Dubbelklik om afbeelding in te voegen, of verwijder dit veld

Creating an interactive weather forecasting dashboard in Python

Training manual

HKV

GLOBAL CENTER ON ADAPTATION

TAHMO

IRI

Deltares

Global Center on Adaptation

GLOBAL
CENTER ON
ADAPTATION

Creating an interactive weather forecasting dashboard in Python

Training manual

Author(s)
Thomas Stolp
Dorien Lugt

PR4932.10
November 2023

Table of contents

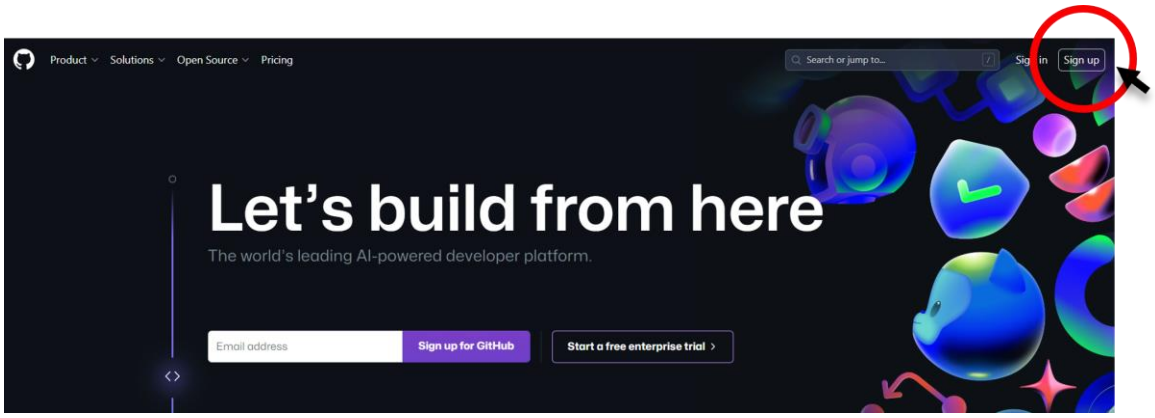
1 Getting started

In this chapter we introduce the environment that we will use in this training, that is GitHub Codespaces and Jupyter Notebooks. If you are already familiar with both, you can go directly to the Chapter 2.

