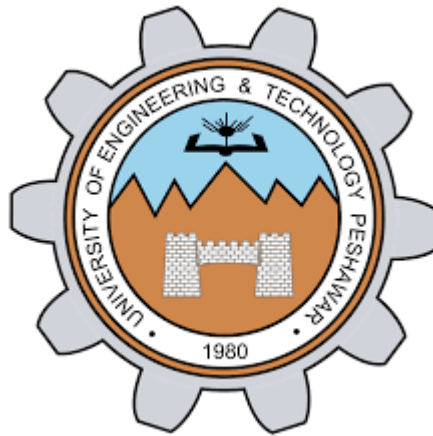


JANUARY 1, 2025



OOP & DATA STRUCTURE LAB REPORT TASKS

SYED MUHAMMAD ISMAEEL KAKAKHEL

21JZELE0424

OOP& DATA STRUCTURE

LAB REPORT TASK 1

1- OBJECTIVES:

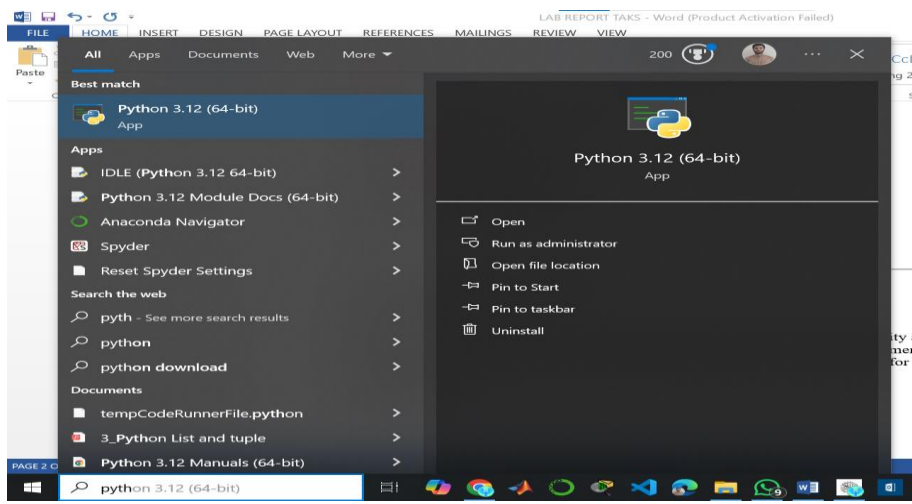
1. Install IDE: Recommended ide python or anaconda.
 2. Make a GitHub account
 3. Introduction to Helping websites
 4. Introduction Jupyter Notebook
 5. Introduction Virtual environments
 6. Introduction online code editor
 7. Courses
-

1-Installation of Python and Anaconda:

Introduction to Python

Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used in various fields such as data science, web development, machine learning, and more. Python's syntax is easy to learn, making it a popular choice for beginners and professionals alike.

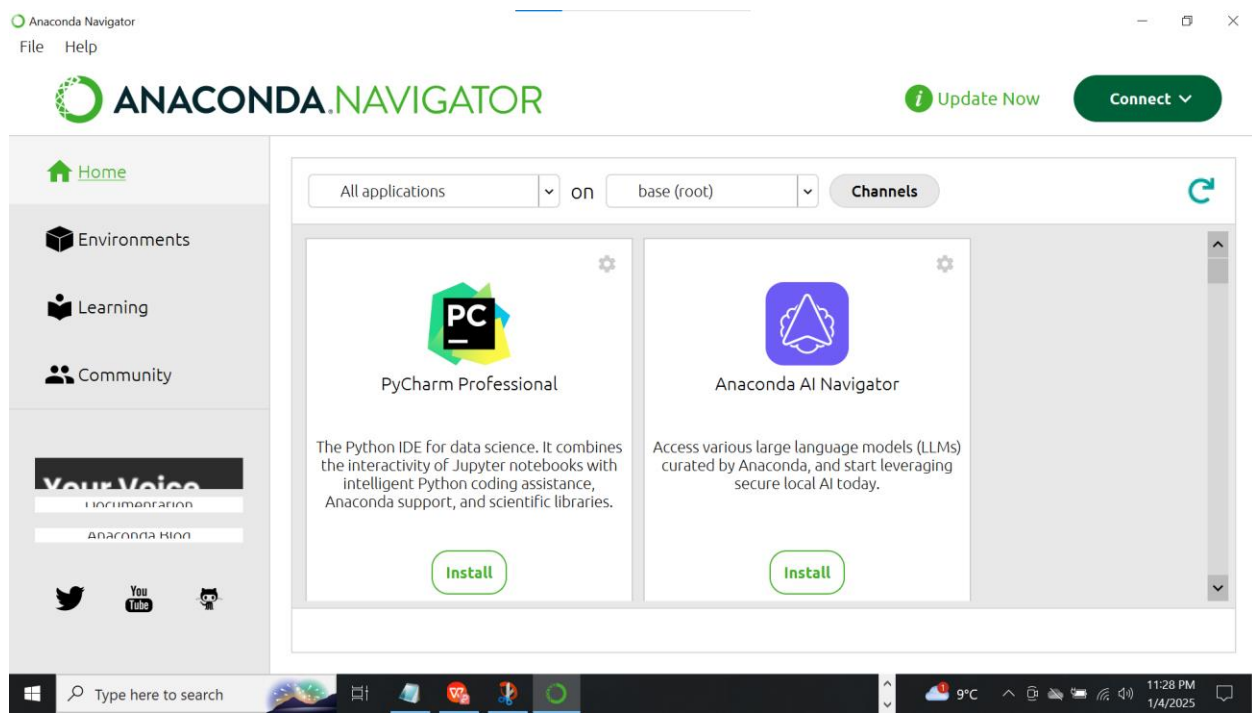
INSTALLED:



Introduction to Anaconda

Anaconda is an open-source distribution of Python and R designed for data science, machine learning, and scientific computing. It comes with many pre-installed libraries and tools, making it easier to set up and manage Python environments.

