Documentation for frontend of Weather application

WeatherApp Class

Purpose: The main class managing the weather application's frontend.

Methods:

build(): Constructs the entire layout of the weather application. Divides the interface into sections for city and temperature display, hourly forecast, air conditions, and 7-day forecast. Binds functions to update views when weather data changes.

updateTopView(): Updates the top section of the interface, including city name, temperature, real feel, weather description, and corresponding weather image.

updateMiddleView(): Updates the hourly forecast section with weather information for different times.

updateBottomView(): Updates the air condition details such as humidity, pressure, and daybreak (sunrise and sunset times).

updateRightView(): Placeholder function for updating the right section of the interface.

create_city_and_temperature_tile(): Constructs the layout for displaying city information, temperature, and weather description with respective fonts and styles.

get_img_source(): Maps weather descriptions to specific image sources for weather icons.

get_cityLabel_text(), get_tempLabel_text(), get_realfeel_text(), get_weatherDescription_text(), get_hourly_array(), get_sunrise_text(), get_pressure_text(), get_humidity_text(): Functions to retrieve specific weather-related text data to display.

create_hourly_forecast_tile(), create_left_air_conditions_tile(), create_right_air_conditions_tile(), create_7_day_forecast_tile(), create_settings_tile(): Functions to construct individual sections of the interface layout.

Main Components:

Box Layouts: Used to structure different sections of the UI (e.g., city and temperature, hourly forecast, air conditions, 7-day forecast).

Labels: Display textual information such as city names, temperatures, real feel, weather descriptions, and daybreak details.

Images: Represent weather icons based on current weather conditions.

Buttons and Text Inputs: For user interaction to apply settings like changing the city or temperature unit.

Popups: Displays settings options and allows users to modify settings like city or unit.

This frontend_weather.py script integrates with a weather manager (weather_manager.py) to update the UI based on fetched weather data. The UI is structured into sections, each handling different aspects of weather information display.

Step by step analysis of code

1. Configuration and Imports

Path: frontend_weather.py

Purpose: Sets up Kivy configuration, imports necessary modules.

Code Snippet:

```
from kivy.config import Config

Config.set('graphics', 'resizable', '0')

Config.set('graphics', 'width', '1570')

Config.set('graphics', 'height', '800')

from kivy.app import App
```

```
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.core.text import LabelBase
from weather_manager import manager
from kivy.uix.image import Image
# Necessary imports for the application # ...
```

2. Class Definition: WeatherApp

Path: frontend_weather.py

Purpose: Defines the main class managing the weather application's frontend.

Methods: build(), updateTopView(), updateMiddleView(), updateBottomView(), and others.

Code Snippet

```
def updateTopView(self,instance,value):
    self.city_label.text = self.get_cityLabel_text()
    self.temp_label.text = self.get_tempLabel_text()
    self.realfeel_label.text = self.get_realfeel_text()
    self.description_label.text = self.get_weatherDescription_text()
    self.image.source = self.get_img_source()


def updateMiddleView(self, instance, value):
    for i, (label, time_label) in enumerate(zip(self.hourly_labels,
self.hourly_times)):
        label.text = self.get_hourly_array(i)
```

```
        time_label.text = f"{(int(self.current_datetime.hour) +
int(self.get_time_difference()) + i) % 24}:00"


    def updateBottomView(self,instance,value):
        self.humidity.text = self.get_humidity_text()
        self.pressure.text = self.get_pressure_text()
        self.daybreak.text = self.get_sunrise_text()
```

Explanation: updateTopView(self, instance, value): Updates the top section of the UI, specifically updating the city label, temperature label, "Real Feel" label, weather description label, and the weather condition image by retrieving and displaying corresponding data fetched from the weather manager.

updateMiddleView(self, instance, value): Manages the middle section of the UI, responsible for displaying hourly forecast information. It iterates through a set of labels and time labels (hourly_labels and hourly_times) to update them with the hourly temperature forecast and corresponding time, adjusting for time differences.

updateBottomView(self, instance, value): Handles the bottom section of the UI, updating humidity, pressure, and daybreak (sunrise time) labels based on information retrieved from the weather manager.

These functions collectively ensure that the UI accurately reflects the latest weather information fetched from the weather manager and dynamically updates the displayed data as new information becomes available or changes occur.

3. UI Building and Element Creation

Path: frontend_weather.py

Purpose: Constructs various UI elements like tiles for city and temperature display, hourly forecast, air conditions, and 7-day forecast.

