheses containing any parameters. The function body follows, indented appropriately.

The return statement is used to exit a function and send back a value to the caller. If no return statement is provided, the function will return None by default.

Example:

```
def square(x):  
    return x * x  
print(square(4)) # Output: 16
```

Types of Functions

1. Built – In Functions

Built-in functions are pre-defined functions that come packaged with Python. They are ready to use without any additional code. Some commonly used built-in functions include:

Built – in Function	Description
print()	Outputs a message to the console
len()	Returns the length of an object like a string, list, or tuple
type()	Returns the data type of an object
range()	Creates a sequence of numbers between specified start and end points
max()	Returns the maximum value in a list or sequence
min()	Returns the minimum value in a list or sequence
sum()	Calculates the sum of a list or sequence

2. User – defined Functions

User-defined functions are functions created by the programmer to perform specific tasks. They help divide programs into smaller, reusable parts. To define a function in Python, the ‘def’ keyword is used, followed by the function name and parentheses [()] containing any parameters. The body of the function is indented.

User – defined functions can take zero or more arguments. They can return zero or more values. They help reduce code complexity, avoid repetition, increase reusability, and improve code clarity.

Syntax –	<code>def function_name(parameters): # Function body return [expression] # Optional return statement</code>
-----------------	---

Example – <code>def greet(): print("Hello, World!") # Calling the function greet()</code>	Output – Hello, World!
--	----------------------------------

Calling a Function

To execute a function, simply use its name followed by parentheses. If the function requires parameters, they should be included within the parentheses.

Example –

```
def add(num1, num2):  
    # Returns the sum of two numbers.  
    return num1 + num2  
  
# Calling the function with arguments  
result = add(5, 3)  
print(result) # Output: 8
```

Functions can accept various types of arguments –

- 1. Positional Arguments** – Passed in the order defined.
- 2. Keyword Arguments** – Passed by explicitly stating which parameter they correspond to.
- 3. Default Arguments** – Parameters that have default values if not provided.

Example of Default Arguments –

```
def greet(name="Guest"):  
    # Greet the user by name  
    print(f"Hello, {name}!")  
  
greet()      # Output: Hello, Guest!  
greet("Alice") # Output: Hello, Alice!
```

PROGRAMS

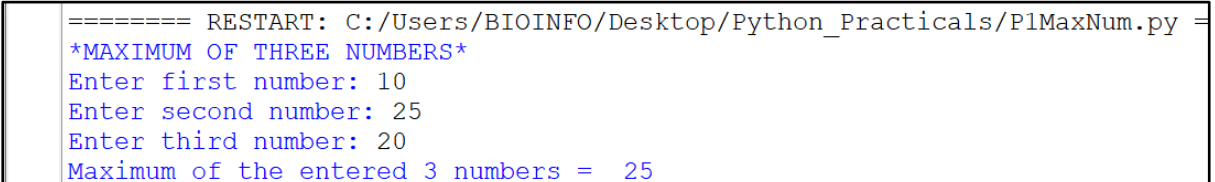
Question 1 –

Write a Python function to find the maximum of three numbers.

CODE –

```
print("*MAXIMUM OF THREE NUMBERS*")
def maximum(num1, num2, num3):
    print("Maximum of the entered 3 numbers = ", max(num1, num2, num3))
    return;
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
maximum(a, b, c)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P1MaxNum.py =
*MAXIMUM OF THREE NUMBERS*
Enter first number: 10
Enter second number: 25
Enter third number: 20
Maximum of the entered 3 numbers = 25
```

Figure 1: Displaying the maximum of the three user – entered numbers

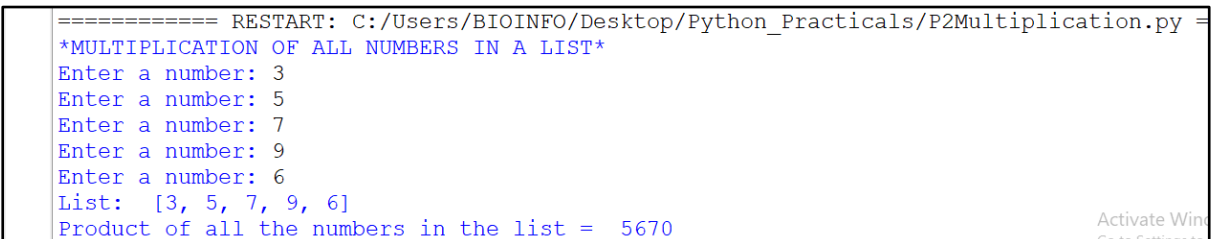
Question 2 –

Write a Python function to multiply all the numbers in a list.

CODE –

```
print("*MULTIPLICATION OF ALL NUMBERS IN A LIST*")
def mul(list1):
    product = 1
    for i in list1:
        product *= i
    return product;
l1 = [ ]
for i in range(5):
    num = int(input("Enter a number: "))
    l1.append(num)
print("List: ", l1)
print("Product of all the numbers in the list = ", mul(l1))
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P2Multiplication.py =
*MULTIPLICATION OF ALL NUMBERS IN A LIST*
Enter a number: 3
Enter a number: 5
Enter a number: 7
Enter a number: 9
Enter a number: 6
List:  [3, 5, 7, 9, 6]
Product of all the numbers in the list = 5670
```

Figure 2: Displaying the product of the multiplication of all the numbers in the user – entered list

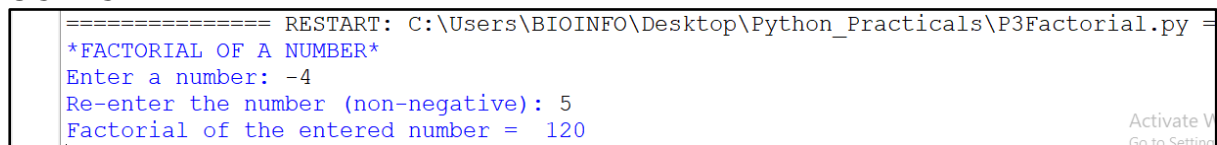
Question 3 –

Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.

CODE –

```
print("*FACTORIAL OF A NUMBER*")
def fact(num):
    factorial = 1
    i = 1
    if(num < 0):
        num = int(input("Re-enter the number (non-negative): "))
    while i <= num:
        factorial *= i
        i += 1
    return factorial;
num = int(input("Enter a number: "))
print("Factorial of the entered number = ", fact(num))
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P3Factorial.py =
*FACTORIAL OF A NUMBER*
Enter a number: -4
Re-enter the number (non-negative): 5
Factorial of the entered number = 120
Activate V
Go to Setting
```

Figure 3: Calculating and displaying the factorial of the user – entered number

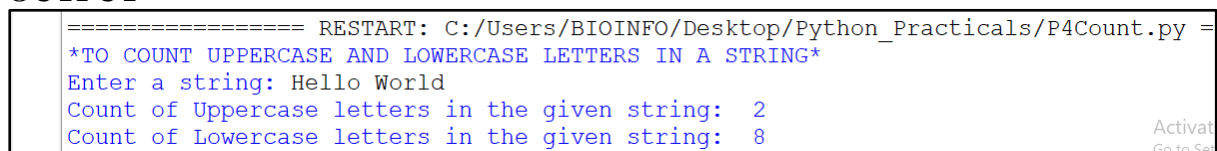
Question 4 –

Write a Python function that accepts a string and calculate the number of upper-case letters and lower-case letters.

CODE –

```
print("*TO COUNT UPPERCASE AND LOWERCASE LETTERS IN A STRING*")
def count(string1):
    count_u = 0
    count_l = 0
    for i in string1:
        if((ord(i) >= 65) and (ord(i) <= 90)):
            count_u += 1
        elif((ord(i) >= 97) and (ord(i) <= 122)):
            count_l += 1
    print("Count of Uppercase letters in the given string: ", count_u)
    print("Count of Lowercase letters in the given string: ", count_l)
string1 = input("Enter a string: ")
count(string1)
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P4Count.py =
*TO COUNT UPPERCASE AND LOWERCASE LETTERS IN A STRING*
Enter a string: Hello World
Count of Uppercase letters in the given string: 2
Count of Lowercase letters in the given string: 8
Activat
Go to Set
```

Figure 4: Displaying the total count of the uppercase and lowercase letters in the user – entered string

Question 5 –

Write a Python function that takes a list and returns a new list with unique elements of the first list.

CODE –

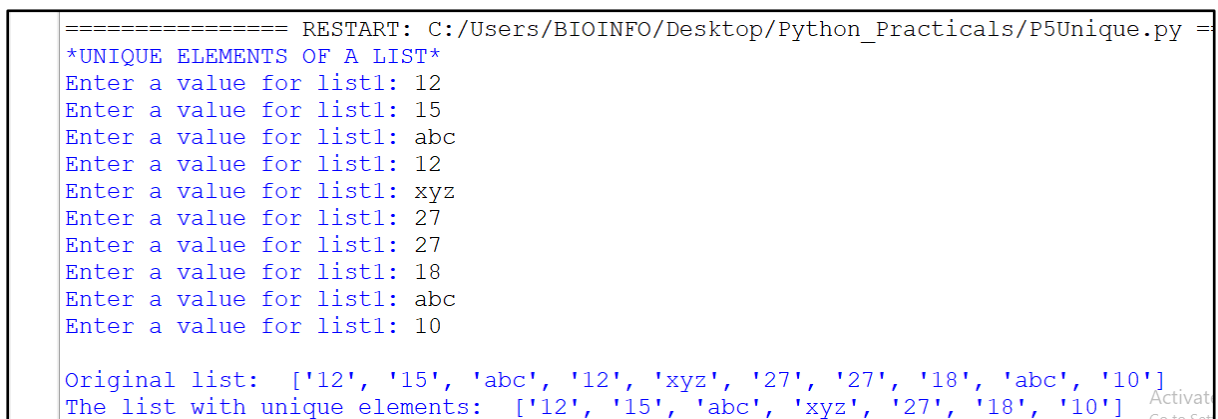
```
print("*UNIQUE ELEMENTS OF A LIST*")

def unique_element(list1):
    list2 = [ ]
    for i in list1:
        if i not in list2:
            list2.append(i)
    return list2

l1 = [ ]
for i in range(10):
    val = input("Enter a value for list1: ")
    l1.append(val)

print("\nOriginal list: ", l1)
print("The list with unique elements: ", unique_element(l1))
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P5Unique.py =
*UNIQUE ELEMENTS OF A LIST*
Enter a value for list1: 12
Enter a value for list1: 15
Enter a value for list1: abc
Enter a value for list1: 12
Enter a value for list1: xyz
Enter a value for list1: 27
Enter a value for list1: 27
Enter a value for list1: 18
Enter a value for list1: abc
Enter a value for list1: 10

Original list:  ['12', '15', 'abc', '12', 'xyz', '27', '27', '18', 'abc', '10']
The list with unique elements:  ['12', '15', 'abc', 'xyz', '27', '18', '10']
```

Figure 5: Displaying the new list consisting of only the unique elements from the original list

RESULTS:

The utility of functions / methods was demonstrated using Python programming language by creating and executing python programs in the Python IDLE to display the maximum of the three user – entered numbers, to display the product of the multiplication of all the numbers in the user – entered list, to calculate and display the factorial of the user – entered number, to display the total count of the uppercase and lowercase letters in the user – entered string and to display the new list consisting of only the unique elements from the original list.

CONCLUSION:

The utility of functions was demonstrated using Python programming language.

REFERENCES:

1. Functions in Python — Easy Python Docs 3.5 documentation. (n.d.). <https://www.easypythondocs.com/functions.html>
 2. GeeksforGeeks. (2024, July 29). Python Functions. GeeksforGeeks. <https://www.geeksforgeeks.org/python-functions/>
 3. Python - Functions. (n.d.). https://www.tutorialspoint.com/python/python_functions.htm
 4. W3Schools.com. (n.d.). https://www.w3schools.com/python/python_functions.asp
-

Practical 5

PYTHON: Class and Object – Oriented Programming (OOPs)

AIM:

To demonstrate the functionality of class and principles of object – oriented programming (OOPs) using Python programming language.

INTRODUCTION:

Python is a popular, high – level programming language renowned for its ease of use, reusability and adaptability. It was created by Guido van Rossum and first released in 1991. Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

Object – Oriented Programming

Object-oriented programming (OOP) in Python is a powerful paradigm that organizes software design around data, or objects, rather than functions and logic. This approach enables developers to create modular, maintainable, and scalable applications by modeling real-world entities.

Benefits of Object-Oriented Programming in Python

1. **Modularity:** Code can be organized into discrete classes.
2. **Reusability:** Classes can be reused across different programs.
3. **Scalability:** Applications can grow more easily by adding new classes.
4. **Maintainability:** Changes in one part of the code can often be made without affecting others.

Python's object-oriented programming capabilities provide a robust framework for building complex applications efficiently. By leveraging classes, inheritance, polymorphism, encapsulation, and abstraction, developers can create well-structured code that models real-world scenarios effectively.

Inheritance

Inheritance allows one class (the child or derived class) to inherit the attributes and methods of another class (the parent or base class). This promotes code reuse and establishes a hierarchical relationship between classes.

Inheritance in Python is a fundamental concept in object-oriented programming that allows one class (the child or derived class) to inherit attributes and methods from another class (the parent or base class). This mechanism promotes code reuse and establishes relationships between classes.

