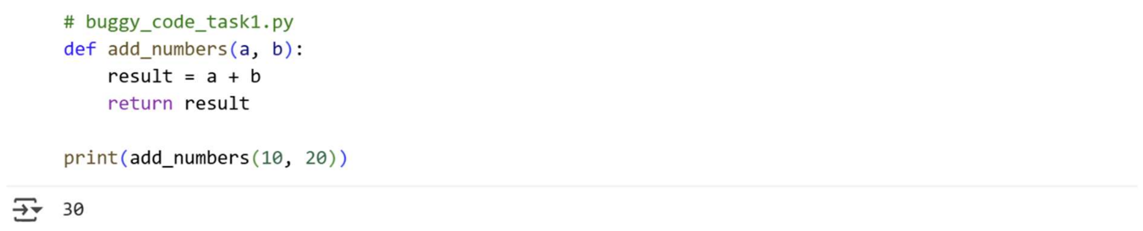
Assignment-10

Task-1:

Code & output:



```
# buggy_code_task1.py
def add_numbers(a, b):
    result = a + b
    return result

print(add_numbers(10, 20))
```

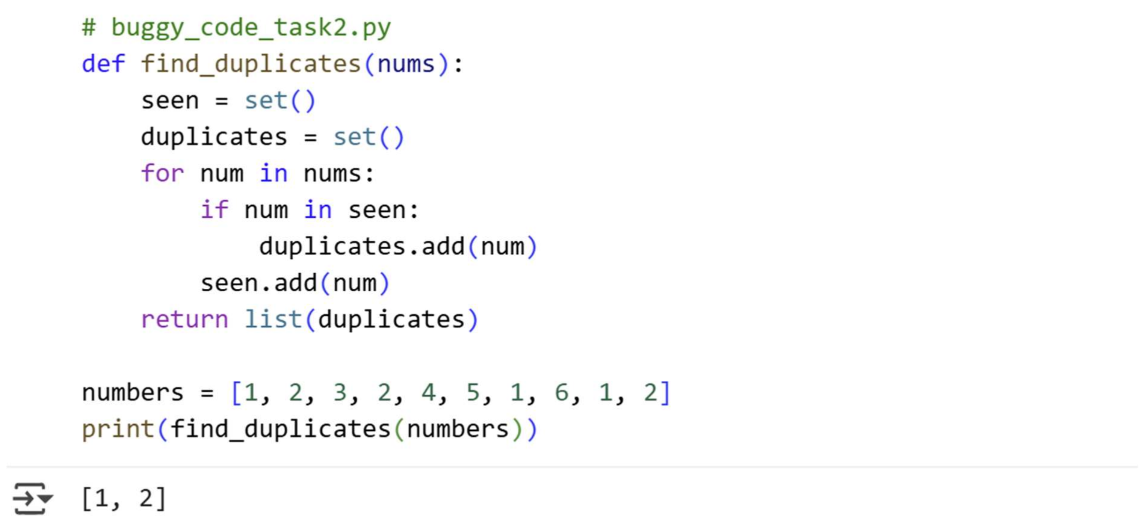
```
30
```

Explanation:

- def add_numbers(a, b):: This line defines a function named add_numbers that accepts two parameters, a and b.
- result = a + b: Inside the function, this line adds the values of a and b and assigns the sum to a variable named result.
- return result: This line returns the value stored in the result variable from the function.
- print(add_numbers(10, 20)): This line calls the add_numbers function with the arguments 10 and 20. The value returned by the function (which is 30) is then printed to the console.

Task-2:

Code & Output:



```python
# buggy_code_task2.py
def find_duplicates(nums):
    seen = set()
    duplicates = set()
    for num in nums:
        if num in seen:
            duplicates.add(num)
        seen.add(num)
    return list(duplicates)

numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2]
print(find_duplicates(numbers))
```

```
[1, 2]
```

