✅ Student Guidance for Assessment (OOP + Dictionary Handling)

🔍 1. Follow Function Names Strictly

- **Function names must exactly match the ones provided in the problem statement:**

✔️ register_book(self, book_title, quantity) # Correct

❌ RegisterBook() # Incorrect (CamelCase, wrong spelling)

Why Important?
If function names are wrong, auto-checkers and unit tests will fail completely — even if your logic is perfect.

🔍 2. __init__() and Empty Dictionary are Mandatory

- **You must define:**

def __init__(self):

self.book_stock = {}

self.member_activity = {}

- **If you miss this or forget the empty dict creation — auto-tests will raise:**

AttributeError: 'LibraryManager' object has no attribute 'book_stock'

****Dont declare above class as global variable**

🔍 3. Always Return the Dictionary (Do Not Print)

return self.book_stock # ✅ Correct

print(self.book_stock) # ❌ Wrong: Returns None to tester

- **Only return statements are tested — print() does nothing useful for automated checking.**

🔍 4. Dictionary Update & Access — Common Mistakes

✔️ Update Correctly:

self.book_stock[book_title] += quantity

❌ Wrong (will cause KeyError if key missing):

self.book_stock[book_title] =
self.book_stock[book_title] + quantity # May fail if key not yet added

---

🔍 5. Handle ‘Out of Stock’ Properly

- When stock reaches zero — assign "Out of Stock" string (as asked):

if self.book_stock[book] == 0:

    self.book_stock[book] = 'Out of Stock'

- Forgetting this = wrong output format.

🔍 6. Handle Division Carefully (Avoid ZeroDivisionError)

For average calculation:

if len(self.member_activity) == 0:

    return 0.0

average = total_pages / total_members

❌ Forgetting this leads to:

ZeroDivisionError: division by zero

🔍 7. Proper Key Checks in Dictionaries

✔️ Before updating member or book:

if member_name not in self.member_activity:

self.member_activity[member_name] = 0

- **Otherwise:**

KeyError: 'UnknownMember'

🔍 8. Consistent Data Types

- **Quantities must remain int or 'Out of Stock' — don’t mix data types unexpectedly.**

🔍 9. Naming & Case Sensitivity

is case-sensitive:

self.book_stock != self.Book_Stock != self.BOOK_STOCK

MCQ Trap: Variable name mismatch = AttributeError in test cases.

🔍 10. No Hardcoding — Always Use Given Parameters

Bad:

self.book_stock[' Book'] = 5 # ❌ Hardcoded value

Good:

self.book_stock[book_title] = quantity # ✅ Uses argument

⚠️ Common Errors from Past Batches:

Mistake	Result
Function name typo	Unit test failure, 0 marks

Using print() instead of return	**None returned to checker**
KeyError due to missing key check	**Runtime failure**
Forgot 'Out of Stock' assignment	**Output mismatch**
Division by zero (average calc)	**ZeroDivisionError**
Hardcoded book/member names	**Wrong output or fixed result**
Wrong data type returned (list instead of dict)	**TypeError in test**

✅ Golden Rules to Remember:

✔️ Function names: Exactly as given
✔️ Use self.book_stock and self.member_activity dictionaries
✔️ Check if key exists before updating
✔️ Do not print() — always return
✔️ Handle division-by-zero gracefully

✔️ **Assign 'Out of Stock' properly**
✔️ **No hardcoding — use function parameters only**
✔️ **Follow return type: dict or float as mentioned**

🎯 **Final Trainer Tip:**

> **“Don’t just read the question — understand the flow of the dictionary updates, and simulate a few operations manually before coding. 90% of student errors are because of blindly typing, not thinking about the dictionary state after each function call.”**