INSTALLED VIEW:

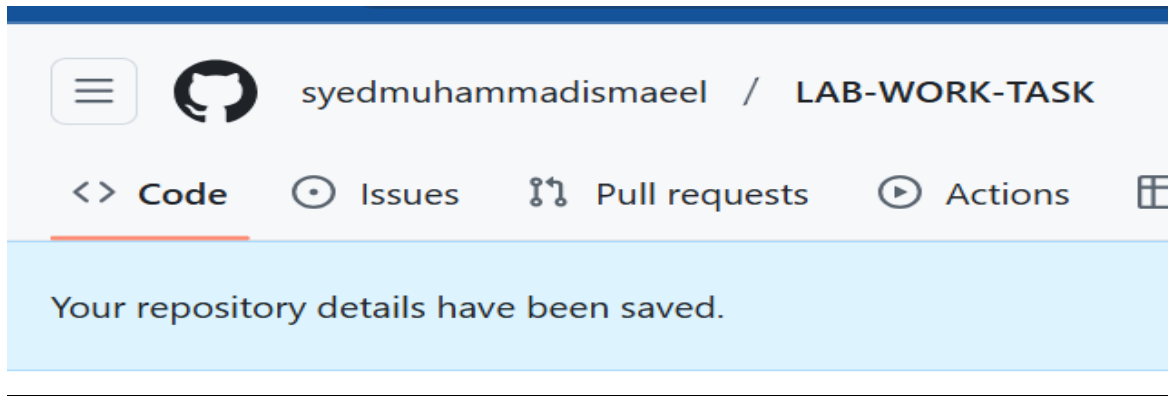


2-Making of GITHUB Account:

GitHub is a web-based platform for version control and collaboration. It allows developers to store, manage, track, and share their code repositories. Built on top of **Git**, which is a version control system, GitHub makes it easy for teams to work on the same project and keep track of changes.

GITHUB ACCOUNT:

GitHub is a fantastic tool for learning, collaborating, and showcasing your programming skills. By mastering it, you'll improve your workflow and increase your chances of contributing to exciting projects!



3-Helping websites:

1-Introduction to stack overflow:

Overflow generally refers to a situation where a system or device exceeds its capacity to handle or store data properly. In the context of computing and programming, overflow occurs when the value being processed exceeds the maximum or minimum value that can be represented within a given data type or memory storage.

2-Real python:

Real Python is a widely recognized online platform and community that provides comprehensive resources for learning Python programming. It caters to beginners, intermediate, and advanced programmers, offering tutorials, articles, videos, and other educational materials to help individuals master Python programming concepts.

4-Introduction to Jupyter Notebook

Jupyter Notebook is an open-source web application that allows users to create and share documents containing live code, equations, visualizations, and explanatory text. It is widely used for data analysis, machine learning, data visualization, and teaching purposes.

5-Introduction to virtual Environment:

Virtual environments are an essential tool for Python developers. They allow you to work on multiple projects simultaneously without worrying about dependency conflicts or modifying the global Python installation. By mastering virtual environments, you can maintain a cleaner, more organized development workflow and ensure your projects remain stable and reproducible.

6-Introduction to online code editor:

An online code editor is a web-based application that allows users to write, edit, and execute code directly in their web browser without the need to install software or set up a development environment on their local machines. These editors are designed to simplify programming, making it accessible and convenient for learners, professionals, and hobbyists.

7-COURSE CERTIFICATE ON PYTHON:

LAB REPORT TASK 2

LAB TASK:

Define a function that will take the parameters as current and resistance and will output the voltage across the resistance.

- a. The argument pass will like positional arguments.
 - b. The arguments passing will like keyword arguments.
 - c. The arguments passing will like default arguments.
 - d. Pass the arguments of one style to the other.
 - e. Passing less number of arguments and check the error message.
 - f. Verify that whether positional argument follows keyword argument.
-

1-Positional arguments: Arguments are passed in the order defined by the function.

2-Keyword arguments: Arguments are passed with their names explicitly.

3-Default arguments: The function provides default values if the arguments are not passed.

4-Passing arguments of one style to another: The code demonstrates how to pass keyword arguments to a function with default values.

5-Passing fewer arguments: This will result in an error, as the function expects two arguments.

6-Positional arguments followed by keyword arguments: This will raise an error since positional arguments must come before keyword arguments in function calls.

CODE:

```
[1]: # Function to calculate voltage using Ohm's Law
def calculate_voltage(current, resistance):
    voltage = current * resistance
    return voltage

# a. Using positional arguments
print("Positional Arguments:")
current = 5 # Current in Amps
resistance = 10 # Resistance in Ohms
print(f"Voltage (V) = {calculate_voltage(current, resistance)} V")

# b. Using keyword arguments
print("\nKeyword Arguments:")
print(f"Voltage (V) = {calculate_voltage(current=5, resistance=10)} V")

# c. Using default arguments
def calculate_voltage_with_default(current=5, resistance=10):
    voltage = current * resistance
    return voltage

print("\nDefault Arguments:")
print(f"Voltage (V) = {calculate_voltage_with_default()} V") # Uses default values
```

```
# d. Pass the arguments of one style to the other
print("\nPass Arguments of One Style to Another:")
# Passing keyword arguments to the default function
print(f"Voltage (V) = {calculate_voltage_with_default(current=8, resistance=15)} V")

# e. Passing Less number of arguments and check the error message
print("\nPassing Less Number of Arguments:")
try:
    print(f"Voltage (V) = {calculate_voltage(5)}") # Missing one argument
except TypeError as e:
    print(f"Error: {e}")

# f. Verify whether positional argument follows keyword argument
print("\nPositional Argument Followed by Keyword Argument:")
try:
    print(f"Voltage (V) = {calculate_voltage(5, resistance=10)}") # Positional argument followed by keyword argument
except SyntaxError as e:
    print(f"Error: {e}")
```

