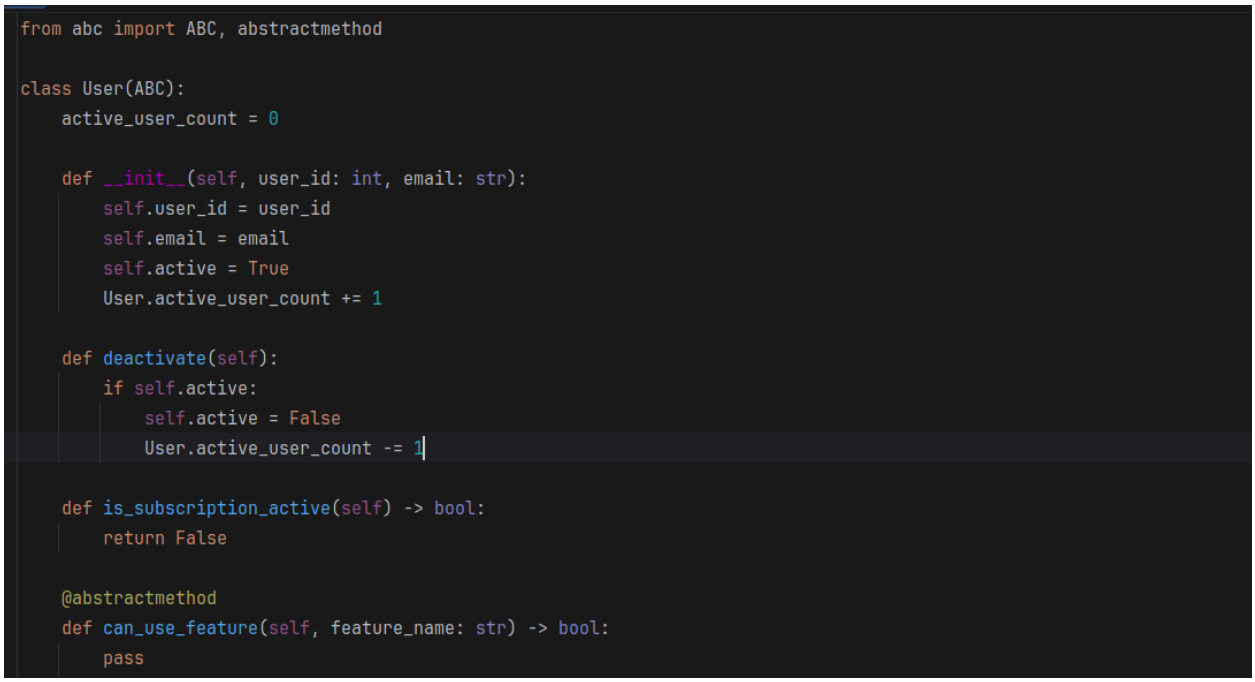
Q1. Object Modeling & Core OOP

Design a **multi-role User Management System** where:

- A base class defines common identity attributes.
- Multiple specialized user types inherit from the base class.
- Each subclass overrides at least one behavior from the parent class.
- A class-level attribute tracks the total number of active users across all subclasses.



```python
from abc import ABC, abstractmethod

class User(ABC):
    active_user_count = 0

    def __init__(self, user_id: int, email: str):
        self.user_id = user_id
        self.email = email
        self.active = True
        User.active_user_count += 1

    def deactivate(self):
        if self.active:
            self.active = False
            User.active_user_count -= 1

    def is_subscription_active(self) -> bool:
        return False

    @abstractmethod
    def can_use_feature(self, feature_name: str) -> bool:
        pass
```

```python
from week4.User import User

class PaidUser(User):
    PAID_FEATURES = {
        "browse_content",
        "playback",
        "search",
        "download_content",
        "ad_free_experience"
    }

    def __init__(self, user_id: int, email: str, package_id: str, payment_method: str):
        super().__init__(user_id, email)
        self.payment_method = payment_method
        self.package_id = package_id
        self.subscription_active = True

    def is_subscription_active(self) -> bool:
        return self.subscription_active

    def can_use_feature(self, feature_name: str) -> bool:
        return self.subscription_active and feature_name in self.PAID_FEATURES

    def cancel_subscription(self):
        self.subscription_active = False
```

```python
from week4.User import User

class FreeUser(User):
    FREE_FEATURES = {"browse_content", "search"}

    def can_use_feature(self, feature_name: str) -> bool:
        return feature_name in self.FREE_FEATURES
```

```python
from week4.FreeUser import FreeUser
from week4.PaidUser import PaidUser
from week4.User import User

free_user = FreeUser( user_id: 1,  email: "preeti.singh@gmail.com")
paid_user = PaidUser( user_id: 2,  email: "pradeep.singh@gmail.com",  package_id: "12345", payment_method: "credit_card")
paid_user_2 = PaidUser( user_id: 3,  email: "kirti.singh@gmail.com",  package_id: "87636", payment_method: "debit_card")

print(free_user.can_use_feature("download_content"))
print(paid_user.can_use_feature("download_content"))

print(paid_user.is_subscription_active())

paid_user.cancel_subscription()
print(paid_user.is_subscription_active())

print(User.active_user_count)

paid_user.deactivate();

print(User.active_user_count)
```