Code Snippet

```
def get_cityLabel_text(self):
```

```
        return f"{manager.city}"

    def get_tempLabel_text(self):
        unit = 'C' if manager.unit == 'Metric' else 'F'
        return f"{round(manager.temperature)}°{unit}"

    def get_realfeel_text(self):
        unit = 'C' if manager.unit == 'Metric' else 'F'
        return "Real Feel: " + f"{round(manager.feels_like)}°"

    def get_weatherDescription_text(self):
        raw = f"{manager.weather_description}              "
        return raw.capitalize()

    def get_hourly_array(self, int):
        return f"{round(manager.hourly[int])}"

    def create_hourly_forecast_tile(self):
        tile = DarkBlueBoxLayout(orientation='vertical', spacing=5)

        tile.add_widget(Label(text="Today's Forecast", font_size=30,
font_name='CustomFontB', color='black', halign='right', valign='middle'))

        forecast_container = BoxLayout(orientation='horizontal', spacing=5)
        self.populate_hourly_forecast(forecast_container)
```

```
        tile.add_widget(forecast_container)
        tile.add_widget(Label())
        return tile

    def create_left_air_conditions_tile(self):
        tile = DarkBlueBoxLayout(orientation='vertical', spacing=5)
        tile.add_widget(Label(text='Daybreak', font_size=25,
font_name='CustomFontB', color='black', halign='right', valign='top'))
        hori = DarkBlueBoxLayout(orientation='horizontal', spacing=5)
        hori.add_widget(Label())
        images = Image(source="images/Sunrise_icon.png", size_hint=(None, None),
size=(80, 80), pos_hint={'center_x': 0.5})
        hori.add_widget(images)
        hori.add_widget(Label())
        images = Image(source="images/Sunset_icon.png", size_hint=(None, None),
size=(80, 80), pos_hint={'center_x': 0.5})
        hori.add_widget(images)
        hori.add_widget(Label())

        tile.add_widget(hori)
        self.daybreak = Label(text=self.get_sunrise_text(), font_size=20, color='black',
halign='right', valign='middle')
        tile.add_widget(self.daybreak)

        return tile
```

Explanation: **get_cityLabel_text()**: Retrieves the city name from the weather manager.

get_tempLabel_text(): Fetches the temperature and formats it with the corresponding unit (Celsius or Fahrenheit) based on the chosen unit in the weather manager.

get_realfeel_text(): Retrieves and formats the "Real Feel" temperature, adjusting it according to the chosen unit.

get_weatherDescription_text(): Formats and capitalizes the weather description fetched from the weather manager.

get_hourly_array(): Fetches and formats hourly temperature data for the forecast display.

Moreover, two functions build specific UI elements:

create_hourly_forecast_tile(): Constructs a vertical tile with today's forecast label and an arrangement for displaying hourly forecasts using populate_hourly_forecast().

create_left_air_conditions_tile(): Builds a vertical tile to represent "Daybreak" information, including sunrise and sunset icons and corresponding times, employing the **get_sunrise_text()** function.

These functions handle data retrieval and formatting for displaying various weather-related information and construct corresponding UI elements for visual representation within the application interface.

4. Weather Data Handling

Path: frontend_weather.py

Purpose: Functions to extract and format weather-related data (e.g., temperature, humidity, sunrise time) fetched from the weather manager.

Code Snippet

```
def get_sunrise_text(self):
    unix = manager.sunrise
```

```
        utc_datetime = datetime.datetime.utcfromtimestamp(unix)
        readable_time = utc_datetime.strftime('%Y-%m-%d %H:%M:%S UTC')
        time_object = datetime.datetime.strptime(readable_time, '%Y-%m-%d
%H:%M:%S UTC')
        hour_part = time_object.hour
        minute_part = time_object.minute

        unix1 = manager.sunset
        utc_datetime = datetime.datetime.utcfromtimestamp(unix1)
        readable_time1 = utc_datetime.strftime('%Y-%m-%d %H:%M:%S UTC')
        time_object1 = datetime.datetime.strptime(readable_time1, '%Y-%m-%d
%H:%M:%S UTC')
        hour_part1 = time_object1.hour
        minute_part1 = time_object1.minute

        return f"{(int(self.get_time_difference()) + int(hour_part)) %
24:02d}:{int(minute_part):02d}" + (" " * 37) + f"{(int(self.get_time_difference()) +
int(hour_part1)) % 24:02d}:{int(minute_part1):02d}"



    def get_pressure_text(self):
        return 'Humidity:    ' + f"{manager.pressure}" + ' hPa'

    def get_humidity_text(self):
        return 'Pressure:           ' + f"{manager.humidity}" + ' %'
```

```
    def create_right_air_conditions_tile(self):
        tile = DarkBlueBoxLayout(orientation='vertical', spacing=5)
        tile.add_widget(Label(text='Additional Information', font_size=25,
font_name='CustomFontB', color='black', halign='right', valign='top'))  #
Placeholder for symmetry


        self.humidity = Label(text=self.get_humidity_text(), font_name='CustomFontB',
color='black', font_size=20, halign='left', valign='middle')
        self.pressure = Label(text=self.get_pressure_text(), font_name='CustomFontB',
color='black', font_size=20, halign='left', valign='middle')


        tile.add_widget(self.humidity)
        tile.add_widget(self.pressure)


        return tile
```