OUTPUT:

```
Positional Arguments:
Voltage (V) = 50 V

Keyword Arguments:
Voltage (V) = 50 V

Default Arguments:
Voltage (V) = 50 V

Pass Arguments of One Style to Another:
Voltage (V) = 120 V

Passing Less Number of Arguments:
Error: calculate_voltage() missing 1 required positional argument: 'resistance'

Positional Argument Followed by Keyword Argument:
Voltage (V) = 50
```

LAB REPORT TASK 3

LAB TASK:

1. Make a list that contains the student name, the last four digits of the registration number as an integer, the CGPA as a float, and a list that contains GPA of all semesters. **(Nested List)**

2. Use list indexing in the following nested list and access each and every element.

x = ["a", ["bb", ["ccc", "ddd"], "ee", "ff"], "g", ["hh", "ii"], "j"]

TASK 1:

CODE:

```
[2]: # Creating a List with student details
student_info = [
    "John Doe", # Student name
    1234, # Last four digits of registration number
    3.85, # CGPA as a float
    [3.7, 3.8, 3.9, 4.0, 3.6] # List of GPA for all semesters
]

# Printing the nested List
print(student_info)
```

RESULT:

```
['John Doe', 1234, 3.85, [3.7, 3.8, 3.9, 4.0, 3.6]]
```



1. |

TASK 2:

CODE:

```
# Given nested list
x = ["a", ["bb", ["ccc", "ddd"], "ee", "ff"], "g", ["hh", "ii"], "j"]

# Accessing each element using list indexing
print(x[0]) # 'a'

# Accessing the second element which is a list
print(x[1]) # ['bb', ['ccc', 'ddd'], 'ee', 'ff']

# Accessing elements in the second sublist
print(x[1][0]) # 'bb'
print(x[1][1]) # ['ccc', 'ddd']
print(x[1][2]) # 'ee'
print(x[1][3]) # 'ff'

# Accessing elements in the nested list inside the second sublist
print(x[1][1][0]) # 'ccc'
print(x[1][1][1]) # 'ddd'

# Accessing other elements in the main list
print(x[2]) # 'g'
print(x[3]) # ['hh', 'ii']

# Accessing elements in the fourth sublist
print(x[3][0]) # 'hh'
print(x[3][1]) # 'ii'

# Accessing the last element
print(x[4]) # 'j'
```

RESULT:

```
a
['bb', ['ccc', 'ddd'], 'ee', 'ff']
bb
['ccc', 'ddd']
ee
ff
ccc
ddd
g
['hh', 'ii']
hh
ii
j
```

LAB REPORT TASK 4

LAB TASK:

Consider the point class below

1-Make an object and print its x and y coordinates.

```
import math
class Point:
    """Represents a point in two-dimensional geometric
    coordinates
    Parameters
    -----
    x : float
    y : float
    """
    def __init__(self, x: float, y: float)->None:
        self.x = x
        self.y = y
    def distance(self, p2)->float:
        return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)
```

a-Define point1 and pass two numbers.

b. Make another instance of the point class, say its name is p2.

c. Print p1 and p2, for this use the print command and pass the point as input.

d. Print the coordinate by using the object and dot operator.

e. Add a new method to the point class that can effectively print the points. Print both the point using that function.

f. Calculate the distance between these two pints.

TASK 1 :**CODE:**

```
import math

class Point:
    """Represents a point in two-dimensional geometric coordinates

    Parameters
    -----
    x : float
    y : float
    """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, p2) -> float:
        """Returns the distance between two points"""
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

# Creating a Point object
point1 = Point(3.0, 4.0)
```

CODE OUTPUT:

```
Point coordinates: (3.0, 4.0)
```

TASK 2:

a-Define point1 and pass two numbers.

CODE:

```
# Define the Point class
import math

class Point:
    """Represents a point in two-dimensional geometric coordinates

    Parameters
    -----
    x : float
    y : float
    """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, p2) -> float:
        """Returns the distance between two points"""
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

# Define point1 and pass two numbers (x=3.0, y=4.0)
point1 = Point(3.0, 4.0)

# Printing point1's coordinates
print(f"point1 coordinates: ({point1.x}, {point1.y})")
```

RESULT OF CODE:

```
point1 coordinates: (3.0, 4.0)
```

```
1: |
```

Passing Two Numbers: In this case, `x_value` is set to 5.0 and `y_value` is set to 7.0. These values are passed as arguments to create the `point1` object with coordinates (5.0, 7.0).

TASK 3:

Make another instance of the point class, say its name is p2.

CODE:

```
# Define the Point class
import math

class Point:
    """Represents a point in two-dimensional geometric coordinates

    Parameters
    -----
    x : float
    y : float
    """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, p2) -> float:
        """Returns the distance between two points"""
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

# Define point1 by passing two numbers (x and y)
x_value = 5.0 # First number (x-coordinate)
y_value = 7.0 # Second number (y-coordinate)
```

OUTPUT OF CODE:

```
point1 coordinates: (5.0, 7.0)
p2 coordinates: (2.0, 3.0)
```

TASK 4:

Print p1 and p2, for this use the print command and pass the point as input

CODE:

```
# Define the Point class
import math

class Point:
    """Represents a point in two-dimensional geometric coordinates

    Parameters
    -----
    x : float
    y : float
    """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, p2) -> float:
        """Returns the distance between two points"""
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

    def __str__(self) -> str:
        """Override the __str__ method to provide a custom string representation for printing"""
        return f"({self.x}, {self.y})"
```

OUTPUT OF CODE:

```
point1: (5.0, 7.0)
p2: (2.0, 3.0)
```

TASK 5:

Add a new method to the point class that can effectively print the points. Print both the point using that function.

CODE:

```
# Define the Point class
import math

class Point:
    """Represents a point in two-dimensional geometric coordinates

    Parameters
    -----
    x : float
    y : float
    """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, p2) -> float:
        """Returns the distance between two points"""
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

    def print_point(self) -> None:
        """Prints the coordinates of the point"""
        print(f"Point coordinates: ({self.x}, {self.y})")
```

OUTPUT OF CODE:

```
Using print_point method:
Point coordinates: (5.0, 7.0)
Point coordinates: (2.0, 3.0)
```

TASK 6:

Calculate the distance between these two points.