Type of Inheritance	Description	Syntax	Example
Single Inheritance	A derived class inherits from a single parent class.	<pre>class Parent: # Parent class methods and attributes class Child(Parent): # Child class methods and attributes</pre>	<pre>class Parent: def show(self): print("This is the parent class.") class Child(Parent): def display(self): print("This is the child class.") obj = Child() obj.show() # Output: This is the parent class. obj.display() # Output: This is the child class.</pre>
Multiple Inheritance	A derived class can inherit from more than one base class.	<pre>class Parent1: pass class Parent2: pass class Child (Parent1, Parent2): pass</pre>	<pre>class Mother: def mother(self): print("This is the mother.") class Father: def father(self): print("This is the father.") class Son(Mother, Father): def parents(self): print("Son's parents:") self.mother() self.father() s1 = Son() s1.parents() # Output: Son's parents: This is the mother. This is the father.</pre>

Multilevel Inheritance	A derived class inherits from another derived class, forming a chain of inheritance.	<pre> class Grandparent: pass class Parent(Grandparent): pass class Child(Parent): pass </pre>	<pre> class Grandparent: def show_grandparent(self): print("This is the grandparent.") class Parent(Grandparent): def show_parent(self): print("This is the parent.") class Child(Parent): def show_child(self): print("This is the child.") c = Child() c.show_grandparent() # Output: This is the grandparent. c.show_parent() # Output: This is the parent. c.show_child() # Output: This is the child. </pre>
Hierarchical Inheritance	Multiple derived classes inherit from a single base class.	<pre> class Parent: pass class Child1(Parent): pass class Child2(Parent): pass </pre>	<pre> class Animal: def speak(self): print("Animal speaks") class Dog(Animal): def bark(self): print("Dog barks") class Cat(Animal): def meow(self): print("Cat meows") dog = Dog() cat = Cat() dog.speak() # Output: Animal speaks dog.bark() # Output: Dog barks cat.speak() # Output: Animal speaks </pre>

			<pre>cat.meow() # Output: Cat meows</pre>
Hybrid Inheritance	<p>Hybrid inheritance combines two or more types of inheritance. It usually involves a mix of hierarchical and multiple inheritance.</p>	<pre>class Base1: pass class Base2: pass class Derived1(Base1): pass class Derived2(Base1, Base2): pass</pre>	<pre>class Vehicle: def type(self): print("This is a vehicle.") class Car(Vehicle): def wheels(self): print("Car has 4 wheels.") class Bike(Vehicle): def wheels(self): print("Bike has 2 wheels.") class ElectricCar(Car): # Hybrid with multiple inheritance as well def battery(self): print("Electric car has battery.") e_car = ElectricCar() e_car.type() # Output: This is a vehicle. e_car.wheels() # Output: Car has 4 wheels. e_car.battery() # Output: Electric car has battery.</pre>

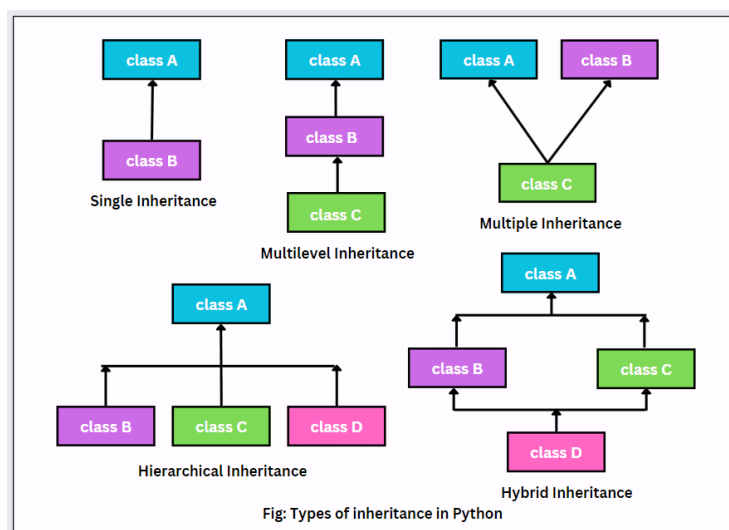


Figure 1: Types of Inheritance

Polymorphism

Polymorphism enables objects of different classes to be treated as objects of a common superclass. It allows methods to be defined in multiple forms, enhancing flexibility. Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. The term "polymorphism" means "many forms," and in programming, it refers to the ability to invoke methods or functions with the same name on different objects, enabling different behaviors based on the object's class.

Polymorphism enhances flexibility and integration in Python programming by allowing functions and methods to operate on objects of various types through a common interface. This capability not only simplifies code but also promotes code reuse and maintainability by enabling developers to write more generic and adaptable code structures.

Types of Polymorphism in Python

1. Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows the subclass to customize or extend the behavior of the inherited method.

Example:

```
class Dog():
    def speak(self):
        return "Woof!"
class Cat():
    def speak(self):
        return "Meow!"
dog = Dog()
dog.speak() # Output: Woof!
cat = Cat()
cat.speak()# Output: Meow!
```

2. Method Overloading

Although Python does not support method overloading in the traditional sense (as seen in languages like Java), it can be achieved using default arguments or variable-length arguments. This allows functions to behave differently based on the number or type of arguments passed.

Example:

```
def add(x, y, z=0):
    return x + y + z
print(add(2, 3))    # Output: 5
print(add(2, 3, 4)) # Output: 9
```

Here, the add function can take either two or three arguments.

Encapsulation

Encapsulation is the bundling of data (attributes) and methods that operate on the data into a single unit or class. It restricts direct access to some of an object's components, which can prevent the accidental modification of data.

Encapsulation is a core principle of object-oriented programming (OOP) that refers to the bundling of data (attributes) and methods (functions) that operate on that data into a single unit called a class. This concept not only organizes code but also protects the internal state of an object from unintended interference and misuse.

Data Hiding

Encapsulation allows for data hiding, where the internal representation of an object is hidden from the outside. This means that attributes of a class can be made private or protected, restricting direct access to them from outside the class.

Access Modifiers

Python uses naming conventions rather than strict access modifiers like some other programming languages (e.g., Java or C++). The common conventions are:

1. **Public:** Attributes and methods that can be accessed from outside the class.
2. **Protected:** Attributes and methods prefixed with a single underscore (_) are intended for internal use and should not be accessed directly from outside the class.
3. **Private:** Attributes and methods prefixed with a double underscore (__) are considered private and are subject to name mangling, making them harder to access from outside the class.

Abstraction

Abstraction involves hiding complex implementation details while exposing only the necessary parts of an object. This simplifies interaction with objects by providing a clear interface. Abstraction is a fundamental concept in object-oriented programming (OOP) that focuses on hiding the complex implementation details of a system while exposing only the essential features to the user. In Python, abstraction helps in designing modular and maintainable code by providing a clear separation between the interface and the implementation.

Purpose of Abstraction

1. **Simplification:** Abstraction reduces complexity by hiding unnecessary details, allowing users to interact with a simplified model of the system.
2. **Enhancement of Efficiency:** By focusing on high-level functionalities, developers can create more efficient applications without getting bogged down by implementation specifics.

Abstract Classes and Methods

In Python, abstraction is typically achieved through the use of abstract classes and methods. An abstract class serves as a blueprint for other classes and cannot be instantiated directly. It may contain abstract methods, which are methods declared without an implementation. Subclasses are required to implement these abstract methods.

Classes

A class is a blueprint for creating objects. It defines a set of attributes (data) and methods (functions) that the created objects will have. For example, a Car class might have attributes like color and model, and methods like drive() and stop().

Example:

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model
    def drive(self):
        print("The", self.color, self.model, "is driving.")
```

Objects

An object is an instance of a class. Each object can hold different values for its attributes. For instance, you can create multiple Car objects with different colors and models.

Example:

```
car1 = Car("red", "Toyota")
car2 = Car("blue", "Honda")
car1.drive() # Output: The red Toyota is driving.
```

Benefits of Using Abstraction

- 1. Code Reusability:** Abstract classes allow for code reuse by providing a common interface for different implementations.
- 2. Flexibility:** Changes in one part of the code (the implementation) do not affect other parts (the interface).
- 3. Maintainability:** Code is easier to maintain and understand since only relevant details are exposed.

PROGRAMS

Question 1 –

Write a Python class named Employee with three attributes Emp_ID, Emp_Name and Emp_Salary. Display the entire attribute and their values of the class.

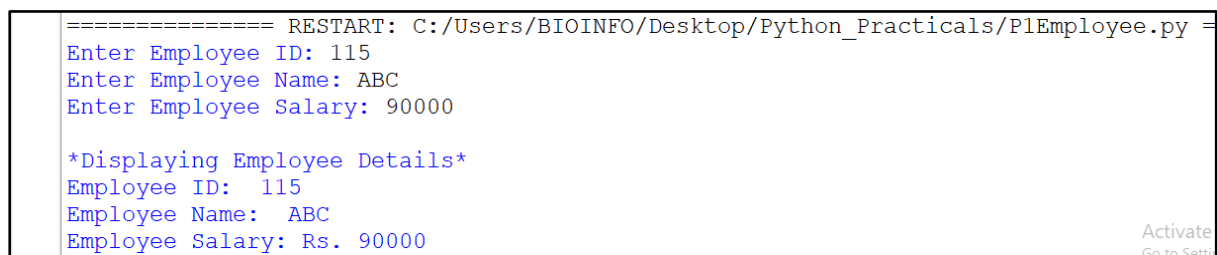
CODE –

```
class Employee:
    def __init__(self, emp_id, emp_name, emp_salary):
        self.emp_id = emp_id
        self.emp_name = emp_name
        self.emp_salary = emp_salary

    def display(self):
        print("\n*Displaying Employee Details*")
        print("Employee ID: ", self.emp_id)
        print("Employee Name: ", self.emp_name)
        print("Employee Salary: Rs.", self.emp_salary)

e_id = int(input("Enter Employee ID: "))
e_name = input("Enter Employee Name: ")
e_sal = int(input("Enter Employee Salary: "))
obj1 = Employee(e_id, e_name, e_sal)
obj1.display()
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/PlEmployee.py =
Enter Employee ID: 115
Enter Employee Name: ABC
Enter Employee Salary: 90000

*Displaying Employee Details*
Employee ID: 115
Employee Name: ABC
Employee Salary: Rs. 90000
```

Figure 2: Displaying the attributes Emp_ID, Emp_Name and Emp_Salary and their values (input taken from the user)

Question 2 –

Write a Python class to reverse a string word by word.

CODE –

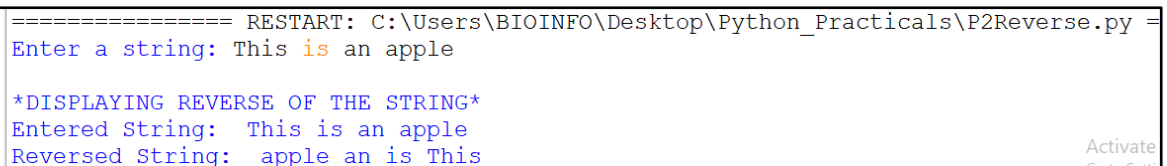
```
class Reverse:
    reverse_string = ""
    def __init__(self, string1):
        self.string1 = string1

    def rev(self):
        print("\n*DISPLAYING REVERSE OF THE STRING*")
        words = self.string1.split()
        self.reverse_string = " ".join(reversed(words))

    def display_rev(self):
        print("Entered String: ", self.string1)
        print("Reversed String: ", self.reverse_string)

s1 = input("Enter a string: ")
obj1 = Reverse(s1)
obj1.rev()
obj1.display_rev()
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P2Reverse.py =
Enter a string: This is an apple

*DISPLAYING REVERSE OF THE STRING*
Entered String:  This is an apple
Reversed String:  apple an is This
```

Figure 3: Displaying the reverse of the entered string

Question 3 –

Write a Python class which has two methods 'get_String' and 'print_String'. 'get_String' accepts a string from the user and print the string in upper case.

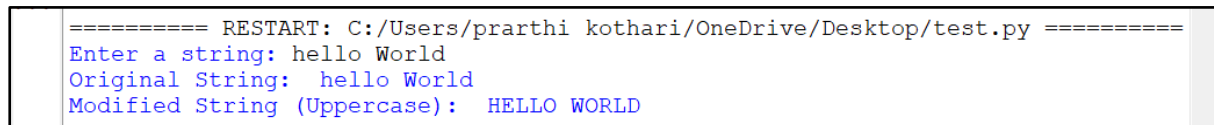
CODE –

```
class UpperString:
    def __init__(self, string=None):
        self.string = string

    def get_string(self):
        self.string = input("Enter a string: ")
        return self.string

    def print_string(self):
        u_str = self.string.upper()
        print("Original String: ", self.string)
        print("Modified String (Uppercase): ", u_str)

obj = UpperString()
obj.get_string()
obj.print_string()
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter a string: hello World
Original String: hello World
Modified String (Uppercase): HELLO WORLD
```

Figure 4: Displaying the string after converting it to uppercase

Question 4 –

Write a Python class named Circle constructed by a radius and two methods which will compute the area and the Circumference perimeter of a circle.

CODE –

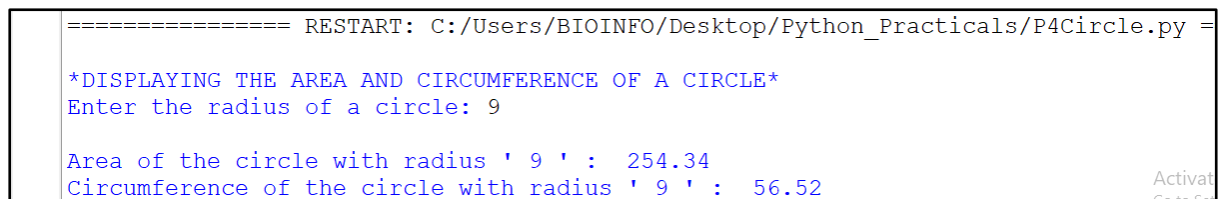
```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        print("\nArea of the circle with radius ", self.radius, " : ", (3.14 * (self.radius **
2)))

    def circumference(self):
        print("Circumference of the circle with radius ", self.radius, " : ", (2 * 3.14 *
self.radius))

print("\n*DISPLAYING THE AREA AND CIRCUMFERENCE OF A CIRCLE*")
radius = int(input("Enter the radius of a circle: "))
obj1 = Circle(radius)
obj1.area()
obj1.circumference()
```

OUTPUT –



