DAY 22:EVENING ASSESSMENT

11. Use property decorators (@property) for available\_books\_count.

@property

def count\_available\_books(self) -> int:

return sum(1 for bk in self.books.values() if bk.available)

12. Create an abstract base class (ABC) Person, inherited by Member and Librarian.

from abc import ABC, abstractmethod

from abc import ABC, abstractmethod

class Human(ABC):

def \_\_init\_\_(self, uid: str, full\_name: str) -> None:

self.uid = uid

self.full\_name = full\_name

@abstractmethod

def identity(self) -> str: ...

class Patron(Human):

def identity(self) -> str:

return "Member"

class Curator(Human):

def identity(self) -> str:

return "Librarian"

13. Demonstrate multiple inheritance by creating a ResearchScholar who is both

StudentMember and FacultyMember.

class StudentMember(Member):

def role(self) -> str:

return "student\_member"

class FacultyMember(Member):

def role(self) -> str:

return "faculty\_member"

class ResearchScholar(StudentMember, FacultyMember):

def role(self) -> str:

return "research\_scholar"

14. Override \_\_str\_\_ and \_\_repr\_\_ methods for clean debugging outputs.

class Book:

def \_\_init\_\_(self, book\_id: str, title: str, author: str, isbn: str, available: bool = True):

self.book\_id = book\_id

self.title = title

self.author = author

self.isbn = isbn

self.available = available

def \_\_repr\_\_(self):

return (f"Book(book\_id={self.book\_id!r}, title={self.title!r}, "

f"author={self.author!r}, isbn={self.isbn!r}, available={self.available!r})")

def \_\_str\_\_(self):

status = "Available" if self.available else "Issued"

return f"[{self.book\_id}] {self.title} by {self.author} ({self.isbn}) — {status}"

class Member(Person):

def role(self) -> str:

return "member"

def \_\_repr\_\_(self):

return f"Member(user\_id={self.user\_id!r}, name={self.name!r})"

def \_\_str\_\_(self):

return f"{self.user\_id} – {self.name} ({self.role()})"

15. Add a Singleton pattern for the Library class (only one instance should exist).

class \_SingletonMeta(type):

\_instance = None

def \_\_call\_\_(cls, \*args, \*\*kwargs):

if cls.\_instance is None:

cls.\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instance

class LibrarySystem(metaclass=\_SingletonMeta):

def \_\_init\_\_(self):

self.books = {}

self.users = {}

self.transactions = []

self.\_load()

16. Implement a Factory Method for creating different types of members (StudentMember,

FacultyMember).

class PatronFactory:

@staticmethod

def build(kind: str, uid: str, name: str) -> Human:

k = (kind or "").strip().lower()

if k == "student":

return StudentPatron(uid, name)

if k == "faculty":

return FacultyPatron(uid, name)

if k == "librarian":

return Curator(uid, name)

return Patron(uid, name) # default

17. Add Method Chaining support, e.g., library.add\_book(...).register\_member(...).

def insert\_book(self, bid, title, author, isbn):

if bid in self.books:

raise ValueError("Book already exists")

self.books[bid] = Book(bid, title, author, isbn)

self.\_save()

return self

18. Create a mixin class that adds to\_json() and from\_json() methods for books and members.

class DictJsonMixin:

def to\_dict(self) -> dict:

return dict(self.\_\_dict\_\_)

@classmethod

def from\_dict(cls, data: dict):

return cls(\*\*data)

class Book(DictJsonMixin):

def \_\_init\_\_(self, book\_id, title, author, isbn, available=True):

self.book\_id = book\_id

self.title = title

self.author = author

self.isbn = isbn

self.available = available

19. Write custom exception classes: BookNotAvailableError, MemberNotFoundError.

class BookNotAvailableError(Exception):

pass

class MemberNotFoundError(Exception):

pass

class BookNotFoundError(Exception):

pass

def issue\_book(self, user\_id: str, book\_id: str):

user = self.users.get(user\_id)

if user is None:

raise MemberNotFoundError("User not found")

book = self.books.get(book\_id)

if book is None:

raise BookNotFoundError("Book not found")

if not book.available:

raise BookNotAvailableError("Book already issued")

issued\_at = datetime.now()

due\_at = issued\_at + timedelta(days=LOAN\_DAYS)

self.transactions.append({

"book\_id": book\_id,

"user\_id": user\_id,

"issue\_date": issued\_at.isoformat(),

"due\_date": due\_at.isoformat(),

"return\_date": None,

})

book.available = False

self.\_save()

logging.info(f"Issued {book\_id} to {user\_id} (due {due\_at.date()})")

return self

20. Add a try-except-else-finally block around file reading/writing with proper error logging.

def \_hydrate\_store(self):

try:

if not os.path.exists(STORE):

logging.warning("Store missing; creating a fresh one.")

self.books, self.users, self.transactions = {}, {}, []

self.\_save()

return

with open(STORE, "r", encoding="utf-8") as fh:

payload = json.load(fh)

self.books = {b["book\_id"]: Book(\*\*b) for b in payload.get("books", [])}

self.users = {u["user\_id"]: User(\*\*u) for u in payload.get("users", [])}

self.transactions = payload.get("transactions", [])

except (json.JSONDecodeError, FileNotFoundError) as err:

logging.error(f"Error loading data: {err}")

self.books, self.users, self.transactions = {}, {}, []

self.\_save()

else:

logging.info("Data loaded successfully.")

finally:

logging.debug("Load sequence completed.")