Q2. Advanced Class Construction

Implement:

- A `@classmethod` that creates an object from a single serialized string.
- A `@staticmethod` that performs validation logic independent of class state.
- Demonstrate the difference in responsibility between the two methods through usage.

```python
class Subscriber:
    active_user_count = 0

    def __init__(self, user_id: int, email: str, is_paid: bool):
        self.user_id = user_id
        self.email = email
        self.is_paid = is_paid
        self.active = True
        Subscriber.active_user_count += 1

    @staticmethod
    def is_valid_email(email: str) -> bool:
        return "@" in email and "." in email

    @classmethod
    def from_serialized(cls, data: str):
        user_id_str, email, plan = data.split("|")

        if not cls.is_valid_email(email):
            raise ValueError("Invalid email address")

        is_paid = plan.lower() == "paid"
        return cls(int(user_id_str), email, is_paid)
```

```python
from week4.Subscriber import Subscriber

print(Subscriber.is_valid_email("test@example.com"))
print(Subscriber.is_valid_email("my-test-email"))


serialized_user = "101|paid_user@example.com|paid"

user = Subscriber.from_serialized(serialized_user)
print(user.user_id)
print(user.email)
print(user.is_paid)
```

Q3. Deep Usage of Special Methods

Enhance one core entity by implementing:

- `__str__` for human-readable output
- `__repr__` for developer-friendly debugging

- `__len__` to represent a business-meaningful metric
- `__eq__` and `__lt__` for object comparison
- `__call__` to make the object behave like a function

