Pragadeesh M | 81630

Q.1 :

CityLink Farebox Engine

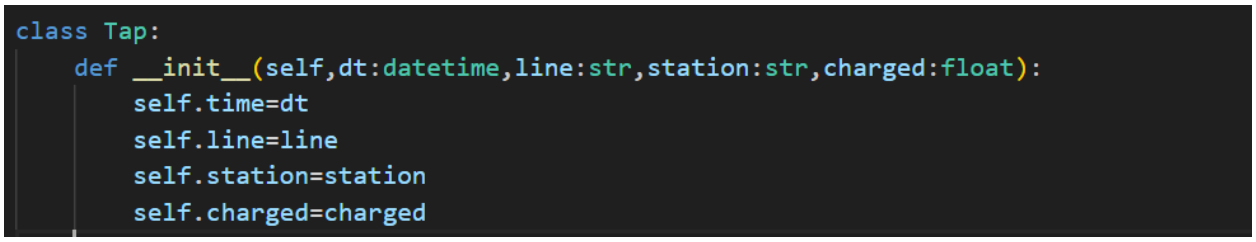
1. Introduction

The CityLink Farebox Engine is a modular Python program that computes metro card fares using historical billing rules.
 It processes tap events and sequentially applies configurable fare rules, making it suitable for hypothesis testing and system validation.

2. Module Overview

2.1 Tap Class

Purpose: Represents a single card tap event at a station.



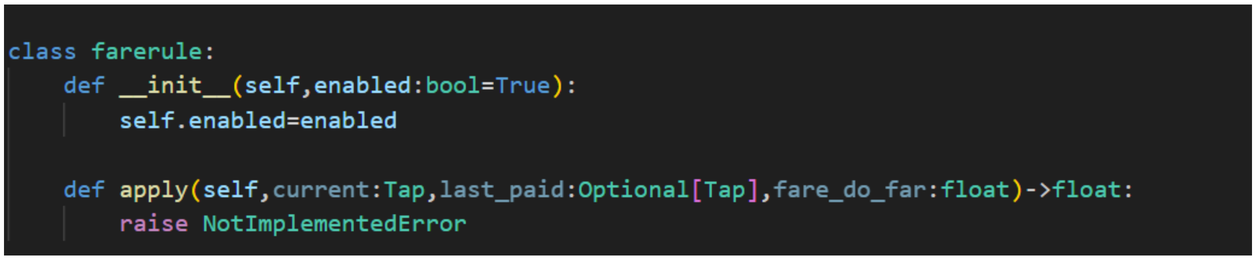
```python
class Tap:
    def __init__(self,dt:datetime,line:str,station:str,charged:float):
        self.time=dt
        self.line=line
        self.station=station
        self.charged=charged
```

- **Attributes:**
    - `time`: Date/time of tap (`datetime`)
    - `line`: Metro line (str)
    - `station`: Station code (str)

    ○ charged: Fare charged (float, for validation)

2.2 farerule (Base Rule Class)

Purpose: Abstract base class for all fare rules.
All rules inherit from this class and override apply.



```
class farerule:
    def __init__(self,enabled:bool=True):
        self.enabled=enabled

    def apply(self,current:Tap,last_paid:Optional[Tap],fare_do_far:float)->float:
        raise NotImplementedError
```

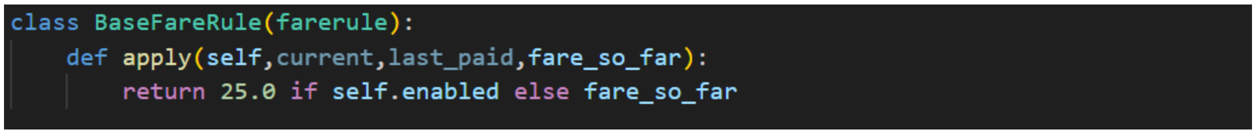
- **Attributes:**
    - enabled: Toggle rule on/off (for A/B testing)
- **Methods:**
    - apply: Override in derived classes to implement rule logic.