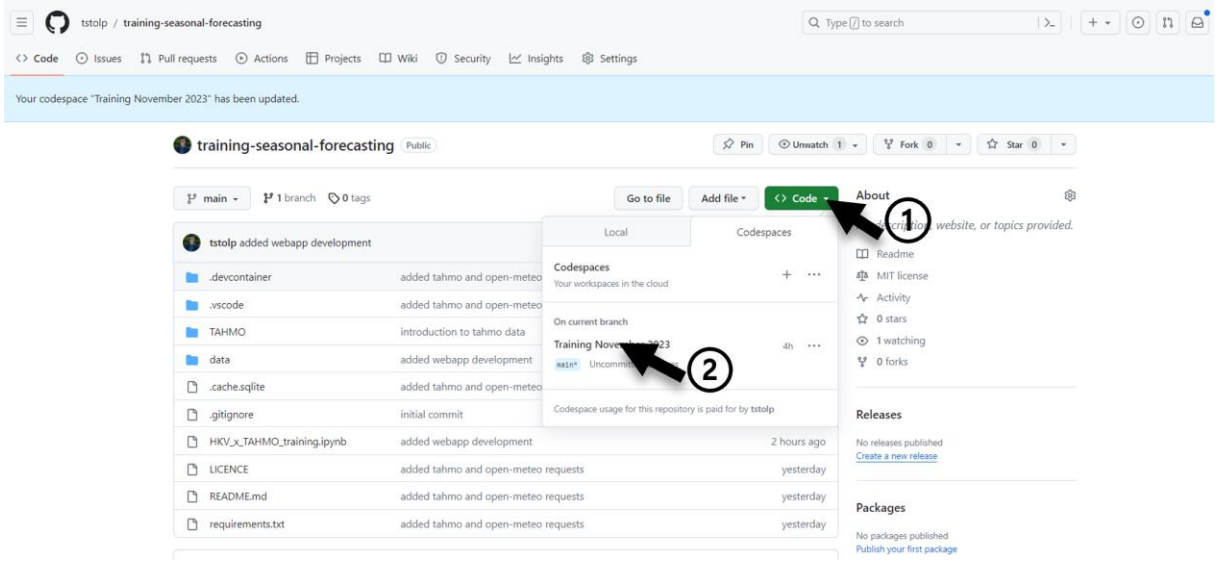
1.1 GitHub Codespaces

GitHub Codespaces is a cloud-based development environment provided by GitHub, designed to streamline the process of coding and collaboration. With GitHub Codespaces, you can create and manage your development environment entirely online, eliminating the need for complex local setup and configuration. A GitHub repository was created for this training and it can be found on https://github.com/tstolp/training-seasonal-forecasting.

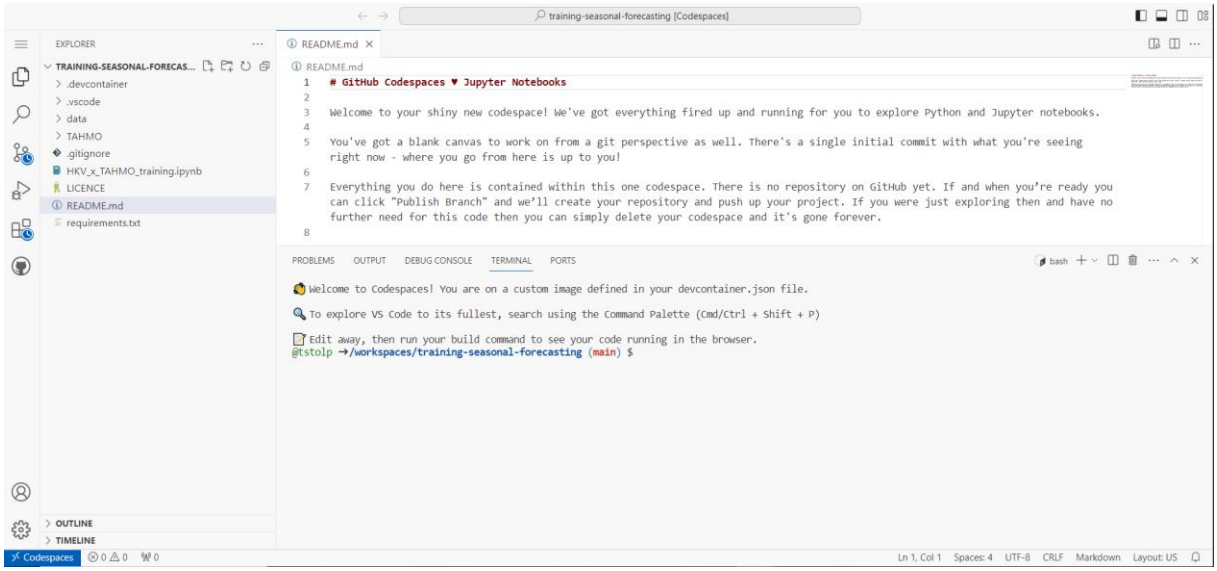
For using GitHub Codespaces, an account is required. If you do not have an account yet, please visit https://github.com and click the "Sign up" button.



Once you are logged in, go to the repository (https://github.com/tstolp/training-seasonal-forecasting) and click the green "Code" button.



