Theoritical activity 2.1.1 Description of python basic concepts

1. **Data Types**

Python has several built-in data types, including:

- ✓ **Integers** (int): Whole numbers, e.g., 5, -3.
- ✓ **Floating** Point Numbers (float): Decimal numbers, e.g., 3.14, -0.001.
- ✓ **Strings** (str): Sequences of characters, e.g., "Hello, World!".
- ✓ **Booleans** (bool): Represents True or False.
- ✓ **Lists**: Ordered, mutable collections, e.g., [1, 2, 3].
- ✓ **Tuples**: Ordered, immutable collections, e.g., (1, 2, 3).
- ✓ **Dictionaries** (dict): Key-value pairs, e.g., {"name": "Alice", "age": 25}.
- ✓ **Sets**: Unordered collections of unique elements, e.g., {1, 2, 3}.
- ✓ **None** : Once you have a variable and you didn't assign any value to it

2. **Variables**

Variables in Python are used to store data. You can create a variable by assigning a value to it using the = operator. For example:

```
x = 10       # Integer

name = "Bob"  # String

is_active = True  # Boolean
```

Variable names should be descriptive and can include letters, numbers, and underscores, but they cannot start with a number.

3. **Comments**

Comments are used to explain code and are ignored by the Python interpreter. You can create a single-line comment by using the # symbol:

```
# This is a single-line comment

x = 5  # Assign 5 to x
```

For multi-line comments, you can use triple quotes:

```
"""

This is a

multi-line comment
```

"""

4. Operators

Operators are special symbols that perform operations on variables and values. Common operators in Python include:

4.1 Arithmetic Operators:

- ✓ + (Addition)
- ✓ - (Subtraction)
- ✓ * (Multiplication)
- ✓ / (Division)
- ✓ // (Floor Division)
- ✓ % (Modulus)
- ✓ ** (Exponentiation)

4.2 Comparison Operators:

- ✓ == (Equal to)
- ✓ != (Not equal to)
- ✓ > (Greater than)
- ✓ < (Less than)
- ✓ >= (Greater than or equal to)
- ✓ <= (Less than or equal to)

4.3 Logical Operators:

- ✓ and
- ✓ or
- ✓ not

4.4 Assignment Operators:

- ✓ = (Assign)
- ✓ += (Add and assign)
- ✓ -= (Subtract and assign)
- ✓ *= (Multiply and assign)
- ✓ /= (Divide and assign)

Practical activity 2.1.2 Application of python basic concepts

1. Data Types
Application: Using different data types to store and manipulate various kinds of data.

```
# Different data types
integer_value = 42              # Integer
float_value = 3.14              # Float
string_value = "Hello, Python!"    # String
boolean_value = True            # Boolean
list_value = [1, 2, 3, 4, 5]      # List
tuple_value = (1, 2, 3)          # Tuple
dict_value = {"name": "Alice", "age": 30}  # Dictionary
set_value = {1, 2, 3}           # Set

print(f"Integer: {integer_value}, Float: {float_value}, String: '{string_value}'")
print(f"Boolean: {boolean_value}, List: {list_value}, Tuple: {tuple_value}")
print(f"Dictionary: {dict_value}, Set: {set_value}")
```

2. Variables
Application: Storing user input and performing operations.

```
# Using variables to store user input
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Displaying the stored variables
print(f"Hello, {name}! You are {age} years old.")
```

3. Comments
Application: Documenting code for better understanding.

```
# This program calculates the area of a rectangle

# Function to calculate area
def calculate_area(length, width):
    return length * width

# Main execution
length = 5  # Length of the rectangle
width = 3   # Width of the rectangle
area = calculate_area(length, width)  # Calculate area
```

```
# Display the result
print(f"The area of the rectangle is: {area}")
```

4. Operators

Application: Performing calculations and comparisons.