2.3 Individual Fare Rules

Each concrete class below implements a specific fare rule, enabling modular construction and easy testing.

a) BaseFareRule

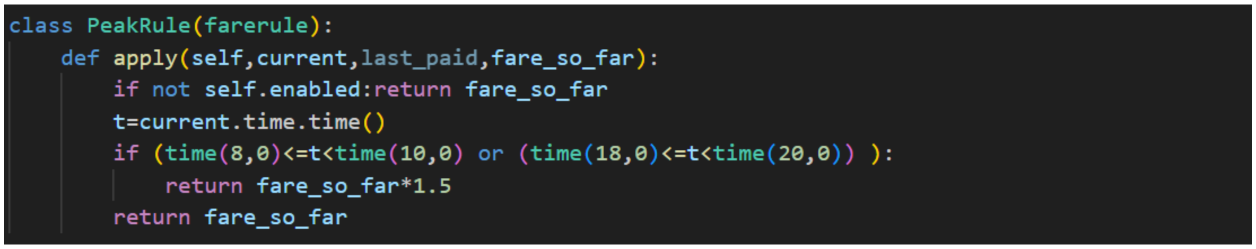
Charges the base fare if enabled.



```
class BaseFareRule(farerule):
    def apply(self,current,last_paid,fare_so_far):
        return 25.0 if self.enabled else fare_so_far
```

b) PeakRule

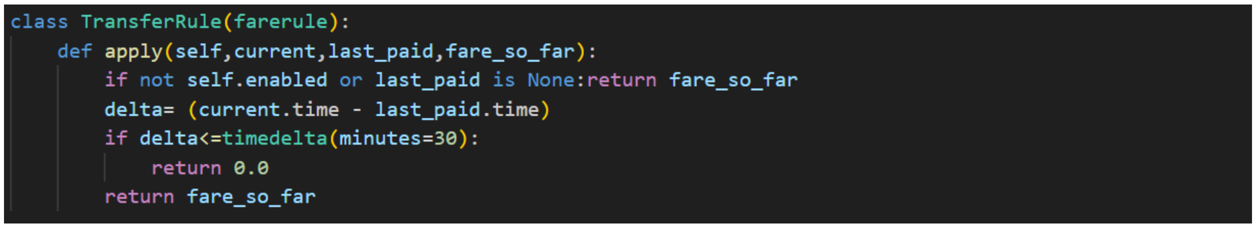
Applies a peak multiplier during specified times.



```python
class PeakRule(farerule):
    def apply(self,current,last_paid,fare_so_far):
        if not self.enabled:return fare_so_far
        t=current.time.time()
        if (time(8,0)<=t<time(10,0) or (time(18,0)<=t<time(20,0)) ):
            return fare_so_far*1.5
        return fare_so_far
```

c) TransferRule

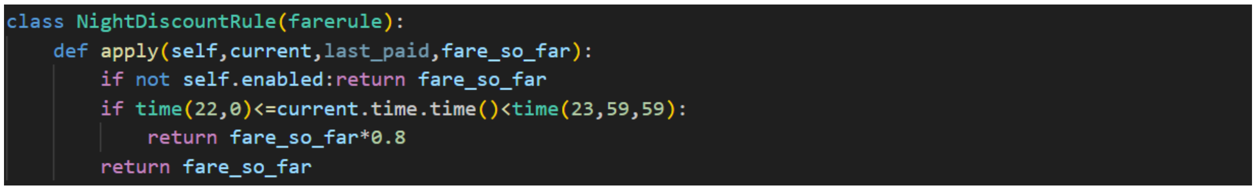
Implements the free transfer window (within 30 minutes of last paid tap).



```python
class TransferRule(farerule):
    def apply(self,current,last_paid,fare_so_far):
        if not self.enabled or last_paid is None:return fare_so_far
        delta= (current.time - last_paid.time)
        if delta<=timedelta(minutes=30):
            return 0.0
        return fare_so_far
```

