P24DS1P1

DATA STRUCTURE AND PYTHON PROGRAMMING LAB

Developed by

Ms. C. Linda Hepsiba

NAME

_____

ROLL NUMBER

_____

PG Department of Data Science

Bishop Heber College(Autonomous)

Tiruchirappalli-620017

PG Department of Data Science

Bishop Heber College (Autonomous)

Tiruchirappalli-620 017, Tamil Nadu

Ranked **34th** at the National Level by MHRD through **NIRF 2023**

(Nationally Reaccredited at the **A++** Grade by NAAC with a CGPA of **3.69 out of 4**)

BONAFIDE CERTIFICATE

Name :_____

Register No : _____

Class : _____

Course Title : **P24DS1P1**

Course Title : **Data Structures and Python Programming Lab**

Certified that this is the bonafide record of work done by me during Odd Semester of 2024 - 2025 and submitted to the Practical Examination on_____

Staff In-Charge **Head of the Department**

Examiners

1._____

2._____

INDEX

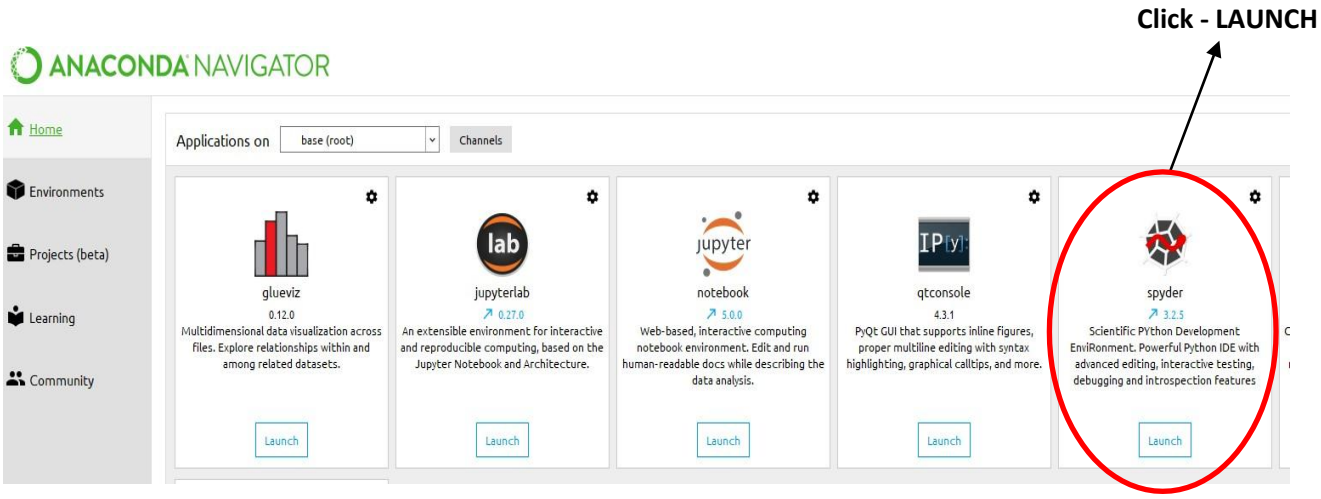
INSTALLING PYTHON - SPYDER

1. To install spyder, through - ANACONDA NAVIGATOR, click on the link below: -

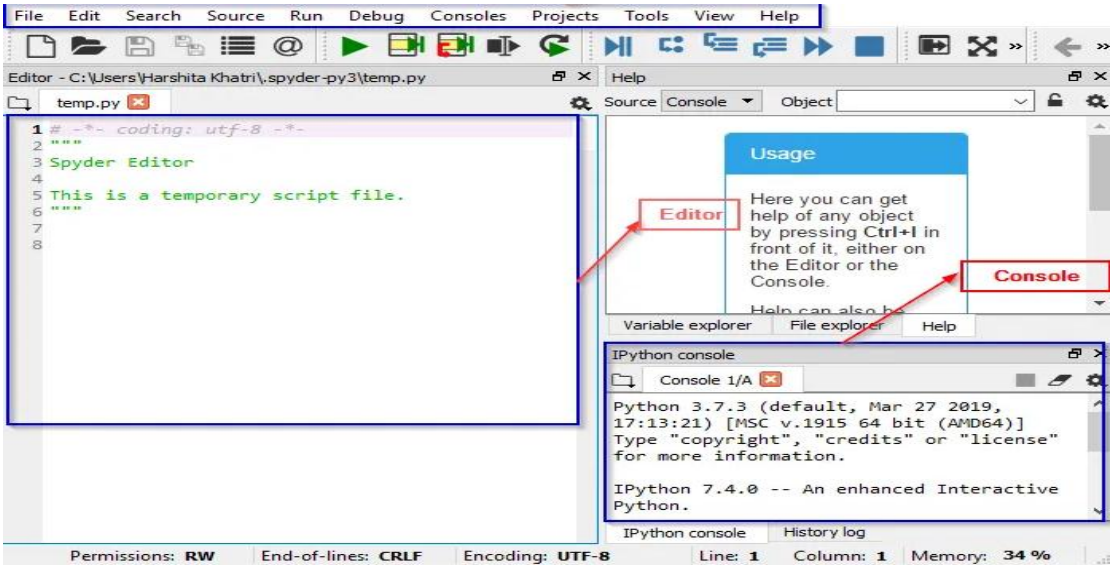
https://repo.anaconda.com/archive/

2. Next choose the version based on the OS: -
   - **a.** Windows- **Anaconda3-5.1.0-Windows-x86_64.exe**
   - **b.** Mac- **Anaconda3-5.1.0-MacOSX-x86_64.pkg**
   - **c.** Linux- **Anaconda3-5.1.0-Linux-x86_64.sh**