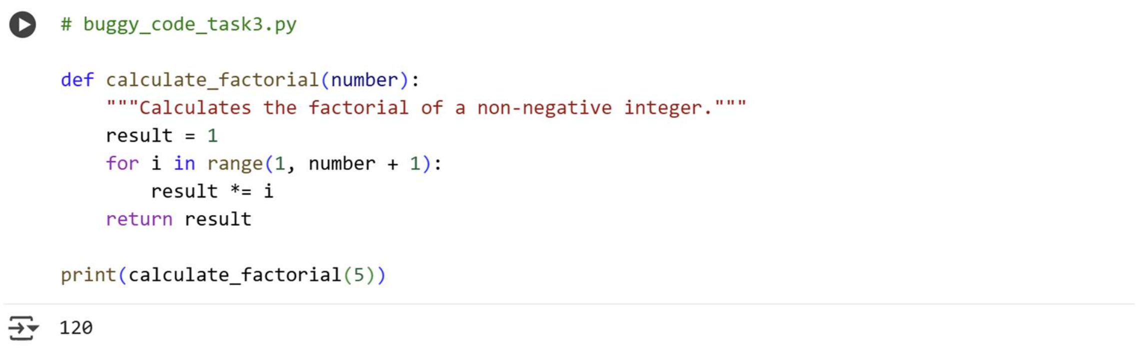
Explanation:

- def find_duplicates(nums):: This line defines a function named find_duplicates that takes one argument, a list of numbers named nums.
- seen = set(): This line initializes an empty set called seen. This set will be used to keep track of the numbers encountered so far in the input list.
- duplicates = set(): This line initializes an empty set called duplicates. This set will store the numbers that are found to be duplicates. Using a set automatically handles uniqueness, so each duplicate number will only be added once.
- for num in nums:: This line starts a for loop that iterates through each element in the input list nums, assigning the current element to the variable num in each iteration.
- if num in seen:: Inside the loop, this line checks if the current number num is already present in the seen set. If it is, it means this is a duplicate number.
- duplicates.add(num): If the if condition is true (the number is a duplicate), this line adds the duplicate number num to the duplicates set.
- seen.add(num): This line adds the current number num to the seen set, regardless of whether it was a duplicate or not. This marks the number as having been encountered.
- return list(duplicates): After the loop finishes, this line converts the duplicates set into a list and returns it. The list contains all the unique duplicate numbers found in the original list.
- numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2]: This line creates a list of integers and assigns it to the variable numbers. This is the input data for the function.

- print(find_duplicates(numbers)): This line calls the find_duplicates function with the numbers list as input and prints the returned list of duplicates to the console.

Task-3:

Code&Output:



```
# buggy_code_task3.py

def calculate_factorial(number):
    """Calculates the factorial of a non-negative integer."""
    result = 1
    for i in range(1, number + 1):
        result *= i
    return result

print(calculate_factorial(5))
```

```
120
```