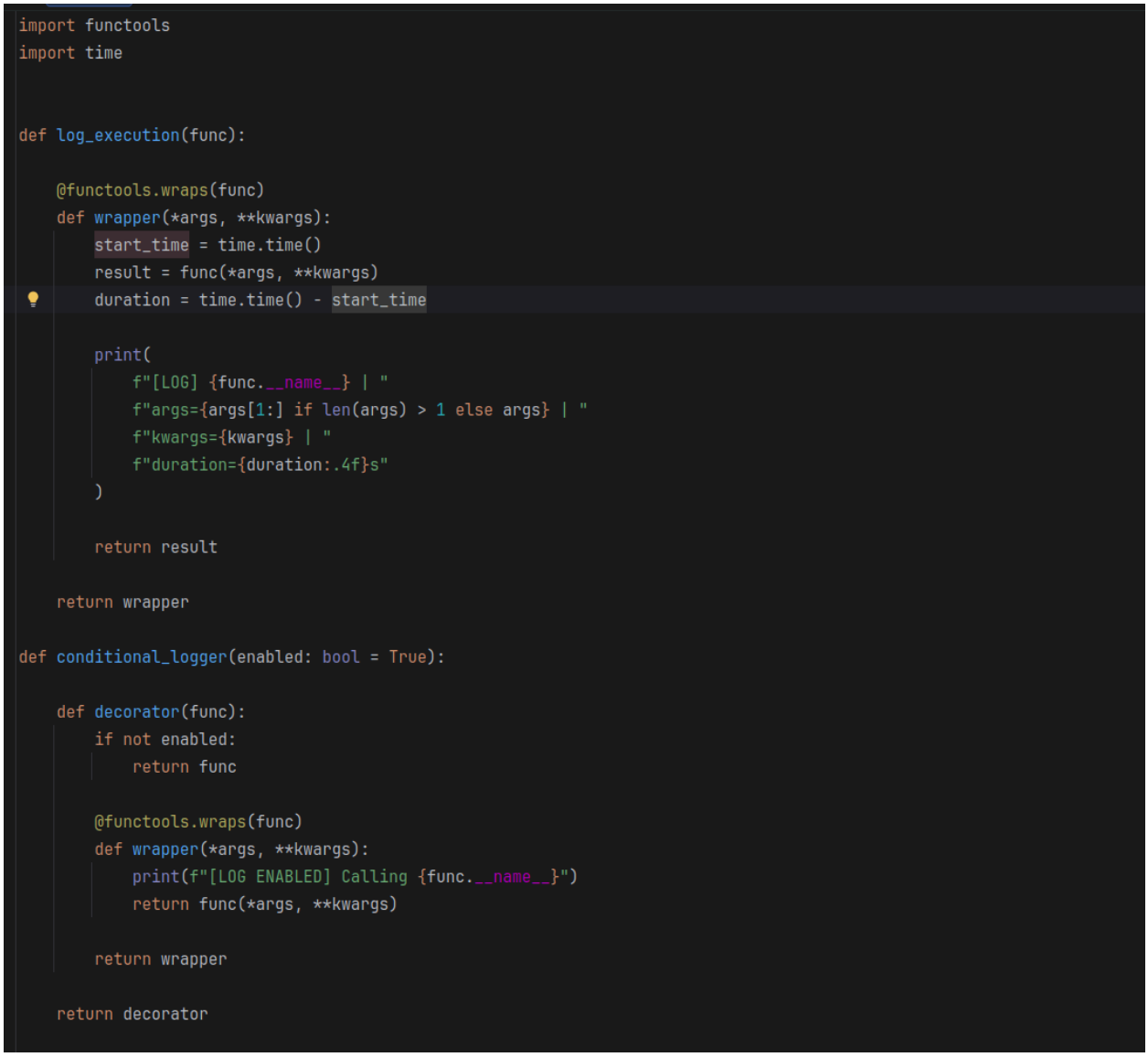
```python
class User:

    def __init__(self, user_id: int, email: str, is_paid: bool, features: list[str]):
        self.user_id = user_id
        self.email = email
        self.is_paid = is_paid
        self.features = features
        self.active = True
        User.active_user_count += 1

    def __str__(self):
        plan = "Paid" if self.is_paid else "Free"
        return f"User {self.user_id} ({plan}) - {self.email}"

    def __repr__(self):
        return (
            f"User(user_id={self.user_id}, "
            f"email='{self.email}', "
            f"is_paid={self.is_paid}, "
            f"features={self.features})"
        )

    def __len__(self):
        return len(self.features)

    def __eq__(self, other):
        if not isinstance(other, User):
            return NotImplemented
        return self.user_id == other.user_id


    def __lt__(self, other):
        if not isinstance(other, User):
            return NotImplemented
        return self.is_paid < other.is_paid

    def __call__(self, feature_name: str) -> bool:
        return feature_name in self.features
```

Q4. Decorator-Driven Behavior Control

Create:

- A decorator that logs function execution metadata.

- A parameterized decorator that conditionally enables or disables logging.
- Apply decorators across multiple class methods without changing their internal logic.



```python
import functools
import time


def log_execution(func):

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        duration = time.time() - start_time

        print(
            f"[LOG] {func.__name__} | "
            f"args={args[1:] if len(args) > 1 else args} | "
            f"kwargs={kwargs} | "
            f"duration={duration:.4f}s"
        )

        return result

    return wrapper

def conditional_logger(enabled: bool = True):

    def decorator(func):
        if not enabled:
            return func

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print(f"[LOG ENABLED] Calling {func.__name__}")
            return func(*args, **kwargs)

        return wrapper

    return decorator
```

```python
class UserService:

    @log_execution
    def create_user(self, user_id: int, email: str):
        return {"user_id": user_id, "email": email}

    @conditional_logger(enabled=True)
    def deactivate_user(self, user_id: int):
        return f"User {user_id} deactivated"

    @conditional_logger(enabled=False)
    def internal_health_check(self):
        return "OK"


service = UserService()

service.create_user( user_id: 1,  email: "test.user@example.com")

service.deactivate_user(1)

service.internal_health_check()
```

Q5. Generator-Based Data Streaming

Design a generator that:

- Lazily processes a large dataset or collection.
- Supports partial consumption and resumption.