CODE:

```
# Define the Point class
import math

class Point:
    """Represents a point in two-dimensional geometric coordinates

    Parameters
    -----
    x : float
    y : float
    """

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def distance(self, p2) -> float:
        """Returns the distance between two points"""
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

    def print_point(self) -> None:
        """Prints the coordinates of the point"""
        print(f"Point coordinates: ({self.x}, {self.y})")

# Define point1 by passing two numbers (x and y)
x value = 5.0 # First number (x-coordinate)
```

OUTPUT OF CODE:

The distance between point1 and p2 is: 5.0

CONCLUSION:

By performing these steps, we demonstrated how to define classes, work with object instances, access attributes, implement methods, and perform calculations such as distance computation in Python.

LAB REPORT TASK 5

LAB TASK:

Define a class circle. Your class must have the appropriate `__init__` method.

- i. Add appropriate property methods. (`@property` and `@property.setter`)
 - ii. In addition, add an instance method for the volume of a cylinder with the given radius.
 - iii. Add property method for area, circumference, and diameter.
 - iv. Add `__repr__` and `__str__` to the class.
 - v. Add proper annotation and doc string to every class and instance method.
 - a. Define `inst_1` and pass two numbers.
 - b. Make another instance `inst2`.
 - c. Print `inst_1` and `inst_2`, for this use the `print` command and pass the `inst_1` and `inst_2`.
 - d. Call the `__dict__` by the class name.
 - e. Also pass the class name to the `vars` built-in function.
 - f. Call the `__dict__` on the object of the class.
 - g. Pass the class name to `help`.
 - h. Print the doc-string and annotations of both the class and each instance method.
 - i. Modify the `__init__` by making its parameters default and verify by instances.
-

TASK 1:

INTRODUCTION TO CLASS CIRCLE:

A class `Circle` is a blueprint or template in object-oriented programming (OOP) that defines how a circle object should behave and what data it should store. In Python, a class is used to define objects with attributes (properties) and methods (functions). The `Circle` class represents a circle, and its behavior might include storing information about its radius and providing methods to calculate properties like the area or circumference of the circle.

CODE FOR CLASS:

```
class Circle:
    """Represents a circle with a given radius.

    Parameters
    -----
    radius : float
        The radius of the circle.
    """

    def __init__(self, radius: float) -> None:
        """Initializes a circle object with a specific radius."""
        self.radius = radius

# Example of creating a Circle object
circle1 = Circle(7.0)

# Printing the radius of the circle
print(f"Radius of circle1: {circle1.radius}")
```

OUTPUT OF CODE:

```
Radius of circle1: 7.0
```

TASK 2:

Add appropriate property methods. (@property and @property.setter)

DEFINITION TO @Property and @property.setter:

In Python, we can use the @property decorator to define a method as a property, which allows us to access it as if it were an attribute, while still using a method to calculate or return its value. The @property.setter decorator allows us to set the value of the property in a controlled manner.

CODE:

```
import math

class Circle:
    """Represents a circle with a given radius."""

    def __init__(self, radius: float) -> None:
        """Initializes the circle with a radius."""
        self._radius = radius

    @property
    def radius(self) -> float:
        """Gets the radius of the circle."""
        return self._radius

    @radius.setter
    def radius(self, value: float) -> None:
        """Sets the radius of the circle ensuring it's positive."""
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self) -> float:
```

OUTPUT OF CODE:

```

Area of circle1: 78.53981633974483
Circumference of circle1: 31.41592653589793
Updated area of circle1: 314.1592653589793
Updated circumference of circle1: 62.83185307179586
Error: Radius cannot be negative

```

TASK 3:

. In addition, add an instance method for the volume of a cylinder with the given radius.

CODE:

```

import math

class Circle:
    """Represents a circle with a given radius."""

    def __init__(self, radius: float) -> None:
        """Initializes the circle with a radius."""
        self._radius = radius

    @property
    def radius(self) -> float:
        """Gets the radius of the circle."""
        return self._radius

    @radius.setter
    def radius(self, value: float) -> None:
        """Sets the radius of the circle ensuring it's positive."""
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self) -> float:
        """Returns the area of the circle."""

```

OUTPUT OF CODE:

```

Area of circle1: 78.53981633974483
Circumference of circle1: 31.41592653589793
Volume of cylinder with radius 5 and height 10: 785.3981633974483
Updated volume of cylinder with radius 7 and height 12: 1847.2564803107982

```

TASK 4:

Add property method for area, circumference, and diameter.

CODE:

```
import math

class Circle:
    """Represents a circle with a given radius."""

    def __init__(self, radius: float) -> None:
        """Initializes the circle with a radius."""
        self._radius = radius

    @property
    def radius(self) -> float:
        """Gets the radius of the circle."""
        return self._radius

    @radius.setter
    def radius(self, value: float) -> None:
        """Sets the radius of the circle ensuring it's positive."""
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self) -> float:
```

OUTPUT OF CODE:

```
Area of circle1: 78.53981633974483
Circumference of circle1: 31.41592653589793
Diameter of circle1: 10
Volume of cylinder with radius 5 and height 10: 785.3981633974483
Updated area of circle1: 153.93804002589985
Updated circumference of circle1: 43.982297150257104
Updated diameter of circle1: 14
```

TASK 5:

Add `__repr__` and `__str__` to the class.

CODE:

```

import math

class Circle:
    """Represents a circle with a given radius."""

    def __init__(self, radius: float) -> None:
        """Initializes the circle with a radius."""
        self._radius = radius

    @property
    def radius(self) -> float:
        """Gets the radius of the circle."""
        return self._radius

    @radius.setter
    def radius(self, value: float) -> None:
        """Sets the radius of the circle ensuring it's positive."""
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self) -> float:
        """Returns the area of the circle."""
        return math.pi * (self._radius ** 2)

```

OUTPUT OF CODE:

```

Circle with radius 5, area 78.54, and circumference 31.42
Circle(radius=5)
Area of circle1: 78.53981633974483
Circumference of circle1: 31.41592653589793
Diameter of circle1: 10
Volume of cylinder with radius 5 and height 10: 785.3981633974483
Updated area of circle1: 153.93804002589985
Updated circumference of circle1: 43.982297150257104
Updated diameter of circle1: 14

```

TASK 6:

Add proper annotation and doc string to every class and instance method.