3. Click on the version to download
4. Post the download, run the setup file
5. Set the path
6. Open the ANACONDA NAVIGATOR



7. Start working with SPYDER.



Week: 1 Basic Python Programs

Objective:

To practice basic Python programming concepts such as operators, loops, conditional statements, list comprehension, and error handling by implementing a set of simple programs.

Exercise 1: Arithmetic operators

Aim: To write a Python program that performs arithmetic operations (addition, subtraction, multiplication, division, and modulus) on two user-input integers.

Procedure:

1. Start by taking two integer inputs from the user.
2. Perform the arithmetic operations: addition, subtraction, multiplication, division, and modulus.
3. Display the results of each operation using formatted print statements.
4. Test the program with various inputs to ensure accuracy.

Program:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print(f"Addition: {a + b}")
print(f"Subtraction: {a - b}")
print(f"Multiplication: {a * b}")
print(f"Division: {a / b}")
print(f"Modulus: {a % b}")
```

Output:

Exercise 2: Comparison Operators

Aim: To write a Python program that compares two user-input integers using relational operators and displays the results.

Procedure:

1. Prompt the user to input two integers.
2. Use relational operators (`==`, `!=`, `>`, `<`, `>=`, `<=`) to compare the two integers.
3. Display the results of each comparison using formatted print statements.
4. Test the program with different inputs to check for equality and inequalities between the numbers.

Program:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print(f"{a} == {b}: {a == b}")
print(f"{a} != {b}: {a != b}")
print(f"{a} > {b}: {a > b}")
print(f"{a} < {b}: {a < b}")
print(f"{a} >= {b}: {a >= b}")
print(f"{a} <= {b}: {a <= b}")
```

Output:

Exercise 3: Logical Operators

Aim: To write a Python program that demonstrates the use of logical operators (`and`, `or`, `not`) on two user-input boolean values.

Procedure:

1. Take two boolean inputs from the user by converting integers (0 or 1) to boolean.
2. Use logical operators (`and`, `or`, `not`) to evaluate the values.
3. Display the results of each operation using formatted print statements.
4. Test with different inputs to verify all cases.

Program:

```
a = bool(int(input("Enter first boolean value (0 or 1): ")))
b = bool(int(input("Enter second boolean value (0 or 1): ")))

print(f"{a} and {b}: {a and b}")
print(f"{a} or {b}: {a or b}")
print(f"not {a}: {not a}")
```

Output:

Exercise 4: If-Else Statements

Aim: To write a Python program that determines whether a number is positive, zero, or negative using if-else statements.

Procedure:

1. Accept an integer input from the user.
2. Use if-elif-else statements to check if the number is positive, zero, or negative.
3. Print the corresponding message based on the condition.
4. Test the program with positive, zero, and negative inputs.

Program:

```
num = int(input("Enter a number:"))
if num > 0:
    print("The number is positive")
elif num == 0:
    print("The number is zero")
else:
    print("The number is negative")
```

Output:

Exercise 5: For Loop

Aim: To write a Python program that prints numbers from 1 to a user-specified value using for loop.

Procedure:

1. Accept a positive integer `n` from the user.
2. Use a for loop to iterate from 1 to `n`.
3. Print each number during each iteration.
4. Ensure the loop stops at the specified number.

Program:

```
n = int(input("Enter a number: "))
for i in range(1, n + 1):
print(i)
```

Output:

Exercise 6: While Loop

Aim: To write a Python program that prints numbers from 1 to a user-specified number using a while loop.

Procedure:

1. Accept an integer input from the user.
2. Initialize a variable to 1 and use a while loop to print numbers until it reaches the input.
3. Increment the variable inside the loop.
4. Test with different numbers to ensure proper functionality.

Program:

```
num = int(input("Enter a number: "))
i = 1
while i <= num:
    print(i)
    i += 1
```

Output:

Exercise 7: Nested Loop

Aim: To write a Python program that prints a right-angled triangle pattern using nested loops.

Procedure:

1. Accept the number of rows for the triangle from the user.
2. Use a nested loop where the outer loop controls the rows and the inner loop prints the stars.
3. Print stars incrementally on each row.
4. Test with different values for the number of rows.

Program:

```
n = int(input("Enter the number of rows: "))
for i in range(1, n + 1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()
```

Output:

Exercise 8: Break and Continue

Aim: To write a Python program demonstrating the use of `break` and `continue` in loops.

Procedure:

1. Use a for loop to iterate from 1 to 10.
2. Skip the iteration when `i` equals 5 using `continue`.
3. Exit the loop when `i` equals 8 using `break`.
4. Test the loop to ensure it works as expected.