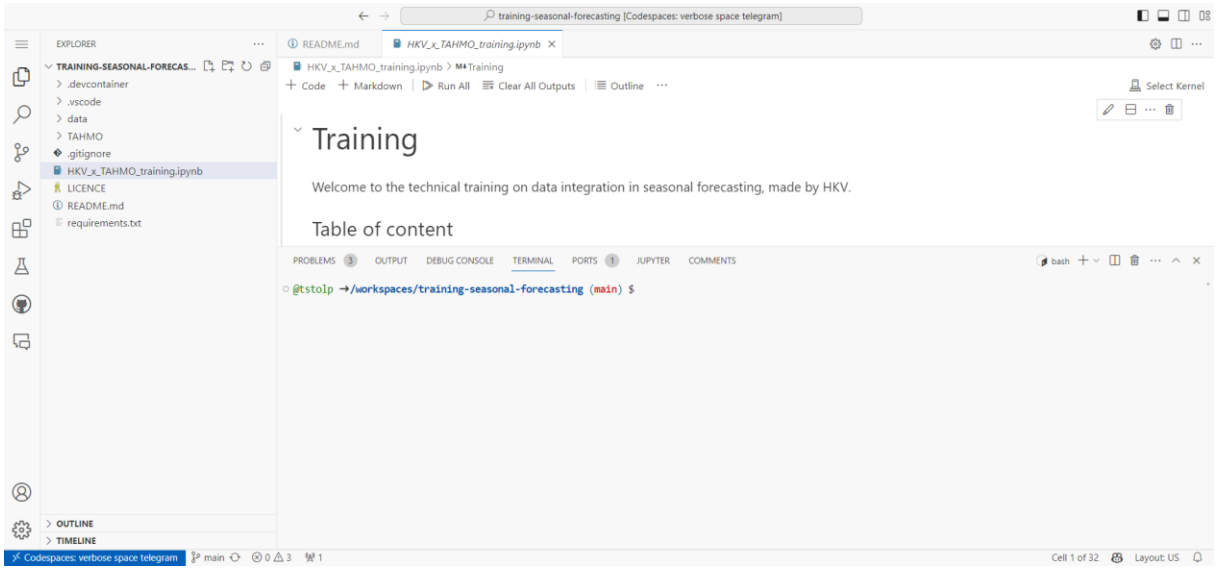
After that, click on “Create a new Codespaces on main”. Once you have opened the link to Codespaces, a developing environment is created for you (can take a minute).



You will see the following screen, click on the notebook in the left side of the screen “training-notebook.ipynb”. This will open the notebook which the training is based on. Go to the next section to learn more about Jupyter Notebooks and how to interact with the code.

1.2 Jupyter Notebooks

Once you opened the Jupyter Notebook, you see cells which are the base components. You can run a cell by clicking the run button on the upper left corner of the cell. Another option is to use the keyboard shortcut “shift + enter” or “control + enter”. The first runs the cell and automatically moves to the next.



You can close the lower window, this is the terminal which we won ‘t use in this training.

2 Introduction to Application Programming Interfaces (APIs)

In this chapter we give a brief introduction to API's, specifically RESTfull API's and we explain how you can use API clients in Python.

2.1 REST APIs

An API (Application Programming Interface) is a protocol that defines how systems can communicate with each other. A REST API is built following the design principles of Representational State Transfer (REST). REST is very flexible, and therefore, it can be found all over the internet. It uses standard HTTP protocols, which are:

1. **GET**: This method is used to request data from a specified resource.
2. **POST**: This method is used to submit data to a specified resource.
3. **PUT**: Used to update data.
4. **DELETE**: Used to remove data.

An API needs an **endpoint**, which is a specific URL to which the API sends requests and from which it receives responses. In simpler terms, an API endpoint is a designated route or path on a server that the API uses to perform a particular function. Each endpoint represents a specific operation or resource in the API.

2.2 Structure

API endpoints are typically organized into a hierarchical structure, and they serve as the means by which clients (applications or systems) interact with the functionality provided by the API. Endpoints are crucial for specifying the type of operation to be performed, the location of a resource, or the data to be retrieved or manipulated.

Here's a breakdown of the components of an API endpoint:

- **Base URL**: Provides the root address of the API, the location where the API is hosted. For example, 'https://api.example.com'.
- **Path**: Comes after the base URL and represents the specific resource. For example '/users'.
- **HTTP Method**: The actual method, for example GET to retrieve data.