Consolidated possible functions in OOPs

Just created two dictionary for this program but actually You will get only one dictionary

```
import pandas as pd
class LibraryManager:
    def __init__(self):
        """
        Initializes two empty dictionaries:
        1. book_stock: To manage library book inventory.
           Key = Book Title, Value = Quantity
        2. member_activity: To manage member reading activity.
           Key = Member Name, Value = Total Pages Read
        """
        self.book_stock = {}      # Book stock dictionary
        self.member_activity = {} # Member activity dictionary

    # ------------------- Book Inventory Operations -------------------

    def register_book(self, book_title: str, quantity: int) -> dict:
        if book_title in self.book_stock:
            self.book_stock[book_title] += quantity
        else:
            self.book_stock[book_title] = quantity
        return self.book_stock
```

```
def lend_books(self, lend_request: dict) -> dict:
    for book, qty in lend_request.items():
        if book in self.book_stock:
            if isinstance(self.book_stock[book], int) and self.book_stock[book] >= qty:
                self.book_stock[book] -= qty
                if self.book_stock[book] == 0:
                    self.book_stock[book] = 'Out of Stock'
            else:
                self.book_stock[book] = 'Out of Stock'
        else:
            self.book_stock[book] = 'Out of Stock'
    return self.book_stock


def restock_books(self, restock_quantity: int) -> dict:
    for book in self.book_stock:
        if self.book_stock[book] == 'Out of Stock':
            self.book_stock[book] = restock_quantity
    return self.book_stock


def low_stock_report(self, threshold: int) -> dict:
    report = {}
    for book, qty in self.book_stock.items():
        if isinstance(qty, int) and qty < threshold:
            report[book] = qty
    return report
```

```
# ------------------ Member Activity Operations ------------------

def register_member(self, member_name: str) -> dict:
    if member_name not in self.member_activity:
        self.member_activity[member_name] = 0
    return self.member_activity

def log_reading(self, member_name: str, pages_read: int) -> dict:
    if member_name in self.member_activity:
        self.member_activity[member_name] += pages_read
    else:
        self.member_activity[member_name] = pages_read
    return self.member_activity

def average_pages_read(self) -> float:
    if len(self.member_activity) == 0:
        return 0.0
    total_pages = sum(self.member_activity.values())
    total_members = len(self.member_activity)
    return total_pages / total_members

def reading_progress_report(self) -> dict:
    return self.member_activity
```

OOPs Cheatsheet (For MCQs)

---

1. Class & Object Creation

```
class Car:
    def __init__(self, brand):
        self.brand = brand

car1 = Car("Toyota")   # object creation
```

✔️ **init()** is called automatically when an object is created.

---

2. Instance Variables & Methods

```
class Student:
    def __init__(self, name):
        self.name = name   # instance variable

    def display(self):    # instance method
        print(self.name)
```

✔️ Access using `self`.

---

3. Class Variables & Class Methods

```
class Employee:
    company = "TCS"      # class variable

    @classmethod
    def company_name(cls):
        return cls.company
```

✔️ `@classmethod` uses `cls`, can access **class-level data**.

---

4. Static Methods

```
class Math:
    @staticmethod
    def add(x, y):
        return x + y
```

✔️ **No self, no cls** — behaves like a regular function in class.

---

5. Encapsulation (Private Variables)

```
class Bank:
    def __init__(self):
        self._balance = 1000   # protected
        self.__pin = 1234      # private
```

✔️ `__var` → **private variable**
✔️ `_var` → **protected variable** (convention)

---

6. Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    pass
```

```
d = Dog()
d.speak()  # Inherited
```

✔️ **Dog inherits Animal**

---

7. Method Overriding

```
class Dog(Animal):
    def speak(self):    # overrides Animal.speak()
        print("Dog barks")
```

✔️ Child class method replaces parent method.

---

8. Polymorphism

```
for animal in [Dog(), Animal()]:
    animal.speak()
```

✔️ **Same method name**, different behavior based on object type.

---

9. Super() Function

```
class Dog(Animal):
    def __init__(self):
        super().__init__()   # Call parent constructor
```

✔️ Used to **call parent class methods**.

---

10. Magic / Dunder Methods