```
===== RESTART: C:/Users/BIOINFO/Desktop/Python_Practicals/P4Circle.py =
*DISPLAYING THE AREA AND CIRCUMFERENCE OF A CIRCLE*
Enter the radius of a circle: 9
Area of the circle with radius ' 9 ' :  254.34
Circumference of the circle with radius ' 9 ' :  56.52
```

Figure 5: Displaying the area and the circumference of the circle with the radius entered by the user

Question 5 –

Create a class Area for calculating area of a square and rectangle using method overloading.

CODE –

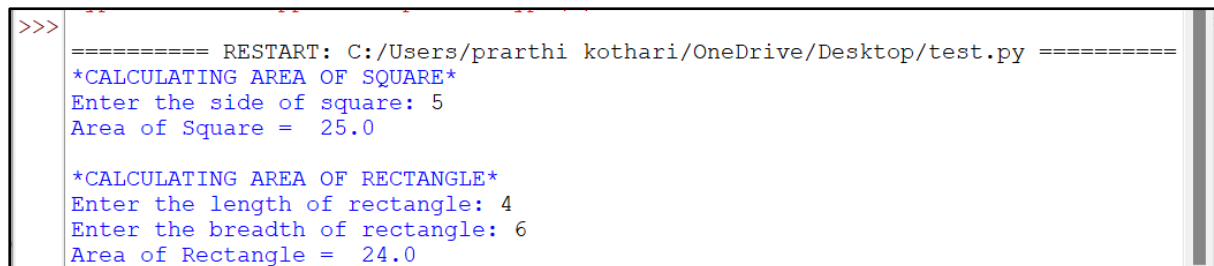
```
class Area:
    def calc_area(self, side=None, length=None, breadth=None):
        if side is not None:
            area = side * side
            print("Area of Square = ", area)
        elif length is not None and breadth is not None:
            area1 = length * breadth
            print("Area of Rectangle = ", area1)
        else:
            print("Invalid parameters provided.")

obj = Area()

print("*CALCULATING AREA OF SQUARE*")
side = float(input("Enter the side of square: "))
obj.calc_area(side=side)

print("\n*CALCULATING AREA OF RECTANGLE*")
length = float(input("Enter the length of rectangle: "))
breadth = float(input("Enter the breadth of rectangle: "))
obj.calc_area(length=length, breadth=breadth)
```

OUTPUT –



```
>>>
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
*CALCULATING AREA OF SQUARE*
Enter the side of square: 5
Area of Square = 25.0

*CALCULATING AREA OF RECTANGLE*
Enter the length of rectangle: 4
Enter the breadth of rectangle: 6
Area of Rectangle = 24.0
```

Figure 6: Calculating and displaying the area of a square and a rectangle using method overloading

Question 6 –

Create a class Sequence (seq_id, seq_name) and inherit in class RNA and DNA (use any attributes) using hierarchical inheritance.

CODE –

```
class Sequence:
    def __init__(self, seq_id, seq_name):
        self.seq_id = seq_id
        self.seq_name = seq_name
        self.seq = input("Enter a nucleotide sequence: ")

class RNA(Sequence):
    def check_seq(self):
        print("\nSequence ID: ", self.seq_id)
        print("Sequence Name: ", self.seq_name)
        print("The given sequence is an RNA Sequence.")

class DNA(Sequence):
    def check_seq(self):
        print("\nSequence ID: ", self.seq_id)
        print("Sequence Name: ", self.seq_name)
        print("The given sequence is a DNA Sequence.")

def validate_sequence(sequence, seq_type):
    if seq_type == "DNA":
        return all(base in ['A', 'T', 'G', 'C'] for base in sequence.upper())
    elif seq_type == "RNA":
        return all(base in ['A', 'U', 'G', 'C'] for base in sequence.upper())
    return False

seq_id = input("Enter sequence ID: ")
seq_name = input("Enter sequence name: ")
dna = DNA(seq_id, seq_name)

if validate_sequence(dna.seq, "DNA"):
    dna.check_seq()
else:
    rna = RNA(seq_id, seq_name)

    if validate_sequence(dna.seq, "RNA"):
        rna.check_seq()
    else:
        print("Not a valid nucleotide sequence.")
```


OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====  
Enter sequence ID: GNK115  
Enter sequence name: Sample  
Enter a nucleotide sequence: ATCGGCATCAG  
  
Sequence ID: GNK115  
Sequence Name: Sample  
The given sequence is a DNA Sequence.
```

Figure 7: Displaying the output when a DNA sequence has been entered

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====  
Enter sequence ID: GNK115  
Enter sequence name: Sample1  
Enter a nucleotide sequence: augcgauacg  
Enter a nucleotide sequence:  
  
Sequence ID: GNK115  
Sequence Name: Sample1  
The given sequence is an RNA Sequence.
```

Figure 8: Displaying the output when an RNA sequence has been entered

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====  
Enter sequence ID: GNK115  
Enter sequence name: Sample2  
Enter a nucleotide sequence: bciwncacbhwiwn  
Enter a nucleotide sequence:  
Not a valid nucleotide sequence.
```

Figure 9: Displaying the output when an invalid (non – nucleotide) sequence is entered

RESULTS:

The functionality of class and principles of object – oriented programming (OOPs) was demonstrated using Python programming language by creating and executing python programs in the Python IDLE to create and display the attributes Emp_ID, Emp_Name and Emp_Salary and their values (input taken from the user), to display the reverse of the entered string, to display the string after converting it to uppercase, to display the area and the circumference of the circle with the radius entered by the user, to calculate and display the area of a square and a rectangle using method overloading and to display whether the entered sequence is a nucleotide sequence (DNA / RNA) or not.

CONCLUSION:

The functionality of class and principles of object – oriented programming (OOPs) was demonstrated using Python programming language.

REFERENCES:

1. GeeksforGeeks. (2024, September 5). Python OOPs Concepts. GeeksforGeeks. <https://www.geeksforgeeks.org/python-oops-concepts/>
 2. OOPs Concepts in Python - javatpoint. (n.d.). www.javatpoint.com. <https://www.javatpoint.com/python-oops-concepts>
 3. GeeksforGeeks. (2022, July 7). Types of inheritance Python. GeeksforGeeks. <https://www.geeksforgeeks.org/types-of-inheritance-python/>
 4. W3Schools.com. (n.d.). https://www.w3schools.com/python/python_polymorphism.asp
-

Practical 6

PYTHON: Regular Expression

AIM:

To demonstrate the functionality of regular expressions using Python programming language.

INTRODUCTION:

Python is a popular, high – level programming language renowned for its ease of use, reusability and adaptability. It was created by Guido van Rossum and first released in 1991. Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

Regular Expressions

Regular expressions (regex or RegEx) are a powerful tool for matching patterns in strings. In Python, the re module provides a robust framework for working with regular expressions, allowing you to search, match, and manipulate strings based on specified patterns.

A regular expression is a sequence of characters that defines a search pattern. This pattern can be used for various string operations such as searching, replacing, or splitting strings. Regular expressions are widely used in data validation, parsing, and text processing.

Alternative			“cat mat” == “cat” or “mat” “python jython” == “python” or “jython”
Grouping	()	more than 1 pattern embedded in a single line	“gr (e a) y” == “grey” or “gray” “ra (mil n (ny el))” == “ramil” or “ranny” or “rael”
Quantification	?	0 or 1 of the preceding element	“rani?el” == “rael” or “rael”
	*	0 or more of the preceding element	“fo*ot” == “foot” or “fooooot” or “foooooot”
	+	1 or more of the preceding element	“too+fan” == “toofan” or (REMAINING)
	{m, n}	m to n times of the preceding element	“go{2, 3}gle” == “google” or “goooogle” “6{3}” == “666” “s{2, }” == “ss” or “sss” or “ssss”...
Anchors	^ (start / begin anchors)	Matches the starting position within the string	“^obje” == “object” or “object – oriented” “^2014” == “2014” or “2014/20/07”

	\$ (end operators)	Matches the ending position within the string	"gram\$" == "program" or "kilogram" "2014\$" == "20/07/2014" or "2013 – 2014"
Metacharacters	.	Matches any single character	"bat." == "bat" or "bats" or "bata" "87.1" == "8741" or "8751" or "8761"
	[]	Matches a single character that is contained within the brackets	"[xyz]" == "x" or "y" or "z" "[aeiou]" == any vowel "[0123456789]" == any digit
	[-]	Matches a single character that is contained within the brackets and the specified range	"[a – c]" == "a" or "b" or "c" "[a – z A – Z]" == all letters (both lowercase and uppercase) "[0 – 9]" == all digits
	[^]	Matches a single character that is not contained within the brackets	"[^aeiou]" == any non – vowel "[^0 – 9]" == any non – digit "[^xyz]" == any character but not "x" or "y" or "z"
Character class	\d	Matches a decimal digit	
	\D	Matches non – digits	
	\s	Matches any white space character	
	\S	Matches any non – white space character	
	\w	Matches alphanumeric character class	
	\W	Matches non – alphanumeric character class	
	\w+	Matches 1 or more words / characters	
	\A	Matches the beginning of the string	
	\Z	Matches the end of the string	
	\b	Returns a match where the specified characters are at the beginning or at the end of a word	
	\B	Returns a match where the specified characters are present, but not at the beginning or at the end of a word	

The re Module

To work with regular expressions in Python, you need to import the re module using:

import re

The search () function

- Scans through the input string and tries to match at any location
- Searches for the 1st occurrence of RE pattern within the string with optional flags

Syntax – re.search (pattern, string, flags = 0)

Modifier / Option Flags	Description
re.I	Performs case – insensitive matching
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w, \W, \b, \B)
re.M	Makes \$ match the end of a line (not just the end of the string) Makes ^ match the start of any line (not just the start of the string)
re.S	Makes a . (period / dot) match any character, including a new line
re.X	Allows comments in Regex

The “match object”

- Used for information about the matching strings
- “match object” instances also have several methods and attributes –

Method	Purpose
group ()	Return the string matched by the RE
start ()	Return the starting position of the match
end ()	Return the ending position of the match
span ()	Return a tuple containing the (start, end) positions of the match

Findall

- Returns all non – overlapping matches of pattern in string, as a list of strings
- The string is scanned left – to – right → matches are returned in the order found
- If 1 or more groups are present in the pattern → returns a list of groups → this will be a list containing each element as a tuple if the pattern has more than 1 group
- Empty matches are included in the result unless they touch the beginning of another match

Syntax – re.findall (pattern, strings, flags = 0)

The re.compile () method

- Combines a regular expression pattern into pattern objects, which can be used for pattern matching
- It also helps to search a pattern again without rewriting it

Syntax – re.compile (pattern)

Example	Output
import re pattern = re.compile ('TP') result = pattern.findall ('TP Tutorialspoint TP') print(result) result2 = pattern.findall ('TP is most popular tutorials site of India') print(result2)	['TP', 'TP'] ['TP']

Modifying strings

Regular expression is compiled into pattern objects, which have methods for various operators such as searching for pattern matches or performing string substitutions.

RE are also commonly used to modify strings in various ways using the following pattern methods –

Method	Purpose	Syntax	Example	Output
split ()	Split the string into a list, splitting it wherever the RE matches	re.split (string, [maxsplit])	txt = "hello, my name is Peter, I am 26 years old" x = txt.split(",") print (x)	['hello', 'my name is Peter', 'I am 26 years old']
			txt = "apple # banana # cherry # orange" x = txt.split("#", 1) print(x)	['apple', '# banana # cherry # orange']
sub ()	<p>Find all substrings where the RE matches and replace them with a different string.</p> <p>Replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided.</p> <p>Returns the modified string.</p>	re.sub (pattern, repl, string, max = 0)	<pre>import re DOB = "25-01-1991 # This is Date of Birth" # Delete Python – style commands Birth = re.sub (r '#.*\$', "", DOB) print ("Date of Birth : ", Birth) # Remove anything other than digits Birth1 = re.sub(r '\D', "", Birth) print ("Before substituting DOB : ", Birth1) # Substituting the ' - ' with '.' (dot) New = re.sub (r '\W', ".", Birth) print ("After substituting DOB : ", New)</pre>	
subn ()	Does the same thing as sub (), but returns the new string and the no. of replacements			

PROGRAMS

Question 1 –

Write a Python program to check that a string contains only a certain set of characters (in this case a-z, A-Z and 0-9).

CODE –

```
import re
print("*CHECKING FOR A PARTICULAR SET OF CHARACTERS IN A STRING*")
string1 = input("Enter a string : ")
print("\nEntered string: ", string1)
if re.match("[a-zA-Z0-9]+$", string1):
    print("String contains Valid characters")
else:
    print("String contains Invalid characters")
```

OUTPUT –

```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P1Characters.py =
*CHECKING FOR A PARTICULAR SET OF CHARACTERS IN A STRING*
Enter a string : hELLO World 1234$

Entered string:  hELLO World 1234$
String contains Invalid characters
```

Figure 1: Displaying the output when the user – entered string has invalid characters

```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P1Characters.py =
*CHECKING FOR A PARTICULAR SET OF CHARACTERS IN A STRING*
Enter a string : Apple123

Entered string:  Apple123
String contains Valid characters
```

Figure 2: Displaying the output when the user – entered string has valid characters

Question 2 –

Write a Python program to remove leading zeros from an IP address.