d) NightDiscountRule

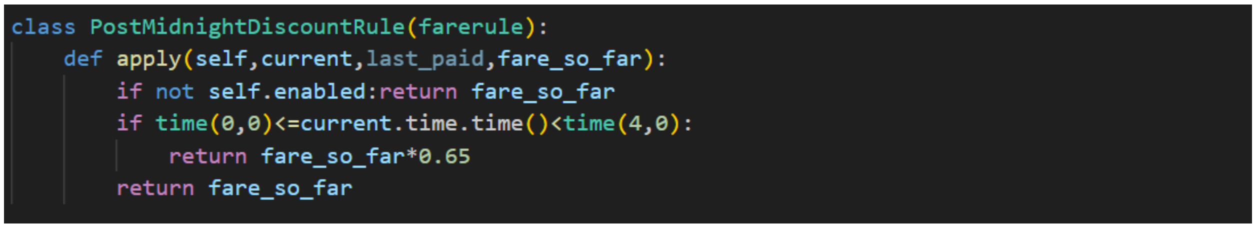
Discount for taps after 22:00.



```python
class NightDiscountRule(farerule):
    def apply(self,current,last_paid,fare_so_far):
        if not self.enabled:return fare_so_far
        if time(22,0)<=current.time.time()<time(23,59,59):
            return fare_so_far*0.8
        return fare_so_far
```

e) PostMidnightDiscountRule

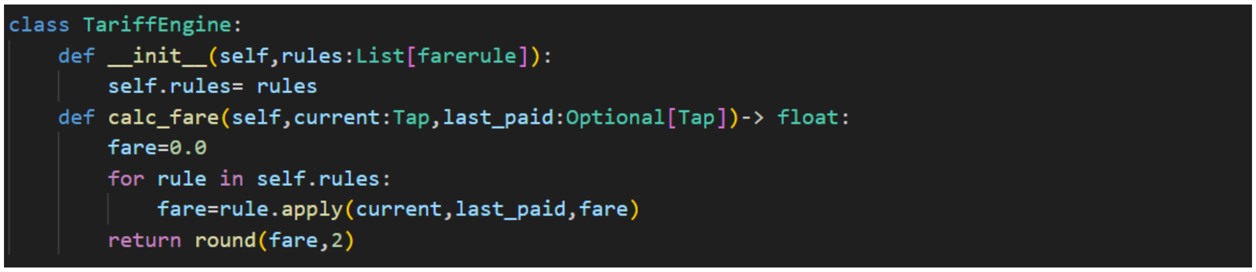
Discount for taps between midnight and 4 am.



```python
class PostMidnightDiscountRule(farerule):
    def apply(self,current,last_paid,fare_so_far):
        if not self.enabled:return fare_so_far
        if time(0,0)<=current.time.time()<time(4,0):
            return fare_so_far*0.65
        return fare_so_far
```

2.4 TariffEngine

Central engine. Applies configured rules to a tap event in order.

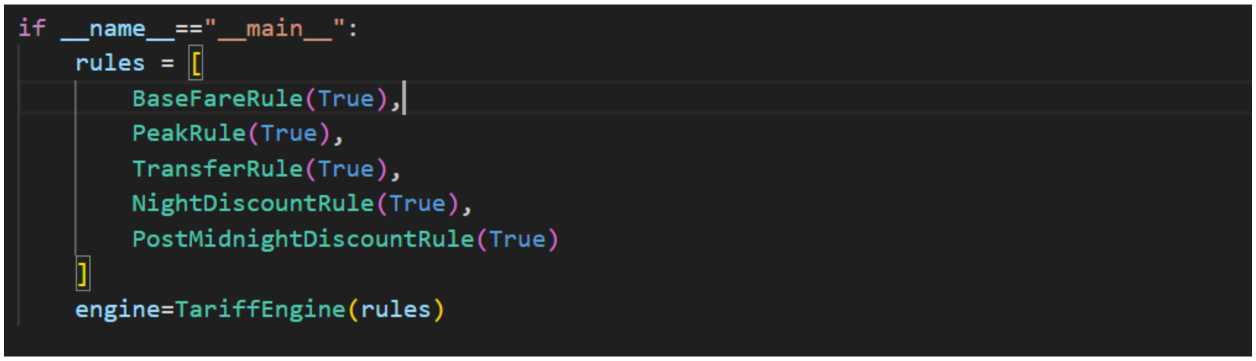


```
class TariffEngine:
    def __init__(self,rules:List[farerule]):
        self.rules= rules
    def calc_fare(self,current:Tap,last_paid:Optional[Tap])-> float:
        fare=0.0
        for rule in self.rules:
            fare=rule.apply(current,last_paid,fare)
        return round(fare,2)
```

- **Configuration:** Accepts a list of rule instances in desired order.
- **Flow:** Sequentially applies rules to compute fare.