```
def __str__(self):   # string representation
    return "Object info"
```

✔️ Examples: `__init__()`, `__str__()`, `__len__()`

---

11. Multiple Inheritance

```
class A: pass
class B: pass
class C(A, B): pass
```

✔️ **C inherits from both A & B**

---

12. Abstraction (ABC)

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

✔️ Must implement `area()` in derived class.

---

13. Common MCQ Errors to Watch:

Mistake	Error/Problem
Forgetting `self` in methods	`TypeError: missing 1 required arg 'self'`

Accessing private `__var` directly	`AttributeError`
Incorrect use of `@staticmethod`, `@classmethod`	Unexpected behavior
Overriding method without correct signature	Logic errors
Using parent method without `super()`	Inheritance problems

---

14. Quick Concept Reference Table:

Concept	Keyword/Decorator	Note
Class Variable	Defined outside methods	Shared across all objects
Instance Variable	`self.var`	Unique to each object
Class Method	`@classmethod` + `cls`	Acts on class, not instance
Static Method	`@staticmethod`	No self/cls, utility method
Private Variable	`__var`	Name mangling (`_Class__var`)
Inheritance	`(ParentClass)`	Acquires parent features
Abstraction	`@abstractmethod`	Must be overridden
Polymorphism	Same method diff. class	Depends on object
Superclass Call	`super()`	Calls parent methods

Python OOPs MCQs (Medium & Difficult)

---

1. Which of the following is true about Python class variables?

A) They are shared across all instances.
B) They are unique to each object.
C) They cannot be changed after creation.
D) They must be private.

✔️ **Answer:** A

---

2. What will be the output?

python

```
class Test:
    def __init__(self):
        self.x = 1
t1 = Test()
t2 = Test()
t1.x = 5
print(t2.x)
```

A) 1
B) 5
C) Error
D) None

✔️ **Answer:** A

---

3. What does the `@staticmethod` decorator do?

A) Makes the method receive class (`cls`) as first parameter
B) Makes the method independent of class and instance
C) Prevents method overriding
D) Converts method to abstract

✔️ **Answer:** B

---

4. What is output?

python

```
class A:
    __secret = 5
a = A()
print(a.__secret)
```

A) 5
B) Error
C) None
D) '__secret'

✔️ **Answer:** B (`AttributeError` — private variable)

---

5. What will happen if you call an abstract method in an abstract class without implementing it?

A) Executes successfully
B) Raises TypeError
C) Raises AttributeError
D) Returns None

✔️ **Answer:** B

---

6. Which is the correct method to define a destructor in Python?

A) `__destroy__()`
B) `__delete__()`
C) `__del__()`
D) `destroy()`

✔️ **Answer:** C

---

7. Which of the following is true for method overloading in Python?

A) Python supports method overloading by default.
B) Python does not support method overloading by default.
C) You can overload by giving default parameters.
D) Both B and C.

✔️ **Answer:** D

---

8. How do you access a private variable `__value` of class `A` from outside?

A) `A.__value`
B) `A._A__value`
C) `A.value`
D) Not possible

✔️ **Answer:** B

---

9. What is the output?

python

```
class Parent:
    def show(self):
        print("Parent")
class Child(Parent):
    pass
c = Child()
c.show()
```

A) Error
B) Nothing
C) Parent
D) Child

✔️ **Answer:** C

---

10. Which of the following does NOT create multiple inheritance?

A) `class C(A, B): pass`
B) `class C: pass`
C) `class C(A): pass`
D) None of these

✔️ **Answer:** B

---

11. The default method resolution order (MRO) in Python is:

A) Depth First
B) Breadth First
C) C3 Linearization
D) None

✔️ **Answer:** C

---

12. In Python, multiple inheritance allows:

A) Only one base class
B) No base classes
C) More than one base class
D) Only abstract base class

✔️ **Answer:** C

---

13. What happens if `__str__()` is not defined in a class but `print(object)` is called?

A) Prints address
B) Error
C) Prints 'None'
D) Prints 0

✔️ **Answer:** A

---

14. What will be the output?

python