CODE:

```
# Create a circle object with a radius of 5
circle1 = Circle(5)

# Access and print area, circumference, and diameter
print(f"Area of circle1: {circle1.area}")
print(f"Circumference of circle1: {circle1.circumference}")
print(f"Diameter of circle1: {circle1.diameter}")

# Calculate and print the volume of the cylinder with the circle's radius and a height of 10
height = 10
print(f"Volume of cylinder with radius {circle1.radius} and height {height}: {circle1.volume_of_cylinder(height)}")

# Change the radius and recalculate
circle1.radius = 7
print(f"Updated area of circle1: {circle1.area}")
print(f"Updated circumference of circle1: {circle1.circumference}")
print(f"Updated diameter of circle1: {circle1.diameter}")
```

OUTPUT OF CODE:

```
Area of circle1: 78.53981633974483
Circumference of circle1: 31.41592653589793
Diameter of circle1: 10
Volume of cylinder with radius 5 and height 10: 785.3981633974483
Updated area of circle1: 153.93804002589985
Updated circumference of circle1: 43.982297150257104
Updated diameter of circle1: 14
```

TASK 7:

Define inst_1 and pass two numbers

CODE:

```
# Assuming Circle class is already defined as before

# Define inst_1 by passing a radius and an optional height for volume calculation
inst_1 = Circle(5) # Passing 5 as the radius for the circle

# Now we can access attributes and methods
print(inst_1) # Calls the __str__ method to print the circle details
print(f"Volume of cylinder with radius {inst_1.radius} and height 10: {inst_1.volume_of_cylinder(10)}")
```

OUTPUT OF CODE:

```
Circle with radius 5, area 78.54, and circumference 31.42
Volume of cylinder with radius 5 and height 10: 785.3981633974483
```

TASK 8:

```
# Assuming Circle class is already defined as before

# Define inst_1 by passing a radius of 5
inst_1 = Circle(5)

# Define inst_2 by passing a radius of 7
inst_2 = Circle(7)

# Now we can access attributes and methods for both instances
print("Details of inst_1:")
print(inst_1) # Calls the __str__ method to print inst_1 details
print(f"Volume of cylinder with radius {inst_1.radius} and height 10: {inst_1.volume_of_cylinder(10)}")

print("\nDetails of inst_2:")
print(inst_2) # Calls the __str__ method to print inst_2 details
print(f"Volume of cylinder with radius {inst_2.radius} and height 10: {inst_2.volume_of_cylinder(10)}")
```

OUTPUT OF CODE:

```
Details of inst_1:
Circle with radius 5, area 78.54, and circumference 31.42
Volume of cylinder with radius 5 and height 10: 785.3981633974483

Details of inst_2:
Circle with radius 7, area 153.94, and circumference 43.98
Volume of cylinder with radius 7 and height 10: 1539.3804002589986
```

TASK 9:

Print inst_1 and inst_2, for this use the print command and pass the inst_1 and inst_2.

CODE:

```
# Assuming Circle class is already defined as before

# Define inst_1 by passing a radius of 5
inst_1 = Circle(5)

# Define inst_2 by passing a radius of 7
inst_2 = Circle(7)

# Print the details of inst_1 and inst_2 using the print command
print(inst_1) # Prints the details of inst_1
print(inst_2) # Prints the details of inst_2
```


OUTPUT OF CODE:

Circle with radius 5, area 78.54, and circumference 31.42
 Circle with radius 7, area 153.94, and circumference 43.98

TASK 10:

Call the `__dict__` by the class name.

CODE:

```
# Assuming Circle class is already defined as before

# Call the __dict__ of the class Circle to see its attributes and methods
print(Circle.__dict__)
```

OUTPUT OF CODE:

```
{'__module__': '__main__', '__doc__': 'Represents a circle with a given radius.\n\n Parameters\n -----\n radius : float\n\n The radius of the circle.\n\n Attributes\n -----\n radius : float\n\n The radius of the circle.\n\n ', '__init__': <function Circle.__init__ at 0x000001F11C0DFB00>, 'radius': <property object at 0x000001F11C4C1080>, 'area': <property object at 0x000001F11C4C0B30>, 'circumference': <property object at 0x000001F11C4C0A40>, 'diameter': <property object at 0x000001F11C4C0A90>, 'volume_of_cylinder': <function Circle.volume_of_cylinder at 0x000001F11C4A9620>, '__repr__': <function Circle.__repr__ at 0x000001F11C4A96C0>, '__str__': <function Circle.__str__ at 0x000001F11C4A9760>, '__dict__': <attribute '__dict__' of 'Circle' objects>, '__weakref__': <attribute '__weakref__' of 'Circle' objects>}
```

TASK 11:

Also pass the class name to the `vars` built-in function.

CODE:

```
# Assuming Circle class is already defined as before

# Call the vars() function on the Circle class to see its attributes and methods
print(vars(Circle))
```

OUTPUT OF CODE:

TASK 12:

Call the `__dict__` on the object of the class.