Program:

```
for i in range(1, 11):

    if i == 5:
        continue # Skip the rest of the code inside the loop for current iteration
    if i == 8:
        break  #  Exit
    the loopprint(i)
```

Output:

Exercise 9: List Comprehension with Condition

Aim: To write a Python program that generates a list of even numbers from 0 to 19 using list comprehension.

Procedure:

1. Use a list comprehension to iterate through numbers from 0 to 19.
2. Include a condition to select only even numbers (`x % 2 == 0`).
3. Store and print the resulting list.
4. Verify the output to ensure only even numbers are included.

Program:

```
numbers = [x for x in range(20) if x % 2 == 0]
print(numbers)
```

Output:

Exercise 10: Try-Except for Error Handling

Aim: To write a Python program that handles division by zero and invalid input using try-except blocks.

Procedure:

1. Accept two numbers from the user for division.
2. Use a try-except block to handle `ZeroDivisionError` and `ValueError`.
3. Print appropriate error messages for division by zero or invalid input.
4. Test with various inputs to ensure proper error handling.

Program:

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    result = a / b
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed")
except ValueError:
    print("Error: Invalid input, please enter a number")
```

Output:

Result:

The programs successfully demonstrate arithmetic and logical operations, control flow using loops and conditionals, and handle errors with try-except blocks.

Week: 2 Control Flow and Modular Programs

Objective:

To implement various Python programs that focus on control structures, functions, and real world applications such as calculators, converters, and parsers. The programs provide hands on practice with switch case like structures, conditional logic, and function handling.

Exercise 1: Calculator Program

Aim: To perform basic arithmetic operations (addition, subtraction, multiplication, division) based on user input.

Procedure:

1. Define a `calculator` function with parameters for operation, number 1, and number 2.
2. Use a dictionary to map the operation to the corresponding arithmetic operation.
3. Return the result based on the user's operation.
4. Handle division by zero by returning an error message.
5. Test the function with different operations.

Program:

```
def calculator(operation, num1, num2):
    switcher = {
        'add': num1 + num2,
        'subtract': num1 - num2,
        'multiply': num1 * num2,
        'divide': num1 / num2 if num2 != 0 else 'Error: Division by zero'
    }
    return switcher.get(operation, 'Invalid operation')

# Example usage:
print(calculator('add', 10, 5))        # Output: 15
```

```
print(calculator('subtract', 10, 5))   # Output: 5
print(calculator('multiply', 10, 5))  # Output: 50
print(calculator('divide', 10, 5))     # Output: 2.0
print(calculator('divide', 10, 0))     # Output: Error: Division by zero
print(calculator('mod', 10, 5))        # Output: Invalid operation
```

Output:

Exercise 2: Day of the Week Program

Aim: To return the day of the week based on a number input.

Procedure:

1. Define a function `day_of_week` with a day number as input.
2. Use a dictionary to map numbers (1–7) to days of the week.
3. Return the day of the week corresponding to the input number.
4. If the input is invalid, return an error message.
5. Test the function with different inputs.

Program:

```
def day_of_week(day_number):

switcher = {

        1: "Sunday",

        2: "Monday",

        3: "Tuesday",

        4: "Wednesday",

        5: "Thursday",

        6: "Friday",

        7: "Saturday"
    }
    return switcher.get(day_number, "Invalid day number")
 # Example usage:

 print(day_of_week(1)) # Output: Sunday

 print(day_of_week(8))  # Output: Invalid day number
```

Output:

Exercise 3: Grade to GPA Converter

Aim: To convert letter grades (A–F) into GPA values.

Procedure:

1. Define a function `grade_to_gpa` that takes a grade as input.
2. Use a dictionary to map grades (A–F) to GPA values.
3. Return the GPA value based on the input grade.
4. Return an error message for invalid grades.
5. Test the function with different grade inputs.

Program:

```
def grade_to_gpa(grade):
    switcher = {
        'A': 4.0,
        'B': 3.0,
        'C': 2.0,
        'D': 1.0,
        'F': 0.0
    }
    return switcher.get(grade, "Invalid grade")
# Example usage:
print(grade_to_gpa('A'))  # Output: 4.0
print(grade_to_gpa('E'))  # Output: Invalid grade
```

Output:

Exercise 4: Month Days Program

Aim: To display the number of days in a month based on user input.

Procedure:

1. Define a function `days_in_month` that takes a month as input.
2. Use a dictionary to map each month to its corresponding number of days.
3. Return the number of days based on the input month.
4. Return an error message for invalid months.
5. Test the function with various months.

Program:

```
def days_in_month(month):
switcher = {
    'January': 31,
    'February': 28, # Ignoring leap year for simplicity
    'March': 31,
    'April': 30,
    'May': 31,
    'June': 30,
    'July': 31,
    'August': 31,
    'September': 30,
    'October': 31,
    'November': 30,
    'December': 31
}
return switcher.get(month, "Invalid month")
```