3. Main Program & Usage

- **Rule set configuration:** Each rule can be enabled/disabled.
- **Processing tap events:** Calculates fare and prints comparison with "given" fare.



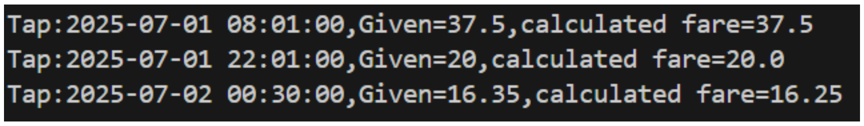
```
if __name__=="__main__":
    rules = [
        BaseFareRule(True),
        PeakRule(True),
        TransferRule(True),
        NightDiscountRule(True),
        PostMidnightDiscountRule(True)
    ]
    engine=TariffEngine(rules)
```

4. Test Cases & Output

Tap Time	Line	Given Fare	Calculated Fare	Station
2025-07-01 08:01	G	37.5	37.5	NC
2025-07-01 22:01	Y	20	20.0	BD

2025-07-02 00:30	X	16.35	16.25	NC

4.1 Test Cases & Output



```
Tap:2025-07-01 08:01:00,Given=37.5,calculated fare=37.5
Tap:2025-07-01 22:01:00,Given=20,calculated fare=20.0
Tap:2025-07-02 00:30:00,Given=16.35,calculated fare=16.25
```

5. Extension Points

- Add custom rules by subclassing `farerule`
- Toggle individual rules for A/B testing
- Rearrange rules for different fare strategies

Pragadeesh M | 81630

Q.1 :

Telecom Smart Plan Recommender — Technical Documentation

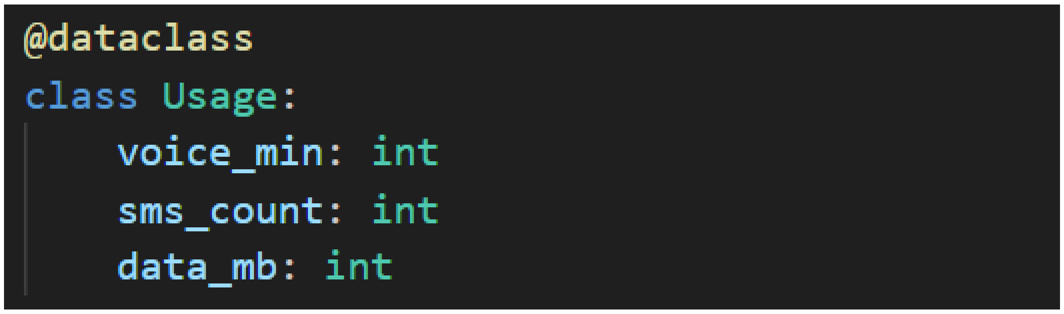
1. Introduction

The **Telecom Smart Plan Recommender** is a Python OOP solution for comparing and recommending mobile telecom plans.
It fairly compares plans of different validities (28, 30, etc.) by normalizing all plan costs/quotas to a 30-day window, matches user OTT app needs, computes overages, and recommends the cheapest eligible plan.

2. Module Documentation

2.1 Usage Model

Purpose: Models the user's projected monthly (30-day) usage.

