Rethik Manoharan 81643

Mobile Plan Selection System Documentation

Overview

This system helps users find the most cost-effective mobile plan based on **usage** and **OTT streaming needs**, calculating potential overage costs.

Core Components

1. `OTTApp` Enum (from `base_plan.py`)

- Supported streaming apps: `NETFLIX`, `PRIME`, `HOTSTAR`, `SPOTIFY`.
    - *(Note: `AMAZON_PRIME` is used in `meets_requirements` but `PRIME` in plan definitions. Check consistency.)*

2. `Usage` Class (from `base_plan.py`)

- Dataclass for user's estimated monthly: `voice_mins`, `sms_count`, `data_mb`.

3. `OTTRequirement` Class (from `base_plan.py`)

- Dataclass for user's required OTT services (`needs_netflix`, etc.).
- `meets_requirements(plan_ott_apps)`: Checks if a plan offers the required OTT apps.

4. `PlanQuote` Class (from `base_plan.py`)

- Dataclass for estimated plan costs: `plan_name`, `total_cost`, `rental_30d`, `data_overage`, `voice_extra`, `sms_extra`.

5. `Plan` Abstract Base Class (from `base_plan.py`)

- Base for all plans. Includes `name`, `base_cost`, `valid_days`, `ott_apps`.

- **Overage Rates**: `DATA_OVERAGE_10MB` ($0.70), `VOICE_OVERAGE_1MIN` ($0.75), `SMS_OVERAGE_1SMS` ($0.20).
- **Methods**:
    - `prorata_cost()`: 30-day base rental.
    - `data_overage(...)`, `voice_overage(...)`, `sms_overage(...)`: Calculate overage costs (returns 0.0 for unlimited, indicated by negative quota).
    - `calculate_30day_cost(usage)` (abstract): Calculates total plan cost including overages.
    - `meets_ott_requirements(req)`: Checks OTT compatibility.

6. Specific Plan Classes (from `plans.py`)

- (`BasicLite`, `Saver30`, `UnlimitedTalk30`, `DataMax20`, `StudentStream56`, `FamilyShare30`, `DataMaxPlus30`, `PremiumUltra30`).
- Each defines specific quotas (e.g., `data_gb_1day`, `voice_mins`, `sms_count`; `-1` means unlimited) and implements `calculate_30day_cost` by prorating, calculating overages, and summing up.

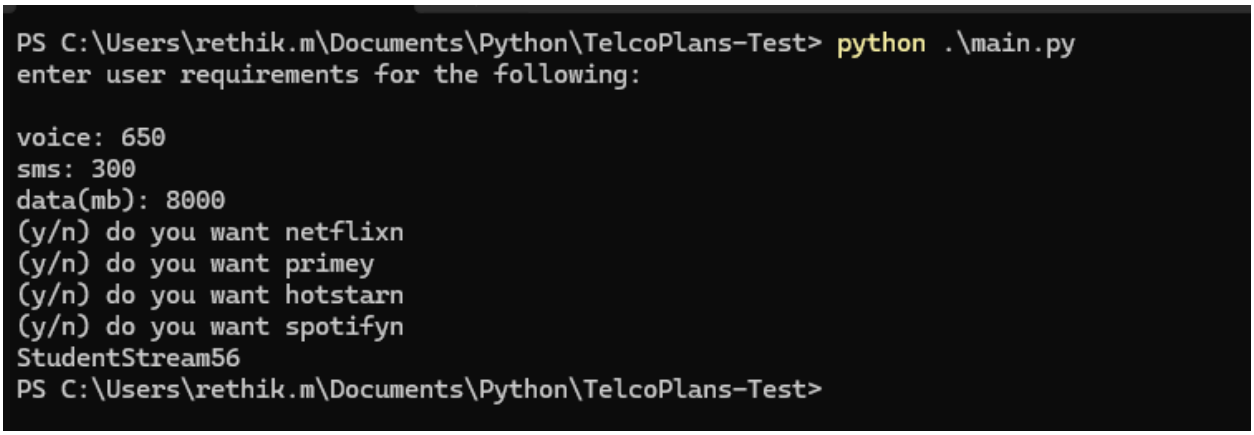
7. `PlanChooser` Class (from `plan_chooser.py`)

- Selects the best plan from a `List[Plan]`.
- `choose_plan(self, usage, req)`: Filters plans by OTT requirements, calculates `PlanQuote` for valid candidates, and returns the one with the `minimum total_cost`.

Example Usage (from `main.py`)

1. User inputs `usage` and `OTTRequirement`.
2. `PlanChooser` selects the best plan.
3. Prints the recommended `plan_name` or "no plan".

Input/Output:



Tariff Engine System Documentation

Overview

This system is a **fare calculation engine** that applies predefined rules to transit "tap" events. The `TariffEngine` class orchestrates the application of various `Rule` objects to determine the final fare.

Core Components

1. Tap Class (from `tap.py`)

Stores information for a single tap event:

- `time` (datetime): Timestamp of the tap.
- `line` (Line Enum): Transit line ('g', 'r', 'y', 'x' for unknown).
- `station_code` (StationCode Enum): Station code ('BD', 'NC', etc., 'UK' for unknown).

- `tap_from_str(...)`: Factory method to create a Tap object from strings, handling invalid inputs.

2. `Fare` Class (from `tap.py`)

Manages the fare amount during calculation:

- `price` (float): Current fare value.
- `apply_peak_hour_charge(percentage)`: Increases `price`.
- `apply_discount(percentage)`: Decreases `price`.
- `get_price()`: Returns current `price`.
- `set_price(new_price)`: Sets `price` to a new value.

3. `TariffEngine` Class (from `main.py`)

The central unit for fare calculation:

- **Initialization (`__init__`)**: Takes boolean flags (`r1` through `r5`) to enable/disable specific rules. Defaults to all rules enabled.
- **`calc_fare(self, tap, last_tap=None)`**:
    - Initializes `Fare` object.
    - Applies `BaseFare` first.
    - Sequentially applies other enabled rules to adjust the fare, if passed as true.
    - Returns the final calculated `fare` price.

Fare Rules (from `rules.py`)

All rules inherit from a base `Rule` class with an `is_enabled` flag.

Specific Fare Rules:

1. **`BaseFare` (R1)**: Sets the initial fare to 25.0.
2. **`PeakSurchargeFare` (R2)**: Applies a **50% surcharge** during peak hours: **8-10 AM** and **6-8 PM**.
3. **`TransferWindowFare` (R3)**: Sets fare to 0.0 if a tap occurs within **30 minutes** of the `last_tap`.
4. **`NightFare` (R4)**: Applies a **20% discount** between **10 PM and midnight**.
5. **`PostMidnightFare` (R5)**: Applies a **35% discount** between **midnight and 4 AM**.

Input:

```
tap1 = ('2025-07-01 08:20', 'g', 'BD')

tap = Tap.tap_from_str(*tap1)
print(tap.time.time())
fare = engine.calc_fare(tap,last_tap=None)
print(fare)
```

Output:

```
PS C:\Users\rethik.m\Documents\Python\Metro-Test> python .\main.py
08:20:00
37.5
PS C:\Users\rethik.m\Documents\Python\Metro-Test>
```