Explanation: This block of code is dedicated to generating additional information about the weather conditions. The **get_sunrise_text()** function calculates the sunrise and sunset times based on Unix timestamps, converting them into human-readable hour and minute formats adjusted by time differences.

Meanwhile, **get_pressure_text()** and **get_humidity_text()** prepare strings displaying humidity and pressure data fetched from the weather manager, respectively.

The function **create_right_air_conditions_tile()** assembles a vertical layout **(DarkBlueBoxLayout)** to organize and display this additional information. It includes a title label for "Additional Information" and places labels for humidity and pressure using the previously prepared text. This block focuses on presenting essential weather-related details in a structured manner within the application's user interface.

5. User Interaction and Settings

Path: frontend_weather.py

Purpose: Functions handling user interactions, such as applying settings for city or temperature unit changes.

Code Snippet

```
def create_settings_tile(self):
    tile = GreyBoxLayout(orientation='vertical', spacing=20,
size_hint_x=1,padding=10)

    # city
    box = BoxLayout(orientation='horizontal')
    box.add_widget(Label(text='City:',size_hint_x=0.3))

    self.search_bar = TextInput(font_size=20,
                    multiline=False,
                    size_hint=(1, 1),
                    padding=(10, 10),
                    hint_text='City',
                    font_name='CustomFontB',
                    halign='center',
                    text=manager.city)
    self.search_bar.bind(on_text_validate=self.applySettings)

    box.add_widget(self.search_bar)
    tile.add_widget(box)
```

```
    # units
    box = BoxLayout(orientation='horizontal')
    box.add_widget(Label(text='Unit:',size_hint_x=0.3))
    if manager.unit == 'Metric':
        unit_text = 'Celsius'
    else:
        unit_text = 'Fahrenheit'
    self.unit_spinner = Spinner(text=unit_text, values=(
        'Celsius', 'Fahrenheit'), size_hint=(1, 1))
    box.add_widget(self.unit_spinner)
    tile.add_widget(box)

    # apply button
    btn = Button(text='Apply',
        on_press=self.applySettings,
        background_color=(.0, 1, .0, 1))
    tile.add_widget(btn)

    return tile

def applySettings(self, instance):
    if self.unit_spinner.text == 'Celsius':
        unit = 'Metric'
```

```
else:
    unit = 'Imperial'


manager.updateSettings(self.search_bar.text,unit)
self.settings_popupsettings_popup.dismiss()
```

Explanation: This code segment is responsible for creating the settings interface within the weather application. The **create_settings_tile()** function builds a vertical layout **(GreyBoxLayout)** comprising input fields and dropdowns for modifying settings.

It includes:

A text input box allowing users to enter a city name **(search_bar)** and a dropdown **(unit_spinner)** to select temperature units in Celsius or Fahrenheit.

An "Apply" button triggers the **applySettings()** function, capturing user selections to update the settings.

The **applySettings()** function retrieves the chosen unit (Celsius or Fahrenheit) and the entered city name from the respective UI components. It then invokes the weather manager's **updateSettings()** method to apply these changes. Finally, it dismisses the settings popup **(settings_popupsettings_popup)**.

This segment encapsulates the settings interface, enabling users to modify city names and temperature units, and ensures the changes are applied effectively to the weather application.

6. Main Execution

Path: frontend_weather.py

Purpose: Checks if the script is executed as the main program and starts the WeatherApp.

Code Snippet

```
if __name__ == '__main__': WeatherApp().run()
```

Explanation:

The code begins by configuring the Kivy application window and importing necessary modules.

The WeatherApp class orchestrates the frontend, containing methods to build UI elements and update views based on weather data changes.

Various sections of the UI (city and temperature, hourly forecast, air conditions, 7-day forecast) are created and structured using Kivy's layout components.

Functions handle data retrieval from the weather manager, formatting weather-related text, and managing user settings.

The script then checks if it's the main program and initiates the WeatherApp to run the application.

This code effectively constructs a Kivy-based frontend for a weather application, displaying weather-related information and allowing user interaction to modify settings.