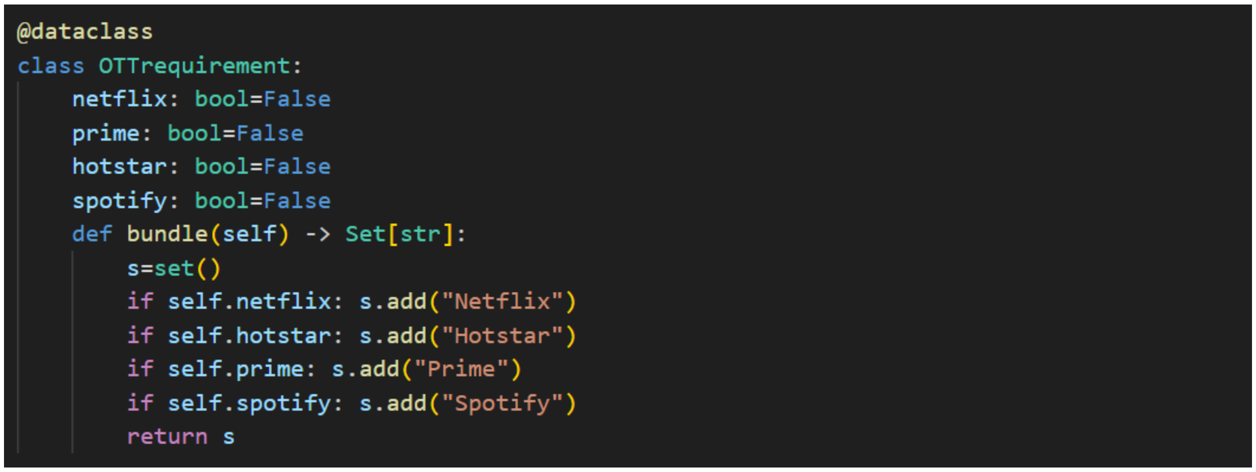


```python
@dataclass
class Usage:
    voice_min: int
    sms_count: int
    data_mb: int
```

2.2 OTT Requirement

Purpose: Records which OTT (Netflix, Prime, Hotstar, Spotify) apps the user requires.

Python

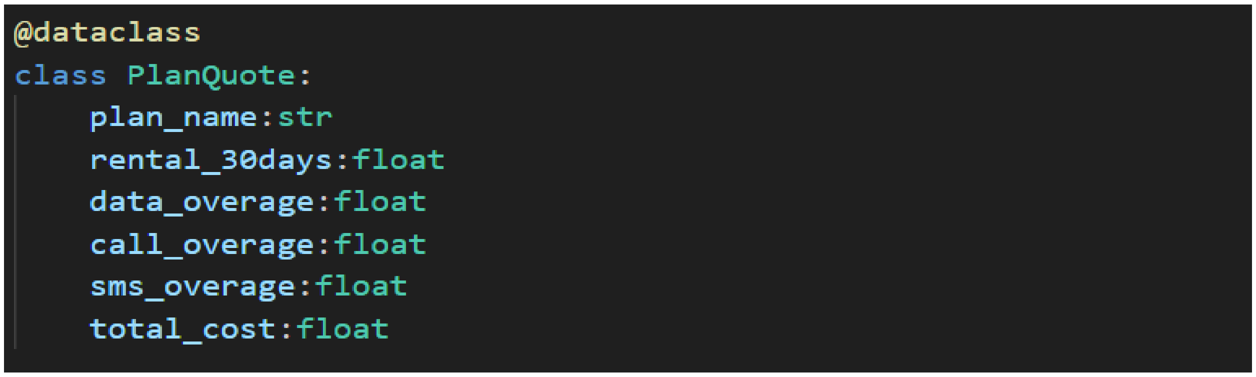


```python
@dataclass
class OTTrequirement:
    netflix: bool=False
    prime: bool=False
    hotstar: bool=False
    spotify: bool=False
    def bundle(self) -> Set[str]:
        s=set()
        if self.netflix: s.add("Netflix")
        if self.hotstar: s.add("Hotstar")
        if self.prime: s.add("Prime")
        if self.spotify: s.add("Spotify")
        return s
```

- `bundle()`: Returns the set of required OTT apps for filtering/plan matching.

2.3 PlanQuote

Purpose: Result object holding the computed cost breakdown for a single plan.