- Is integrated into a class method instead of returning a full list.

```python
class UserRepository:
    def __init__(self, users: list[dict]):
        self._users = users

    def iter_active_users(self):
        for user in self._users:
            if user.get("active"):
                yield user

users = [
    {"id": 1, "email": "ankit.kumar@gmail.com", "active": True},
    {"id": 2, "email": "brijesh.singh@gmail.com", "active": False},
    {"id": 3, "email": "diya.kumari@gmail.com", "active": True},
    {"id": 4, "email": "gaurav.bhatt@gmail.com", "active": True},
]

repo = UserRepository(users)

active_users = repo.iter_active_users()

print(next(active_users))

for user in active_users:
    print(user)
```

Q6. Immutable Data Modeling

Use `namedtuple` to:

- Represent immutable configuration or metadata objects.
- Store and retrieve these objects within a class.

- Demonstrate why immutability is important in this context.

```python
from collections import namedtuple

UserConfig = namedtuple( typename: "UserConfig",  field_names: ["max_features", "plan_name"])

class User:
    def __init__(self, user_id: int, email: str, config: UserConfig):
        self.user_id = user_id
        self.email = email
        self._config = config

    def get_plan_name(self) -> str:
        return self._config.plan_name

    def can_add_feature(self, current_feature_count: int) -> bool:
        return current_feature_count < self._config.max_features

FREE_CONFIG = UserConfig(max_features=2, plan_name="Free")
PAID_CONFIG = UserConfig(max_features=10, plan_name="Paid")

free_user = User( user_id: 1,  email: "free.user@gmail.com", FREE_CONFIG)
paid_user = User( user_id: 2,  email: "paid.user@gmail.com", PAID_CONFIG)

print(free_user.get_plan_name())
print(paid_user.get_plan_name())
```

Q7. Control Flow with Loop `else`

Implement a search or validation routine where:

- A loop scans through a collection.
- The `else` clause executes only when the search fails.
- The logic avoids using flags or additional state variables.

```python
class Validator:
    def __init__(self, allowed_features: list[str]):
        self.allowed_features = allowed_features

    def validate_feature(self, feature: str):
        for allowed in self.allowed_features:
            if allowed == feature:
                print("Feature allowed")
                break
        else:
            raise ValueError("Feature not allowed")
```

Q8. Module Execution Boundary

Structure the application such that:

- Business logic is import-safe.
- Execution logic runs only when the module is executed directly.
- Multiple modules interact without circular dependency issues.

```python
class User:
    def __init__(self, user_id: int, features: list[str]):
        self.user_id = user_id
        self.features = features
```

```python
class FeatureValidator:
    def __init__(self, allowed_features: list[str]):
        self.allowed_features = allowed_features

    def validate(self, feature: str):
        for allowed in self.allowed_features:
            if allowed == feature:
                return True
        else:
            raise ValueError("Feature not provided")
```

```python
from week4.app.core.models import User
from week4.app.core.validators import FeatureValidator


class FeatureService:
    def __init__(self, validator: FeatureValidator):
        self.validator = validator

    def enable_feature_for_user(self, user: User, feature: str):
        self.validator.validate(feature)
        user.features.append(feature)
        return user
```

```python
from week4.app.core.models import User
from week4.app.services.feature_service import FeatureService
from week4.app.core.validators import FeatureValidator


def main():
    validator = FeatureValidator(
        allowed_features=["browse", "search", "download"]
    )

    service = FeatureService(validator)
    user = User(user_id=1, features=[])

    service.enable_feature_for_user(user, feature: "browse")
    print(user.features)


if __name__ == "__main__":
    main()
```

Q9. Cross-Cutting Concerns via Decorators

Implement decorators to handle:

- Execution timing
- Access control or permission checks

- Reusable logic applied across unrelated classes

erator_test.py | named_tuple.py | validator.py | UserTest.py | Subscriber.py | week4/main.py | PaidUser.py | FreeUser.py

```python
import time
import functools


def time_execution(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"[TIMER] {func.__name__} took {(end - start):.4f}s")
        return result
    return wrapper

def require_permission(required_permission: str):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            user = getattr(args[0], "current_user", None)

            if not user or required_permission not in user.get("permissions", []):
                raise PermissionError(
                    f"Permission '{required_permission}' required"
                )

            return func(*args, **kwargs)
        return wrapper
    return decorator
```

erator_test.py | named_tuple.py | validator.py | UserTest.py | Subscriber.py | week4/main.py | PaidUser.py | FreeUser.py

```python
import time

from week4.decorators.decorator import time_execution, require_permission


class UserService:
    def __init__(self, current_user: dict):
        self.current_user = current_user

    @time_execution
    @require_permission("create_user")
    def create_user(self, email: str):
        time.sleep(0.1)  # simulate work
        return f"User {email} created"
```