4.1 Arithmetic Operators

```
# Arithmetic Operators
a = 10
b = 3

# Addition
addition = a + b
print(f"Addition: {a} + {b} = {addition}")

# Subtraction
subtraction = a - b
print(f"Subtraction: {a} - {b} = {subtraction}")

# Multiplication
multiplication = a * b
print(f"Multiplication: {a} * {b} = {multiplication}")

# Division
division = a / b
print(f"Division: {a} / {b} = {division}")

# Floor Division
floor_division = a // b
print(f"Floor Division: {a} // {b} = {floor_division}")

# Modulus
modulus = a % b
print(f"Modulus: {a} % {b} = {modulus}")

# Exponentiation
exponentiation = a ** b
print(f"Exponentiation: {a} ** {b} = {exponentiation}")
```

4.2 Comparison Operators

```
# Comparison Operators
x = 5
y = 10

# Equal to
print(f"{x} == {y}: {x == y}")

# Not equal to
print(f"{x} != {y}: {x != y}")

# Greater than
```

```
print(f"{x} > {y}: {x > y}")

# Less than
print(f"{x} < {y}: {x < y}")

# Greater than or equal to
print(f"{x} >= {y}: {x >= y}")

# Less than or equal to
print(f"{x} <= {y}: {x <= y}")
```

4.3 Logical Operators

```
# Logical Operators
a = True
b = False

# Logical AND
print(f"a and b: {a and b}")

# Logical OR
print(f"a or b: {a or b}")

# Logical NOT
print(f"not a: {not a}")
```

4.4 Assignment Operators

```
# Assignment Operators
num = 10

# Add and assign
num += 5
print(f"After += 5, num = {num}")

# Subtract and assign
num -= 3
print(f"After -= 3, num = {num}")

# Multiply and assign
num *= 2
print(f"After *= 2, num = {num}")

# Divide and assign
num /= 4
print(f"After /= 4, num = {num}")
```

2.2 APPLYING PYTHON CONTROL STRUCTURES

Practical activity 2.2.1 Application of Conditional Statements

Conditional statements in Python allow you to execute different blocks of code based on certain conditions.
The most common conditional statements are

- ✓ if,
- ✓ elif
- ✓ else.

Here are some applications with program examples:

1. Simple if Statement

Application: Checking a condition and executing a block of code if the condition is true.

```
# Simple if statement
age = 18

if age >= 18:
    print("You are eligible to vote.")
```

2. if and else Statement

Application: Executing one block of code if the condition is true, and another if it is false.

```
# If-else statement
number = 7

if number % 2 == 0:
    print(f"{number} is even.")
else:
    print(f"{number} is odd.")
```

3. if, elif, and else Statement

Application: Checking multiple conditions using elif.

```
# If-elif-else statement
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
```

```
else:
    grade = 'D'

print(f"Your grade is: {grade}")
```

4. Nested Conditional Statements
Application: Using conditional statements within other conditional statements.

```
# Nested if statement
temperature = 30

if temperature > 0:
    print("The water is liquid.")
    if temperature > 100:
        print("The water is boiling.")
else:
    print("The water is frozen.")
```

5. Using Logical Operators in Conditions
Application: Combining conditions using logical operators (and, or, not).

```
# Logical operators in conditions
age = 20
has_id = True
if age >= 18 and has_id:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

Practical activity 2.2.2 Application of Looping Statements

Looping statements in Python allow you to execute a block of code multiple times. The two primary types of loops are for loops and while loops. Here are some applications with sample examples for each:

1. for Loop
Application: Iterating over a sequence (like a list, tuple, or string).

```
# Using a for loop to iterate over a list
fruits = ["apple", "banana", "cherry"]
```

```
for fruit in fruits:
    print(f"I like {fruit}.")
```

2. for Loop with range()
Application: Repeating a block of code a specific number of times.

```
# Using a for loop with range()
for i in range(5):  # This will iterate from 0 to 4
    print(f"Iteration {i + 1}")
```

3. while Loop
Application: Repeating a block of code as long as a condition is true.

```
# Using a while loop
count = 0

while count < 5:
    print(f"Count is: {count}")
    count += 1  # Increment count
```

4. Nested Loops
Application: Using a loop inside another loop.

```
# Using nested loops
for i in range(3):
    for j in range(2):
        print(f"Outer loop {i}, Inner loop {j}")
```

5. Looping with Conditional Statements
Application: Combining loops with conditional statements to filter results.

```
# Using a loop with a conditional statement
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print("Even numbers:")
for number in numbers:
    if number % 2 == 0:
        print(number)
```

6. Using break and continue Statements

Application: Controlling the flow of loops using break to exit a loop and continue to skip to the next iteration.

```
# Using break and continue
for number in range(1, 11):
    if number == 5:
        print("Breaking the loop at number 5.")
        break  # Exit the loop when number is 5
    print(number)

print("\nUsing continue:")
for number in range(1, 11):
    if number % 2 == 0:
        continue  # Skip even numbers
    print(number)  # Print only odd numbers
```

Practical activity 2.2.3 Using Jump Statements

Jump statements in Python control the flow of loops and can alter the normal execution sequence.
The main jump statements are break, continue, and pass. Here are examples of each:

1. break Statement

Application: Exits the nearest enclosing loop when a specified condition is met.