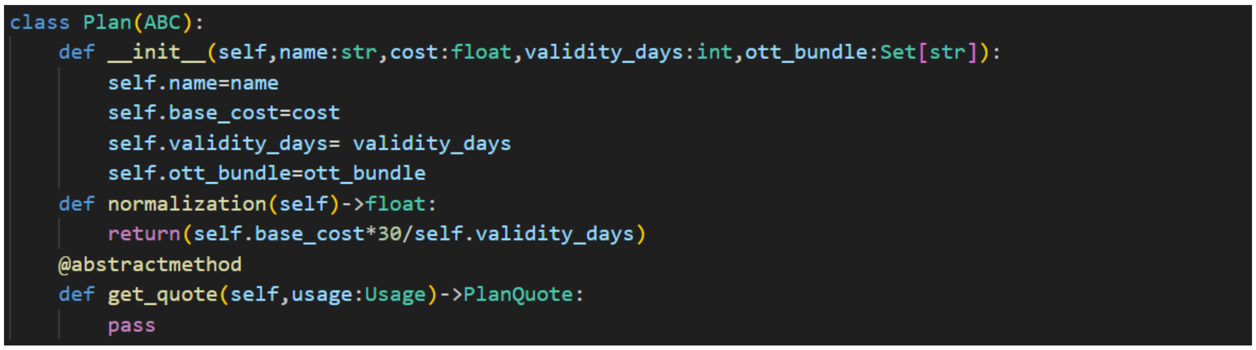


```
@dataclass
class PlanQuote:
    plan_name:str
    rental_30days:float
    data_overage:float
    call_overage:float
    sms_overage:float
    total_cost:float
```

- Fields cover plan name, normalized rental, calculated data/voice/sms overage charges, and total cost.

2.4 Plan (Abstract Base Class)

Purpose: The foundation/interface for all plan types.
Implements normalization logic needed to fairly compare plans of varying validities.



```
class Plan(ABC):
    def __init__(self,name:str,cost:float,validity_days:int,ott_bundle:Set[str]):
        self.name=name
        self.base_cost=cost
        self.validity_days= validity_days
        self.ott_bundle=ott_bundle
    def normalization(self)->float:
        return(self.base_cost*30/self.validity_days)
    @abstractmethod
    def get_quote(self,usage:Usage)->PlanQuote:
        pass
```

- `normalization()`: Converts plan rental to a per-30-day price for fair comparison.

3. Concrete Plan Classes

Each plan has its own class with specific quota, pricing, and OTT details.
All override `get_quote(Usage)` to produce a cost estimate, including rental and any usage overages.

Example: BasicLite

```
class BasicLite(Plan):
    def __init__(self):
        super().__init__("Basic Lite",249,28,set())
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        included_data = 1*1024*30   # GB/day for validity, normalized
to 30 days
        included_voice= 300
        included_sms= 100
        data_overage = max(0,usage.data_mb - included_data)
        voice_overage = max(0,usage.voice_min - included_voice)
        sms_overage = max(0,usage.sms_count - included_sms)
        total= rent+ data_overage + voice_overage + sms_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,total)
```

Here is a detailed breakdown **for all plan subclasses**, expanding on quotas, OTT bundles, and overage pricing, presented in a professional, easy-to-paste format for inclusion in your documentation. Each plan class is explained with its business rules and the matching code fragment so you can use this directly in Word.

Plan Subclass Details

1. Saver30

- **Validity:** 30 days
- **Price:** ₹499
- **Quotas:**
  - Data: 1.5 GB/day (1.5 × 1024 × 30 MB total)
  - Voice: 300 mins included
  - SMS: 100 included
- **OTT:** Hotstar
- **Overage Pricing:**
  - Data: ₹0.70/10 MB
  - Voice: ₹0.75/min after 300 mins
  - SMS: ₹0.20/SMS after 100
- **Code:**

```
class Saver30(Plan):
    def __init__(self):
        super().__init__("Saver 30",499,30,{"Hotstar"})
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        included_data = 1.5*1024*30
        included_voice= 300
        included_sms= 100
        data_overage = max(0,usage.data_mb - included_data)
        voice_overage = max(0,usage.voice_min - included_voice)
        sms_overage = max(0,usage.sms_count - included_sms)
        total= rent+ data_overage + voice_overage + sms_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,total)
```

