






Question: 1(a)(i)

```
def IterativeFactorial(Number):  
    if Number == 0:  
        return 1  
    else:  
        Result = 1  
        for Count in range(1, Number + 1):  
            Result *= Count # Multiply Result by the loop counter  
        return Result
```

Answer

One mark each to a max of 5



-  **Function header:** Correct definition of `IterativeFactorial(Number)` taking an integer parameter.
-  **Base Case:** Properly handling the base case (`if Number == 0: return 1`)
-  **Iterative Loop:** Using a `for` loop (or equivalent) to iterate from 1 to `Number`.
-  **Multiplication:** Multiplying the `Result` by the loop counter within the loop.
-  **Return:** Returning the calculated factorial (`Result`) at the end of the function.

Marks: 5

Question: 1(a)(ii)

```
print(IterativeFactorial(5))    # Output: 120  
print(IterativeFactorial(0))    # Output: 1  
print(IterativeFactorial(3))    # Output: 6
```

One mark each

-  **Calling the function:** Calling `IterativeFactorial()` with given values.
-  **Output:** Printing (or otherwise outputting) the returned result.

Marks: 2

One mark for a screenshot clearly demonstrating these outputs:


-  The result of calculating the factorials. (which should be 120,1,6).

Marks: 1

Question: 1(b)(i)


```
def RecursiveFactorial(Number):  
    if Number == 0: # Base case  
        return 1  
    else:  
        return Number * RecursiveFactorial(Number - 1) # Recursive case
```

One mark each, One additional for correct recursive case code

 **Function Header:** Correct definition of `RecursiveFactorial(Number)` taking an integer parameter.

 **Base Case:** Handling the base case (`if Number == 0: return 1`).

 **Recursive Case:** The logic for making the recursive call:


 If `Number` is greater than 0: `return Number * RecursiveFactorial(Number - 1)`

Marks: 5

Question: 1(b)(ii)

```
print(R RecursiveFactorial(5)) # Output: 120  
print(R RecursiveFactorial(0)) # Output: 1  
print(R RecursiveFactorial(3)) # Output: 6
```

One mark each

 **Calling the function:** Calling `RecursiveFactorial()` with given values.

 **Output:** Printing (or otherwise outputting) the returned result.


Marks: 2


One mark for a screenshot clearly demonstrating these outputs:

 The result of calculating the factorials. (which should be 120,1,6).

Marks: 1

Important Notes

 **Flexibility:** Some flexibility will be inherent in judging code correctness, especially given potential variations in Python code.

 **Partial Credit:** Consider partial credit if a student's code has minor errors but demonstrates understanding of the core concepts.

Question 2

Part (a) - Program Code Declaration

```
# UDT (User Defined Type)
class ProductUDT:
    def __init__(self):
        self.ID = 0          # None can be used instead of 0
        self.Name = ''      # None can be used instead of ''


# ADT (Abstract Data Type) - Stack to store ProductUDT
ProductStack = [ProductUDT() for i in range(50)]

# Global Variable - TopPointer to indicate the
# last occupied position in the stack
TopPointer = -1
```


Marks: 4

 Correct UDT definition:

Marks: 2

 Declaration of ProductStack with space for 50 records:

Mark: 1


 Correct initialization of TopPointer to -1:


Mark:1


Question 2 Part (b)(i)

```
def push(Product):
    global TopPointer
    if TopPointer == 49:
        print("Stack overflow! Unable to add more products.")
    else:
        TopPointer += 1
        ProductStack[TopPointer] = Product
```

Marks: 3

 **Overflow Check (1 mark):** Award 1 mark for correctly checking if the stack is full (TopPointer == 49) to prevent overflow.


 **Pointer Increment (1 mark):** Award 1 mark for correctly incrementing TopPointer before adding a new product.


 **Product Addition (1 mark):** Award 1 mark for correctly adding the product to ProductStack at the new TopPointer position.