```
# Example of break
for number in range(1, 11):
    if number == 6:
        print("Breaking the loop at number 6.")
        break  # Exit the loop
    print(number)

# Output will be: 1, 2, 3, 4, 5
```

2. continue Statement

Application: Skips the current iteration of the nearest enclosing loop and continues with the next iteration.

```
# Example of continue
for number in range(1, 11):
    if number % 2 == 0:
        continue  # Skip even numbers
    print(number)  # Print only odd numbers
# Output will be: 1, 3, 5, 7, 9
```

3. pass Statement

Application: A null operation; it is syntactically required but does nothing when executed. It's often used as a placeholder.

```
# Example of pass
for number in range(1, 6):
    if number == 3:
        pass  # Placeholder for future code
    print(number)

# Output will be: 1, 2, 3, 4, 5
```

Summary

- **break**: Terminates the loop entirely when a condition is met.
- **continue**: Skips the current iteration and continues with the next one.
- **pass**: Does nothing and is useful for maintaining the structure of code where a statement is syntactically required.

Applying functions in Python

Theoretical activity 2.3.1 Description of function in python

Definition of Function
A function in Python is a block of reusable code that performs a specific task. It is defined using the def keyword, followed by the function name and parentheses containing any parameters. Functions help organize code, making it more modular and easier to understand.
Example of a Function Definition:

```
def greet(name):
    """This function greets the person passed as a parameter."""
    print(f"Hello, {name}!")
```

Characteristics of Functions

1. **Modularity**: Functions allow you to break your program into smaller, manageable pieces.
2. **Reusability**: Once defined, functions can be reused multiple times throughout the code.
3. **Parameters and Return Values**: Functions can accept parameters and return values, making them flexible.
4. **Encapsulation**: Functions encapsulate the logic of a task, which can improve code clarity.
5. **Scope**: Variables defined inside a function are local to that function unless specified otherwise.

Advantages of Functions

1. **Improved Readability**: Breaking code into functions improves readability and organization.
2. **Easier Maintenance**: Functions can be modified independently, making maintenance simpler.
3. **Code Reusability**: Functions can be reused across different parts of a program or even in different programs.
4. **Debugging**: Isolating functionality into functions simplifies debugging since you can test each function independently.

5. **Abstraction**: Functions allow you to abstract complex operations, making it easier to understand and use.

Types of Functions

1. Built-in Functions

These are functions that are pre-defined in Python and can be used without any additional code.

Examples include:

- **print():** Outputs data to the console.
- **len():** Returns the length of an object.
- **type():** Returns the type of an object.
- **sum():** Returns the sum of a collection of numbers.

Example of a Built-in Function:

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)  # Using the built-in sum function
print(f"The total is: {total}")
```

2. User-Defined Functions

These are functions that you define yourself to perform specific tasks. You can create them using the def keyword.

Example of a User-Defined Function:

```
def add(a, b):
    """This function returns the sum of two numbers."""
    return a + b

result = add(5, 3)  # Calling the user-defined function
print(f"The sum is: {result}")
```

Summary

Functions are fundamental building blocks in Python programming. They promote code reuse, readability, and maintainability. Understanding both built-in and user-defined functions enables you to write efficient and organized code.

Practical activity 2.3.2 creation of function in python

Creation of Function in Python

1. Defining a Function

You define a function using the def keyword, followed by the function name and parentheses. You can include parameters within the parentheses.

Example:

```
def greet(name):
    """This function greets the person passed as a parameter."""
    print(f"Hello, {name}!")
```

2. Arguments

Arguments are the values you pass to a function when calling it. You can define functions with different types of arguments:

Positional Arguments:

These must be provided in the correct order.

```
def add(a, b):
    return a + b

result = add(5, 3)  # 5 and 3 are positional arguments
print(f"The sum is: {result}")
```

Keyword Arguments:

You can specify arguments by name, allowing you to pass them in any order.