2. UnlimitedTalk30

- **Validity:** 30 days
- **Price:** ₹650
- **Quotas:**
    - Data: 5 GB (5120 MB) total
    - Voice: Unlimited
    - SMS: Unlimited
- **OTT:** Spotify
- **Overage Pricing:**
    - Data: ₹0.70/10 MB after 5GB
    - Voice/SMS: No overage
- **Code:**

-
  ```
  class UnlimitedTalk30(Plan):
      def __init__(self):
          super().__init__("Unlimited Talk v30",650,30,{"Spotify"})
      def get_quote(self, usage:Usage)->PlanQuote:
          rent=self.normalization()
          included_data = 5*1024
          data_overage = max(0,usage.data_mb - included_data)
          voice_overage = 0
          sms_overage = 0
  ```

```
        total= rent+ data_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,t
otal)
```

3. DataMax20

- **Validity:** 20 days
- **Price:** ₹749
- **Quotas:**
    - Data: Unlimited
    - Voice: 100 mins included (normalized for 30 days: 100 × (30/20))
    - SMS: Unlimited
- **OTT:** Hotstar
- **Overage Pricing:**
    - Voice: ₹0.75/min after quota
    - Data/SMS: No overage
- **Code:**

```
class DataMax20(Plan):
    def __init__(self):
        super().__init__("Data Max 20",749,20,{"Hotstar"})
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        included_voice= 100* (30/self.validity_days)
        data_overage = 0
        voice_overage = max(0,usage.voice_min - included_voice) * .75
        sms_overage = 0
        total= rent+ data_overage + voice_overage + sms_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,total)
```

4. StudentStream56

- **Validity:** 30 days
- **Price:** ₹435

- **Quotas:**
  - Data: 2 GB/day (2 × 1024 × 30 MB total)
  - Voice: 300 mins included
  - SMS: 200 included
- **OTT:** Spotify
- **Overage Pricing:**
  - Data: ₹0.07/MB
  - Voice: ₹0.75/min after 300 mins
  - SMS: ₹0.20/SMS after 200
- **Code:**

```
class StudentStream56(Plan):
    def __init__(self):
        super().__init__("Student Stream 56",435,30,{"Spotify"})
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        included_data = 2*1024*30
        included_voice= 300
        included_sms= 200
        data_overage = max(0,usage.data_mb - included_data) * 0.07
        voice_overage = max(0,usage.voice_min - included_voice) * .75
        sms_overage = max(0,usage.sms_count - included_sms) * 0.20
        total= rent+ data_overage + voice_overage + sms_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,total)
```

5. FamilyShare30

- **Validity:** 28 days
- **Price:** ₹500
- **Quotas:**
  - Data: 50 GB total, pro-rated for 30 days and further multiplied by ₹0.07 per MB overage
  - Voice: 1000 mins included, pro-rated for 30 days and further multiplied by ₹0.6 per min overage
  - SMS: 500 included, pro-rated for 30 days and further multiplied by ₹0.20 per SMS overage
- **OTT:** Prime

- **Overage Pricing:**
    - Data: As above
    - Voice: As above
    - SMS: As above
- **Code:**

```
class FamilyShare30(Plan):
    def __init__(self):
        super().__init__("Family Share 30",500,28,{"Prime"})
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        included_data = 50*1024*(30/self.validity_days) * 0.07
        included_voice= 1000*(30/self.validity_days) * .6
        included_sms= 500*(30/self.validity_days) * .20
        data_overage = max(0,usage.data_mb - included_data)
        voice_overage = max(0,usage.voice_min - included_voice)
        sms_overage = max(0,usage.sms_count - included_sms)
        total= rent+ data_overage + voice_overage + sms_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,total)
```