CODE –

```
import re
print("*REMOVING LEADING ZEROS FROM AN IP ADDRESS*")
ip = input("Enter an IP Address: ")
pattern1 = re.sub(r'^[0]+', '', ip)
print("\nModified IP Address (after removing leading 0s): ", pattern1)
pattern2 = re.sub(r'\.[0]*', '.', pattern1)
print("Final Modified IP Address: ", pattern2)
```

OUTPUT –

```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P2IPZero.py =
*REMOVING LEADING ZEROS FROM AN IP ADDRESS*
Enter an IP Address: 000021.025.031

Modified IP Address (after removing leading 0s):  21.025.031
Final Modified IP Address:  21.25.31
```

Figure 3: Displaying the modified IP address after removing the leading zeros

Question 3 –

Write a Python program to search the numbers (0-9) of length between 1 to 3 in a given string.

CODE –

```
import re
print("*CHECK LENGTH OF DIGITS*")
txt = input("Enter the digits: ")
x = re.search("^\\d{1,3}$", txt)
if x:
    print("\nThe entered digits are between the length 1 to 3")
else:
    print("\nThe entered digits are not between the length 1 to 3")
```

OUTPUT –

```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P3NumberSearch.py =
*CHECK LENGTH OF DIGITS*
Enter the digits: 123456789

The entered digits are not between the length 1 to 3
```

Figure 4: Displaying the output when the length of the entered digits is not between 1 to 3

```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P3NumberSearch.py =
*CHECK LENGTH OF DIGITS*
Enter the digits: 900

The entered digits are between the length 1 to 3
```

Figure 5: Displaying the output when the length of the entered digits is between 1 to 3

Question 4 –

Write a Python program to replace whitespaces with an underscore and vice versa.

CODE –

```
import re
def replace_spaces(text):
    text = re.sub(r' ', '__REPLACE__', text)
    text = re.sub(r'\s+', '_', text)
    text = re.sub(r'__REPLACE__', ' ', text)
    return text

txt = input("Enter the text: ")
mod_txt = replace_spaces(txt)
print("\nOriginal String:", txt)
print("Modified String:", mod_txt)
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter the text: Hello_world_this is_an apple

Original String: Hello_world_this is_an apple
Modified String: Hello world this_is an_apple
```

Figure 6: Modifying the string by replacing whitespaces with an underscore and vice versa and displaying both the original and modified

Question 5 –

Write a Python program to extract year, month and date from a URL.

CODE –

```
import re
def extract_date(url):
    return re.search(r'(\d{4})/(\d{1,2})/(\d{1,2})', url)
url = input("Enter URL: ")
match = extract_date(url)
if match:
    year, month, day = match.groups()
    print("Year: ", year, "\nMonth: ", month, "\nDay: ", day)
else:
    print("No date found")
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter URL: https://bion.com/2025/01/28
Year: 2025
Month: 01
Day: 28
```

Figure 7: Displaying the extracted year, month and date from a URL

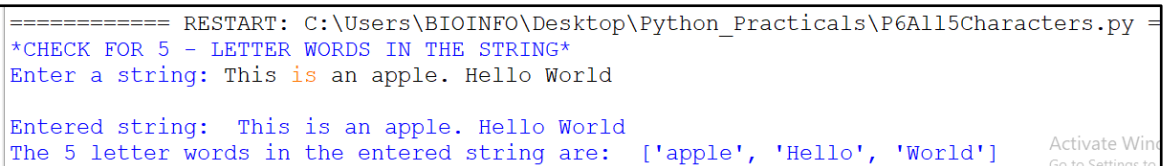
Question 6 –

Write a Python program to find all five characters long word in a string.

CODE –

```
import re
print("*CHECK FOR 5 - LETTER WORDS IN THE STRING*")
string1 = input("Enter a string: ")
matches = re.findall(r'\b\w{5}\b', string1)
print("\nEntered string: ", string1)
print("The 5 letter words in the entered string are: ", matches)
```

OUTPUT –



```
===== RESTART: C:\Users\BIOINFO\Desktop\Python_Practicals\P6All5Characters.py =
*CHECK FOR 5 - LETTER WORDS IN THE STRING*
Enter a string: This is an apple. Hello World

Entered string: This is an apple. Hello World
The 5 letter words in the entered string are: ['apple', 'Hello', 'World']
```

Figure 8: Displaying all the words from a string that are five – character long

Question 7 –

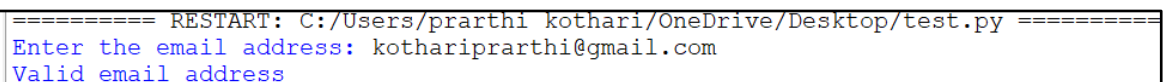
Write a Python regex code for email validation.

CODE –

```
import re
def validate_email(email):
    if re.match(r'^[a-zA-Z0-9_+]+@[a-zA-Z0-9]+\.[a-zA-Z0-9-]+\$', email):
        return "Valid email address"
    else:
        return "Invalid email address"

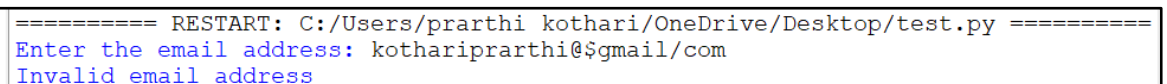
email_id = input("Enter the email address: ")
print(validate_email(email_id))
```

OUTPUT –



```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter the email address: kothariprarthi@gmail.com
Valid email address
```

Figure 9: Displaying the output when valid email address has been entered



```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter the email address: kothariprarthi@gmail/com
Invalid email address
```

Figure 10: Displaying the output when an invalid email address has been entered

RESULTS:

The functionality of regular expressions was demonstrated using Python programming language by creating and executing python programs in the Python IDLE to create and display the output when the user – entered string has either valid or invalid characters, to display the modified IP address after removing the leading zeros, to display the output when the length of the entered digits is either between or not between 1 to 3, to display the modified string after replacing whitespaces with an underscore and vice versa, to display the extracted year, month and date from a URL, to display all the words from a string that are five – character long and to display the output when a valid or invalid email address has been entered.

CONCLUSION:

The functionality of regular expressions was demonstrated using Python programming language.

REFERENCES:

1. GeeksforGeeks. (2024, September 5). Regular Expression (Regex) in Python with Examples. GeeksforGeeks. <https://www.geeksforgeeks.org/regular-expression-python-examples/>
 2. Python Regex (With Examples). (n.d.). <https://www.programiz.com/python-programming/regex>
 3. W3Schools.com. (n.d.). https://www.w3schools.com/python/python_regex.asp
-

Practical 7

PYTHON: Biopython

AIM:

To demonstrate the functionality of Biopython using Python programming IDLE.

INTRODUCTION:

Python is a popular, high – level programming language renowned for its ease of use, reusability and adaptability. It was created by Guido van Rossum and first released in 1991. Python executes code immediately without first requiring compilation since it is an interpreted language. Moreover, variables can store values of many data types without explicit declaration as it is dynamically typed.

Biopython

Biopython is an extensive library designed for biological computation, providing tools for handling DNA, RNA, and protein sequences, as well as interfacing with biological databases and software. Biopython is an open-source collection of modules tailored for biological computation. It encompasses a wide range of functionalities, including:

1. **Sequence Analysis:** Tools for working with DNA, RNA, and protein sequences.
2. **Sequence Alignments:** Methods for aligning biological sequences to identify similarities.
3. **Population Genetics:** Tools for analyzing genetic variation within populations.
4. **Molecular Structures:** Utilities to manage and analyze molecular structures.
5. **Database Interfaces:** Access to common biological databases like GenBank and tools such as BLAST.

Features of Biopython

1. **Interpreted and Interactive:** Biopython is built on Python, making it easy to use interactively.
2. **Object-Oriented:** It employs object-oriented programming (OOP) to organize data and methods efficiently.
3. **Support for Multiple Formats:** Handles various biological data formats such as FASTA, PDB, GenBank, and more.
4. **Access to Online Services:** Interfaces with NCBI services (e.g., BLAST, Entrez) and ExPASy services.

Biopython is a comprehensive toolkit that simplifies various tasks in bioinformatics, from sequence manipulation to database querying and analysis. Its modular design and extensive functionality make it an invaluable resource for researchers and developers working in computational biology. By leveraging its features—such as sequence objects, record handling, file I/O capabilities, and direct database access—users can efficiently conduct biological computations and analyses.

Core Components

1. Sequence Handling

Biopython uses the Seq class from the Bio.Seq module to represent biological sequences.

Example:

```
from Bio.Seq import Seq
seq = Seq("AGCT")
print(seq) # Output: AGCT
```

Alphabet Objects

The type of sequence is represented by an alphabet object (e.g., DNA or RNA). For instance:

```
from Bio.Alphabet import IUPAC
my_seq = Seq('AGTACACTGGT', IUPAC.unambiguous_dna)
```

2. Basic Operations on Sequences

- a. **Slicing:** seq returns 'A'.
- b. **Length:** len(seq) gives the length of the sequence.
- c. **Concatenation:** seq1 + seq2 combines two sequences.

3. Special Methods for Sequences

Function	Description	Syntax
Reverse Complement	The reverse complement of a DNA sequence is obtained by reversing the sequence and replacing each nucleotide with its complement (A ↔ T, C ↔ G).	my_seq.reverse_complement()
GC Content Calculation	The GC content of a DNA sequence is the percentage of nucleotides in the sequence that are either guanine (G) or cytosine (C). This can be calculated using the GC() function from the Bio.SeqUtils module.	from Bio.SeqUtils import GC GC(my_seq)
Transcription to RNA	Transcription is the process of converting DNA into RNA. In Biopython, this can be done using the transcribe() method on a Seq object.	rna_seq = my_seq.transcribe()
Translation to Protein	Translation is the process of converting an RNA sequence into a protein sequence. This is achieved using the translate() method on a Seq object. The translation considers stop codons, which are represented by an asterisk (*).	protein_seq = my_seq.translate()

4. SeqRecord Object

The SeqRecord object encapsulates a sequence along with its metadata (annotations). It includes attributes like .id, .description, and .features.

Example:

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

test_seq = Seq('GATC')
test_record = SeqRecord(test_seq, id='xyz', description='This is only a test')
print(test_record)
```

5. File Handling with SeqIO

The SeqIO module allows reading and writing sequence files in various formats (e.g., FASTA, FASTQ).

Reading a FASTA File:	<pre>from Bio import SeqIO gene = SeqIO.read("NC_005816.fna", "fasta") print(gene.seq)</pre>
Parsing Multiple Records:	<pre>handle = open("example.fasta", "r") seq_list = list(SeqIO.parse(handle, "fasta")) handle.close()</pre>

6. Direct Database Access with Entrez

Biopython provides modules to access online databases directly using the Entrez API.

Example of Fetching Data from NCBI:

```
from Bio import Entrez
Entrez.email = "your_email@example.com"
handle = Entrez.efetch(db="nucleotide", id="186972394", rettype="gb",
retmode="text")
record = SeqIO.read(handle, "genbank")
print(record)
```

7. Working with BLAST

Biopython facilitates interaction with BLAST databases for sequence alignment.

Example of Running a BLAST Query:

```
from Bio.Blast import NCBIWWW

query = SeqIO.read("test.fasta", format="fasta")
result_handle = NCBIWWW.qblast("blastn", "nt", query.seq)
with open("my_blast.xml", "w") as blast_file:
    blast_file.write(result_handle.read())
result_handle.close()
```

PROGRAMS

Question 1 –

Write a Biopython script to store DNA sequence using Seq class. Perform following operation:

- a. To get the second value in sequence
- b. To print the first two values
- c. To perform length and count number of 'g'
- d. Create second sequence and to add two sequences and display
- e. Turn a Seq object into a string

CODE –

```
from Bio.Seq import Seq

string1=input("Enter a DNA Sequence: ")
seq1=Seq(string1)

print ("Second value in sequence : ", seq1[1])
print("First 2 values of the sequence: ", seq1[0:2])
print ("Length of the sequence : ", len(seq1))
print ("Count of 'g' in the sequence: ", seq1.count('g'))

seq2=input("\nEnter a new DNA Sequence:")
seq21=Seq(seq2)
seq3=seq1+seq21

print("First sequence: ", seq1)
print("Second sequence: ", seq21)
print("Combined sequence: ", seq3)

seq_to_string=str(seq1)
print ("\nConverted string: ",seq_to_string)
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter a DNA Sequence: agctgactca
Second value in sequence :  g
First 2 values of the sequence:  ag
Length of the sequence :  10
Count of 'g' in the sequence:  2

Enter a new DNA Sequence:tagctgtta
First sequence:  agctgactca
Second sequence:  tagctgtta
Combined sequence:  agctgactcatagctgtta

Converted string:  agctgactca
```

Figure 1: Displaying the output after retrieving the second value, first two values in the sequence, length of the sequence and count number of 'g', combining two sequences and converting a Seq object into a string

Question 2 –

Write a Biopython script to store DNA sequence. Perform following operation:

- Find reverse complement
- Calculate GC percentage in DNA sequence
- Convert DNA to RNA
- Convert DNA to protein

CODE –