erator_test.py | named_tuple.py | validator.py | UserTest.py | Subscriber.py | week4/main.py | PaidUser.py | FreeUser.py

```python
import time

from week4.decorators.decorator import time_execution, require_permission


class ReportService:
    def __init__(self, current_user: dict):
        self.current_user = current_user

    @time_execution
    @require_permission("generate_report")
    def generate_report(self):
        time.sleep(0.2)  # simulate work
        return "Report generated"
```

```python
from week4.decorators.ReportService import ReportService
from week4.decorators.UserService import UserService

admin_user = {
    "name": "Admin",
    "permissions": ["create_user", "generate_report"]
}

limited_user = {
    "name": "Viewer",
    "permissions": []
}

user_service = UserService(admin_user)
print(user_service.create_user("test@site.com"))

report_service = ReportService(admin_user)
print(report_service.generate_report())
```

Q10. System-Level Integration

Combine all components into a single runnable program that:

- Demonstrates object lifecycle management
- Uses generators, decorators, inheritance, and special methods together
- Produces structured output using f-strings

- Follows a clean, modular file structure

```python
from functools import wraps
from datetime import datetime


def lifecycle_logger(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        cls_name = args[0].__class__.__name__
        print(f"[{datetime.now().strftime('%H:%M:%S')}] {cls_name}.{func.__name__}() called")
        return func(*args, **kwargs)

    return wrapper

class User:
    def __init__(self, user_id: int, email: str):
        self.user_id = user_id
        self.email = email

    def __str__(self):
        return f"User(id={self.user_id}, email={self.email})"


class Subscriber(User):

    active_user_count = 0

    def __init__(self, user_id: int, email: str, is_paid: bool):
        super().__init__(user_id, email)
        self.is_paid = is_paid
        self.active = True
        Subscriber.active_user_count += 1
```

```
    def __repr__(self):
        return (
            f"Subscriber(user_id={self.user_id}, "
            f"email='{self.email}', "
            f"is_paid={self.is_paid}, "
            f"active={self.active})"
        )

    @lifecycle_logger
    def deactivate(self):
        if self.active:
            self.active = False
            Subscriber.active_user_count -= 1

    @lifecycle_logger
    def activate(self):
        if not self.active:
            self.active = True
            Subscriber.active_user_count += 1

    def __del__(self):
        if self.active:
            Subscriber.active_user_count -= 1
        print(f"Subscriber {self.user_id} deactivated")

    def __enter__(self):
        print(f"Entering context for Subscriber {self.user_id}")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Exiting context for Subscriber {self.user_id}")
        self.deactivate()
```

```python
def active_subscribers(subscribers):
    for sub in subscribers:
        if sub.active:
            yield sub


def print_subscribers(subscribers):
    print("\n********** Subscriber Details **********")
    for sub in subscribers:
        print(
            f"Subscriber ID={sub.user_id:<3} | "
            f"Subscriber Email={sub.email:<25} | "
            f"Is Paid Subscriber={str(sub.is_paid):<5} | "
            f"Is Active Subscriber={str(sub.active):<5}"
        )

    print(f"\n ********** Active subscriber count :: {Subscriber.active_user_count}")


def main():
    print("******************** Entry Point **************************\n")

    subscribers = [
        Subscriber( user_id: 1,  email: "kiran@gmail.com",  is_paid: True),
        Subscriber( user_id: 2,  email: "bisht@yahoo.com",  is_paid: False),
        Subscriber( user_id: 3,  email: "kandari@gmail.com",  is_paid: True),
    ]

    print_subscribers(subscribers)

    print("\n********** Active Subscribers using generator **********")
    for sub in active_subscribers(subscribers):
        print(f"→ {sub}")
```

```python
    print("\n***** Context Manager *****")
    with Subscriber( user_id: 4,  email: "kiran@gmail.com",  is_paid: False) as temp_sub:
        print(f"Inside context :: {temp_sub}")

    print(f"\n ***** Active subscriber count after context :: {Subscriber.active_user_count}")

    print("\n********** Deactivating Subscriber 1 **********")
    subscribers[0].deactivate()

    print_subscribers(subscribers)

    print("\n********** Exit **********\n")


if __name__ == "__main__":
    main()
```

📌 Deliverables

- Python source code organized into modules
- Clear separation of concerns
- Public GitHub repository contains only one python file, where all the answers will be there.