CODE:

```
# Assuming Circle class is already defined as before

# Define inst_1 and inst_2 by passing radius values
inst_1 = Circle(5)
inst_2 = Circle(7)

# Access the __dict__ of inst_1 (instance of Circle class)
print(inst_1.__dict__)

# Access the __dict__ of inst_2 (another instance of Circle class)
print(inst_2.__dict__)
```

OUTPUT OF CODE;

```
{ '_radius': 5 }  
{ '_radius': 7 }
```

TASK 13:

Pass the class name to help

CODE:

```
# Assuming Circle class is already defined as before

# Get the help documentation for the Circle class
help(Circle)
```

OUTPUT:

Help on class Circle in module __main__:

```
class Circle(builtins.object)
| Circle(radius: float) -> None
|
| Represents a circle with a given radius.
|
| Parameters
| -----
| radius : float
|     The radius of the circle.
|
| Attributes
| -----
| radius : float
|     The radius of the circle.
```

TASK 14:

Print the doc-string and annotations of both the class and each instance method

CODE:

```
# Assuming Circle class is already defined as before

# Print the docstring for the Circle class
print("Class Docstring:")
print(Circle.__doc__)

# Print the annotations for the Circle class
print("\nClass Annotations:")
print(Circle.__annotations__)

# Print the docstring and annotations for each method
print("\nMethod Docstrings and Annotations:")

# Iterate through the methods in the Circle class
for method_name in dir(Circle):
    method = getattr(Circle, method_name)

    # Print the method's docstring if available
    if callable(method):
        print(f"\nMethod: {method_name}")
        print("Docstring:", method.__doc__)
        print("Annotations:", method.__annotations__)
```

OUTPUT OF CODE:

TASK 15:

Modify the `__init__` by making its parameters default and verify by instances

CODE:

```
class Circle:
    """Represents a circle with a given radius."""

    def __init__(self, radius: float = 1.0) -> None:
        """Initialize the circle with the given radius. Default radius is 1.0."""
        self._radius = radius

    @property
    def radius(self) -> float:
        """Getter for the radius."""
        return self._radius

    @radius.setter
    def radius(self, value: float) -> None:
        """Setter for the radius."""
        if value <= 0:
            raise ValueError("Radius must be a positive value.")
        self._radius = value

    def __repr__(self) -> str:
```

OUTPUT OF CODE:

```
Circle with radius 5
Circle with radius 1.0
```

CONCLUSION:

In the tasks performed, we explored various concepts related to class and instance methods in Python. We began by defining a `Circle` class with an `__init__` method, incorporating default parameters for the radius. The class was enhanced with property methods using `@property` and `@property.setter` to control access to the radius attribute, as well as methods to calculate the area, circumference, diameter, and volume of a cylinder. We also implemented `__repr__` and `__str__` methods to provide string representations of the class and its instances. Additionally, we examined how to access the class and instance-level docstrings and annotations using the `__doc__` and `__annotations__` attributes. Instances of the `Circle` class were created with and without passing values to the `radius` parameter, confirming that the default value of `1.0` was used when no radius was provided. Overall, these tasks helped deepen the understanding of object-oriented programming in Python, focusing on class design, encapsulation, and method functionality.

LAB 6

Problem 1:

a. Take the example of the square class in the last lab and inherit a cube class with its own volume and surface area instance method.

CODE:

```
[1]: class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

    def perimeter(self):
        return 4 * self.side

class Cube(Square):
    def __init__(self, side):
        super().__init__(side) # Inherit the side attribute from Square

    def volume(self):
        return self.side ** 3 # Volume of cube = side³

    def surface_area(self):
        return 6 * super().area() # Surface area = 6 * area of one face

# Example Usage
square = Square(4)
print("Square Area:" square.area())
```

OUTPUT:

```
Square Area: 16
Square Perimeter: 16
Cube Volume: 64
Cube Surface Area: 96
```

b. Create two instances of the cube class and call both the instance methods

CODE:

```
class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

    def perimeter(self):
        return 4 * self.side

class Cube(Square):
    def __init__(self, side):
        super().__init__(side) # Inherit the side attribute from Square

    def volume(self):
        return self.side ** 3 # Volume of cube = side³

    def surface_area(self):
        return 6 * super().area() # Surface area = 6 * area of one face

# Creating two instances of Cube
cube1 = Cube(3)
cube2 = Cube(5)
```

OUTPUT:

```
Cube 1:
Side Length: 3
Volume: 27
Surface Area: 54

Cube 2:
Side Length: 5
Volume: 125
Surface Area: 150
```

Problem 2:

- a. Take the example of the square class in the last lab and inherit a rectangle class.

CODE:

```
class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

    def perimeter(self):
        return 4 * self.side

class Rectangle(Square):
    def __init__(self, length, width):
        super().__init__(length) # Using `length` as `side` in the parent class
        self.width = width

    def area(self): # Override area method
        return self.side * self.width # Rectangle area = length × width

    def perimeter(self): # Override perimeter method
        return 2 * (self.side + self.width) # Rectangle perimeter = 2(l + w)
```

OUTPUT:

```
Rectangle Area: 24
Rectangle Perimeter: 20
```

- b. Create two instances of the rectangle class and call the instance methods.

CODE:

```

class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

    def perimeter(self):
        return 4 * self.side

class Rectangle(Square):
    def __init__(self, length, width):
        super().__init__(length) # Using `length` as `side` in the parent class
        self.width = width

    def area(self): # Override area method
        return self.side * self.width # Rectangle area = length x width

    def perimeter(self): # Override perimeter method
        return 2 * (self.side + self.width) # Rectangle perimeter = 2(l + w)

```

Creating two instances of Rectangle

OUTPUT:

```

Rectangle 1:
Length: 4
Width: 6
Area: 24
Perimeter: 20

```

```

Rectangle 2:
Length: 5
Width: 8
Area: 40
Perimeter: 26

```

Problem 3:

a. Take the example of the Point2D class in the last lab and inherit a cube class with its own distance_from_origin instance methods.


```
import math

class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.sqrt(self.x**2 + self.y**2) # Distance formula  $\sqrt{x^2 + y^2}$ 

class Cube(Point2D):
    def __init__(self, x, y, z, side):
        super().__init__(x, y) # Inheriting x and y coordinates
        self.z = z # Adding third dimension
        self.side = side # Cube's side length
```

OUTPUT:

```
Point2D Distance from Origin: 5.0

Cube Distance from Origin: 7.0710678118654755
Cube Volume: 216
Cube Surface Area: 216
```

B Create two instances and call the instance methods.