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet(pet_name="Buddy", animal_type="dog")  # Using keyword arguments
```

3. Default Parameter Value

You can set default values for parameters in a function. If a value is not provided during the function call, the default value will be used.

Example:

```
def greet(name="Guest"):
    """This function greets the person passed as a parameter with a default value."""
    print(f"Hello, {name}!")

greet()        # Uses default value
greet("Alice")  # Overrides default value
```

4. Passing a List as an Argument

You can pass a list (or any other collection) as an argument to a function. Inside the function, you can manipulate it as needed.

Example:

```
def print_fruits(fruits):
    """This function prints each fruit in the list."""
    for fruit in fruits:
        print(fruit)

fruit_list = ["apple", "banana", "cherry"]
print_fruits(fruit_list)  # Passing a list as an argument
```

5. Calling a Function

To call a function, simply use its name followed by parentheses. If the function requires arguments, provide them within the parentheses.

Example:

```
def multiply(a, b):
    return a * b

# Calling the function with arguments
result = multiply(4, 5)
print(f"The product is: {result}")  # Outputs: The product is: 20
```

Summary

Functions in Python allow you to encapsulate code for reuse and better organization.
You can define functions with various types of arguments, including default parameter values and lists, and call these functions as needed.
This enhances code readability and maintainability.

Practical activity 2.3.3. Applying special purpose functions

1. Lambda Functions

Definition: A lambda function is a small anonymous function defined with the lambda keyword. It can take any number of arguments but can only have one expression.

Example:

```
# Lambda function to add two numbers
add = lambda x, y: x + y
result = add(5, 3)
print(f"The sum using lambda is: {result}")  # Outputs: The sum using lambda is: 8
```

2. Python Generators

Definition: Generators are a type of iterable, like lists or tuples. Unlike lists, they do not store their contents in memory; instead, they generate items on-the-fly using the yield keyword.

Example:

```
# Generator function to yield numbers
```

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

# Using the generator
for number in count_up_to(5):
    print(number)  # Outputs: 1, 2, 3, 4, 5
```

3. Python Closures

Definition: A closure is a function that remembers its enclosing lexical scope even when the program flow is no longer in that scope.

Example:

```
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

# Create a closure
my_greeting = outer_function("Hello, World!")
my_greeting()  # Outputs: Hello, World!
```

4. Python Decorators

Definition: Decorators are a way to modify or enhance functions or methods without changing their code. They are applied using the @decorator syntax.

Example:

```
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed before {}".format(original_function.__name__))
        return original_function()
    return wrapper_function

@decorator_function
def display():
    print("Display function executed.")

# Calling the decorated function
display()
# Outputs:
# Wrapper executed before display
```

```
# Display function executed.
```

5. Recursive Function
Definition: A recursive function is a function that calls itself in order to solve a problem.

Example:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Calling the recursive function
result = factorial(5)
print(f"The factorial of 5 is: {result}")  # Outputs: The factorial of 5 is: 120
```

6. Higher-Order Function
Definition: A higher-order function is a function that takes one or more functions as arguments or returns a function as its result.
Example:

```
def apply_function(func, value):
    return func(value)

# Function to square a number
def square(x):
    return x * x

# Using a higher-order function
result = apply_function(square, 4)
print(f"The square of 4 is: {result}")  # Outputs: The square of 4 is: 16
```

Summary

These special-purpose functions in Python enhance your programming capabilities. lambda functions provide concise function definitions, generators create iterables efficiently, closures maintain state, decorators modify behavior, recursive functions solve problems through self-reference, and higher-order functions facilitate function manipulation. Understanding these concepts will help you write more effective and elegant Python code.

2.4 Applying of Python Collections

Theoretical activity 2.4.1 Description of python Collections

Collection Types in Python

1. Lists

Definition: Lists are ordered, mutable collections that can hold a variety of object types. Elements can be added, removed, or modified.

Syntax: my_list = [1, 2, 3, 'apple', 4.5]

Example:

```
fruits = ['apple', 'banana', 'cherry']
fruits.append('orange')  # Add an item
print(fruits)  # Outputs: ['apple', 'banana', 'cherry', 'orange']
```

2. Tuples

Definition: Tuples are ordered, immutable collections. Once created, their elements cannot be changed.

Syntax: my_tuple = (1, 2, 3)

Example:

```
colors = ('red', 'green', 'blue')
print(colors[1])  # Outputs: green
```

3. Dictionaries

Definition: Dictionaries are unordered collections of key-value pairs. Keys must be unique and immutable, while values can be of any type.

Syntax: my_dict = {'key1': 'value1', 'key2': 'value2'}

Example:

```
student = {'name': 'Alice', 'age': 25}
student['age'] = 26  # Modify value
print(student)  # Outputs: {'name': 'Alice', 'age': 26}
```

4. Sets

Definition: Sets are unordered collections of unique elements. They are mutable and do not allow duplicate values.

Syntax: my_set = {1, 2, 3}

Example:

```
unique_numbers = {1, 2, 2, 3}
print(unique_numbers)  # Outputs: {1, 2, 3}
```

5. Frozen Set

Definition: A frozen set is an immutable version of a set. Once created, its elements cannot be changed.

Syntax: my_frozenset = frozenset([1, 2, 3])

Example:

```
immutable_set = frozenset([1, 2, 3, 4])
print(immutable_set)  # Outputs: frozenset({1, 2, 3, 4})
```

6. ChainMaps

Definition: A ChainMap groups multiple dictionaries into a single view. It allows for searching through multiple dictionaries as if they were one.

Syntax: from collections import ChainMap

Example:

```
from collections import ChainMap

dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
combined = ChainMap(dict1, dict2)

print(combined['b'])  # Outputs: 2 (from dict1)
```

7. Deques

Definition: Deques (double-ended queues) are mutable sequences that allow fast appends and pops from both ends.

Syntax: from collections import deque

Example:

```
from collections import deque

my_deque = deque(['a', 'b', 'c'])
my_deque.append('d')  # Add to the right
my_deque.appendleft('z')  # Add to the left
print(my_deque)  # Outputs: deque(['z', 'a', 'b', 'c', 'd'])
```

Specialized Tools from the Collections Module

1. Counter

Definition: A Counter is a dictionary subclass for counting hashable objects. It makes it easy to count occurrences of elements.

Example:

```
from collections import Counter

count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'banana'])
print(count)  # Outputs: Counter({'banana': 3, 'apple': 2, 'orange': 1})
```

2. OrderedDict

Definition: An OrderedDict is a dictionary subclass that maintains the order of keys based on their insertion order.

Example:

```
from collections import OrderedDict

ordered_dict = OrderedDict()
ordered_dict['apple'] = 1
ordered_dict['banana'] = 2
ordered_dict['cherry'] = 3
print(ordered_dict)  # Outputs: OrderedDict([('apple', 1), ('banana', 2), ('cherry', 3)])
```

3. defaultdict

Definition: A defaultdict is a dictionary subclass that provides a default value for a nonexistent key. It avoids KeyErrors.

Example:

```
from collections import defaultdict

default_dict = defaultdict(int)  # Default value is 0
default_dict['a'] += 1
default_dict['b'] += 2
print(default_dict)  # Outputs: defaultdict(<class 'int'>, {'a': 1, 'b': 2})
```

Summary

Python provides various built-in collection types, each serving different purposes. Lists, tuples, dictionaries, sets, frozen sets, ChainMaps, and deques offer flexibility for handling data. Additionally, the collections module enhances functionality with specialized tools like Counter, OrderedDict, and defaultdict, making it easier to manage and manipulate data structures efficiently.

Practical activity 2.4.2 Perform common operations on collection

1. Adding and Removing Elements

Lists

Adding:

```
fruits = ['apple', 'banana']
fruits.append('cherry')  # Add to the end
fruits.insert(1, 'orange')  # Add at index 1
print(fruits)  # Outputs: ['apple', 'orange', 'banana', 'cherry']
```

Removing:

```
fruits.remove('banana')  # Remove by value
popped_fruit = fruits.pop()  # Remove last item and return it
print(fruits)  # Outputs: ['apple', 'orange']
print(f"Popped fruit: {popped_fruit}")  # Outputs: Popped fruit: cherry
```

Dictionaries

Adding:

```
student = {'name': 'Alice'}
student['age'] = 25  # Add new key-value pair
print(student)  # Outputs: {'name': 'Alice', 'age': 25}
```

Removing:

```
del student['age']  # Remove key-value pair by key
print(student)  # Outputs: {'name': 'Alice'}
```

2. Accessing and Iterating Over Elements

Lists

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

# Iterating over elements
for fruit in fruits:
    print(fruit)
```

Dictionaries

```
# Accessing values
print(student['name'])  # Outputs: Alice

# Iterating over keys and values
for key, value in student.items():
    print(f"{key}: {value}")
```

3. Filtering and Sorting

Lists

Filtering:

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Outputs: [2, 4, 6]
```

Sorting:

```
numbers.sort()  # Sorts in place
print(numbers)  # Outputs: [1, 2, 3, 4, 5, 6]

# Sorting with a custom key
sorted_fruits = sorted(fruits, key=len)  # Sort by length
print(sorted_fruits)
```

Dictionaries