```
from Bio.Seq import Seq, translate, transcribe
from Bio.SeqUtils import gc_fraction

s=input("Enter a DNA Sequence: ")
seq=Seq(s)

print("\nOriginal Sequence: ", seq)
print("Reverse Complement: ", seq.reverse_complement())
print("GC Percentage:", 100*gc_fraction(seq),"%")
print("DNA to RNA: ", transcribe(seq))
print("RNA to Protein: ",translate(seq))
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter a DNA Sequence: atgccatgtcaa

Original Sequence:  atgccatgtcaa
Reverse Complement:  ttgacatggcat
GC Percentage: 41.66666666666667 %
DNA to RNA:  augccaugucaa
RNA to Protein:  MPCQ
```

Figure 2: Displaying the output of the reverse complement, GC percentage of a DNA sequence, converting the DNA to RNA and DNA to protein

Question 3 –

Write a Biopython script to store protein sequence and calculate molecular weight.

CODE –

```
from Bio.Seq import Seq
from Bio.SeqUtils import molecular_weight
seq1 = input("Enter a sequence: ")
seq2 = Seq(seq1)
print("The molecular weight of the entered sequence = ", molecular_weight(seq2))
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter a sequence: atgccatgtcatcga
The molecular weight of the entered sequence = 4631.959
```

Figure 3: Displaying the molecular weight of the entered sequence

Question 4 –

Write a Biopython script to create sequence record using sequence record by adding parameters like seq.id, Name, description and display the whole sequence record and also access only sequence id and sequence.

CODE –

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
seq1 = input("Enter a Sequence: ")
sequence = Seq(seq1)
record = SeqRecord(sequence,id='P00115', name="Example Protein",
description="Protein sequence")
print("\n*SEQUENCE RECORD*")
print("Sequence record is: ",record)
print("Sequence ID is: ",record.id)
print("Sequence name is: ",record.name)
```

OUTPUT –

```
===== RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py =====
Enter a Sequence: MPHJMKPN

*SEQUENCE RECORD*
Sequence record is: ID: P00115
Name: Example Protein
Description: Protein sequence
Number of features: 0
Seq('MPHJMKPN')
Sequence ID is: P00115
Sequence name is: Example Protein
```

Figure 4: Displaying the complete sequence record of a user – entered sequence

Question 5 –

Write a Biopython script to read sequence which is downloaded from genbank i.e fasta file and save named as NC_005816.fna. Perform the following:

- Display id and seq
- Find reverse complement
- Calculate GC percentage in DNA sequence
- Convert DNA to RNA
- Convert DNA to protein

CODE –

```
from Bio import SeqIO
from Bio.Seq import Seq, transcribe, translate
from Bio.SeqUtils import gc_fraction
gene = SeqIO.read("C:\\Users\\prarthi kothari\\Downloads\\NC_005816.fna", "fasta")
print("ID: ", gene.id)
print("Sequence: ", gene.seq)
print("Reverse Complement: ", gene.seq.reverse_complement())
print("GC%: ", (100 * gc_fraction(gene.seq)))
print("RNA: ", transcribe(gene.seq))
print("Protein: ", translate(gene.seq))
```

OUTPUT –

[illegible]

Figure 5: Displaying the ID, sequence, the reverse complement, GC percentage of the provided sequence (NC_005816.fna) and displaying the output after converting it to RNA and protein

Question 6 –

Write a Biopython script to access sequence record from genbank directly.

CODE –

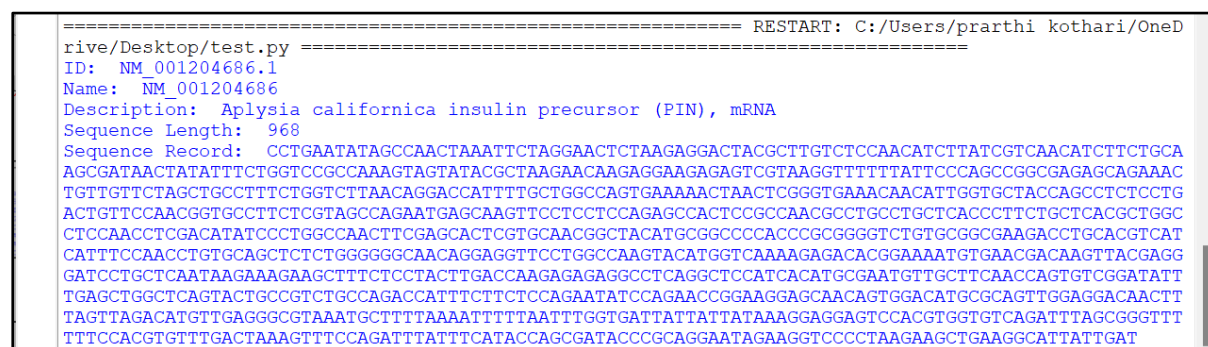
```
from Bio import Entrez
from Bio import SeqIO

Entrez.email = "email@gmail.com"
accession_number = "NM_001204686.1"
handle = Entrez.efetch(db="nucleotide", id=accession_number, rettype="gb",
retmode="text")
record = SeqIO.read(handle, "genbank")

print("ID: ", record.id)
print("Name: ", record.name)
print("Description: ", record.description)
print("Sequence Length: ", len(record.seq))
print("Sequence Record: ", record.seq)

handle.close()
```

OUTPUT –



```
restart: C:/Users/prarthi kothari/OneDrive/Desktop/test.py
ID: NM_001204686.1
Name: NM_001204686
Description: Aplysia californica insulin precursor (PIN), mRNA
Sequence Length: 968
Sequence Record: CCTGAATATAGCCAACATAAATCTAGGAACCTAAGAGGACTACGCTTGTCTCCAACATCTTATCGTCAACATCTTCTGCA
AGCGATAACTATATTTCTGGTCCGCCAAAGTAGTATACGCTAAGAACAAGAGGAAGAGAGTCGTAAGGTTTTTATCCAGCCGGCGAGAGCAGAAAC
TGTTGTTCTAGCTGCCTTTCTGGTCTTAACAGGACCATTTTGTGGCCAGTGAAAACTAACTCGGGTGAAACAACATTGGTGCTACCGCCTCTCCTG
ACTGTTCCAACGGTGCCCTTCTCGTAGCCAGATGAGCAAGTTCCCTCCAGAGCCACTCCGCCAACGCCTGCCTGCTCACCCCTTCTGCTCACGCTGGC
CTCCAACCTCGACATATCCCTGGCCAACTTCGAGCACTCGTGCAACGGCTACATGCGGCCCCACCGCGGGTCTGTGCGGCGAAGACCTGCACGTCAT
CATTCCAACCTGTGCAGCTCTCTGGGGGGCAACAGGAGGTTCTTGCCCAAGTACATGGTCAAAAGAGACACGGAAAATGTGAACGACAAGTTACGAGG
GATCCTGCTCAATAAGAAAGAAGCTTTCTCTACTTGACCAAGAGAGAGGCCTCAGGCTCCATCACATGCGAATGTTGCTTCAACCAAGTGTGCGATATT
TGAGCTGGCTCAGTACTGCCGCTCTGCCAGACCATTTCTCTCCAGAATATCCAGAACCAGGAGGCAACAGTGGACATGCGCAGTTGGAGGACAACCTT
TAGTTAGACATGTTGAGGGCGTAAATGCTTTTAAATTTTAAATTTGGTGATTATTATTATAAAGGAGGAGTCCACGTGGTGTGAGATTAGCGGGTTT
TTCCACGTGTTTGACTAAAGTTTCAGATTATTTATACAGCGATACCCGAGGAATAGAAGGTCCCTAAGAAGCTGAAGGCATTATTGAT
```

Figure 6: Displaying the sequence record directly from the GenBank database

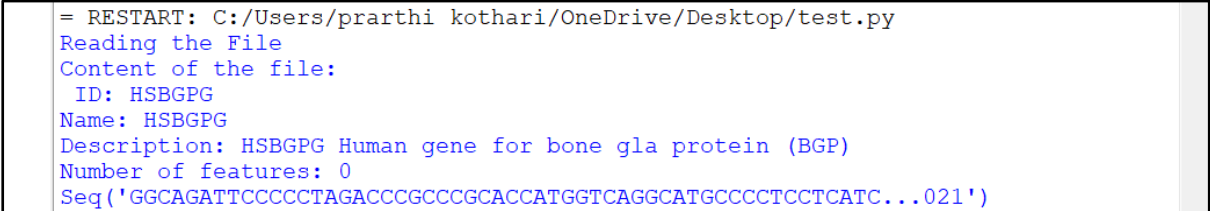
Question 7 –

Write a Biopython script to any read fasta file query using sequence I/O and run that query using blast with any blast algorithm. Save the output file in xml format.

CODE –

```
from Bio import SeqIO
from Bio.Blast import NCBIWWW,NCBIXML
query=SeqIO.read("C:\\Users\\prarthi kothari\\Downloads\\test.fasta",format="fasta")
print("Reading of the File")
result_handle=NCBIWWW.qblast("blastn","nt",query.seq)
print("Content of file...\n",query)
blast_file=open("C:\\Users\\BIOINFO\\Downloads\\test.fasta","w")
print("Creating a blast file in XML format!")
blast_file.write(result_handle.read())
```

OUTPUT –



```
= RESTART: C:/Users/prarthi kothari/OneDrive/Desktop/test.py
Reading the File
Content of the file:
  ID: HSBGPG
Name: HSBGPG
Description: HSBGPG Human gene for bone gla protein (BGP)
Number of features: 0
Seq('GGCAGATTCCCCCTAGACCCGCCCGCACCATGGTCAGGCATGCCCTCCTCATC...021')
```

Figure 7: Displaying the output after running the query for BLAST

RESULTS:

The functionality of Biopython was demonstrated using Python programming language by creating and executing python programs in the Python IDLE to create and display the output after retrieving the second value, first two values in the sequence, length of the sequence and count number of 'g', combining two sequences and converting a Seq object into a string, to display the output of the reverse complement, GC percentage of a DNA sequence, converting the DNA to RNA and DNA to protein, to display the molecular weight of the entered sequence, to display the complete sequence record of a user – entered sequence, to display the ID, sequence, the reverse complement, GC percentage of the provided sequence (NC_005816.fna) and displaying the output after converting it to RNA and protein, to display the sequence record directly from the GenBank database and to display the output after running the query for BLAST.

CONCLUSION:

The functionality of Biopython was demonstrated using Python programming language.

REFERENCES:

1. Biopython - Introduction. (n.d.).
https://www.tutorialspoint.com/biopython/biopython_introduction.htm
 2. Chapman, B., & Chang, J. (2000). Biopython: Python tools for computation biology.
<http://biopython.org/DIST/docs/acm/ACMbiopy.pdf>
 3. GeeksforGeeks. (2020, October 1). Introduction to Biopython. GeeksforGeeks.
<https://www.geeksforgeeks.org/introduction-to-biopython/>
-

Practical 8
Machine Learning

AIM:

To predicting drug response and disease classification using genomic and protein data

INTRODUCTION:

Machine Learning (ML) is a branch of artificial intelligence (AI) that enables systems to learn from data, identify patterns, and make decisions with minimal human intervention. By leveraging algorithms and statistical models, ML allows computers to improve their performance on tasks over time without being explicitly programmed to do so. From recommendation engines and speech recognition to medical diagnostics and autonomous vehicles, ML is at the heart of many advanced technologies transforming industries today.

Types of ML Training

ML models are trained using different approaches depending on the availability and type of data. The three primary types of ML training are:

1. Supervised Learning

- In supervised learning, the model is trained on a labeled dataset, meaning each input comes with a corresponding output (or label). The goal is for the model to learn the relationship between the inputs and outputs to make predictions on new, unseen data.
- **Applications:** Image classification, spam detection, medical diagnosis.
- **Algorithms:** Linear regression, decision trees, support vector machines (SVM), neural networks.

2. Unsupervised Learning

- In unsupervised learning, the model is given an unlabeled dataset, meaning it only has access to input data without any predefined output. The objective is to uncover hidden patterns, relationships, or structures within the data.
- **Applications:** Market segmentation, anomaly detection, clustering.
- **Algorithms:** K-means clustering, principal component analysis (PCA), autoencoders.

3. Reinforcement Learning

- In reinforcement learning, an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. The goal is for the agent to maximize cumulative rewards over time by learning an optimal strategy or policy.
- **Applications:** Game playing (e.g., AlphaGo), robotics, autonomous driving.
- **Algorithms:** Q-learning, deep Q-networks (DQN), policy gradients.

Each of these training methods has its own strengths and is suited to different kinds of problems, depending on the availability of labeled data, the complexity of the task, and the desired outcomes.

Python libraries used for Machine Learning:

Python offers a robust ecosystem for Machine Learning (ML) and Artificial Intelligence (AI), with key libraries that streamline various tasks:

1. **NumPy:** Multi-dimensional arrays and high-performance mathematical operations for numerical algorithms.
2. **Pandas:** Simplifies data manipulation with high-level data structures like DataFrames.
3. **Matplotlib:** Flexible plotting library for static, animated, and interactive visualizations.
4. **Seaborn:** Enhances statistical data visualization, built on top of Matplotlib.
5. **Scikit-learn:** Comprehensive tools for ML, covering classification, regression, clustering, and more.
6. **TensorFlow, Keras, and PyTorch:** Advanced deep learning frameworks for neural networks, image recognition, and NLP.

Together, these libraries enable efficient data manipulation, visualization, and model building in Python.

NumPy

NumPy (Numerical Python) is one of the fundamental Python libraries for numerical computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. It serves as the foundation for many other scientific computing libraries like SciPy, Pandas, and Scikit-learn, making it a cornerstone in the Python ecosystem for Machine Learning (ML) and data science.