You can also provide parameters and headers when making API requests, and they play a crucial role in customizing and specifying the details of your request. Here's an explanation of how

parameters and headers work in API requests. They are usually added as key-value pairs and are appended to the endpoint's URL. There are two main types of parameters:

1. **Query Parameters**: '?key1=value1&key2=value2'

2. **Path Parameters**: '/resource/{parameter}'

Many APIs require authentication. To use them, you often need an API key, a unique identifier that grants you access. The authentication is an example of a header, it provides additional information about the request or the client making the request.

2.3 Response

Understanding the API documentation is crucial. It provides details on available endpoints, request formats, parameters, and response structures. The responses usually come in JSON or XML format. When a request fails or was successful, a HTTP status code is often returned.

- **200: OK**
  The request was successful.
- **400: Bad request**
  The request cannot be fulfilled due to incorrect syntax or invalid parameters.
- **401: Unauthorized**
  Authentication is required, and the provided credentials are invalid.
- **403: Forbidden**
  The server understood the request, but it refuses to authorize it.
- **404: Not Found**
  The requested resource could not be found on the server.
- **500: Internal Server Error**
  The requested resource could not be found on the server.

2.4 API Clients

An API client is software or code that uses the features of an API by sending the requests (possibly with parameters and headers) and response handling. In python the requests library (https://pypi.org/project/requests/) is commonly used to send HTTP requests and handling the responses.

The website httpbin.org is an API endpoints that let's you test and inspect data that you sent to a service. We can use it to test the request module in python. In the cell below we create a GET request using the command 'request.get()' with query parameters and header provided as arguments. From the response we can get de status code which is 200 meaning that our request was successful.

```python
# Import the 'requests' library
import requests

# Define a dictionary 'params' with key-value pairs to be included as query parameters
params = {'key1': 'value1', 'key2': 'value2'}

# Define a dictionary 'headers' to include additional headers in the GET request.
headers = {'user-name': 'password123'}

# Use the 'requests.get()' method to make a GET request to the specified URL 'https://httpbin.org/get'.
# Include the defined 'params' and 'headers' in the request.
r = requests.get('https://httpbin.org/get', params=params, headers=headers)

# The response object 'r' now contains the server's response to the GET request.
# You can access various properties of the response object, such as 'status_code', 'text', 'json()', etc.
```

[27] ✓ 0.5s Python

```python
# Below are a few examples of using the response object:
# Print the HTTP status code received in the response.
print(f"Status Code: {r.status_code}")
```

[28] ✓ 0.0s Python

```
Status Code: 200
```

3 TAHMO station API

In this chapter we describe how you can access the TAHMO API endpoint and retrieve data for a variety of variables and stations. We will create an interactive visualization of precipitation measurements at one of the stations.

3.1.1 TAHMO API

The Trans-African Hydro-Meteorological Observatory (TAHMO) maintains a network of weather stations across Africa. The data of these stations can be retrieved using the API. We can use the API-V2 client that can be found on the TAHMO GitHub page (https://github.com/TAHMO/API-V2-Python-examples).

In this training we use the demo credentials which gives us access to the data of three stations in total. The credentials are already in the Notebook, all we have to do is run the cell and we are good to go.

```
[12]  ✓ 0.0s                                                    Python

1  # Import the TAHMO module
2  import TAHMO
3
4  # The demo credentials listed below give you access to three pre-defined stations.
5  api = TAHMO.apiWrapper()
6
7  # set the credentials
8  api.setCredentials('demo', 'DemoPassword1!')
```

The TAHMO.apiWrapper() is a client that handles the request and responses of the API for us. In the following cell we make a request to retrieve the stations that we actually have access to. We can see from the response that these stations are identified with codes.

```
[13]  ✓ 0.4s                                                    Python

1  # list other stations that are available
2  stations = api.getStations()
3  print('Account has access to stations: %s' % ', '.join(list(stations)))

···  API request: services/assets/v2/stations
     Account has access to stations: TA00134, TA00252, TA00567
```

3.1.2 Variables

Next, we can get the variables that are recorded and available for these stations. This is done with the 'getVariables()' method. Run the cell in the Notebook and we get a list of variables with corresponding abbreviations and units. In this training we will focus on precipitation, but requesting data for other variables goes in a similar way.