Question 2 Part (b)(ii)


```
def EncodeShipment():  
    file = open("shipment_data.txt", "rt")  
    for line in file:  
        thisRecord = ProductUDT()  
        thisRecord.ID, thisRecord.Name = line.split()  
        push(thisRecord)  
    file.close()
```

Marks: 4

 **File Handling (1 mark):** Properly opening and closing the shipment data file. Award 1 mark for correctly using `open("shipment_data.txt", "rt")` for reading and ensuring the file is closed with `file.close()`.

 **Looping Through File Lines (1 mark):** Correctly iterating over each line in the file. Award 1 mark for the correct use of `for` for reading file: to process each line of the file.


 **Data Parsing and Variable Creation (1 mark):** Successfully parsing each line into product ID and name, and creating a `ProductUDT` variable with these values. Award 1 mark for splitting the line and assigning values to `thisRecord.ID` and `thisRecord.Name` correctly.


 **Pushing to Stack (1 mark):** Correctly using the `push` function to add the `thisRecord` variable to the stack. Award 1 mark for the correct use of `push(thisRecord)` within the loop to add each product record to the stack.

Question 2 Part b(iii)

```
def pop():  
    global TopPointer  
    if TopPointer == -1:  
        print("Stack underflow! No products to remove.")  
        return None  
    else:  
        thisRecord = ProductStack[TopPointer]  
        TopPointer -= 1  
        return thisRecord
```

Marks: 3

 **Underflow Check (1 mark):** Award 1 mark for correctly checking if the stack is empty to handle stack underflow.

 **Pointer Decrement and Item Removal (1 mark):** Award 1 mark for correctly decrementing `TopPointer` and removing the top item from the stack.


 **Returning the Removed Item (1 mark):** Award 1 mark for correctly returning the removed item from the stack, or `None` in case of underflow.


Question 2 Part b(iv)


```
import struct
def DecodeShipment():
    record_format = 'i46s'
    record_size = struct.calcsize(record_format)


    file = open("shipment_data.dat", "wb")
    while TopPointer != -1:
        thisRecord = pop()
        file.write(struct.pack(record_format, int(thisRecord.ID),
                               thisRecord.Name.encode()))
    file.close()
```

Marks: 4

 **Correct Use of struct for Data Formatting (1 mark):** Award 1 mark for correctly defining and using the struct module to specify the format of the binary data (`record_format = 'i46s'`), including correct usage of `struct.calcsize` to determine the size of each record.

 **Proper File Handling (1 mark):** Award 1 mark for correctly opening the binary file for writing (`"wb"`) and ensuring it is properly closed with `file.close()`.



 **Looping and Pop Operations (1 mark):** Award 1 mark for correctly using a loop to pop records from the stack until it is empty (`while TopPointer != -1:`), ensuring the function handles the stack data correctly.

 **Binary Data Writing (1 mark):** Award 1 mark for the correct use of `struct.pack` to format each record according to `record_format` and writing it to the file (`file.write(struct.pack(...))`), including the correct encoding of the string data (`thisRecord.Name.encode()`).

Part (c) - Direct Product Lookup in Binary File

```
def RetrieveProductName(product_id):  
    record_format = 'i46s'  
    record_size = struct.calcsize(record_format)  
    offset = (product_id - 1) * record_size  
    # * 50 can be used instead of record_size as  
    # the record size is fixed at 50 bytes  
  
    file = open("shipment_data.dat", "rb")  
    file.seek(offset)  
    data = file.read(record_size)  
    file.close()  
  
    if not data:  
        return "Product not found"  
    else:  
        thisID, thisName = struct.unpack(record_format, data)  
        if thisID == product_id:  
            return thisName.decode()  
        else:  
            return "Product not found"
```