```
# Sorting by keys
sorted_keys = sorted(student.keys())
print(sorted_keys)

# Sorting by values
sorted_by_value = sorted(student.items(), key=lambda item: item[1])
print(sorted_by_value)
```

4. Set Operations and Counting

Sets

Basic Set Operations:

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
union_set = set_a | set_b  # or set_a.union(set_b)
print(union_set)  # Outputs: {1, 2, 3, 4, 5}

# Intersection
```

```
intersection_set = set_a & set_b  # or set_a.intersection(set_b)
print(intersection_set)  # Outputs: {3}

# Difference
difference_set = set_a - set_b  # or set_a.difference(set_b)
print(difference_set)  # Outputs: {1, 2}
```

Counting Elements (Using Counter):

```
from collections import Counter

elements = ['apple', 'banana', 'apple', 'orange', 'banana', 'banana']
count = Counter(elements)
print(count)  # Outputs: Counter({'banana': 3, 'apple': 2, 'orange': 1})
```

5. Stack and Queue Operations

Stack Operations (Using Lists)

Stack: LIFO (Last In, First Out)

```
stack = []

# Push
stack.append('A')
stack.append('B')
stack.append('C')

# Pop
top_element = stack.pop()
print(top_element)  # Outputs: C
print(stack)  # Outputs: ['A', 'B']
```

Queue Operations (Using deque)

Queue: FIFO (First In, First Out)

```
from collections import deque

queue = deque()

# Enqueue
queue.append('A')
queue.append('B')
queue.append('C')

# Dequeue
first_element = queue.popleft()
print(first_element)  # Outputs: A
```

```
print(queue)  # Outputs: deque(['B', 'C'])
```

Summary

These are common operations you can perform on various collection types in Python. Lists, dictionaries, sets, and specialized tools such as deques provide powerful ways to manage and manipulate data, making Python a versatile language for handling collections.

2.5 Performing File handling

Theoretical activity 2.5.1 Description of file Handling libraries

File handling libraries in Python, include os, pathlib, shutil, and pandas. Each library serves different purposes and offers various functionalities for file and directory manipulation.

1. os Module

Description: The os module provides a way to use operating system-dependent functionality like reading or writing to the file system, working with directories, and handling environment variables.

Common Functions:

- ✓ os.listdir(path): Returns a list of files and directories in the specified path.
- ✓ os.mkdir(path): Creates a directory at the specified path.
- ✓ os.remove(path): Deletes a file at the specified path.
- ✓ os.rename(src, dst): Renames a file or directory.
- ✓ os.path: Contains functions to manipulate pathnames (e.g., os.path.join, os.path.exists).

Example:

```
import os

# List files in the current directory
files = os.listdir('.')
print(files)

# Create a new directory
os.mkdir('new_folder')
```

```
# Remove a file (make sure it exists)
# os.remove('file_to_delete.txt')
```

2. pathlib Module

Description: The pathlib module offers an object-oriented approach to file system paths. It allows easier manipulation of paths and provides a more intuitive syntax.

Common Classes and Methods:

- ✓ Path: Represents a filesystem path.
- ✓ Path.exists(): Checks if the path exists.
- ✓ Path.mkdir(): Creates a new directory.
- ✓ Path.rmdir(): Removes a directory.
- ✓ Path.read_text(): Reads the contents of a text file.

Example:

```
from pathlib import Path

# Create a Path object
path = Path('example.txt')

# Check if the file exists
if path.exists():
    print(f"{path} exists.")
else:
    # Create a new file
    path.write_text("Hello, World!")

# Read the file
content = path.read_text()
print(content)
```

3. shutil Module

Description: The shutil module provides a higher-level interface for file operations, particularly for copying and removing files and directories. It simplifies tasks like file and directory management.

Common Functions:

- ✓ shutil.copy(src, dst): Copies a file from src to dst.
- ✓ shutil.move(src, dst): Moves a file or directory from src to dst.
- ✓ shutil.rmtree(path): Deletes an entire directory tree.
- ✓ shutil.make_archive(base_name, format, root_dir): Creates a zip or tar archive.

Example:

```
import shutil

# Copy a file
shutil.copy('source.txt', 'destination.txt')

# Move a directory
shutil.move('old_directory', 'new_directory')

# Remove a directory tree
# shutil.rmtree('directory_to_delete')
```

4. pandas Library

Description: While primarily a data analysis library, pandas provides powerful tools for reading from and writing to various file formats, including CSV, Excel, JSON, and more. It simplifies data manipulation and analysis.