```
# Example usage:
print(days_in_month('February'))  # Output: 28
print(days_in_month('April'))     # Output: 30
print(days_in_month('Invalid'))  # Output: Invalid month
```

Output:

Exercise 5: Animal Sound Program

Aim: To return the sound made by different animals based on user input.

Procedure:

1. Define a function `animal_sound` that takes an animal name as input.
2. Use a dictionary to map animals to their respective sounds.
3. Return the sound associated with the input animal.
4. Return an error message for unknown animals.
5. Test the function with various animal names.

Program:

```
def animal_sound(animal):
    switcher = {
        'dog': 'bark',
        'cat': 'meow',
        'cow': 'moo',
        'lion': 'roar'
    }
    return switcher.get(animal, "Unknown sound")

# Example usage:

print(animal_sound('dog')) # Output: bark
print(animal_sound('cat'))  # Output: meow
print(animal_sound('giraffe'))  # Output: Unknown sound
```

Output:

Exercise 6: Shape Area Calculator

Aim: To calculate the area of different shapes (circle, square, triangle) based on user input.

Procedure:

1. Define a function `shape_area` with shape type and dimensions as inputs.

2. Use conditionals to calculate the area based on the shape type.

3. For circles, calculate area as $\pi * r^2$; for squares, $side^2$; for triangles, 0.5 * base * height.

4. Handle invalid shapes with an error message.

5. Test the function with different shapes and dimensions.

Program:

```
def shape_area(shape, dimension):
    if shape == 'circle':
        return 3.14 * (dimension ** 2)
    elif shape == 'square':
        return dimension ** 2
    elif shape == 'triangle':
        return 0.5 * dimension[0] * dimension[1]  # dimension is (base, height)
    else:
        return "Invalid shape"
# Example usage:
print(shape_area('circle', 5))        # Output: 78.5
print(shape_area('square', 4))        # Output: 16
print(shape_area('triangle', (3, 4))) # Output: 6.0
print(shape_area('hexagon', 5))       # Output: Invalid shape
```

Output:

Exercise 7: Traffic Light Program

Aim: To simulate traffic light behavior by displaying corresponding actions based on the light color.

Procedure:

1. Define a function `traffic_light` that takes the light color as input.
2. Use a dictionary to map colors (red, yellow, green) to corresponding actions (stop, slow down, go).
3. Return the appropriate action based on the input color.
4. Return an error message for invalid colors.
5. Test the function with different traffic light colors.

Program:

```
def traffic_light(action):
    switcher = { 'red':
        'Stop',
        'yellow': 'Slow down',
        'green': 'Go'
    }
    return switcher.get(action, "Invalid action")
# Example usage:
print(traffic_light('red'))     # Output: Stop
print(traffic_light('yellow')) # Output: Slow down
print(traffic_light('green')) # Output: Go
print(traffic_light('blue'))    # Output: Invalid action
```

Output:

Exercise 8: Temperature Converter

Aim: To convert temperatures between Celsius and Fahrenheit.

Procedure:

1. Define a function `temperature_converter` with temperature scale and value as inputs.
2. Convert temperatures using appropriate formulas: (C → F) and (F → C).
3. Return the converted value based on the input scale.
4. Return an error message for invalid temperature scales.
5. Test the function with different temperature values.

Program:

```
def temperature_converter(scale, temperature):
    if scale == 'C_to_F':
        return (temperature * 9/5) + 32
    elif scale == 'F_to_C':
        return (temperature - 32) * 5/9
    else:
        return "Invalid scale"
# Example usage:
print(temperature_converter('C_to_F', 0))    # Output: 32.0
print(temperature_converter('F_to_C', 32)) # Output: 0.0
print(temperature_converter('K_to_C', 273))  # Output: Invalid scale
```

Output:

Exercise 9: Simple Command Line Argument Parser

Aim: To simulate a command line argument parser that interprets user commands.

Procedure:

1. Define a function `cli_parser` that takes a command as input.
2. Use a dictionary to map commands (start, stop, restart) to appropriate responses.
3. Return the corresponding message based on the command.
4. Use `sys.argv` to get the command from the command line.
5. Test the function with different command line inputs.

Program:

```
import sys
def cli_parser(command):
    switcher = {
        'start': "Starting the program...",
        'stop': "Stopping the program...",
        'restart': "Restarting the program..."
    }
    return switcher.get(command, "Unknown command")
# Example usage:
if len(sys.argv) > 1:
    print(cli_parser(sys.argv[1]))
else:
    print("No command provided")
```

Output:

Exercise 10: Currency Converter

Aim: To convert an amount of money between different currencies using predefined exchange rates.

Procedure:

1. Define a function `currency_converter` that takes an amount and currency pair as input.
2. Use a dictionary to store exchange rates for different currency pairs.
3. Multiply the amount by the exchange rate to convert the currency.
4. Return an error message for invalid currency pairs.
5. Test the function with different currency pairs and amounts.

Program:

```
def currency_converter(amount, currency):
    rates = {
        'USD_to_EUR': 0.85,
        'EUR_to_USD': 1.18,
        'USD_to_GBP': 0.75,
        'GBP_to_USD': 1.33
    }
    rate = rates.get(currency, None)if
    rate:
        return amount * rate
    else:
        return "Invalid currency pair"
# Example usage:
print(currency_converter(100, 'USD_to_EUR')) # Output: 85.0
print(currency_converter(100, 'EUR_to_USD')) # Output: 118.0
```