CODE:

```
import math

class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.sqrt(self.x**2 + self.y**2) # Distance formula  $\sqrt{x^2 + y^2}$ 

class Cube(Point2D):
    def __init__(self, x, y, z, side):
        super().__init__(x, y) # Inheriting x and y coordinates
        self.z = z # Adding third dimension
        self.side = side # Cube's side length

    def volume(self):
        return self.side ** 3 # Volume of a cube = side3
```

OUTPUT:

```
Cube 1:
Coordinates (x, y, z): (3, 4, 5)
Distance from Origin: 7.0710678118654755
Volume: 216
Surface Area: 216
```

```
Cube 2:
Coordinates (x, y, z): (7, 1, 2)
Distance from Origin: 7.3484692283495345
Volume: 64
Surface Area: 96
```

Problem 4:

The **HR system** needs to process payroll for the company's employees, but there are different types of employees depending on how their payroll is calculated.

1. Implement a base class, `Employee`, that handles the common interface for every employee.

CODE:

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

    def calculate_payroll(self):
        """This method should be implemented by subclasses."""
        raise NotImplementedError("Subclasses must implement calculate_payroll()")

    def __str__(self):
        return f"Employee Name: {self.name}, ID: {self.employee_id}"
```

The **HR system** needs to process payroll for the company's employees, but there are different types of employees depending on how their payroll is calculated.

1. Implement a base class, `Employee`, that handles the common interface for every employee.
2. Administrative workers have a fixed salary, so every week they get paid the same amount.
 - a) You create a derived class, `SalaryEmployee`, that inherits from `Employee`. The class initializes with the `.id` and `.name` required by the base class, and you use `super()` to initialize the members of the base class.
 - b) `SalaryEmployee` also requires a `weekly_salary` initialization parameter that represents the amount that the employee makes per week.
 - c) The class provides the required `.calculate_payroll()` method that the HR system uses. The implementation just returns the amount stored in `weekly_salary`.
3. The company also employs manufacturing workers who are paid by the hour, so you add `HourlyEmployee` to the HR system
 - a) You create a derived class, `HourlyEmployee`, that inherits from `Employee`. The class initializes with the `.id` and `.name` required by the base class, and you use `super()` to initialize the members of the base class.
 - b) `HourlyEmployee` also requires `hours_worked` and the `hourly_rate` required to calculate the payroll.
 - c) The class provides the required `.calculate_payroll()` method by returning the hours worked times the hourly rate.
4. the company employs sales associates who are paid through a fixed salary plus a commission

based on their sales, so you create a `CommissionEmployee` class) Derive `CommissionEmployee` from `SalaryEmployee` because both classes have a `weekly_salary` to consider. At the same time, you initialize `CommissionEmployee` with a `commission` value that's based on the sales for the employee.

b) In `.calculate_payroll()`, you leverage the implementation of the base class to retrieve the fixed salary, and you add the commission value.

5. Implementing a `PayrollSystem` class that processes payroll.

a) `PayrollSystem` implements a `.calculate_payroll()` method that takes a collection of employees and prints their `.id`, `.name`, and check amount using the `.calculate_payroll()` method exposed on each employee object.

Task: creates the employees and passes them to the payroll system to process payroll

CODE:

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

    def calculate_payroll(self):
        """This method should be implemented by subclasses."""
        raise NotImplementedError("Subclasses must implement calculate_payroll()")

    def __str__(self):
        return f"ID: {self.employee_id}, Name: {self.name}"

class SalaryEmployee(Employee):
    def __init__(self, name, employee_id, weekly_salary):
        super().__init__(name, employee_id)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, name, employee_id, hours_worked, hourly_rate):
        super().__init__(name, employee_id)
```

OUTPUT:

```
Processing Payroll
=====
ID: 101, Name: Alice Johnson - Payroll Amount: $1200
ID: 102, Name: Bob Smith - Payroll Amount: $600
ID: 103, Name: Charlie Davis - Payroll Amount: $1500
=====
```

Problem 5:

Design a class for the employee of electrical department.

CODE:

```
class ElectricalEmployee:
    def __init__(self, name, employee_id, designation, experience, hourly_rate):
        self.name = name
        self.employee_id = employee_id
        self.designation = designation # Example: "Electrician", "Engineer", etc.
        self.experience = experience # Years of experience
        self.hourly_rate = hourly_rate # Wage per hour

    def calculate_payroll(self, hours_worked):
        """Calculate payroll based on hours worked."""
        return hours_worked * self.hourly_rate

    def __str__(self):
        return (f"Employee ID: {self.employee_id}, Name: {self.name}, "
                f"Designation: {self.designation}, Experience: {self.experience} years")

# Creating an instance of ElectricalEmployee
electrical_worker = ElectricalEmployee("John Doe", 201, "Electrician", 5, 20)

# Display employee details
print(electrical_worker)
```

OUTPUT:

```
Employee ID: 201, Name: John Doe, Designation: Electrician, Experience: 5 years
Payroll for 40 hours: $800
```

LAB 8

1. Start adding the new classes to the existing class.

ProductivitySystem tracks productivity based on employee roles. There are different employee roles:

- **Managers:** They walk around yelling at people, telling them what to do. They're salaried employees and make more money.
 - **Secretaries:** They do all the paperwork for managers and ensure that everything gets billed and paid on time. They're also salaried employees but make less money.
 - **Sales employees:** They make a lot of phone calls to sell products. They have a salary, but they also get commissions for sales.
 - **Factory workers:** They manufacture the products for the company. They're paid by the hour.
- First, you add a `Manager` class that derives from `SalaryEmployee`. The class exposes a `.work()` method that the productivity system will use. The method takes the `hours` that the employee worked. Then you add `Secretary`, `SalesPerson`, and `FactoryWorker` and then implement the `.work()` interface, so they can be used by the productivity system

CODE:

```
# Base Employee Class
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def calculate_payroll(self):
        """Must be implemented by subclasses."""
        raise NotImplementedError("Subclasses must implement calculate_payroll()")

    def work(self, hours):
        """Must be implemented by subclasses."""
        raise NotImplementedError("Subclasses must implement work()")

    def __str__(self):
        return f"ID: {self.employee_id}, Name: {self.name}"

# Salaried Employees (Base for Manager & Secretary)
class SalaryEmployee(Employee):
    def __init__(self, name, employee_id, weekly_salary):
        super().__init__(name, employee_id)
```