Common Functions:

- ✓ pandas.read_csv(filepath): Reads a CSV file into a DataFrame.
- ✓ DataFrame.to_csv(filepath): Writes a DataFrame to a CSV file.
- ✓ pandas.read_excel(filepath): Reads an Excel file into a DataFrame.
- ✓ DataFrame.to_excel(filepath): Writes a DataFrame to an Excel file.

Example:

```
import pandas as pd

# Read a CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the first few rows
print(df.head())

# Write DataFrame to a new CSV file
df.to_csv('new_data.csv', index=False)
```

Summary

These libraries provide robust tools for file handling in Python:

- ✓ **os**: For interacting with the operating system and file system.
- ✓ **pathlib**: For an object-oriented approach to path manipulation.
- ✓ **shutil**: For high-level file operations such as copying and moving files.

- ✓ **pandas**: For reading and writing data in various formats, primarily used for data analysis.

Practical activity 2.5.2 Practice to read file

Practice reading a file in Python, covering how to open a file and check file permissions.

1. Open a File

To open a file in Python, you use the built-in open() function. This function takes at least one argument: the path to the file. You can also specify a second argument to indicate the mode in which you want to open the file.

Common File Modes:

- ✓ 'r': Read (default mode) - Opens a file for reading.
- ✓ 'w': Write - Opens a file for writing (creates a new file or truncates an existing file).
- ✓ 'a': Append - Opens a file for appending (data will be written at the end).
- ✓ 'b': Binary - Opens a file in binary mode.
- ✓ 't': Text - Opens a file in text mode (default).

Example:

```
# Open a file for reading
file_path = 'example.txt'  # Ensure this file exists before running

try:
    with open(file_path, 'r') as file:
        content = file.read()  # Read the entire file
        print(content)  # Print the file content
except FileNotFoundError:
    print(f"The file {file_path} does not exist.")
except IOError:
    print("An error occurred while reading the file.")
```

2. Read File Permissions

Before opening a file, you may want to check its permissions to ensure you have the appropriate access rights. You can use the os module to check file permissions.

Example:

```
import os

file_path = 'example.txt'  # Ensure this file exists

# Check if the file exists
if os.path.exists(file_path):
    # Get file permissions
    permissions = os.stat(file_path).st_mode
    # Check read permission
```

```
    can_read = bool(permissions & 0o400)  # Owner can read
    can_write = bool(permissions & 0o200)  # Owner can write
    can_execute = bool(permissions & 0o100)  # Owner can execute

    print(f"Read permission: {can_read}")
    print(f"Write permission: {can_write}")
    print(f"Execute permission: {can_execute}")
else:
    print(f"The file {file_path} does not exist.")
```

Summary

- ✓ **Opening a File**: Use the open() function with the appropriate mode to read from or write to a file.
- ✓ **Reading a File**: You can read the entire content using file.read(), or read line by line using file.readline() or file.readlines().
- ✓ **Checking Permissions**: Use the os module to check whether you have read, write, or execute permissions for a file before attempting to open it.

Practical activity 2.5.3 Performing write/create and delete file

1. Create a New File

To create a new file in Python, you can use the open() function with the 'w' (write) or 'x' (exclusive creation) mode. The 'w' mode will create a new file or overwrite an existing file, while 'x' will raise an error if the file already exists.

Example:

```
# Create a new file
file_path = 'new_file.txt'

try:
    with open(file_path, 'w') as file:
        file.write("This is a new file created with Python.\n")
    print(f"File {file_path} created successfully.")
except IOError:
    print("An error occurred while creating the file.")
```

2. Write to an Existing File

To write to an existing file, you can open it in 'a' (append) or 'w' mode. The 'a' mode will add content to the end of the file without deleting the current content.

Example:

```
# Write to an existing file
existing_file_path = 'new_file.txt'
```

```
try:
    with open(existing_file_path, 'a') as file:
        file.write("Appending new content to the existing file.\n")
    print(f"Content appended to {existing_file_path} successfully.")
except IOError:
    print("An error occurred while writing to the file.")
```

3. Remove a File

To delete a file, you can use the os.remove() function from the os module.