Variable	Short code	Unit
Atmospheric pressure	ap	kPa
Depth of water	dw	mm

Electrical conductivity of precipitation	ec	mS/cm
Electrical conductivity of water	ew	mS/cm
Lightning distance	ld	km
Lightning events	le	-
Shortwave radiation	ra	W/m2
Soil moisture content	sm	m3/m3
Soil temperature	st	degrees Celsius
Surface air temperature	te	degrees Celsius
Vapor pressure	vp	kPa
Wind gusts	wg	m/s
Wind speed	ws	m/s
Temperature of humidity sensor	ht	degrees Celsius
X-axis level	tx	degrees
Y-axis level	ty	degrees
Logger battery percentage	lb	-
Logger reference pressure	lp	kPa
Logger temperature	lt	degrees Celsius
Cumulative precipitation	cp	mm
Water level	wl	m
Water velocity	wv	m/s
Precipitation	pr	mm
Relative humidity	rh	-
Wind direction	wd	degrees
Soil electrical conductivity	se	mS/cm
Water temperature	tw	degrees Celsius
Water discharge	wq	m3/s

3.2 Retrieve and plot daily precipitation data

In this paragraph we will retrieve the actual data of a weather stations using a GET request. We choose to use the station 'TA00567'. With the following code, we can get the coordinates and name of this stations. It turns out to be the weather station at the Accra Girls Senior High School.

```
# choose a station
station = 'TA00567'

# get the data
station_data = api.getStations()[station]

print()
print( f"Station name =  {station_data['location']['name']}")
print( f"Longitude =  {station_data['location']['longitude']:.02f}")
print( f"Latitude =  {station_data['location']['latitude']:.02f}")
```

[45] ✓ 0.9s Python

```
API request: services/assets/v2/stations

Station name =  Accra Girls SHS
Longitude =  -0.19
Latitude =  5.60
```

In the cell below, we create the get request using the short code for precipitation and a time interval to get only the data from the present year. The response is a pandas DataFrame, which is

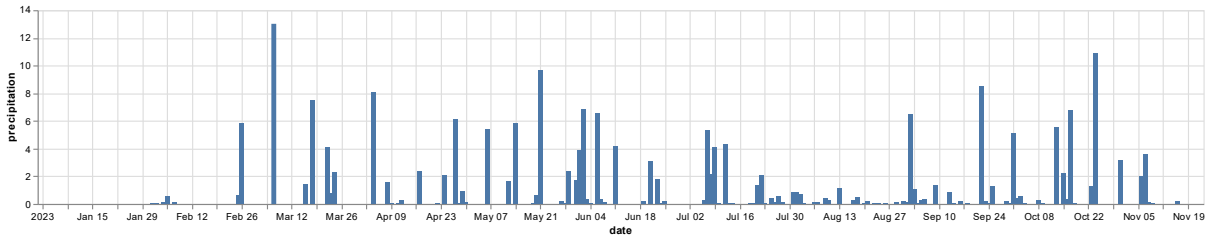
a data structure consisting of rows and columns. In this case two columns, the precipitation that was measured and the time of measurement.

```python
# Example 3: Retrieve a pandas dataframe containing the time serie of surface air observations and save to C
startDate = '2023-01-01'
endDate = '2023-11-22'
variables = ['pr']

df = api.getMeasurements(station, startDate=startDate, endDate=endDate, variables=variables)
df.index.name = 'Timestamp'
df.to_csv('data/timeseries.csv', na_rep='', date_format='%Y-%m-%d %H:%M')
print('Timeseries saved to file "timeseries.csv"')

```

[46] ✓ 5.6s Python

```
API request: services/measurements/v2/stations/TA00567/measurements/controlled
Timeseries saved to file "timeseries.csv"
```

Next we can show what the data looks like in a bar plot. In het notebook we use Vega-Altair (https://altair-viz.github.io/) to create an interactive visualization in which you can zoom and hover over the bars to see the exact data and measured precipitation.