Marks 6


-  **Correct Use of struct for Data Unpacking (1 mark):** Award 1 mark for correctly using the struct module to specify the binary data format and unpack the data (struct.unpack(record_format, data)).
-  **Proper File Handling (1 mark):** Award 1 mark for correctly opening the file in binary read mode ("rb"), properly closing the file with file.close(), and handling file operations correctly.
-  **Correct Calculation of Offset (1 mark):** Award 1 mark for correctly calculating the offset to find the specific product record based on its ID (offset = (product_id - 1) * record_size). This includes understanding how the record size relates to the product ID and file structure.
-  **Seeking and Reading the Correct Record (1 mark):** Award 1 mark for using file.seek(offset) to navigate to the correct position in the file and file.read(record_size) to read the data of the correct size.
-  **Handling Product Not Found (1 mark):** Award 1 mark for correctly handling cases where the product is not found either due to reading an empty data segment or the product ID not matching (if not data: and else: return "Product not found").
-  **Correctly Decoding and Returning Product Name (1 mark):** Award 1 mark for correctly decoding the product name from the binary data (thisName.decode()) and returning it when the product ID matches the requested ID. This includes understanding and handling string encoding in binary data files.


Part (d) - Integration and Testing


Part (d)(i):


```
EncodeShipment()  
DecodeShipment()  
print(RetrieveProductName(7))
```

Marks 4: Focusing on the core functions and their execution in sequence:

 **EncodeShipment Implementation (1 mark):** For correctly reading and encoding shipment data, creating product objects, and pushing them onto a stack. This involves correctly handling file operations and data structure manipulation.

 **DecodeShipment Execution (1 mark):** For correctly popping products from the stack, encoding their information using a predetermined structure (`struct.pack`), and writing this information to a binary file. This step requires proper stack handling and binary file operations.

 **RetrieveProductName Accuracy (1 mark):** For correctly calculating the position of the desired product record in the binary file, reading, and decoding it to retrieve the product name for a given product ID, particularly ID 7 in this scenario. Accuracy in seeking, reading, and unpacking the record is crucial.

 **Integration and Flow (1 mark):** For the seamless integration and correct sequential execution of `EncodeShipment`, `DecodeShipment`, and `RetrieveProductName(7)`, leading to the successful retrieval and display of the product name. This mark assesses the overall flow and integration of the functions to achieve the desired outcome.

Part (d)(ii):


Marks 2:


Correct product lookup and resulting output screenshot (2 mark): Award 2 marks for correct attempt for retrieving and displaying the name of the product with ID 7 using

```
RetrieveProductName(7).
```

In the original text file, records are listed sequentially from 1 to 30. When these records are pushed onto the stack, the first record (ID=1) ends up at the bottom of the stack, and the last record (ID=30) ends up at the top. Given the LIFO (Last-In, First-Out) nature of a stack, when records are popped from the stack and written into the binary file (`shipment_data.dat`), the order is reversed compared to the original order in the text file (`shipment_data.txt`). In a stack containing 30 records, the 24th record from the text file would be the 7th record to be popped and written to the binary file. This reversal means that direct access based on the original sequential IDs won't correctly align with their new positions in the binary file and output if the function `RetrieveProductName(7)` will be "Product not found".

Important Notes

 **Flexibility:** Some flexibility will be inherent in judging code correctness, especially given potential variations in Python code.

 **Partial Credit:** Consider partial credit if a student's code has minor errors but demonstrates understanding of the core concepts.

Question 3:

Question 3. Part (a) (i):

```
class SmartDevice:
    # Attributes
    # DECLARE name: STRING
    # DECLARE type: STRING
    # DECLARE serial_number: STRING
    # DECLARE setup_year: STRING
    # DECLARE is_on: BOOLEAN

    # Constructor
    def __init__(self, name, type, serial_number, setup_year, is_on):
        self.__name = name
        self.__type = type
        self.__serial_number = serial_number
        self.__setup_year = setup_year
        self.__is_on = is_on
```

Marking Scheme (Total: 2 Marks)

1. Class Definition and Attribute Encapsulation (1 Mark)

- **0.5 Mark:** Correct class definition. The student has correctly defined a class named SmartDevice.

- **0.5 Mark:** Proper use of encapsulation. The student has correctly declared all attributes (name, type, serial_number, setup_year, is_on) as private by prefixing them with double underscores, indicating a solid understanding of encapsulation and privacy in Python classes.

2. Constructor Definition and Attribute Initialization (1 Mark)

- **0.5 Mark:** Correct constructor (__init__) definition. The student has successfully defined an __init__ method that takes self and five additional parameters (name, type, serial_number, setup_year, is_on).