```
print(currency_converter(100, 'USD_to_INR'))  # Output: Invalid currency pair
```

Output:

Result:

The programs successfully perform specific operations like calculating values, converting units, interpreting commands, and simulating real world scenarios.

Week: 3 Core Data Structures and OOP Principles

Objective:

To implement and practice Python programs involving lists, dictionaries, tuples, strings, regular expressions, and object oriented programming (OOP) concepts such as inheritance, encapsulation, polymorphism, and abstraction.

Exercise 1: Reverse a List

Aim: To reverse the elements of a given list using Python.

Procedure:

1. Define a function that takes a list as input.
2. Use list slicing to reverse the list.
3. Return the reversed list.

Program:

```
def reverse_list(lst):
    return lst[::-1]
sample_list = [1, 2, 3, 4, 5]
print("Original list:", sample_list)
print("Reversed list:", reverse_list(sample_list))
```

Output:

Exercise 2: Count Frequency of Elements (Dictionaries)

Aim: To count the frequency of elements in a list using a dictionary.

Procedure:

1. Define a function that iterates over a list.
2. Check if the element exists in the dictionary.
3. If it exists, increment the count, otherwise, initialize it.

Program:

```
def count_frequency(lst):
    freq_dict = {}
    for item in lst:
        if item in freq_dict:
            freq_dict[item] += 1
        else:
            freq_dict[item] = 1
    return freq_dict

sample_list = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
print("Frequency count:", count_frequency(sample_list))
```

Output:

Exercise 3: Find Max and Min in a Tuple

Aim: To find the maximum and minimum elements in a tuple.

Procedure:

1. Define a function that accepts a tuple.
2. Use the built-in `max()` and `min()` functions to find the values.
3. Return the maximum and minimum values.

Program:

```
def max_min_tuple(tpl):
    return max(tpl), min(tpl)
sample_tuple = (4, 7, 1, 3, 9, 2)
max_val, min_val = max_min_tuple(sample_tuple)
print("Maximum value:", max_val)
print("Minimum value:", min_val)
```

Output:

Exercise 4: Check Palindrome (Strings)

Aim: To check whether a given string is a palindrome.

Procedure:

1. Define a function that takes a string as input.
2. Reverse the string and compare it with the original string.
3. Return `True` if it's a palindrome, otherwise `False`.

Program:

```
def is_palindrome(s):
    return s == s[::-1]
sample_string = "radar"
print(f"Is '{sample_string}' a palindrome?", is_palindrome(sample_string))
```

Output:

Exercise 5: Extract Emails (Regular Expressions)

Aim: To extract all email addresses from a given text using regular expressions.

Procedure:

1. Define a function that uses a regular expression pattern to match email addresses.
2. Use the `re.findall()` method to find all email addresses.
3. Return the list of extracted emails.

Program:

```
importre
def extract_emails(text):
    email_pa6ern = r'[a-zA-Z0- 9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'
    return re.findall(email_pa6ern, text)
sample_text = "Please contact us at support@example.com or sales@example.org."
emails = extract_emails(sample_text)
print("Extracted emails:", emails)
```

Output:

Exercise 6: Define a Class (`Person`)

Aim: To create a class `Person` with attributes for name and age, and a method to greet.

Procedure:

1. Define a `Person` class with attributes for `name` and `age`.
2. Implement a method `greet()` that prints a greeting message.
3. Instantiate the class and call the `greet()` method.

Program:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
person1 = Person("Alice", 30)
person1.greet()
```

Output:

Exercise 7: Inheritance (`Student` Subclass)

Aim: To demonstrate inheritance by creating a subclass `Student` that inherits from `Person`.

Procedure:

1. Define a `Student` class inheriting from `Person`.
2. Add an additional attribute `student_id` and a method to display it.
3. Instantiate the `Student` class and call the methods.

Program:

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__Init__(name,age)
        self.student_id = student_id
    def display_id(self):
        print(f"My student ID is {self.student_id}.")
student1 = Student("Bob", 20, "S12345")
student1.greet()
student1.display_id()
```

Output:

Exercise 8: Encapsulation (Private Attributes)

Aim: To demonstrate encapsulation by using private attributes in a class `BankAccount`.

Procedure:

1. Define a `BankAccount` class with private attributes for account number and balance.
2. Implement methods to deposit, withdraw, and check balance.
3. Demonstrate the encapsulation by performing operations.

Program:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount}. New balance is {self.balance}.")
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance is {self._balance}.")
        else:
            print("Insufficient balance.")

account = BankAccount("123456789", 1000)
account.deposit(500)
account.withdraw(200)
```

Output:

Exercise 9: Polymorphism (Method Overriding)

Aim: To demonstrate polymorphism by overriding the `sound()` method in subclasses of `Animal`.

Procedure:

1. Define a base class `Animal` with a method `sound()`.
2. Create subclasses `Dog` and `Cat` and override the `sound()` method.
3. Instantiate the classes and call the `sound()` method to demonstrate polymorphism.