4 Open-Meteo weather API

In this chapter we show how the API of Open-Meteo can be used to get forecast data for a given location (latitude and longitude coordinates). We will visualize this data within an interactive plot.

4.1 Open-Meteo

Open-Meteo (https://open-meteo.com) is a weather API, open source and free for non-commercial use (no credentials needed!). It utilizes the weather forecasts provided by weather services such as ECMWF. Normally these numerical weather forecasts are challenging to work with due to the knowledge required to handle the projections and data formats.

4.1.1 Available APIs

There are various APIs available on Open-Meteo, an overview is given below:

1. **Forecast API** – weather forecasts up to 14 days.
2. **Historical weather API** – past weather data (hourly) going back to 1940.
3. **Ensemble models API** – combines over 200 individual forecasts.
4. **Climate Change API** – downscaled IPCC climate predictions at 10km resolution.

4.2 ECMWF IFS

With the Open-Meteo forecasting API, one can request open-data ECMWF weather forecasts generated by the IFS weather model. Access to the open data is limited to a resolution of 40 km and 3-hourly values. Despite this limitation, the model still has impressive accuracy in predicting large-scale weather patterns.

4.2.1 Forecast API

We will request forecasts of the ECMWF model by using the GET method from the base url: https://api.open-meteo.com/v1/forecast. We can give some parameters such as the variable name (we use "precipitation") and the amount of days of the forecast (for example 10).

Another option in the forecast API is the parameter "past_days". If this parameter is set, past weather data can be returned. In this example we set it to 90 days.

In the code block below we created a function that returns the data we get as the response of our GET request. The format of this data is JSON and the frequency of the measurements will need to be converted to daily sums. We use the latitude and longitude of the TAHMO weather station at the Accra Girls Senior High School.

```python
import requests


def get_ecmwf_precipitation(lon, lat):
    """Retrieve the ECMWF precipitation forecast from the Open-Meteo API and return a JSON object"""

    base_url = "https://api.open-meteo.com/v1/forecast"

    # Specify the parameters for the ECMWF precipitation forecast
    params = {
        "longitude" : lon,
        "latitude" : lat,
        "daily" : "precipitation_sum",
        "past_days" : 90,
        "timezone" : "auto",
        "hourly" : "precipitation",
        "start" : "current",
        "forecast_days" : 10,
        "models" : "ecmwf_ifs04"}

    try:
        # Make a request to the Open-Meteo API
        response = requests.get(base_url, params=params)
        data = response.json()
        return data
    except requests.RequestException as e:
        print(f"Error: {e}")

data = get_ecmwf_precipitation(lon=station_data['location']['longitude'], lat=station_data['location']['latitude'])