Key Features of NumPy:

1. Multi-dimensional Arrays (ndarray)

The core feature of NumPy is its powerful N-dimensional array object, `ndarray`. These arrays allow for the storage of large datasets in a grid-like structure and can have multiple dimensions, making it easy to represent vectors, matrices, or higher-dimensional tensors.

2. Broadcasting

NumPy allows broadcasting, which is a way of performing element-wise operations on arrays of different shapes without having to duplicate data. This is particularly useful for performing arithmetic operations on arrays of unequal dimensions efficiently.

3. High-performance Mathematical Operations

NumPy provides optimized, precompiled functions for a wide range of mathematical operations, including element-wise addition, subtraction, multiplication, division, and trigonometric functions. It leverages low-level languages like C and Fortran for performance, making it faster than native Python loops.

4. Linear Algebra and Random Number Generation

NumPy offers comprehensive support for linear algebra operations such as matrix multiplication, inversion, eigenvalue computation, and decomposition. Additionally, it includes a random number generation module for creating arrays of random numbers for stochastic processes or simulations.

5. Indexing and Slicing

NumPy arrays can be sliced and indexed in multiple ways, providing flexible access to array elements. Users can slice arrays using traditional indexing as well as boolean indexing, allowing for fine-grained control over data selection and manipulation.

6. Memory Efficiency

NumPy arrays are more memory-efficient compared to Python's built-in lists because they store elements in contiguous memory blocks and use fixed data types, reducing overhead and improving performance.

7. Integration with Other Libraries

Since many other libraries like Pandas, Scikit-learn, and TensorFlow rely on NumPy, it integrates seamlessly into the broader data science and ML ecosystem. NumPy arrays are often used as input to these libraries, making it a foundational tool.

These features make NumPy indispensable for efficient numerical computing and a must-have library for anyone working in data science, ML, or scientific computing.

Matplotlib

Matplotlib is a comprehensive and flexible Python library used for creating static, animated, and interactive visualizations. It is one of the most widely used data visualization libraries in Python, serving as the foundation for other libraries such as Seaborn and Pandas' plotting capabilities. Matplotlib allows users to generate a wide variety of plots and charts, making it essential for exploratory data analysis (EDA), scientific visualization, and the communication of results in data science and machine learning.

Key Features of Matplotlib:

1. Flexible Plotting Capabilities

Matplotlib offers a vast range of plots, including line plots, bar charts, histograms, scatter plots, pie charts, 3D plots, and more. This flexibility allows users to choose the best visualization to represent their data effectively. Custom plots and complex visualizations can also be created by combining different plot types.

2. Object-Oriented API and Pyplot Interface

Matplotlib provides two main ways to create plots:

- a. **Pyplot Interface:** A state-based interface that mimics MATLAB's plotting style, making it intuitive for users to quickly generate plots with a few lines of code.
- b. **Object-Oriented API:** Allows for more control and customization of plots by directly manipulating figure objects, axes, and properties. This method is highly useful when creating complex or multi-plot figures.

3. Customization and Control

One of the most significant strengths of Matplotlib is the degree of customization it offers. Users can control every aspect of a plot, including colors, labels, line styles, markers, titles, fonts, legends, gridlines, and much more. This granular control allows for the creation of publication-quality plots and figures.

4. Interactive Plots

Matplotlib supports interactive visualizations when integrated with IPython or Jupyter Notebooks, allowing users to zoom, pan, and modify plots in real time. This is particularly useful in exploratory data analysis, where immediate feedback from the data is required.

5. Subplots and Figure Layouts

The library allows for the creation of complex layouts with multiple subplots and figures. It includes functions to adjust the size, spacing, and alignment of multiple plots within a single figure, enabling users to compare different datasets or variables side by side.

6. Integration with Other Libraries

Matplotlib integrates seamlessly with other Python libraries like NumPy, Pandas, and Seaborn. For instance, Pandas can directly call Matplotlib's plotting functions, enabling

easy plotting of DataFrames. Seaborn also extends Matplotlib's functionality, providing more advanced statistical visualizations.

7. 3D Plotting and Animations

Matplotlib supports 3D plotting using the mplot3d module, enabling the visualization of three-dimensional data. Additionally, users can create animated plots using the FuncAnimation class, which is useful for visualizing time-series data or dynamic systems.

8. Output in Various Formats

Matplotlib allows saving plots in numerous formats such as PNG, JPEG, SVG, and PDF, making it convenient for embedding plots in websites, research papers, presentations, or other forms of documentation. It also provides fine control over DPI (dots per inch) and figure size for high-quality output.

These features make Matplotlib a versatile and powerful tool for visualizing data, providing users with extensive customization options while integrating smoothly with other Python libraries in the data science ecosystem.

Seaborn

Seaborn is a high-level Python data visualization library built on top of Matplotlib. It is designed to make it easier to create informative and aesthetically pleasing statistical plots. Seaborn integrates closely with Pandas DataFrames, allowing for quick and efficient visualization of data, especially when dealing with statistical relationships. It simplifies the process of creating complex visualizations with just a few lines of code and adds a layer of abstraction over Matplotlib, making the syntax more intuitive and the plots more visually attractive.

Key Features of Seaborn:

1. Built-in Themes and Color Palettes

Seaborn comes with several built-in themes and color palettes to improve the visual appeal of plots. This aesthetics make it easy to create publication-quality visualizations without needing much customization. The color palettes are designed for both categorical and continuous data, providing visually balanced and distinct colors.

2. Statistical Plotting Functions

Seaborn is specifically designed for creating statistical plots and offers various functions to visualize distributions and relationships. Key plot types include:

- a. **Distribution plots:** `distplot()`, `kdeplot()`, and `rugplot()` for visualizing data distributions.
- b. **Categorical plots:** `boxplot()`, `violinplot()`, `barplot()`, and `stripplot()` for exploring the relationship between categorical variables.
- c. **Relational plots:** `scatterplot()` and `lineplot()` for showing relationships between numerical variables.

3. Faceted and Multi-plot Grids

One of Seaborn's standout features is its ability to create multi-plot grids, allowing users to visualize different subsets of their data simultaneously. Functions like `FacetGrid` and `pairplot()` allow for the easy creation of complex multi-plot visualizations. These are particularly useful when exploring multi-dimensional datasets, as they help in identifying patterns and relationships across different subsets of data.

4. Easy Integration with Pandas DataFrames

Seaborn works seamlessly with Pandas, making it easy to visualize data stored in DataFrames. Many of Seaborn's plotting functions allow users to directly pass DataFrames and specify columns to be used as axes or variables, making data visualization much more straightforward and reducing the need for manual data wrangling.

5. Automatic Estimation and Plotting of Statistical Relationships

Seaborn has built-in functions to estimate and plot statistical relationships between variables, such as regression lines, confidence intervals, and error bars. For instance,

lmpplot() allows for the easy plotting of linear regression models, and regplot() provides tools for visualizing simple regression relationships with confidence intervals.

6. High-level Abstraction for Complex Visualizations

Seaborn abstracts away many of the complexities of Matplotlib, providing high-level functions for creating complex visualizations with minimal code. This means that users can generate intricate plots like violin plots, heatmaps, and pair plots using intuitive functions without the need for deep customization.

7. Heatmaps and Correlation Matrices

Seaborn provides specialized support for creating heatmaps, especially for visualizing correlation matrices. This is particularly useful when analyzing relationships between variables in a dataset. The heatmap() function allows for the easy creation of visually appealing correlation heatmaps, which can be further customized to display annotations, color bars, and more.

8. Joint and Pair Plots

Seaborn offers the ability to plot multiple types of relationships simultaneously with functions like jointplot() and pairplot(). These functions create visualizations that show the relationship between multiple variables in a dataset by combining scatter plots, histograms, and density plots in one figure. This is especially useful for quickly gaining insights into high-dimensional data.

In summary, Seaborn simplifies the process of creating complex statistical visualizations by offering high-level plotting functions, built-in aesthetics, and seamless integration with Pandas. Its ability to handle complex visualizations with minimal code makes it a favorite tool for data scientists looking to quickly explore and present data insights.

Pandas

Pandas is a powerful and versatile open-source Python library used for data manipulation and analysis. Built on top of NumPy, it provides high-level data structures and tools designed to make working with structured data fast, easy, and flexible. Pandas is widely used in data science, machine learning, and data analysis for handling, cleaning, and transforming datasets efficiently. Its two primary data structures, Series and DataFrame, offer a range of functionalities to handle time series, numerical tables, and data from various file formats.

Key Features of Pandas:

1. Data Structures: Series and DataFrame

- a. **Series:** A one-dimensional labeled array capable of holding data of any type (integers, strings, floats, etc.). It's like a column in a spreadsheet or a SQL table.
- b. **DataFrame:** A two-dimensional labeled data structure, akin to a table with rows and columns, where each column can hold different types of data. DataFrames are the primary structure used for data manipulation in Pandas, making it easy to handle structured datasets.

2. Data Import and Export

Pandas allows users to read and write data from various file formats with minimal effort. This includes support for CSV, Excel, and JSON files, SQL databases, HDF5 and Parquet formats. This functionality makes Pandas a go-to library for data preprocessing, enabling seamless import/export of datasets.

3. Data Cleaning and Preprocessing

Pandas simplifies data cleaning with functions for handling missing data, filtering rows and columns, and performing data transformations. It includes methods for:

- a. **Handling missing data:** With functions like `isnull()`, `fillna()`, and `dropna()`, Pandas helps handle missing or null values in datasets efficiently.
- b. **Data filtering and selection:** Users can easily slice, filter, and select data using intuitive labels or Boolean indexing, allowing for flexible manipulation of subsets of data.

4. Flexible Indexing and Selection

Pandas offers powerful tools for indexing, which allows for easy access to specific rows, columns, or slices of data. With multi-level indexing (also called hierarchical indexing), users can work with high-dimensional data in an intuitive and easy-to-read format. This makes it easy to reshape and access data efficiently.

5. Data Aggregation and Grouping

With methods like `groupby()`, Pandas simplifies the process of grouping data by one or more columns and applying aggregate functions (like `sum`, `mean`, `count`, etc.) to these groups. This is especially useful for summarizing large datasets and performing data aggregations based on categorical variables.

6. Data Alignment and Merging

Pandas offers robust support for merging and joining datasets with functions like `merge()`, `join()`, and `concat()`. These methods allow for database-style joins (inner, outer, left, right) across different DataFrames, making it simple to combine, align, and integrate datasets from multiple sources.

7. Time Series Handling

Pandas provides extensive tools for time series data, including support for date ranges, resampling, frequency conversion, and shifting. It allows for easy handling of datetime objects and provides methods for time-based indexing and analysis, making it highly efficient for analyzing financial or sensor data.

8. Descriptive Statistics and Data Summarization

Pandas offers a wide range of built-in statistical functions for summarizing data, such as `mean()`, `median()`, `std()`, `describe()`, and more. These methods allow for quick and easy calculations of basic statistics across entire datasets or specific subsets of data.

9. Integration with Other Libraries

Pandas works seamlessly with other popular libraries in the Python ecosystem, such as NumPy, Matplotlib, and Scikit-learn. This integration ensures that Pandas can handle data manipulation and analysis tasks before passing data to these libraries for further processing or visualization.

10. Efficient Data Manipulation with Vectorized Operations

Pandas is built on top of NumPy, allowing for fast vectorized operations. This means that functions applied to columns or entire DataFrames are performed in a highly efficient manner, making it well-suited for handling large datasets.

In summary, Pandas provides a robust toolkit for data manipulation, offering functionalities for data cleaning, aggregation, merging, time series analysis, and more. Its ability to handle diverse data types and integrate with other Python libraries makes it a go-to tool for data scientists and analysts.

Scikit-learn (sklearn)

Scikit-learn (sklearn) is a widely used open-source machine learning library for Python, designed to provide simple and efficient tools for data mining, data analysis, and machine learning. Built on top of NumPy, SciPy, and Matplotlib, Scikit-learn offers a user-friendly interface and a consistent API, making it accessible for both beginners and experienced practitioners. The library encompasses a variety of algorithms and techniques, covering classification, regression, clustering, dimensionality reduction, and model selection.

Key Features of Scikit-learn:

1. Comprehensive Library of Algorithms

Scikit-learn includes a diverse array of machine learning algorithms for various tasks, including:

- a. Classification:** Support Vector Machines (SVM), Decision Trees, Random Forests, Gradient Boosting, and Logistic Regression.
- b. Regression:** Linear Regression, Ridge Regression, Lasso Regression, and more advanced techniques like Support Vector Regression.
- c. Clustering:** K-Means, DBSCAN, Hierarchical Clustering, and Gaussian Mixture Models.
- d. Dimensionality Reduction:** Principal Component Analysis (PCA), t-distributed Stochastic Neighbor Embedding (t-SNE), and Singular Value Decomposition (SVD).

2. Easy-to-Use Interface

Scikit-learn is designed to be user-friendly, with a consistent API that follows a fit-predict paradigm. Most functions require only a few lines of code to implement, making it straightforward to use for both beginners and experienced users.

3. Preprocessing and Data Transformation

The library provides various utilities for data preprocessing, including:

- a. Standardization and Normalization:** Scaling features using StandardScaler and MinMaxScaler.
- b. Encoding Categorical Variables:** Using OneHotEncoder and LabelEncoder for transforming categorical data.
- c. Imputation of Missing Values:** Functions like SimpleImputer help handle missing data effectively.
- d. Feature Extraction:** Techniques for extracting features from raw data, such as CountVectorizer for text data.