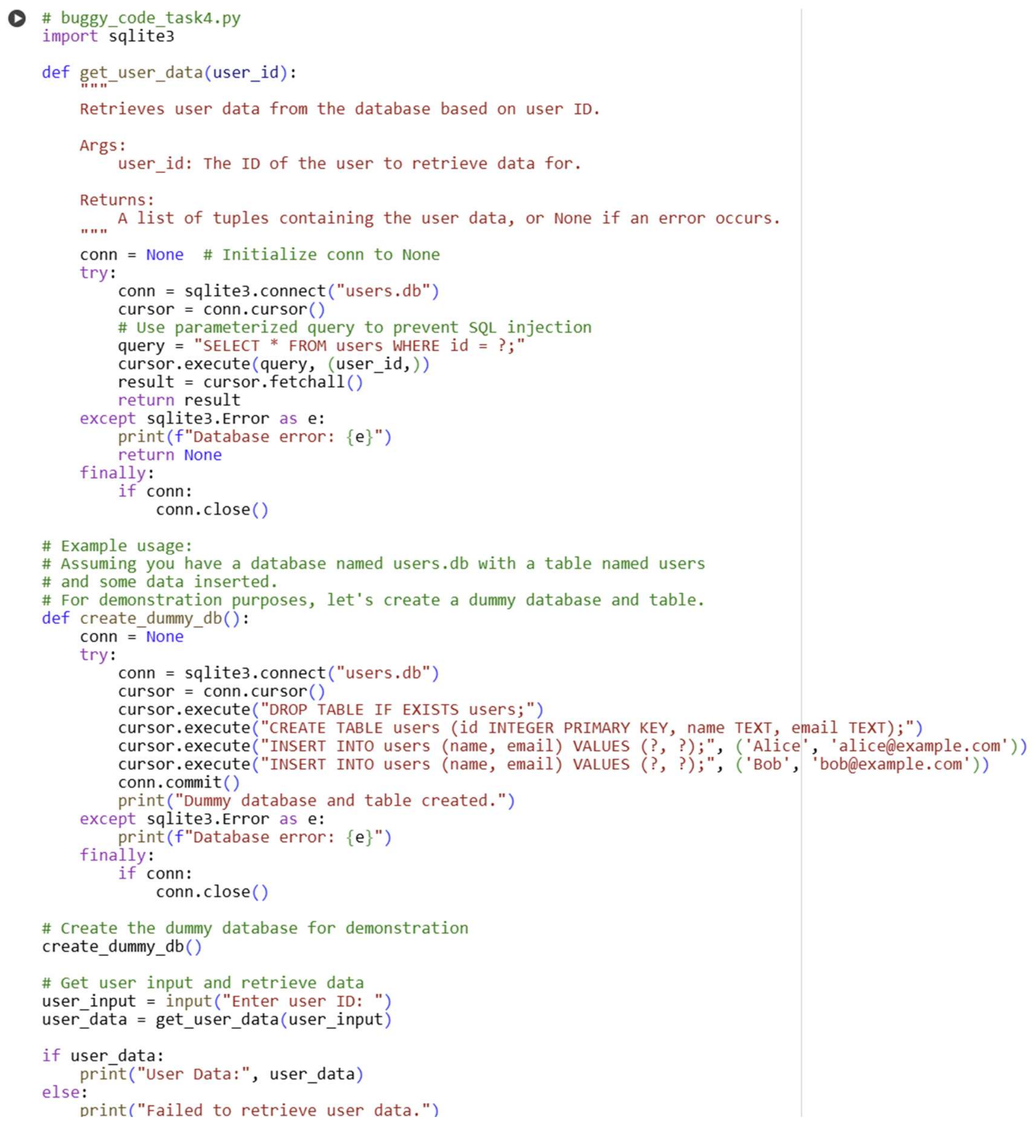
Explanation:

- # buggy_code_task3.py: This is a comment indicating the name of the file the code might have originated from.
- def calculate_factorial(number):: This line defines a function named calculate_factorial that takes one argument, an integer named number. This function is designed to calculate the factorial of this number.
- """Calculates the factorial of a non-negative integer.""": This is a docstring, which provides a brief explanation of what the function does.
- result = 1: This line initializes a variable called result and sets its initial value to 1. This is the starting point for calculating the factorial.
- for i in range(1, number + 1):: This line starts a for loop. The range(1, number + 1) function generates a sequence of numbers starting from 1 and going up to (and including) the value of number. The loop will iterate through each number in this sequence, assigning the current number to the variable i.
- result *= i: Inside the loop, this line multiplies the current value of result by the current value of i and updates result with the new value. This is the core of the factorial calculation, where the result is successively multiplied by each integer from 1 to number.
- return result: After the loop finishes, this line returns the final value stored in the result variable, which is the calculated factorial of the input number.

- print(calculate_factorial(5)): This line calls the calculate_factorial function with the argument 5. The value returned by the function (which is 120, the factorial of 5) is then printed to the console.

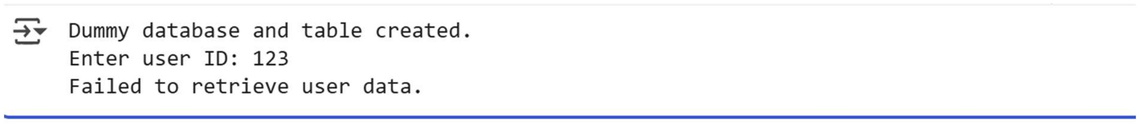
Task-4:

Code:



```python
# buggy_code_task4.py
import sqlite3

def get_user_data(user_id):
    """
    Retrieves user data from the database based on user ID.

    Args:
        user_id: The ID of the user to retrieve data for.

    Returns:
        A list of tuples containing the user data, or None if an error occurs.
    """
    conn = None  # Initialize conn to None
    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()
        # Use parameterized query to prevent SQL injection
        query = "SELECT * FROM users WHERE id = ?;"
        cursor.execute(query, (user_id,))
        result = cursor.fetchall()
        return result
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        return None
    finally:
        if conn:
            conn.close()

# Example usage:
# Assuming you have a database named users.db with a table named users
# and some data inserted.
# For demonstration purposes, let's create a dummy database and table.
def create_dummy_db():
    conn = None
    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()
        cursor.execute("DROP TABLE IF EXISTS users;")
        cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT);")
        cursor.execute("INSERT INTO users (name, email) VALUES (?, ?);", ('Alice', 'alice@example.com'))
        cursor.execute("INSERT INTO users (name, email) VALUES (?, ?);", ('Bob', 'bob@example.com'))
        conn.commit()
        print("Dummy database and table created.")
    except sqlite3.Error as e:
        print(f"Database error: {e}")
    finally:
        if conn:
            conn.close()

# Create the dummy database for demonstration
create_dummy_db()

# Get user input and retrieve data
user_input = input("Enter user ID: ")
user_data = get_user_data(user_input)

if user_data:
    print("User Data:", user_data)
else:
    print("Failed to retrieve user data.")
```

Output:



```
Dummy database and table created.
Enter user ID: 123
Failed to retrieve user data.
```

Explanation:

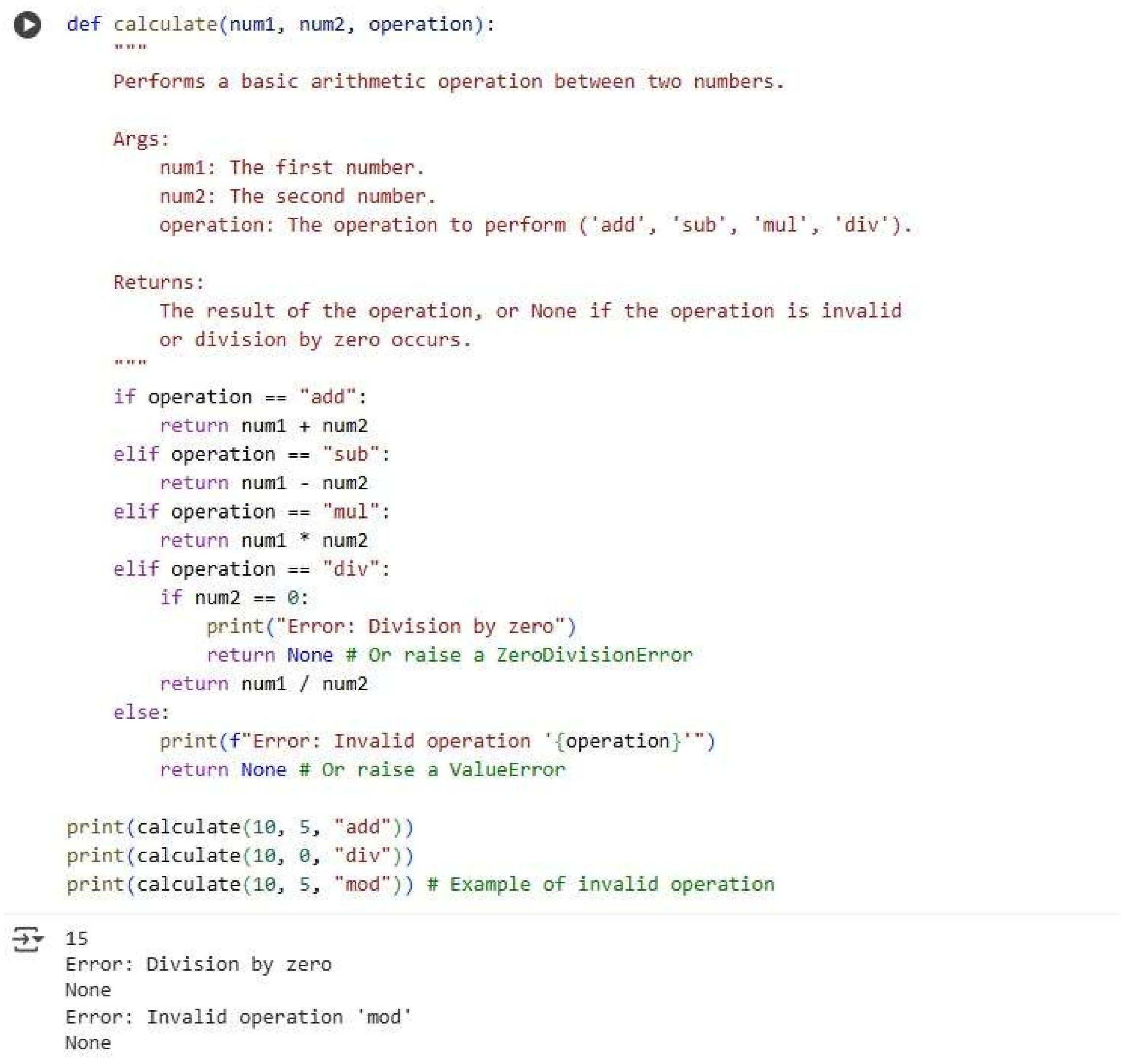
- `# buggy_code_task4.py`: This is a comment indicating the name of the file the code might have originated from.
- `import sqlite3`: This line imports the `sqlite3` module, which provides an interface for working with SQLite databases in Python.
- `def get_user_data(user_id):`: This line defines a function named `get_user_data` that takes one argument, `user_id`, which is intended to be the ID of the user to retrieve data for.
- `"""..."""`: This is a docstring explaining the purpose of the `get_user_data` function, its arguments, and what it returns.
- `conn = None # Initialize conn to None`: This line initializes a variable `conn` to `None`. This variable will hold the database connection object. It's initialized to `None` so that it can be checked in the `finally` block.
- `try:`: This line starts a `try` block, which is used for exception handling. Code within this block will be monitored for potential errors.
- `conn = sqlite3.connect("users.db")`: This line establishes a connection to the SQLite database file named "users.db". If the file doesn't exist, it will be created. The connection object is assigned to the `conn` variable.
- `cursor = conn.cursor()`: This line creates a cursor object from the connection. A cursor is used to execute SQL commands.
- `# Use parameterized query to prevent SQL injection`: This is a comment explaining the purpose of the next line.
- `query = "SELECT * FROM users WHERE id = ?;"`: This line defines the SQL query to select all columns (`*`) from the "users" table where the "id" column matches a specific value. The `?` is a placeholder for a parameter, which is a security measure against SQL injection.
- `cursor.execute(query, (user_id,))`: This line executes the SQL query. The `user_id` is passed as a parameter in a tuple `(user_id,)`. SQLite will safely substitute the `user_id` into the query, preventing malicious input from affecting the SQL command.
- `result = cursor.fetchall()`: This line fetches all the rows returned by the executed query and stores them as a list of tuples in the `result` variable.
- `return result`: This line returns the `result` (the retrieved user data) from the function.
- `except sqlite3.Error as e:`: This line starts an `except` block that catches any `sqlite3.Error` that might occur within the `try` block.
- `print(f"Database error: {e}")`: If a database error occurs, this line prints an error message including the specific error details.
- `return None`: If a database error occurs, this line returns `None` to indicate that the data retrieval failed.
- `finally:`: This line starts a `finally` block, which contains code that will be executed regardless of whether an exception occurred or not.