Example:

```
import os

# Remove a file
file_to_remove = 'new_file.txt'

try:
    os.remove(file_to_remove)
    print(f"File {file_to_remove} deleted successfully.")
except FileNotFoundError:
    print(f"The file {file_to_remove} does not exist.")
except PermissionError:
    print(f"Permission denied to delete the file {file_to_remove}.")
except Exception as e:
    print(f"An error occurred: {e}")
```

4. Delete a Folder

To delete a folder, you can use os.rmdir() for empty directories or shutil.rmtree() for directories that contain files.

Example (Deleting an Empty Folder):

```
# Delete an empty folder
folder_to_remove = 'empty_folder'

try:
    os.rmdir(folder_to_remove)
    print(f"Folder {folder_to_remove} deleted successfully.")
except FileNotFoundError:
    print(f"The folder {folder_to_remove} does not exist.")
except OSError:
    print(f"The folder {folder_to_remove} is not empty or cannot be deleted.")
```

```
except Exception as e:
    print(f"An error occurred: {e}")
```

Example (Deleting a Non-Empty Folder):

```
import shutil

# Delete a non-empty folder
non_empty_folder = 'non_empty_folder'

try:
    shutil.rmtree(non_empty_folder)
    print(f"Folder {non_empty_folder} deleted successfully.")
except FileNotFoundError:
    print(f"The folder {non_empty_folder} does not exist.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Summary

- ✓ **Creating a New File**: Use open(file_path, 'w') or open(file_path, 'x') to create a new file.
- ✓ **Writing to an Existing File**: Use open(file_path, 'a') to append or 'w' to overwrite.
- ✓ **Removing a File**: Use os.remove() to delete a file.
- ✓ **Deleting a Folder**: Use os.rmdir() for empty directories and shutil.rmtree() for non-empty directories.

Practical activity 2.5.3 Application of python best practices

Best practices for writing Python code, focusing on readability and style, the use of built-in features, efficiency and memory usage, and error handling and testing.

1. Readability and Style

PEP 8: Follow the Python Enhancement Proposal (PEP) 8 style guide, which outlines conventions for writing clean and readable code.

- ✓ **Indentation**: Use 4 spaces per indentation level.
- ✓ **Line Length**: Limit lines to 79 characters.
- ✓ **Naming Conventions**: Use descriptive variable and function names. Use snake_case for variables and functions, and CamelCase for classes.

Example:

```
def calculate_area(radius):
    """Calculate the area of a circle given its radius."""
    return 3.14 * radius ** 2
```

- **Docstrings**: Use docstrings to describe the purpose of functions and classes. This helps others understand your code.

2. Use of Built-in Features

Leverage Built-in Functions: Use Python's built-in functions and libraries whenever possible, as they are optimized and thoroughly tested.

Example:

```
# Instead of manually calculating the sum of a list
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)  # Use built-in sum function
```

List Comprehensions: Use list comprehensions for creating lists in a concise and readable way.

Example:

```
squares = [x ** 2 for x in range(10)]  # List comprehension for squares
```

3. Efficiency and Memory Usage

Use Generators: When working with large datasets, use generators to save memory. Generators yield items one at a time and do not load everything into memory.

Example:

```
def generate_numbers(n):
    for i in range(n):
        yield i * 2

# Using the generator
for number in generate_numbers(10):
    print(number)
```

Avoid Unnecessary Copies: Be mindful of operations that create unnecessary copies of data. For instance, use in-place modifications where applicable.

Example:

```
# Instead of creating new lists
my_list = [1, 2, 3]
my_list.append(4)  # Modify in place
```

4. Error Handling and Testing

Use Exceptions: Use try and except blocks for error handling to make your code robust. Handle specific exceptions rather than using a bare except.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

Assertions: Use assertions to enforce conditions that must be true for your program to work correctly.

Example:

```
def divide(a, b):
    assert b != 0, "The denominator cannot be zero."
    return a / b
```

Unit Testing: Write unit tests using the unittest or pytest framework to ensure that your code works as expected. Testing helps catch bugs early and improves code reliability.
Example:

```
import unittest

def add(a, b):
    return a + b

class TestMathFunctions(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(1, 2), 3)

if __name__ == "__main__":
    unittest.main()
```

Summary
By following these best practices, you enhance the quality of your Python code:

- ✓ **Readability and Style**: Adhere to PEP 8 guidelines and use meaningful names and docstrings.
- ✓ **Use of Built-in Features**: Take advantage of Python's built-in functions and libraries for optimized performance.
- ✓ **Efficiency and Memory Usage**: Use generators and avoid unnecessary copies to manage memory effectively.
- ✓ **Error Handling and Testing**: Implement robust error handling and write unit tests to ensure code reliability.