Program:

```
class Animal:
    def sound(self):
        print("Some generic animal sound")
class Dog(Animal):
    def sound(self):
        print("Bark")
class Cat(Animal):
    def sound(self):
        print("Meow")
animals = [Animal(), Dog(), Cat()]
for animal in animals:
    animal.sound()
```

Output:

Exercise 10: Abstraction (Abstract Class)

Aim: To implement an abstract class `Shape` with a method `area()` using Python's `ABC` module.

Procedure:

1. Define an abstract class `Shape` with an abstract method `area()`.
2. Create a subclass `Rectangle` that implements the `area()` method.
3. Instantiate the subclass and calculate the area of a rectangle.

Program:

```
from abc import ABC,
abstractmethod
class Shape(ABC): @abstractmethod
        def area(self):
            pass
class Rectangle(Shape):
    def __init__(self,width,height):
        self.width = width
        self.height = height
    def area(self):
      return self.width * self.height
rectangle = Rectangle(10, 5)
print("Rectangle area:", rectangle.area())
```

Output:

Result:

Successfully executed Python programs using core data structures and OOP principles, demonstrating their use in solving real world problems effectively.

Week: 4

Exercise 1: Pigeonhole Sort Algorithm

Aim: To implement the Pigeonhole Sort algorithm to sort an array of integers.

Procedure:

1. Define the range of values in the array to determine the size of pigeonholes.
2. Create an array of pigeonholes with a size equal to the range of input values.
3. Populate the pigeonholes by incrementing the corresponding hole for each element.
4. Traverse the pigeonholes to retrieve sorted elements.
5. Output the sorted array by placing elements back into the original array.

Program:

```
def pigeonhole_sort(a):
    # size of range of values in the list
    # (ie, number of pigeonholes we need)
    my_min = min(a)
    my_max = max(a)
    size = my_max - my_min + 1
    # our list of pigeonholes
    holes = [0] * size
    # Populate the pigeonholes.
    for x in a:
        assert type(x) is int, "integers only please"
        holes[x - my_min] += 1
    # Put the elements back into the array in order.
    i = 0
    for count in range(size):
        while holes[count] > 0:
            holes[count] -= 1
            a[i] = count + my_min
            i += 1
```

```
a = [8, 3, 2, 7, 4, 6, 8]
print("Sorted order is : ", end =" ")
pigeonhole_sort(a)
for i in range(0, len(a)):
        print(a[i], end =" ")
```

Output:

Result:

Pigeonhole Sort distributes elements into 'holes' based on their value range and then reassembles them in sorted order. It is efficient when the range of elements is small compared to the number of elements.

Exercise 2: Breadth First Search (BFS) Algorithm

Aim: To implement the BFS algorithm for graph traversal.

Procedure:

1. Initialize an empty set for visited nodes and a queue with the starting node.
2. Visit nodes by dequeuing them from the queue and marking them as visited.
3. Traverse all unvisited adjacent nodes and enqueue them.
4. Repeat the process until the queue is empty.
5. Print the order of traversal.

Program:

```
#BFS algorithm in Python
import collections
# BFS algorithm
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
    while queue:
    # Dequeue a vertex from queue
    vertex=queue.popleft()
    print(str(vertex) + " ",end=" ")
    # If not visited, mark it as visited, and
    # enqueue it
    for neighbour in graph[vertex]:
        if neighbour not in visited:
        visited.add(neighbour)
        queue.append(neighbour)
```

```
 if __name__ == '__main__':

graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}

print("Following is Breadth First Traversal: ")

bfs(graph, 0)
```

Output:

Result:

BFS explores all nodes at the present depth level before moving on to nodes at the next depth level, ensuring that the shortest path in un-weighted graphs is found.

Week: 5

Exercise 1: Finding the n^{th} Ugly Number

Aim: To find the n^{th} ugly number, where ugly numbers are positive integers whose prime factors are limited to 2, 3, or 5.

Procedure:

1. Initialize a heap with the first ugly number (1).
2. Repeatedly extract the smallest element from the heap for n iterations.
3. Multiply the current ugly number by 2, 3, and 5 and push the results back into the heap.
4. Continue until the n^{th} ugly number is found.
5. Output the result.

Program:

```
import heapq

class Solution(object):

    #:type n: integer

    #:return type: integer

  def nth_Ugly_Number(self, n):

     ugly_num = 0

     heap = []

     heapq.heappush(heap, 1)

     for _ in range(n):

        ugly_num = heapq.heappop(heap)

        if ugly_num % 2 == 0:

           heapq.heappush(heap, ugly_num * 2)

        elif ugly_num % 3 == 0:

           heapq.heappush(heap, ugly_num * 3)
```

```
        else:
            heapq.heappush(heap, ugly_num * 2)
            heapq.heappush(heap, ugly_num * 3)
            heapq.heappush(heap, ugly_num * 5)
    return ugly_num
n = 7
S = Solution()
result = S.nth_Ugly_Number(n)
print("7th Ugly number:")
print(result)
n = 10
result = S.nth_Ugly_Number(n)
print("\n10th Ugly number:")
print(result)
```

Output:

Result:

This program finds the n^{th} 'ugly number' using a min-heap to track numbers that only have prime factors of 2, 3, or 5, efficiently generating these numbers in sequence.

