SABUDH

Topic 1: Understanding Decorators in Python

Description: This assignment will focus on understanding the concept of decorators in Python. A decorator is a function that takes another function as an argument, modifies its behavior, and returns the modified function. In Python, decorators are a powerful tool that allows you to modify the behavior of functions or cl.

Objective:

- Understand the concept of decorators in Python
- Learn how to create decorators in Python
- Learn how to apply decorators to functions and classes

TASK 1:

1. Create a new Python file called decorators.py.
2. Define a function called uppercase_decorator that takes another function as an argument, modifies its behavior, and returns the modified function. The modified function should convert the result of the original function to uppercase.
3. Define a function called say_hello that takes a name as an argument and returns a greeting string.
4. Apply the uppercase_decorator to the say_hello function and assign the modified function to a new variable called greet.
5. Test the greet function by calling it with a name argument and printing the result.
6. Define a new decorator function called timing_decorator that takes another function as an argument, modifies its behavior, and returns the modified function. The modified function should measure and print the time it takes to run the original function.
7. Apply the timing_decorator to the greet function and assign the modified function to a new variable called timed_greet.

8. Test the timed_greet function by calling it with a name argument and printing the result.
9. Define a class called Math with two methods: add and subtract. Both methods should take two arguments and return the result of adding or subtracting them.
10. Define a new decorator function called logging_decorator that takes another function as an argument, modifies its behavior, and returns the modified function. The modified function should log the arguments and the result of the original function to the console.
11. Apply the logging_decorator to the Math class methods and test them by creating an instance of the class and calling the methods with some arguments.
12. Submit the decorators.py file with all the code and comments explaining each step.
Note: Make sure to follow the best practices for naming conventions and code style in Python.

Topic 2: Generators in Python

Description: In this assignment, you will be exploring the concept of generators in Python. Generators are a powerful feature of Python that allow you to iterate over a sequence of values without creating the entire sequence in memory.

Task 2

1. Write a generator function that generates the first n numbers in the Fibonacci sequence. Test your function by printing out the first 10 numbers in the sequence.
2. Write a generator function that generates all the prime numbers up to n. Test your function by printing out all the prime numbers up to 100.
3. Write a generator function that takes a string as input and generates all the words in the string. Test your function by generating all the words in the following string: "The quick brown fox jumps over the lazy dog."
4. Write a generator function that takes a list of strings as input and generates all the unique words in the list. Test your function by generating all the unique words in the following list: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]

5. Write a generator function that takes a list of integers as input and generates all the sublists of length n. Test your function by generating all the sublists of length 3 from the following list: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Submission Guidelines:

1. Write the code for each exercise in a separate Python file with a descriptive name (e.g."exercise1.py").
2. Include comments in your code to explain what each line does.
3. Include a brief explanation of your approach and any challenges you faced in a README.md file.
4. Submit your assignment as a zip file on the platform you are using.

TOPIC 3: Error and Exception Handling in Python

In this assignment, you will practice handling errors and exceptions in Python, including creating and raising custom exceptions.

Task 3:

1. Write a Python program that prompts the user to enter two numbers, then divides the first number by the second number and prints the result.
2. Use a try-except block to catch any ZeroDivisionError that might occur when dividing the numbers. Print an error message to the user instead of the result.
3. Create a custom exception called NegativeNumberError that raises an error message when a negative number is entered.
4. Modify your program to raise a NegativeNumberError if the user enters a negative number. Handle the exception using a try-except block and print the error message to the user.
5. Use the else block the print the result
6. Use the finally block to print a message to the user, indicating the end of the program execution.

Notes:

- You may want to use the input function to prompt the user for input.
- You may also want to use the int function to convert the user's input to integers.

● You should define your custom exception class before using it in your program.

Submission: Submit your Python program as a text file or code snippet, along with a brief write-up of your findings. In your write-up, discuss the following:
● How does the try-except block handle the ZeroDivisionError?
● What happens when the NegativeNumberError is raised?
● How does using custom exceptions improve the error handling in your program?
● What did you learn about error and exception handling in Python from this assignment?

Grading: Your program will be graded based on the following criteria:
1. Correct implementation of the try-except block.
1. Proper handling of the specified errors.
2. Proper display of error messages.
3. Correct conversion of the input string into a list of integers.
4. Proper display of the list of integers if the input is valid.

Good luck!