- **0.5 Mark:** Proper initialization of attributes. The student correctly initializes all private attributes within the constructor using the provided parameters, demonstrating an understanding of how to set initial values for object attributes in Python.

Question 3. Part (a) (ii):

```
# Accessors/Getters
def get_name(self):
    return self.__name

def get_type(self):
    return self.__type

def get_serial_number(self):
    return self.__serial_number

def get_setup_year(self):
    return self.__setup_year

# Method to check the device's power status
def is_device_on(self):
    return self.__is_on

# Mutators/Setters
def set_name(self, name):
    self.__name = name

def set_type(self, type):
    self.__type = type

def set_serial_number(self, serial_number):
    self.__serial_number = serial_number

def set_setup_year(self, setup_year):
    self.__setup_year = setup_year

def set_is_on(self, is_on):
    self.__is_on = is_on
```

Marking Scheme (Total: 3 Marks)

1. Implementation of Accessor Methods (1 Mark)

- **0.2 Mark** for each correctly implemented accessor method. The student should have provided accessor methods (`get_name`, `get_type`, `get_serial_number`, `get_setup_year`, `is_device_on`) that return the value of the respective private attributes. The implementation of `is_device_on` as a method to check the device's power status, rather than naming it as a getter, can also be awarded full marks for understanding its contextual use.

2. Implementation of Mutator Methods (1 Mark)

- **0.2 Mark** for each correctly implemented mutator method. The student must define mutator methods (`set_name`, `set_type`, `set_serial_number`, `set_setup_year`, `set_is_on`) that correctly assign a new value to the class's private attributes. Marks should be allocated based on whether each method correctly updates the respective attribute.

3. Adherence to Naming Conventions and Functional Correctness (1 Mark)

- **0.5 Mark:** Adherence to standard Python naming conventions and clarity in method names. This includes using the `get_` prefix for accessor methods and `set_` prefix for mutator methods, as well as the use of `is_` prefix for boolean-returning methods which is followed in the `is_device_on` method.

- **0.5 Mark:** Functional correctness of all methods. Each accessor should return the correct attribute value, and each mutator should correctly update the attribute value. The `is_device_on` method should return a boolean value indicating the device's power status.

Question 3. Part (a) (iii):

```
def get_age_in_years(self):  
    return 2024 - self.__setup_year
```

Marking Scheme (Total: 2 Marks)

1. Correctness of Logic and Calculation (1 Mark)

- **1 Mark:** The method correctly calculates the age of the device by subtracting the `setup_year` from the current year (2024). The method must correctly handle the subtraction operation, implying that `self.__setup_year` is properly converted or handled as an integer if necessary, considering Python's dynamic type system.

2. Handling of Data Types and Error Checking (1 Mark)

- **0.5 Mark:** Correct handling of data types. Since `self.__setup_year` is indicated as a string in the class attribute comments, the student must convert this string to an integer for the subtraction operation to work correctly. If the student directly performs the operation without type conversion where needed, they would miss out on this mark.

- **0.5 Mark:** Implementation of error checking or handling potential exceptions. This could involve ensuring that the `setup_year` attribute contains a valid year before attempting the calculation. While not explicitly required by the prompt, demonstrating awareness of potential errors (e.g., `ValueError` from incorrect string-to-integer conversion or handling future setup years) and including safeguards against these issues would merit full marks.

Question 3. Part (a) (iv):

```
def turn_on(self):  
    self.__is_on = True  
  
def turn_off(self):  
    self.__is_on = False
```

Marking Scheme (Total: 3 Marks)

1. Correct Implementation and Functionality (1.5 Marks)

- **0.75 Mark** for `turn_on` method: Award full marks if the method correctly sets the `__is_on` attribute to `True`, indicating the device is turned on. This demonstrates the student's understanding of how to manipulate class attributes to reflect changes in the object's state.

- **0.75 Mark** for `turn_off` method: Award full marks if the method correctly sets the `__is_on` attribute to `False`, indicating the device is turned off. Like the `turn_on` method, this score reflects the student's ability to accurately change object states through method calls.