6. DataMaxPlus30

- **Validity:** 30 days
- **Price:** ₹1499
- **Quotas:**
    - Data: Unlimited
    - Voice: 300 mins included
    - SMS: 200 included
- **OTT:** Prime, Hotstar
- **Overage Pricing:**
    - Voice: ₹0.75/min after 300 mins
    - SMS: ₹0.20/SMS after 200
    - Data: No overage
- **Code:**

```
class DataMaxPlus30(Plan):
    def __init__(self):
        super().__init__("Data Max Plus
30",1499,30,{"Prime","Hotstar"})
 def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        included_voice= 300
        included_sms= 200
        data_overage = 0
        voice_overage = max(0,usage.voice_min - included_voice) *0.75
        sms_overage = max(0,usage.sms_count - included_sms) *0.20
        total= rent+ data_overage + voice_overage + sms_overage
        return
PlanQuote(self.name,rent,data_overage,voice_overage,sms_overage,total)
```

7. PremiumUltra30

- **Validity:** 30 days
- **Price:** ₹2999
- **Quotas:**
    - Data: Unlimited
    - Voice: Unlimited
    - SMS: Unlimited
- **OTT:** Netflix, Prime, Hotstar, Spotify
- **Overage Pricing:** None
- **Code:**

```
class PremiumUltra30(Plan):
    def __init__(self):
        super().__init__("Premium Ultra
30",2999,30,{"Netflix","Prime","Hotstar","Spotify"})
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        total=rent
        return PlanQuote(self.name,rent,0,0,0,total)
```

8. PremiumUltraMax30

- **Validity:** 30 days
- **Price:** ₹200
- **Quotas:**
  - Data: Unlimited
  - Voice: Unlimited
  - SMS: Unlimited
- **OTT:** Netflix, Prime, Hotstar, Spotify
- **Overage Pricing:** None (note: price seems intentionally low for testing/demo)
- **Code:**

```
class PremiumUltraMax30(Plan):
    def __init__(self):
        super().__init__("Premium Ultra Max 
30",200,30,{"Netflix","Prime","Hotstar","Spotify"})
    def get_quote(self, usage:Usage)->PlanQuote:
        rent=self.normalization()
        total=rent
        return PlanQuote(self.name,rent,0,0,0,total)
```

4. PlanRecommender

Purpose: Filters available plans by OTT requirements, computes costs, and recommends the best plan.

```
class PlanRecomender:
    def __init__(self,plans:List[Plan]):
       self.plans=plans

    def recommender(self,usage:Usage,ott:OTTrequirement):
        req_ott = ott.bundle()
        filtered=[]
        for plan in self.plans:
            if req_ott.issubset(plan.ott_bundle):
                quote = plan.get_quote(usage)
                filtered.append((quote,plan))
        if not filtered:
            print("No plan satisfies Requirements")
            return None
        filtered.sort(key=lambda x:(x[0].total_cost,-
len(x[1].ott_bundle),x[0].rental_30days))
        best_quote,best_plan=filtered[0]
        print ("\n---- Plan Cost ----")
        for quote,_ in filtered:
            print(quote,"\n")
        print("Recomended PLan : ",best_quote.plan_name)
        print("Reason: cheapest plan,with the required OTTs")
```

- Filters plans whose OTT bundle contains all user-required apps.
- Sorts by total cost, OTT count (prefer more), rental price (prefer lower).
- Prints detailed cost breakdown and the recommended plan.

5. Main Usage Example

```
if __name__=="__main__":
    usage = Usage(voice_min=650,sms_count=300,data_mb=8*1024)
    ott_req=OTTrequirement(hotstar=True)

plans=[BasicLite(),Saver30(),UnlimitedTalk30(),DataMax20(),StudentStre
am56(),FamilyShare30(),DataMaxPlus30(),PremiumUltra30()]
    recommend= PlanRecomender(plans)
    recommend.recommender(usage,ott_req)
```

6. Test Cases & Output

Below, paste *screenshots of test outputs showing plan cost breakdowns and recommendations for sample user scenarios*.

[Insert output screenshot here]

7. Extensibility & Maintenance

- **Open/Closed Principle:** Add new plans, OTT options, cost rules by subclassing `Plan` without changing core logic.
- **Pro-rated normalization:** All plan costs/quotas are normalized to 30 days, handling disparate validity periods automatically.
- **OTT Filtering:** System ensures only plans with required OTTs are considered, improving user satisfaction.