Exercise 2: Sum of Series Using Recursion

Aim: To calculate the sum of the series 12 + 22 + 32 + ... + N2 using recursion.

Procedure:

1. Create a recursive function to calculate the sum of the squares up to a given number.
2. The base case returns 0 if the input number is 0.
3. For other cases, return the square of the number plus the recursive sum of the remaining series.
4. Input the value of N and call the recursive function.
5. Print the result.

Program:

```
def sum_of_square_series(number):
    if(number == 0):
        return 0
    else:
        return (number * number) + sum_of_square_series(number - 1)
num = int(input("Please Enter any Positive Number  : "))
total = sum_of_square_series(num)
print("The Sum of Series upto {0}  = {1}".format(num, total))
```

Output:

Result:

The program calculates the sum of squares recursively, adding each square to the accumulated sum of the series.

Week: 6 Finding k^{th} Largest Element in an Unsorted Array

Aim: To find the kth largest element in an unsorted array using a heap.

Procedure:

1. Initialize an empty heap.
2. Insert each element of the array into the heap with negative priority.
3. Pop elements from the heap until the k^{th} largest element is found.
4. Return the k^{th} largest element.
5. Output the result for different values of k.

Program:

```
import heapq
class Solution(object):
    def find_Kth_Largest(self, nums, k):
        :type nums: List[int]
        :type of k: int
        :return value type: int
        h = []
        for e in nums:
            heapq.heappush(h, (-e, e))
        for i in range(k):
            w, e = heapq.heappop(h)
            if i == k - 1:
                return e
arr_nums = [12, 14, 9, 50, 61, 41]
s = Solution()
result = s.find_Kth_Largest(arr_nums, 3)
```

```
print("Third largest element:",result)

result = s.find_Kth_Largest(arr_nums, 2)

print("\nSecond largest element:",result)

result = s.find_Kth_Largest(arr_nums, 5)

print("\nFifth largest element:",result)
```

Output:

Result:

This program uses a min-heap to find the k^{th} largest element in an unsorted array, providing efficient retrieval for large datasets.

Week: 7 Printing a Heap as a Tree-like Data Structure

Aim: To print a heap as a tree-like structure to visualize its hierarchy.

Procedure:

1. Define a function to print the heap in a tree-like format using the total width for spacing.
2. Use logarithmic calculations to determine the position of elements.
3. Traverse the heap and print each element in its respective position.
4. Input a sample heap using heap operations (insertion).
5. Output the visualized tree structure.

Program:

```
import math
from io import StringIO
#source https://bit.ly/38HXSoU
def show_tree(tree, total_width=60, fill=' '):
    """Pretty-print a tree.
    total_width depends on your input size"""
    output = StringIO()
    last_row = -1
    for i, n in enumerate(tree):
        if i:
            row = int(math.floor(math.log(i+1, 2)))
        else:
            row = 0
        if row != last_row:
            output.write('\n')
        columns = 2**row
```

```
        col_width = int(math.floor((total_width * 1.0) / columns))
        output.write(str(n).center(col_width, fill))
        last_row = row
    print (output.getvalue())
    print ('-' * total_width)
    return
#test
import heapq
heap = []
heapq.heappush(heap, 1)
heapq.heappush(heap, 2)
heapq.heappush(heap, 3)
heapq.heappush(heap, 4)
heapq.heappush(heap, 7)
heapq.heappush(heap, 9)
heapq.heappush(heap, 10)
heapq.heappush(heap, 8)
heapq.heappush(heap, 16)
heapq.heappush(heap, 14)
show_tree(heap)
```

Output:

Result:

The program visualizes a heap as a tree structure, displaying the hierarchical nature of heap elements in a clear, graphical format.

Week: 8 Drawing a Bear Using Turtle Graphics

Aim: To use the Python turtle graphics library to draw a simple bear.

Procedure:

1. Set the turtle's speed and background color.
2. Draw the bear's head and ears using filled circles.
3. Draw the bear's eyes and nose with smaller filled circles.
4. Draw the bear's smiling mouth using a semicircular curve.
5. Hide the turtle and display the drawing.