2. Adherence to Object-Oriented Principles (1 Mark)

- **0.5 Mark:** Encapsulation. The student uses private attributes (`__is_on`) and provides methods to modify its value, adhering to the encapsulation principle by not allowing direct access to the class's internal state.

- **0.5 Mark:** Simplicity and readability. The methods are straightforward, achieving their intended functionality with minimal and clear code. This also reflects the principle of simplicity in object-oriented design, where actions should be as direct as possible.

3. Impact on Class Functionality and Usability (0.5 Marks)

- **0.5 Mark:** These methods significantly enhance the class's usability by providing clear and direct ways to change the power status of the device. If the methods are correctly implemented, they allow for the intuitive control of the device's state, which is a critical aspect of a smart device's functionality.

Question 3. Part (b) (i):

```
class SmartLight(SmartDevice):  
    # Attributes  
    # DECLARE brightness: INTEGER  
  
    # Constructor  
    def __init__(self, name, type, serial_number, setup_year, is_on, brightness):  
        # Call the parent class constructor  
        super().__init__(name, type, serial_number, setup_year, is_on)  
        self.__brightness = brightness  
  
    # Mutator/Setter  
    def set_brightness(self, brightness):  
        self.__brightness = brightness
```

Marking Scheme (Total: 2 Marks)**1. Integration with Parent Class (1 Mark)**

- **0.5 Mark:** Correctly calling the parent class constructor with `super().__init__(name, type, serial_number, setup_year, is_on)`. This mark is awarded if the student demonstrates a proper understanding of inheritance in Python by using `super()` to initialize the parent class attributes.

- **0.5 Mark:** Proper integration of parent class attributes with subclass-specific attributes. This involves not only calling the parent class's constructor correctly but also correctly initializing the subclass-specific attribute (`__brightness`) in the `__init__` method.

2. Implementation of Subclass-Specific Functionality (1 Mark)

- **0.5 Mark:** Correct declaration and initialization of the subclass-specific attribute (`__brightness`). This mark is awarded for correctly declaring the brightness attribute as private and initializing it within the subclass's constructor.

- **0.5 Mark:** Correct implementation of the `set_brightness` method. This includes correctly defining the method to update the `__brightness` attribute. Full marks are awarded for ensuring the method effectively changes the brightness level, adhering to good practices such as checking if the provided brightness value is within a logical range (if specified or implied by the context, such as brightness levels being within 0-100%).

Question 3. Part (b) (ii):

```
class SmartThermostat(SmartDevice):  
    # Attributes  
    # DECLARE temperature: INTEGER  
  
    # Constructor  
    def __init__(self, name, type, serial_number, setup_year, is_on, temperature):  
        # Call the parent class constructor  
        super().__init__(name, type, serial_number, setup_year, is_on)  
        self.__temperature = temperature
```

```
# Mutator/Setter
def set_temperature(self, temperature):
    self.__temperature = temperature

# Method to suggest temperatures based on the season
def suggest_temperature(self, season):
    if season == "winter":
        # suggest and set any temperature
        print("Suggested temperature: 20°F")
        # set temperature
        self.__temperature = 20 # May call set_temperature() method
    elif season == "summer":
        # suggest and set any temperature
        print("Suggested temperature: 25°F")
        # set temperature
        self.__temperature = 25 # May call set_temperature() method
    else:
        # suggest and set any temperature
        print("Suggested temperature: 22°F")
        # set temperature
        self.__temperature = 22 # May call set_temperature() method
```

Marking Scheme (Total: 2 Marks)

1. Effective Use of Inheritance and Attribute Management (1 Mark)

- **0.5 Mark:** Correct use of inheritance. This includes properly calling the parent class constructor using `super().__init__(name, type, serial_number, setup_year, is_on)` to initialize inherited attributes. Demonstrating a good understanding of inheritance principles is key.

- **0.5 Mark:** Correct declaration and initialization of the new attribute (`__temperature`). Award marks for appropriately handling the temperature attribute as private and initializing it in the subclass constructor, indicating an understanding of encapsulation and attribute management within subclasses.