```

[17] ✓ 0.2s Python

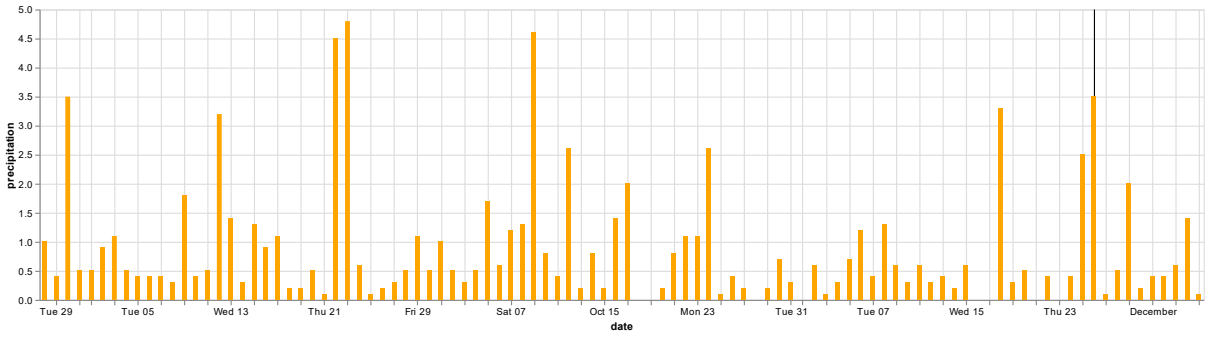
To process the data we define the following function, which converts data types and renames columns.

```python
def process_ecmwf_precip_data(data):
    """Load the precipitation data from the Open-Meteo API and return a pandas dataframe"""
    df = pd.DataFrame.from_dict(data['hourly'])
    df['time'] = pd.to_datetime(df['time'])
    df.loc[:,'date'] = df['time'].dt.date
    df['date'] = pd.to_datetime(df['date'])
    df = df[['date', 'precipitation']].dropna()
    df = df.groupby('date').max().reset_index()# .set_index('date')
    return df

df_ecmwf = process_ecmwf_precip_data(data)
df_ecmwf.head()
```

[19] ✓ 0.0s Python

	date	precipitation
0	2023-08-28	1.0
1	2023-08-29	0.4
2	2023-08-30	3.5
3	2023-08-31	0.5
4	2023-09-01	0.5

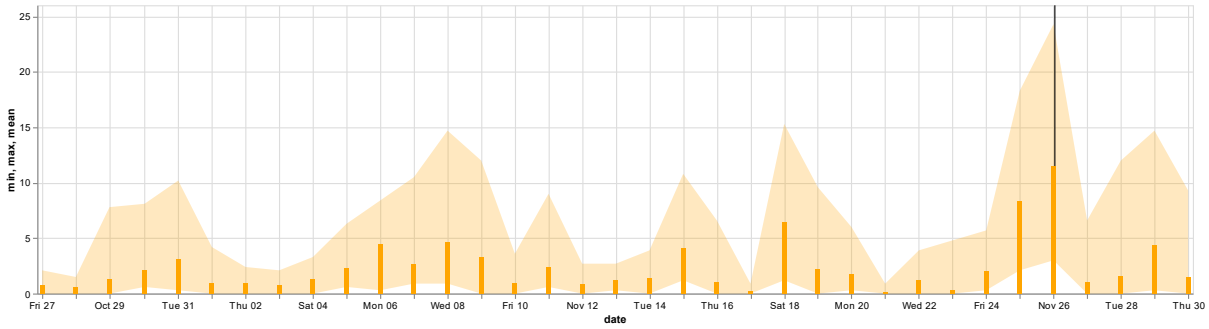
We can plot this data, here we place a black vertical rule at the current date such that we can see where the forecast starts.



4.2.2 Ensemble API

With an ensemble forecast, multiple versions of a numerical weather model are used to generate a diverse set of potential outcomes. Each member is initiated with slightly varied initial conditions or model parameters, accommodating uncertainties and atmospheric variations and producing a collection of perturbed forecasts.

With the Open-Meteo ensemble API, we can retrieve ECMWF IFS ensembles. In the notebook we show how to request data using the ensemble API and plot the minimum and maximum value such that we see a range of plausible events.



5 Building a dashboards

In this chapter we give an introduction to Solara, which is a module for creating web applications that run in Jupyter Notebooks. We will create a dashboard for ensemble forecasts of precipitation at certain TAHMO weather stations.