Program:

```
import turtle
def draw_bear():
    turtle.speed(2)
    turtle.bgcolor("lightblue")
    #Draw the bear’s head
    turtle.penup()
    turtle.goto(0, -100)
    turtle.pendown()
    turtle.color("brown")
    turtle.begin_fill()
    turtle.circle(80)
    turtle.end_fill()
    #Draw the bear’s ear
    turtle.penup()
    turtle.goto(-30, 50)
    turtle.pendown()
```

```
    turtle.color("brown")
    turtle.begin_fill()
    turtle.circle(20)
    turtle.end_fill()

    turtle.penup()
    turtle.goto(30, 50)
    turtle.pendown()
    turtle.color("brown")
    turtle.begin_fill()
    turtle.circle(20)
    turtle.end_fill()

  # Draw the bear’s eyes
    turtle.penup()
    turtle.goto(-20, 10)
    turtle.pendown()
    turtle.color("black")
    turtle.begin_fill()
    turtle.circle(8)
    turtle.end_fill()

    turtle.penup()
    turtle.goto(20, 10)
    turtle.pendown()
```

```
    turtle.color("black")
    turtle.begin_fill()
    turtle.circle(8)
    turtle.end_fill()
    #Draw the bear's nose
    turtle.penup()
    turtle.goto(0, -5)
    turtle.pendown()
    turtle.color("black")
    turtle.begin_fill()
    turtle.circle(3)
    turtle.end_fill()
    #Draw the bear's Smiling mouth
    turtle.penup()
    turtle.goto(-20, -20)
    turtle.pendown()
    turtle.color("black")
    turtle.width(3)
    turtle.right(90)
    turtle.circle(20, 180)  #    #Draw the semicircle for a smile
   # Hide the turtle
    turtle.hideturtle()
    # Keep the window open
    turtle.done()
#Draw the bear
```

```
draw_bear()
```

Output:

Result:

The turtle graphics program uses basic shapes to draw a cartoon bear, demonstrating the use of turtle commands for graphical representation.

Week: 9 Finding Common Items from Two Dictionaries

Aim: To find common keys between two dictionaries.

Procedure:

1. Input keys and values for two dictionaries from the user.
2. Convert the keys of both dictionaries into sets.
3. Use set intersection to find common keys between the two dictionaries.
4. Output the common keys.
5. Display the result.

Program:

```
keys1 =[]
vals1 = []
num1 = int(input("Enter the Number of elements for Dictionary-1: "))
print("Enter Values for Keys")
for i in range(0, num1):
    x = str(input("Enter Key " + str(i + 1) + "="))
    keys1.append(x)
print("Enter Values for Values")
for i in range(0, num1):
    x = str(input("Enter Value "+ str(i + 1) +"="))
    vals1.append(x)
dict1 = dict(zip(keys1, vals1))
print("Dictionary_1 Items:", dict1)
set1 = set(dict1.keys())
keys2 = []
vals2 =[]
```

```
num2 = int(input("Enter the Number of elements for Dictionary-2: "))
print("Enter Values for Keys")
for i in range(0, num2):
    y = str(input("Enter Key " + str(i + 1) + "="))
    keys2.append(y)
print("Enter Values for Values")
for i in range(0, num2):
    y = str(input("Enter Value " + str(i + 1) + "="))
    vals2.append(y)
dict2= dict(zip(keys2, vals2))
print("Dictionary_2 Items:", dict2)
set2 = set(dict2.keys())
common_items = set1 & set2
print("Common items from dictionary 1 and dictionary 2:", common_items)
```

Output:

Result:

This program finds and displays common keys from two dictionaries by converting them to sets and using set intersection.

Week: 10 Rock, Paper, Scissors Game

Aim: To create a game where the user plays rock, paper, scissors against the computer.

Procedure:

1. Define possible choices: rock, paper, and scissors.
2. Get the user's choice and randomly select the computer's choice.
3. Compare the two choices based on the rules of the game.
4. Determine and print the winner or declare a tie.
5. Repeat the process to play the game.

Program:

```
import random
def play_game():
    choices = ["rock", "paper", "scissors"]
    #Get user input for their choice
    user_choice = input("Enter your choice (rock, paper, or scissors): ").lower()
    # Validate user input
    if user_choice not in choices:
        print("Invalid choice. Please choose rock, paper, or scissors.")
        return
    # Computer randomly selects its choice
    computer_choice = random.choice(choices)
    # Display choices
    print(f"\nYou chose: {user_choice}")
    print(f"Computer chose: {computer_choice}\n")
    #Determine the winner
    if user_choice == computer_choice:
```

```
        print("It's a tie!")
    elif (
        (user_choice =="rock" and computer_choice == "scissors") or
        (user_choice=="paper" and computer_choice =="rock") or
        (user_choice =="scissors" and computer_choice== "paper")
    ):
        print("You win!")
    else:
        print("Computer wins!")
if __name__=="__main__":
    play_game()
import random
def play_game():
    choices = ["rock", "paper", "scissors"]
    #Get user input for their choice
    user_choice = input("Enter your choice (rock, paper, or scissors): ").lower()
    # Validate user input
    if user_choice not in choices:
        print("Invalid choice. Please choose rock, paper, or scissors.")
        return
    # Computer randomly selects its choice
    computer_choice = random.choice(choices)
    # Display choices
    print(f"\nYou chose: {user_choice}")
    print(f"Computer chose: {computer_choice}\n")
```

```
    #Determine the winner
    if user_choice == computer_choice:
        print("It's a tie!")
    elif (
        (user_choice =="rock" and computer_choice == "scissors") or
        (user_choice=="paper" and computer_choice =="rock") or
        (user_choice =="scissors" and computer_choice== "paper")
    ):
        print("You win!")
    else:
        print("Computer wins!")
if __name__=="__main__":
    play_game()
```

Output:

Result:

The program simulates a rock-paper-scissors game where a user plays against the computer, randomly selecting from the three choices.

P24DS1P1

PG Department of Data Science

Bishop Heber College(Autonomous)

Tiruchirappalli-620017