- `if conn:`: This line checks if the `conn` variable is not `None` (meaning a database connection was successfully established).
- `conn.close()`: If a connection exists, this line closes the database connection to release resources.
- `# Example usage:`: This is a comment indicating the start of example usage code.
- `def create_dummy_db():`: This line defines a function named `create_dummy_db` to create a sample database and table for demonstration.
- `conn = None`: Initializes `conn` to `None` within the `create_dummy_db` function.
- `try:`: Starts a `try` block for error handling within `create_dummy_db`.
- `conn = sqlite3.connect("users.db")`: Connects to the "users.db" database within `create_dummy_db`.
- `cursor = conn.cursor()`: Creates a cursor for `create_dummy_db`.
- `cursor.execute("DROP TABLE IF EXISTS users;")`: Executes an SQL command to drop the "users" table if it already exists.
- `cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT);")`: Executes an SQL command to create a new "users" table with columns for id, name, and email.
- `cursor.execute("INSERT INTO users (name, email) VALUES (?, ?);", ('Alice', 'alice@example.com'))`: Inserts a row with 'Alice' into the "users" table using parameterized query.
- `cursor.execute("INSERT INTO users (name, email) VALUES (?, ?);", ('Bob', 'bob@example.com'))`: Inserts a row with 'Bob' into the "users" table using parameterized query.
- `conn.commit()`: Commits the changes made to the database within `create_dummy_db`.
- `print("Dummy database and table created.")`: Prints a confirmation message.
- `except sqlite3.Error as e:`: Catches database errors in `create_dummy_db`.
- `print(f"Database error: {e}")`: Prints error message for `create_dummy_db`.
- `finally:`: `finally` block for `create_dummy_db`.
- `if conn:`: Checks if connection exists in `create_dummy_db`.
- `conn.close()`: Closes the connection in `create_dummy_db`.
- `create_dummy_db()`: Calls the `create_dummy_db` function to set up the database.
- `user_input = input("Enter user ID: ")`: Prompts the user to enter a user ID and stores the input in the `user_input` variable.
- `user_data = get_user_data(user_input)`: Calls the `get_user_data` function with the user's input and stores the returned data in the `user_data` variable.
- `if user_data:`: Checks if `user_data` is not `None` (meaning data was successfully retrieved).
- `print("User Data:", user_data)`: If data was retrieved, this line prints the retrieved user data.
- `else:`: If `user_data` is `None` (meaning data retrieval failed or no user with that ID was found).