4. Model Selection and Evaluation

Scikit-learn offers a range of tools for model evaluation and selection, including:

- a. **Cross-Validation:** The `cross_val_score()` function allows for easy model evaluation through k-fold cross-validation.
- b. **Hyperparameter Tuning:** Tools like `GridSearchCV` and `RandomizedSearchCV` help optimize hyperparameters by systematically searching through a specified parameter grid.
- c. **Metrics for Evaluation:** A comprehensive suite of metrics, such as accuracy, precision, recall, F1-score, and ROC-AUC, to evaluate model performance.

5. Pipeline Creation

Scikit-learn allows users to create pipelines that streamline the workflow from preprocessing to model training. The `Pipeline` class enables chaining together preprocessing steps and model training, ensuring that data transformations are applied consistently and efficiently.

6. Support for Custom Models and Transformers

Users can easily extend Scikit-learn's capabilities by implementing custom models and transformers, following the same `fit/predict` interface. This flexibility allows for integration of specialized algorithms or preprocessing techniques into the Scikit-learn ecosystem.

7. Ensemble Methods

Scikit-learn supports ensemble methods, which combine multiple models to improve predictive performance. Techniques like bagging (e.g., Random Forests) and boosting (e.g., AdaBoost, Gradient Boosting) are included, allowing users to leverage the strengths of multiple algorithms for better accuracy.

8. Documentation and Community Support

Scikit-learn has extensive documentation, tutorials, and examples that help users understand how to use the library effectively. Its large community support ensures continuous development, improvement, and troubleshooting resources.

9. Integration with Other Libraries

Scikit-learn seamlessly integrates with other libraries in the Python ecosystem, such as NumPy and Pandas for data handling, Matplotlib for plotting, and Jupyter Notebooks for interactive data exploration. This interoperability enhances its utility in comprehensive data analysis workflows.

10. Support for Different Data Formats

The library can handle various data formats, including NumPy arrays, Pandas DataFrames, and SciPy sparse matrices, allowing users to work with data in the format that best suits their needs.

Data Understanding and EDA (Exploratory Data Analysis)

1. Import Libraries: Start by importing the necessary libraries.

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

[1] ✓ 6.4s Python

Figure 1: Importing essential libraries

2. Load Data:

```
# Load the dataset
data = pd.read_csv('bioinformatics_ml_project.csv')
# data = pd.read_csv('bioinformatics_ml_project.csv', index_col='Patient_ID')

# Display the first few rows of the dataset
data.head()
```

[2] ✓ 0.0s Python

	Patient_ID	Age	Weight	Gene1_Expression	Gene2_Expression	Gene3_Expression	Protein_Descriptor1	Protein_Descriptor2	Drug_Response	Disease
0	1	45.0	93.0	NaN	9.445006	14.14	0.49	1.10	Ineffective	
1	2	79.0	93.0	7.984583	6.053021	7.74	0.47	1.27	Effective	
2	3	70.0	78.0	9.740995	2.121154	9.22	0.46	1.44	Ineffective	
3	4	28.0	NaN	2.856589	7.097806	12.30	0.35	0.90	Ineffective	
4	5	51.0	91.0	9.182268	9.219273	8.06	0.52	1.10	Ineffective	

Figure 2: Loading data from CSV

3. Basic Data Summary:

- Check the shape of the dataset (number of rows and columns).
- Check for null values.
- Get summary statistics.

```
# Check the number of rows and columns
print(data.shape)
```

[3] Python

... (100, 11)

```
# Check for missing values
data.isnull().sum()
```

[4] Python

... Patient_ID 0
Age 10
Weight 8
Gene1_Expression 6
Gene2_Expression 0
Gene3_Expression 0
Protein_Descriptor1 0
Protein_Descriptor2 0
Drug_Response 1
Disease_Status 1
Drug_Dosage 12
dtype: int64

```
# Get summary statistics
data.describe()
```

[5] Python

	Patient_ID	Age	Weight	Gene1_Expression	Gene2_Expression	Gene3_Expression	Protein_Descriptor1	Protein_Descriptor2	Drug_Dosage
count	100.000000	90.000000	92.000000	94.000000	100.000000	100.000000	100.000000	100.000000	88.000000
mean	50.500000	52.833333	75.141704	5.677361	5.165436	9.889400	0.461100	1.142300	12.121637

Figure 3: Basic data summary

4. Visualize Data:

- Plot distributions of features such as age, weight, and gene expressions.
- Use pair plots to see relationships between variables.

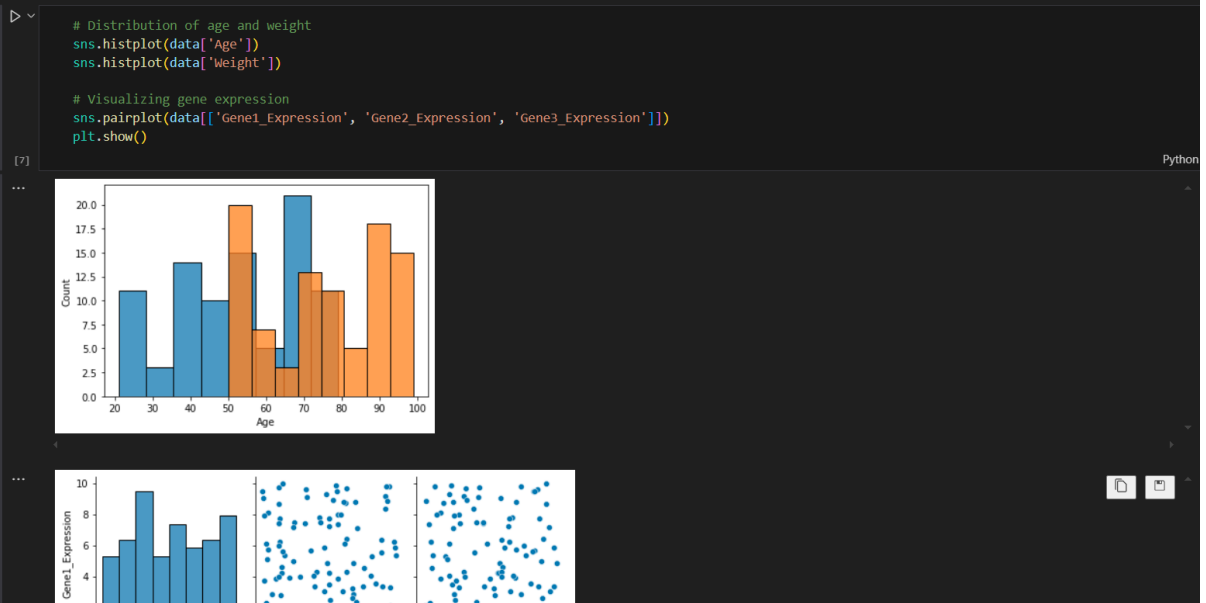


Figure 4: Visualizing data



Figure 5: Visualizing distribution

Data preprocessing

```
▷ ~
# Filling missing values for numeric columns (if necessary)
data.fillna(data.mean(numeric_only=True), inplace=True)

# Check for missing values
data.isnull().sum()

[10] Python

... Patient_ID      0
Age              0
Weight           0
Gene1_Expression 0
Gene2_Expression 0
Gene3_Expression 0
Protein_Descriptor1 0
Protein_Descriptor2 0
Drug_Response     1
Disease_Status    1
Drug_Dosage       0
dtype: int64
```

Figure 6: Handling missing numerical data

```
▷ ~
# Filling missing values for categorical columns with the mode
for column in data.select_dtypes(include=['object']).columns:
    data[column].fillna(data[column].mode()[0], inplace=True)

# Check for missing values
data.isnull().sum()

[13] ✓ 0.0s Python

... C:\Users\raanna\AppData\Local\Temp\ipykernel_21924\306021795.py:3: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through ch
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves a
For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value) instead

    data[column].fillna(data[column].mode()[0], inplace=True)

... Patient_ID      0
Age              0
Weight           0
Gene1_Expression 0
Gene2_Expression 0
Gene3_Expression 0
Protein_Descriptor1 0
Protein_Descriptor2 0
Drug_Response     0
Disease_Status    0
Drug_Dosage       0
dtype: int64
```

Figure 7: Handling missing object data

```
▷ ~
# Encode categorical variables
data['Disease_Status'] = data['Disease_Status'].map({'Cancer': 1, 'Healthy': 0})
data['Drug_Response'] = data['Drug_Response'].map({'Effective': 1, 'Ineffective': 0})

[15] ✓ 0.0s

• The map() function is case-sensitive and expects exact matches for the strings provided.
• This is not possible when there are more number of different categorical values.
• That is when LabelEncoder() can be used so that mentioning exact strings is not needed.

# Another way:
from sklearn.preprocessing import LabelEncoder

# Encode categorical columns
label_encoder = LabelEncoder()
data['Drug_Response'] = label_encoder.fit_transform(data['Drug_Response'])
data['Disease_Status'] = label_encoder.fit_transform(data['Disease_Status'])

[16] ✓ 0.2s
```

Figure 8: Encoding categorical data

<pre># Before encoding data[['Disease_Status', 'Drug_Response']].head()</pre>																				
[14]	✓	0.0s																		
...	<table> <tr> <th></th><th>Disease_Status</th><th>Drug_Response</th></tr> <tr> <td>0</td><td>Cancer</td><td>Ineffective</td></tr> <tr> <td>1</td><td>Healthy</td><td>Effective</td></tr> <tr> <td>2</td><td>Cancer</td><td>Ineffective</td></tr> <tr> <td>3</td><td>Cancer</td><td>Ineffective</td></tr> <tr> <td>4</td><td>Healthy</td><td>Ineffective</td></tr> </table>			Disease_Status	Drug_Response	0	Cancer	Ineffective	1	Healthy	Effective	2	Cancer	Ineffective	3	Cancer	Ineffective	4	Healthy	Ineffective
	Disease_Status	Drug_Response																		
0	Cancer	Ineffective																		
1	Healthy	Effective																		
2	Cancer	Ineffective																		
3	Cancer	Ineffective																		
4	Healthy	Ineffective																		

<pre># After encoding data[['Disease_Status', 'Drug_Response']].head()</pre>																				
[17]	✓	0.0s																		
...	<table> <tr> <th></th><th>Disease_Status</th><th>Drug_Response</th></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>2</td><td>1</td><td>0</td></tr> <tr> <td>3</td><td>1</td><td>0</td></tr> <tr> <td>4</td><td>0</td><td>0</td></tr> </table>			Disease_Status	Drug_Response	0	1	0	1	0	1	2	1	0	3	1	0	4	0	0
	Disease_Status	Drug_Response																		
0	1	0																		
1	0	1																		
2	1	0																		
3	1	0																		
4	0	0																		