2. Logical Implementation of Temperature Control Methods (1 Mark)

- **0.5 Mark:** Correct implementation of the `set_temperature` method. This method should properly update the `__temperature` attribute. Marks are given for accurately enabling temperature adjustments, which is essential for a thermostat's functionality.

- **0.5 Mark:** Appropriate implementation of the `suggest_temperature` method. This method's logic should suggest and optionally set temperatures based on the season. Full marks are for correctly handling different seasons with logical temperature suggestions. Bonus consideration could be given for using the `set_temperature` method to update the temperature, which would demonstrate good practice in utilizing existing methods for attribute modifications. However, the direct assignment in the provided implementation still achieves the method's goal.

Question 3. Part (c) (i):

```
# Create a SmartLight object
living_room_light = SmartLight("Living Room Light", "light", "LR001", 2024, False, 0)

# Turn on the SmartLight object and set its brightness to 75
living_room_light.turn_on()
living_room_light.set_brightness(75)
```

```
# Change the brightness to 50
living_room_light.set_brightness(50)

# Turn off the SmartLight object
living_room_light.turn_off()
```

Marking Scheme (Total: 4 Marks)

1. Correct Object Instantiation (1 Mark)

- **1 Mark:** Properly creating a SmartLight object with the specified attributes ("Living Room Light", "light", "LR001", 2024, False, 0). This step tests the student's ability to correctly instantiate an object of a class, including passing the correct initial values for both inherited and subclass-specific attributes.

2. Using Methods to Manipulate Object State (2 Marks)

- **0.5 Mark:** Correctly turning on the SmartLight object using the turn_on method. This action should effectively change the object's __is_on attribute to True.

- **1 Mark:** Appropriately using the set_brightness method twice to first set the brightness to 75 and then adjust it to 50. This requires the student to understand how to use mutator methods to change an object's internal state.

- **0.5 Mark:** Correctly turning off the SmartLight object using the turn_off method, thereby setting the __is_on attribute to False. This demonstrates the ability to revert the object's state.

3. Understanding and Application of Object-Oriented Principles (1 Mark)

- **0.5 Mark:** Effective use of encapsulation. This is demonstrated through the use of methods to modify private attributes (__is_on and __brightness) instead of direct access, adhering to good object-oriented programming practices.

- **0.5 Mark:** Demonstrating an understanding of how object states are manipulated using methods. The student needs to show they can sequence method calls (turn on, set brightness, change brightness, turn off) to achieve specific outcomes, reflecting an understanding of object behavior over time.

Question 3. Part (c) (ii):

```
# Create a SmartThermostat object
main_thermostat = SmartThermostat("Main Thermostat", "thermostat", "MT001", 2024, True,
68)

# Adjust the temperature to 70°F
main_thermostat.set_temperature(70)

# Use the suggest_temperature() method with "winter" as the parameter
main_thermostat.suggest_temperature("winter")
```

Marking Scheme (Total: 4 Marks)

1. Correct Object Instantiation and Initial State Setting (1 Mark)

- **1 Mark:** Accurately creating a SmartThermostat object with the specified attributes and correctly initializing it with given values. This step assesses the student's ability to instantiate an object with both inherited and new attributes, setting an initial state.

2. Proper Use of Method to Adjust Object Attributes (1 Mark)

- **1 Mark:** Correctly using the `set_temperature` method to adjust the thermostat's temperature to 70°F. This reflects an understanding of object manipulation through methods and encapsulation by modifying private attributes via a public interface.

3. Logical Use of Seasonal Temperature Suggestion Method (1.5 Marks)

- **0.75 Mark:** Successfully calling the `suggest_temperature` method with "winter" as the argument, demonstrating an understanding of how to use object methods with parameters to influence behavior based on external inputs.

- **0.75 Mark:** The logic inside the `suggest_temperature` method correctly suggests and sets a temperature suitable for the winter season. While the explicit output might be more indicative of method functionality rather than state change, the intent to use seasonal data to adjust the object's state is key. In this scenario, acknowledging the attempt to adapt the object's functionality based on seasonal input is crucial, even though the actual temperature suggestion (like "20°F") and setting might not directly reflect realistic or expected values.

