Contents

Analysis / Planning

I do not have a quick way of checking the weather where I live. I would like to not have to open up a website with a potentially bad UI, and would prefer if the program's UI fit with my desktop environment.

The solution is to make a small GUI app written in GTK, which will fit in with my desktop and will have a nice UI. It will also show the information I want from a reliable source. The source I will use is OpenWeather, as they have a free API which I can use to get the actual weather information. I will use Rust use the API as it is fast and I can use it's type system to make sure that I am getting correct data. I can then use the PyO3 binding to use my Rust code in Python, which I am using for the graphical interface.

Success Criteria

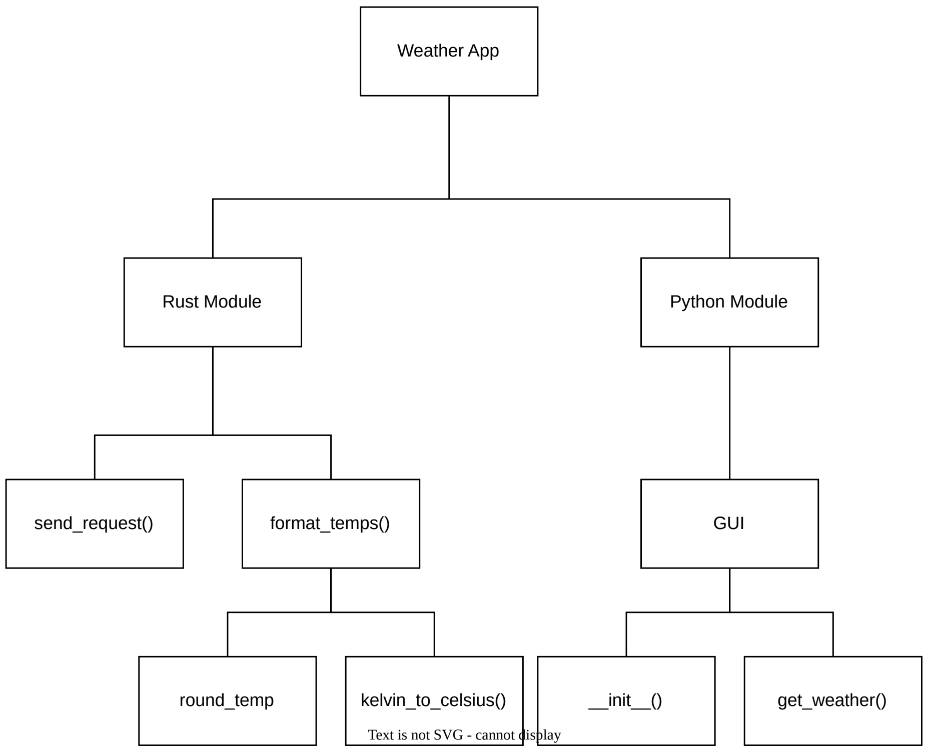
- Able to get data from the API
- Able to display it on the GUI
- Able to ask which city the user wants the weather for

Designing

Algorithm

First we create the GUI with a text input box for the location, and a button to get the weather. We attach the button to a function so that when the button is clicked, it gets the location from the text input box. It then calls the rust function to send an API request for the location from entered in the text input box. When the data is returned it will be returned to Python, and then displayed on the screen.

Structure Diagram



Coding

I am using Rust for the API part, which means I will need certain libraries to avoid re-implementing certain functionality such as an HTTPS library. I will be using `reqwest` for sending API requests. For converting the JSON into a better structure, I will be using `serde` and `serde_json`. I will also be using `PyO3` for the PyO3 bindings to work. The last Rust library I am using is `libmath` which contains a rounding function I need. For Python I will be using the packages `maturin`, and `PyGObject` for generating the `PyO3` rust bindings, and building the GTK interface respectively.

Important Code Snippets

`src/lib.rs`

```rust
#[pyclass]
#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(rename_all = "camelCase")]
pub struct Master {
    #[pyo3(get)]
    #[serde(rename(deserialize = "weather"))]
    pub weather: Vec<Weather>,
    #[pyo3(get)]
    #[serde(rename(deserialize = "main"))]
    pub temp: Temp,
}
#[pyclass]
#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(rename_all = "camelCase")]
pub struct Weather {
    #[pyo3(get)]
    pub main: String,
    #[pyo3(get)]
    pub description: String,
}

#[pyclass]
#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(rename_all = "camelCase")]
pub struct Temp {
    #[pyo3(get)]
    pub temp: f32,
    #[pyo3(get)]
    #[serde(rename = "feels_like")]
    pub feels_like: f32,
    #[pyo3(get)]
    #[serde(rename = "temp_min")]
    pub temp_min: f32,
    #[pyo3(get)]
    #[serde(rename = "temp_max")]
    pub temp_max: f32,
}
```

This code snippet contains the structs which I use to parse the JSON returned from the API into a format that Rust can understand. Each struct is marked with `#[pyclass]`, which tells `PyO3` that this struct needs Python bindings. They are also marked with `#[derive(Default, Debug, Clone, PartialEq, Serialize, Deserialize)]`, which tells the library `Serde` how to deal with them when serialising and deserialise data. These are very important as otherwise accessing the data returned by the API would be very hard to work with.

src/lib.rs

```rust
#[pyfunction]
fn send_request(location: String, api_key: String) -> PyResult<Master> {
    let response = reqwest::blocking::get(format!(
        "https://api.openweathermap.org/data/2.5/weather?q={}&APPID={}",
        location, api_key
    ))
    .unwrap();
    let mut weather = response.json::<Master>().unwrap();
    format_temps(&mut weather);
    Ok(weather)
}
```

This is the function I use to send the API request to OpenWeather, it uses the `reqwest` library mentioned earlier. It sends a request using the `get()` function. This function is not ayncronous, as it is the from the `blocking` module so it will freeze the program while waiting for the data to be returned. The JSON data is then parsed into the variable `weather`, which is of type `Master`. The `format_temps()` function is then called, passing in a mutable reference to the `weather` variable. This will convert the temperatures returned, which are currently stored in Kelvin, to Celsius, and then round them to two decimal places and directly mutate the `weather` variable, so we don't have to return it. This function is marked with `#[pyfunction]`, which indicates to `PyO3` that this function needs Python bindings, which will allow us to call it from Python.

`main.py`

```python
def get_weather(self, action):
    weather = weather.send_request(
        self.location_entry.get_text(), "e391a6cfbcd81421bbc316f0eb5ab74c"
    )
    self.main.set_text(f"The weather is currently: {weather.weather[0].main}")
    self.description.set_text(
        f"The weather is more specifically, : {weather.weather[0].description}"
    )
    self.temp.set_text(f"The temperature is: {weather.temp.temp}")
    self.feels_like.set_text(
        f"The temperature feels like: {weather.temp.feels_like}"
    )
    self.temp_max.set_text(
        f"The maximum temperature today will be: {weather.temp.temp_max}"
    )
    self.temp_min.set_text(
        f"The minimum temperature today will be: {weather.temp.temp_min}"
    )
```

This code calls the rust module's `send_request()` function and passes in the text from the user input box. It then sets the text of each label to the respective data, such as what temperature it currently is.

Evaluation / Testing

Sadly, due to the data being weather data, I cannot perform unit tests on the Rust part of the code, as I wouldn't be able to check data returned from the API against past data as the weather is always changing. We can however check if when we press the button, a request is properly sent. We can also check if the UI will even run, by simply running the Python script.