```
class A:
    def __init__(self):
        print("A", end=' ')
class B(A):
    def __init__(self):
        super().__init__()
        print("B", end=' ')
B()
```

A) B A
B) A B
C) Error
D) None

✔️ **Answer:** B

---

15. What will happen if no constructor is defined in Python class?

A) SyntaxError
B) Python will provide a default `__init__()`
C) Raises Exception
D) Object cannot be created

✔️ **Answer:** B

---

16. Choose the correct way to call a superclass method inside a child class:

A) `super().method()`
B) `ParentClass.method(self)`
C) Both A & B
D) Neither

✔️ **Answer:** C

---

17. What does the `__new__()` method do in Python?

A) Initializes the object
B) Deletes the object
C) Creates the object before `__init__()`
D) Cleans memory

✔️ **Answer:** C

---

18. In multiple inheritance, which class will be called first?

A) Left-most parent class
B) Right-most parent class
C) Random
D) Both at the same time

✔️ **Answer:** A

---

19. Can you instantiate an abstract class directly?

A) Yes
B) No
C) Depends on class
D) Only once

✔️ **Answer:** B

---

20. What is the result of:

python

```
class X:
    count = 0
    def __init__(self):
        X.count += 1
x1 = X()
x2 = X()
print(X.count)
```

A) 0
B) 1
C) 2
D) Error

✔️ **Answer:** C

---

21. The constructor method in Python is:

A) `__init__()`
B) `__new__()`
C) `__construct__()`
D) `__create__()`

✔️ **Answer:** A

---

22. Static methods can be called using:

A) Class Name
B) Object
C) Both A & B
D) Neither

✔️ **Answer:** C

---

23. What is ‘duck typing’ in Python?

A) Type checking at compile time
B) Type checking at run time
C) Checking if object behaves like required type
D) Abstract typing

✔️ **Answer:** C

---

24. What will be the output?

python

```
class Sample:
    pass

print(isinstance(Sample(), Sample))
```

A) True
B) False
C) Error
D) None

✔️ **Answer:** A

---

25. Which dunder method overloads `+` operator?

A) `__add__()`
B) `__sum__()`

C) `__plus__()`
D) `__concat__()`

✔️ **Answer:** A

---

26. Which of the following can be overridden?

A) Constructor
B) `__str__()`
C) `__len__()`
D) All of the above

✔️ **Answer:** D

---

27. Inheritance allows:

A) Code reuse
B) Restriction
C) Security only
D) None

✔️ **Answer:** A

---

28. Private variables can be accessed:

A) Directly
B) Never
C) Using name mangling (`_Class__var`)
D) Only inside class

✔️ **Answer:** C

---

29. Which is not an OOP principle?

A) Inheritance
B) Encapsulation
C) Polymorphism
D) Compilation

✔️ **Answer:** D

---

30. Which is true for an abstract class in Python?

A) Cannot contain normal methods
B) Must be inherited to be useful
C) Can be instantiated directly
D) Is optional in all cases

✔️ **Answer:** B

---

31. When is `__del__()` method called?

A) During object creation
B) When the object is garbage collected
C) At the start of program
D) On class definition

✔️ **Answer:** B

---

32. What is the output?

python

```
class Test:
    def __str__(self):
        return \"Hello\"

print(Test())
```

A) Address of object
B) Hello
C) Error
D) None

✔️ **Answer:** B

---

33. What is true about encapsulation?

A) Hides internal data
B) Forces abstraction

C) Shows private attributes
D) Disables inheritance

✔️ **Answer:** A

---

34. Which is used for abstraction in Python?

A) `@abstractmethod`
B) `super()`
C) `staticmethod`
D) `__dict__`

✔️ **Answer:** A

---

35. Which method is used to override string conversion of an object?

A) `__str__()`
B) `__repr__()`
C) Both A and B
D) `__convert__()`

✔️ **Answer:** C