OUTPUT:

```

Processing Payroll
=====
ID: 101, Name: Alice Johnson - Payroll Amount: $2000
ID: 102, Name: Bob Smith - Payroll Amount: $1200
ID: 103, Name: Charlie Davis - Payroll Amount: $1500
ID: 104, Name: Daniel Lee - Payroll Amount: $800
=====

Productivity Report
=====
Alice Johnson is managing for 8 hours, yelling at people.
Bob Smith is handling paperwork for 8 hours.
Charlie Davis is making sales calls for 8 hours

```

Lab 9

1. It turns out that sometimes temporary secretaries are hired when there's too much paperwork to do. The `TemporarySecretary` class performs the role of a `Secretary` in the context of the `ProductivitySystem`, but for payroll purposes, it's an `HourlyEmployee`.

a) Derive it from both `Secretary` and `HourlyEmployee`:

i.

```
class TemporarySecretary(Secretary, HourlyEmployee):
    pass
```

Run the code

```
temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)
```

Understand the code and error and check `TemporarySecretary.__mro__`

ii. Modify the above class as

```
class TemporarySecretary(HourlyEmployee, Secretary):
    pass
```

Run the code

```
temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)
```

Understand the code and error and check `TemporarySecretary.__mro__`

iii. Modify the above class as

```
class TemporarySecretary(HourlyEmployee, Secretary):
    def __init__(self, id, name, hours_worked, hourly_rate):
        super().__init__(id, name, hours_worked, hourly_rate)
```

Run the code `temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)`

Understand the code and error and check `TemporarySecretary.__mro__`

iv. Modify the above class as

```
class TemporarySecretary(Secretary, HourlyEmployee):
    def __init__(self, id, name, hours_worked, hourly_rate):
        HourlyEmployee.__init__(self, id, name, hours_worked, hourly_rate)
```

Run the code

```
temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)
company_employees = [temporary_secretary]
```

```
productivity_system = productivity.ProductivitySystem()
productivity_system.track(company_employees, 40)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll(company_employees)
```

Understand the code and error and check TemporarySecretary.__mro__

v. Modify the above class as

```
class TemporarySecretary(Secretary, HourlyEmployee):
    def __init__(self, id, name, hours_worked, hourly_rate):
        HourlyEmployee.__init__(self, id, name, hours_worked, hourly_rate)
    def calculate_payroll(self):
        return HourlyEmployee.calculate_payroll(self)
```

Run the code

```
temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)
company_employees = [temporary_secretary]
productivity_system =
productivity.ProductivitySystem()productivity_system.track(company_employees, 40)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll(company_employees)
```

Understand the code and error and check TemporarySecretary.__mro__

CODE :

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

    def calculate_payroll(self):
        raise NotImplementedError("Subclasses must implement calculate_payroll()")

    def work(self, hours):
        raise NotImplementedError("Subclasses must implement work()")

    def __str__(self):
        return f"ID: {self.employee_id}, Name: {self.name}"

class Secretary(Employee):
    def __init__(self, name, employee_id):
        super().__init__(name, employee_id)

    def calculate_payroll(self):
        return 1000 # Example fixed salary for a secretary

    def work(self, hours):
```


OUTPUT:

ID: 5, Name: Robin Williams
Payroll: \$360

Productivity Report
Robin Williams is handling paperwork for 40 hours.

Processing Payroll
ID: 5, Name: Robin Williams - Payroll Amount: \$360

Lab Report

Task 1: Convert Labs 7, 8, and 9 into Modules, Package them with Name ``inheritance``, and Revise the Corresponding Tasks.

Introduction

In this task, we are asked to convert Labs 7, 8, and 9 into Python modules, and then organize them into a package named ``inheritance``. After converting them into modules, we will revise the tasks to include the functionality and structure of the newly created modules and package.

Steps to Convert Labs into Modules

1. ****Create a new directory**** to store the package:
 - The name of the directory will be ``inheritance``.
2. ****Convert Lab 7, 8, and 9 scripts**** into separate Python modules:

- Each lab will be a module in the ``inheritance`` package.
- You will need to create ``*.py`` files for each lab task.

Code :


```
# Import relevant classes from Lab7 module in inheritance package
from inheritance.lab7 import Employee, SalaryEmployee, HourlyEmployee, CommissionEmployee

# Example usage
salary_employee = SalaryEmployee(1, "John Doe", 1000)
print(salary_employee.calculate_payroll()) # Expected Output: 1000
```

```
# Import classes from Lab8 module in inheritance package
from inheritance.lab8 import TemporarySecretary


# Example usage
temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)
print(f"Payroll: ${temporary_secretary.calculate_payroll()}")
|
```

python

 Copy  Edit

```
# Import necessary classes from Lab9 module in inheritance package
from inheritance.lab9 import ProductivitySystem, PayrollSystem

# Example usage
company_employees = [temporary_secretary]
productivity_system = ProductivitySystem()
productivity_system.track(company_employees, 40)

payroll_system = PayrollSystem()
payroll_system.calculate_payroll(company_employees)
```

```
temporary_secretary = TemporarySecretary(5, "Robin Williams", 40, 9)
company_employees = [temporary_secretary]
productivity_system = Productivity_system()
productivity_system.Track(company_employees, 40)
payroll_system = Payroll_system()
payroll_system.calculate_payroll(company_employees)
```

OUTPUT:

```
Productivity System Tracking
-----
Robin Williams efficiently organizes and manages documents for 40 hours.

Calculate Payroll
=====
Payroll for: 5 - Robin Williams
- Check amount: 360
```

LAB 12:

"QUEUE ALGORITHMS"

AIM:

[+ Code](#)[+ Markdown](#)

To understand the concept of queue algorithms in OOP (PYTHON).

A queue is a fundamental data structure in computer science that follows the First-In-First-Out (FIFO) principle. In a queue, elements are added at the rear (enqueue) and removed from the front (dequeue). This ensures that the oldest element in the queue is the first to be

DONE ON VS CODE.