Figure 9: Categorical data before and after encoding

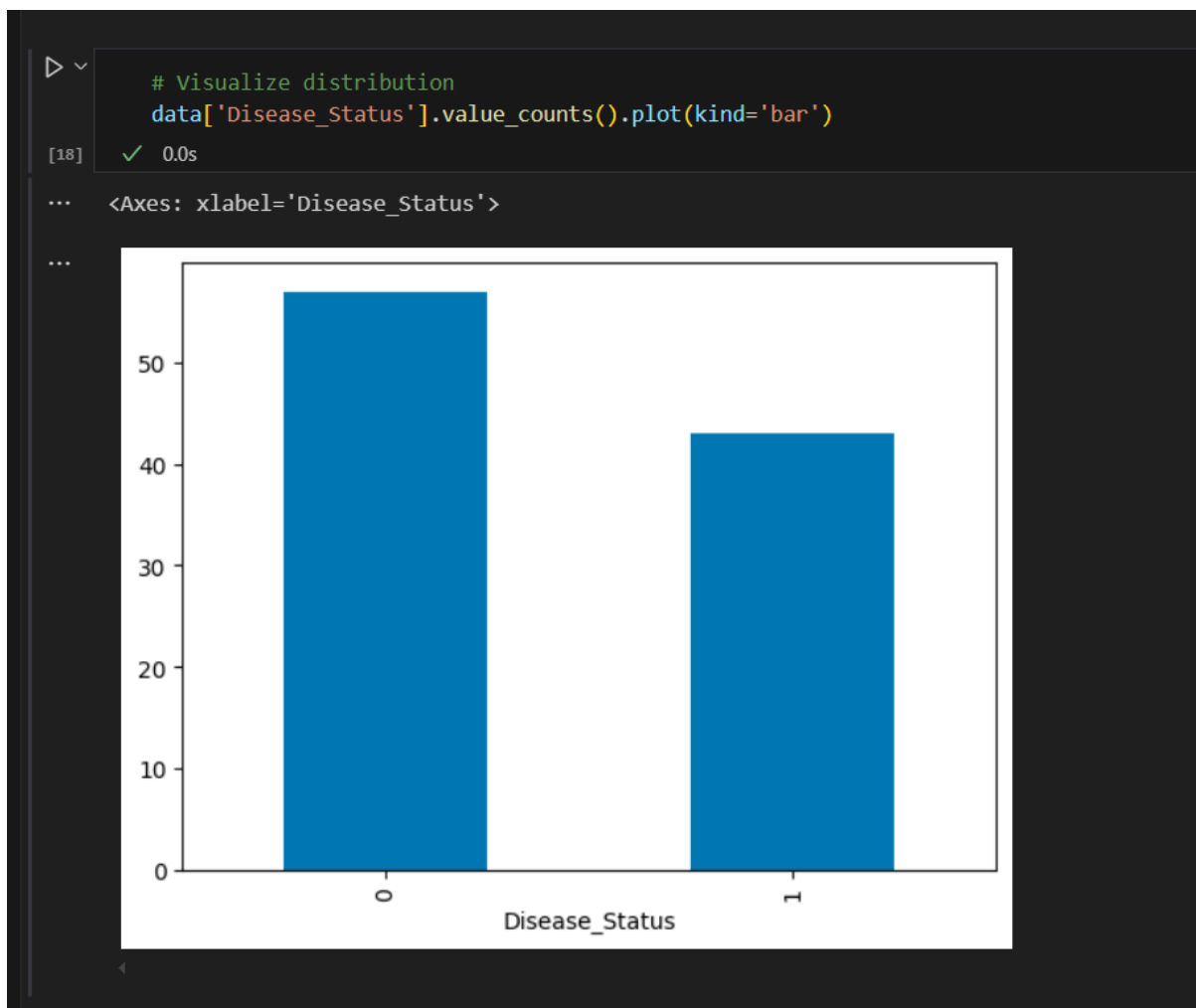


Figure 10: Visualizing data distribution

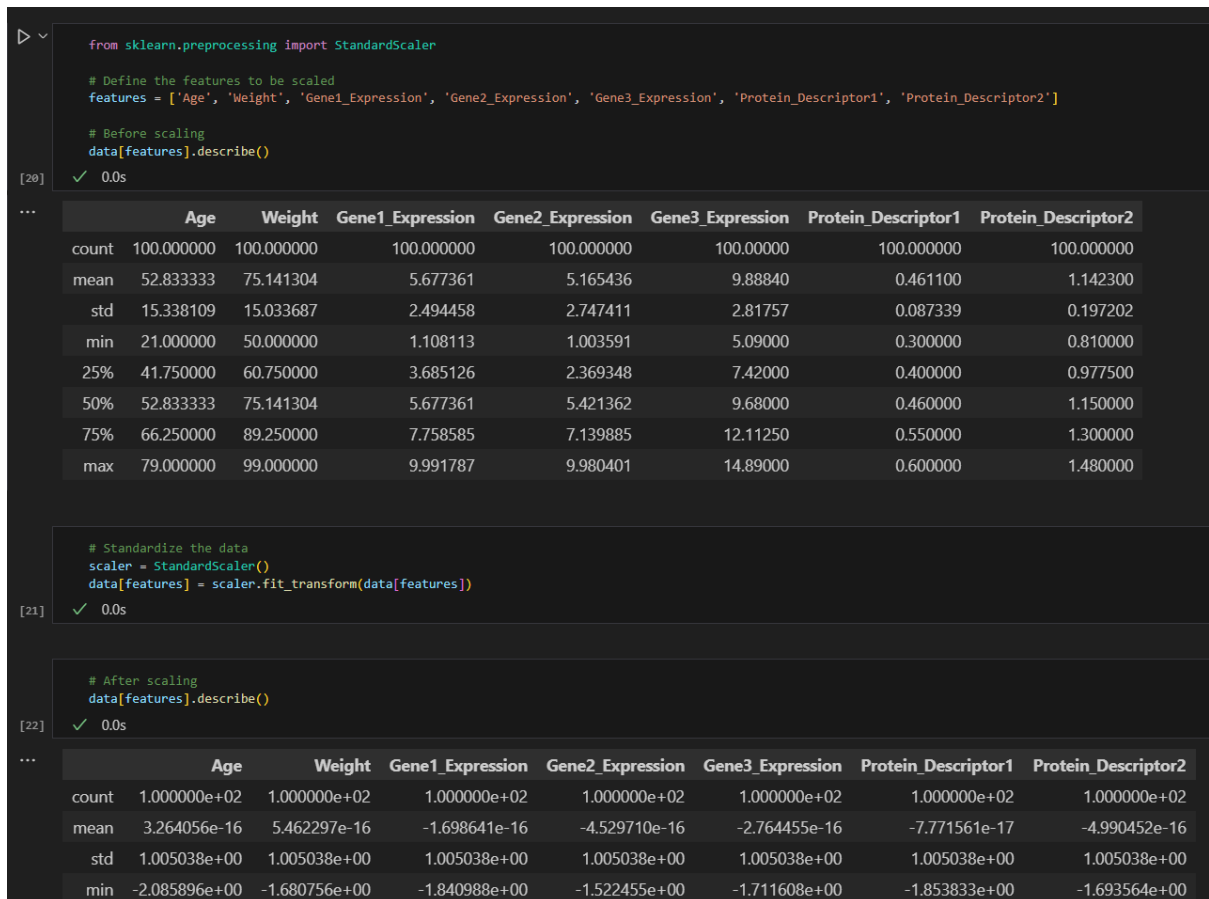


Figure 11: Feature Scaling: Normalizing or Standardizing data

Model Building & Interpretation

Model 1: Linear Regression for Drug Dosage Prediction

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Define features (Age, Weight, Disease_Status) and target (Drug_Dosage)
X = data[['Age', 'Weight', 'Disease_Status']]
y = data['Drug_Dosage']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

[23] ✓ 0.7s

Figure 12: Defining features and target

2. Train the Model:

```
# Train Linear Regression Model
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Make predictions
y_pred = lin_reg.predict(X_test)
```

] ✓ 0.0s

Figure 13: Training model

```
from sklearn.metrics import mean_squared_error, r2_score

# Calculate Mean Squared Error and R-squared
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Mean Squared Error: {mse}')
print(f'R-squared: {r2}')
```

[26] ✓ 0.0s

... Mean Squared Error: 1543.552338424623
R-squared: 0.015031450248899847

Figure 14: Model evaluation

INTERPRETATION:

Mean Squared Error (MSE):

Lower MSE values are better, indicating that predictions are closer to actual values.

An MSE close to 0 suggests accurate predictions, while an MSE of 1543.55 indicates significant error in the predictions.

R-squared (R^2):

R^2 measures the proportion of variance in the target variable explained by the model, with values ranging from 0 to 1.

$R^2 = 1$: Perfect explanation of variance.

$R^2 = 0$: No variance explained (like predicting the mean).

$R^2 < 0$: Worse performance than a horizontal line.

CONCLUSION:

The model's $R^2 = 0.015$ indicates that it explains only 1.5% of the variance, highlighting significant underperformance.

Model 2: Decision Tree for Disease Classification

```
from sklearn.tree import DecisionTreeClassifier

# Features: physical information, gene expressions and protein descriptors
X = data[['Age', 'Weight', 'Gene1_Expression', 'Gene2_Expression', 'Gene3_Expression', 'Protein_Descriptor1', 'Protein_Descriptor2']]
y = data['Disease_Status']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

[28] ✓ 0.2s Python

Figure 15: Defining features and targets

```
# Train Decision Tree
tree_clf = DecisionTreeClassifier()
tree_clf.fit(X_train, y_train)

# Predict
y_pred = tree_clf.predict(X_test)
```

[29] ✓ 0.0s

Figure 16: Training model

```
from sklearn.metrics import accuracy_score

# Calculate Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

[30] ✓ 0.0s

... Accuracy: 0.75

Figure 17: Evaluating model



Figure 18: Confusion matrix for model

INTERPRETATION:

An accuracy of 0.75 indicates that 75% of the model's predictions are correct.

However, accuracy alone may not fully represent model performance, particularly in imbalanced datasets or when false positives/negatives are critical. In bioinformatics and drug discovery, minimizing false negatives (e.g., misclassifying a diseased patient as healthy) is often more important.

Other metrics, such as precision, recall, F1-score, and ROC-AUC, provide a more comprehensive evaluation of classification models.

CONCLUSION:

While a 75% accuracy may be acceptable, it's essential to check for dataset balance and utilize additional evaluation metrics for a thorough performance assessment.

Implementing techniques like hyperparameter tuning and feature engineering can further enhance model accuracy.

Model 3: Random Forest for Drug Response Prediction

```
from sklearn.ensemble import RandomForestClassifier

# Features: protein descriptors and gene expressions
X = data[['Age', 'Weight', 'Protein_Descriptor1', 'Protein_Descriptor2', 'Gene1_Expression', 'Gene2_Expression', 'Drug_Dosage']]
y = data['Drug_Response']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

[32] ✓ 0.1s

Figure 19: Defining features & targets

```
# Train Random Forest
rf_clf = RandomForestClassifier()
rf_clf.fit(X_train, y_train,)

# Predict
y_pred = rf_clf.predict(X_test)
```

[33] ✓ 0.0s

Figure 20: Training model

```
# Calculate Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

[34] ✓ 0.0s

... Accuracy: 0.55

Figure 21: Evaluating model

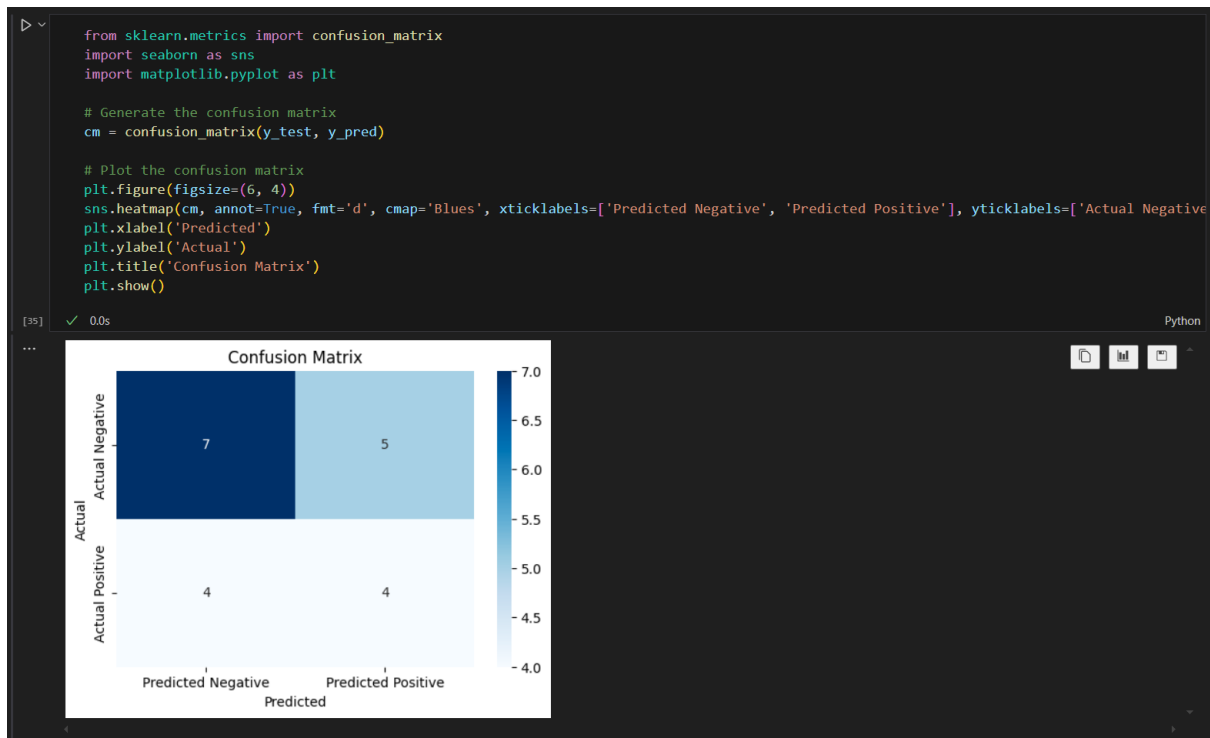


Figure 22: Confusion matrix for model

INTERPRETATION:

An accuracy of 55% means the model correctly predicted outcomes for 55% of the samples in the test set, indicating that 45% of the predictions were incorrect.

CONCLUSION:

The 55% accuracy of the Random Forest model suggests low performance, which may result from factors such as overfitting, poor feature selection, or imbalanced data.

To improve accuracy and overall performance, consider tuning hyperparameters, using cross-validation, addressing class imbalances, and evaluating feature importance.

Additionally, other evaluation metrics to consider include:

Precision: Proportion of predicted positives that are actual positives.

Recall: Proportion of actual positives correctly predicted.

F1-Score: The harmonic means of precision and recall.

ROC-AUC: Measures the model's ability to distinguish between classes.

OVERALL RESULTS

In this project, we employed three distinct machine learning models to address specific bioinformatics challenges:

Linear Regression was used to predict Drug Dosage, providing a quantitative understanding of the relationships between input features and the dosage required for effective treatment.

Decision Trees facilitated the classification of diseases, offering a clear, interpretable model that aids in understanding the decision-making process behind classifying patients based on their symptoms and test results.

Random Forest was applied for predicting Drug Response, leveraging ensemble learning to enhance prediction accuracy and reduce the likelihood of overfitting by aggregating results from multiple decision trees.

Each model was evaluated using relevant metrics, including accuracy, Mean Squared Error (MSE), and R-squared values, allowing for a comprehensive assessment of performance and effectiveness in addressing the respective tasks.

OVERALL CONCLUSION

The project successfully integrated exploratory data analysis (EDA), data preprocessing, and modeling using diverse machine learning techniques within the bioinformatics domain. While the models provided valuable insights, the performance metrics indicated areas for improvement, particularly for the Random Forest model, which demonstrated a 55% accuracy. Factors such as overfitting, class imbalances, and feature selection were identified as potential challenges.

Moving forward, the project can be expanded by enhancing feature engineering, applying hyperparameter tuning, and incorporating additional machine learning techniques. This approach will not only improve the predictive accuracy of the models but also enrich the overall analysis, contributing to more effective decision-making in bioinformatics applications. The framework established here serves as a foundation for further research and exploration in the intersection of machine learning and healthcare.

REFERENCE:

1. GeeksforGeeks. (2024, September 12). Machine Learning Tutorial. GeeksforGeeks. <https://www.geeksforgeeks.org/machine-learning/>
 2. FreeCodeCamp.org. (n.d.). <https://www.freecodecamp.org/learn/machine-learning-with-python/>
-