- `print("Failed to retrieve user data.")`: This line prints a message indicating that data retrieval failed.

Task-5:

Code & Output:



```python
def calculate(num1, num2, operation):
    """
    Performs a basic arithmetic operation between two numbers.

    Args:
        num1: The first number.
        num2: The second number.
        operation: The operation to perform ('add', 'sub', 'mul', 'div').

    Returns:
        The result of the operation, or None if the operation is invalid
        or division by zero occurs.
    """
    if operation == "add":
        return num1 + num2
    elif operation == "sub":
        return num1 - num2
    elif operation == "mul":
        return num1 * num2
    elif operation == "div":
        if num2 == 0:
            print("Error: Division by zero")
            return None # Or raise a ZeroDivisionError
        return num1 / num2
    else:
        print(f"Error: Invalid operation '{operation}'")
        return None # Or raise a ValueError

print(calculate(10, 5, "add"))
print(calculate(10, 0, "div"))
print(calculate(10, 5, "mod")) # Example of invalid operation
```

```
15
Error: Division by zero
None
Error: Invalid operation 'mod'
None
```

Explanation:

Here's a breakdown:

- **`def calculate(num1, num2, operation):`** : This defines the function `calculate` which takes two numbers (`num1` , `num2`) and a string `operation` as input. The variable names are more descriptive than in the original code (`x` , `y` , `z`).
- **Docstring (`""" ... """`)**: This is a docstring, which explains what the function does, its arguments (`Args:`), and what it returns (`Returns:`). This is a good practice for code documentation.
- **`if operation == "add": ... elif operation == "sub": ...`** : This block of `if-elif` statements checks the value of the `operation` string to determine which arithmetic operation to perform.
- **`if operation == "div":`** : Specifically for division, there's an additional check.
- **`if num2 == 0:`** : This checks if the second number (`num2`) is zero.
- **`print("Error: Division by zero")`** : If `num2` is zero, it prints an error message to the console.
- **`return None`** : It returns `None` in case of division by zero, indicating that the operation could not be completed successfully. You could also choose to raise a `ZeroDivisionError` here if you prefer to handle errors with exceptions.
- **`return num1 / num2`** : If `num2` is not zero, it performs the division and returns the result.
- **`else:`** : This block handles cases where the `operation` string does not match any of the defined operations ("add", "sub", "mul", "div").
- **`print(f"Error: Invalid operation '{operation}'")`** : It prints an error message indicating that the provided operation is invalid, including the specific invalid operation entered by the user.
- **`return None`** : It returns `None` for invalid operations. You could also choose to raise a `ValueError` here.
- **`print(calculate(10, 5, "add"))`** : This line calls the `calculate` function to perform addition (10 + 5) and prints the result (15).
- **`print(calculate(10, 0, "div"))`** : This line calls the `calculate` function to perform division (10 / 0). Because of the error handling, it will print "Error: Division by zero" and then print `None` (the return value in case of this error).
- **`print(calculate(10, 5, "mod"))`** : This line calls the `calculate` function with an invalid operation ("mod"). It will print "Error: Invalid operation 'mod'" and then print `None` .

In essence, this refactored code is more robust and user-friendly due to better naming, clear structure, and explicit handling of potential errors like division by zero and invalid operations.