4. Application of Object-Oriented Principles and Method Output Handling (0.5 Marks)

- **0.5 Mark:** Demonstrating an understanding of inheritance and encapsulation through the use and extension of a base class method (`suggest_temperature`) that not only acts based on input but also potentially alters the object's state in a way that's visible externally (via print statements or attribute changes). This mark emphasizes the student's ability to extend base class functionality in a meaningful way, showcasing proficiency in object-oriented programming concepts.

Question 3. Part (c) (iii):

Initialize a SmartLight object for the living room

```
living_room_light = SmartLight("Living Room Light", "light", "LR002", 2024, False, 0)
```

Initialize a SmartThermostat object for the home

```
home_thermostat = SmartThermostat("Home Thermostat", "thermostat", "HT001", 2024, True, 68)
```

Morning (7:00 AM)

```
living_room_light.turn_on()
```

```
living_room_light.set_brightness(60)
```

```
home_thermostat.set_temperature(70)
```

Output the status of each device at 7:00 AM

```
print("Morning Status:")
```

```
print("Living Room Light Status:")
```

```
print("Is On:", living_room_light.is_device_on())
```

```
print("Brightness:", living_room_light.get_brightness())
```

```
print("Home Thermostat Status:")
```

```
print("Temperature:", home_thermostat.get_temperature())
```

Day (9:00 AM to 5:00 PM)

```
living_room_light.turn_off()
```

```
home_thermostat.set_temperature(68)
```

Output the status of each device at 9:00 AM

```
print("Day Status:")
```

```
print("Living Room Light Status:")
```

```
print("Is On:", living_room_light.is_device_on())
```

```
print("Home Thermostat Status:")
```

```
print("Temperature:", home_thermostat.get_temperature())
```

```
# Evening (6:00 PM)
living_room_light.turn_on()
living_room_light.set_brightness(70)
home_thermostat.set_temperature(72)
# Output the status of each device at 6:00 PM
print("Evening Status:")
print("Living Room Light Status:")
print("Is On:", living_room_light.is_device_on())
print("Brightness:", living_room_light.get_brightness())
print("Home Thermostat Status:")
print("Temperature:", home_thermostat.get_temperature())

# Night (10:00 PM)
living_room_light.turn_off()
home_thermostat.set_temperature(65)
# Output the status of each device at 10:00 PM
print("Night Status:")
print("Living Room Light Status:")
print("Is On:", living_room_light.is_device_on())
print("Home Thermostat Status:")
print("Temperature:", home_thermostat.get_temperature())
```

Marking Scheme (Total: 8 Marks)

1. Correct Initialization of Objects (2 Marks)

- **1 Mark** for correctly initializing the SmartLight object with the specified attributes.
- **1 Mark** for correctly initializing the SmartThermostat object with the specified attributes.

2. Accurate Manipulation of Object States (3 Marks)

- **0.75 Mark** for correctly manipulating the SmartLight object's state throughout the day (turning on/off and setting brightness).
- **0.75 Mark** for correctly manipulating the SmartThermostat object's temperature throughout the day.
- **1.5 Marks** for logical sequencing of operations corresponding to the different times of day (Morning, Day, Evening, Night), including adjusting settings appropriate to each time period. This part evaluates the student's ability to envision and implement a realistic scenario where object states are changed to reflect typical daily patterns.

3. Outputting Status of Each Device Correctly (2 Marks)

- **1 Mark** for correctly outputting the status of the SmartLight object, including its power state and brightness level at each specified time.
- **1 Mark** for correctly outputting the status of the SmartThermostat object, including its temperature setting at each specified time.
- **These** marks are awarded based on the student's ability to use accessor methods to retrieve object states and output them, demonstrating an understanding of how to interact with and report on object attributes.

4. Understanding and Application of Object-Oriented Principles (1 Mark)

- **0.5 Mark** for effective use of encapsulation, demonstrated by the manipulation of private attributes through public methods across both objects.

- **0.5 Mark** for demonstrating an understanding of inheritance (especially if SmartLight and SmartThermostat methods are shown to extend or utilize functionality from the SmartDevice base class) and method abstraction.