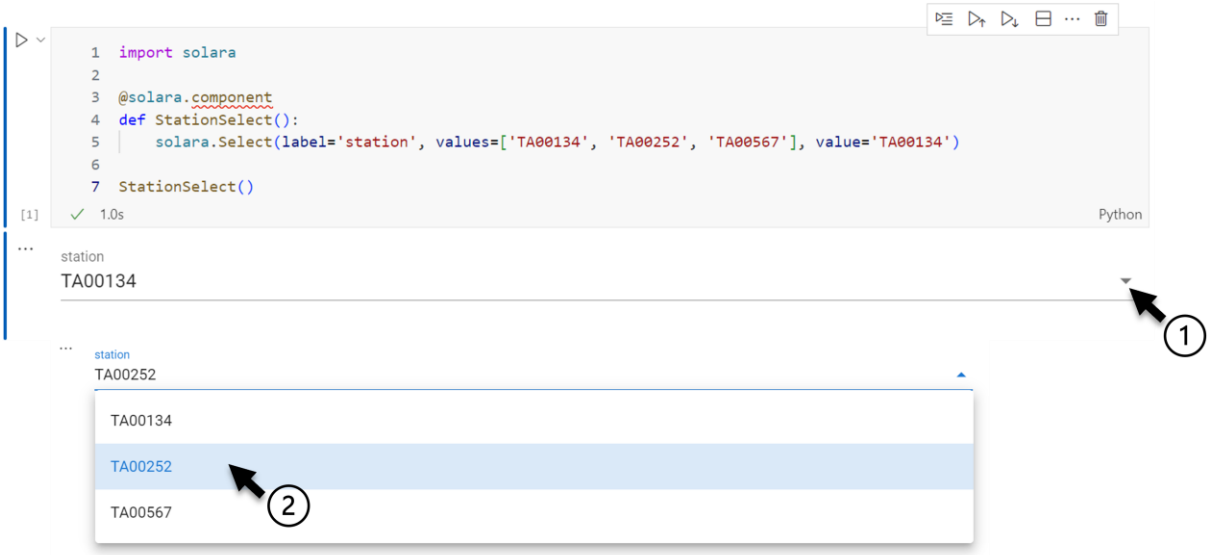
5.1 Introduction to Solara

Solara (https://solara.dev) is an Open Source library for creating web apps in a Jupyter Notebook, but they will also work using professional web frameworks. It builds on top of IPywidgets that allows you to use interactive controls such as sliders and checkboxes.

5.1.1 Components and event handling

Solara uses 'components' which are building blocks of the application and which are reusable. There are two types: **Widget components** and **function components**. Widget components are IPywidgets for creating the interactive application. Function components combine logic state and other components to create complex and dynamic apps.

Inside the function, you can create the component's structure by calling Solara's built-in components or creating custom components to suit your specific needs. In the example below we create a selection menu where you can select the code of the TAHMO stations.



The design philosophy in Solara is to first create all components that you need in your application or dashboard and then to layout these components in a 'Page'. More information about layout can be found on https://solara.dev/docs/howto/layout.

Components are reusable. See the following example, we create a page that contains two elements of the StationSelect component.

Components can capture user input and respond to events. To handle the input of users, you define callback functions and connect them to the component using the Solara's event handling system.

In the example above we created a component which uses a callback function (on_value_change) and connects it to the event handling (on_value).

5.1.2 Arguments and state

The components we have created can accept arguments, the values passed when we call the functions. In this way it is possible to have elements from the same component but with different behavior or appearance.

It is also possible to manage the state of a component in Solara by creating a reactive variable with solara.reactive(). Reactive variables store values that can change over time, and based on that value can trigger other components in your application.

```python
import solara

station = solara.reactive('TA00134')

def set_station(value):
    station.value = value

@ solara.component
def StationDisplay():
    solara.Info(f"Station: {station.value}")

@solara.component
def StationSelect():
    solara.Select(label='station', values=['TA00134', 'TA00252', 'TA00567'], value=station.value, on_value=set_station)

@solara.component
def Page():
    StationSelect()
    StationDisplay()


Page()
```

[10] ✓ 0.0s Python

station
TA00567

Station: TA00567

In the example above we create a state variable (or reactive variable) "station" that has the default value "TA00134" but it can change to other values. By selecting another value in the selection menu, we see that the variable station is changed. We show this by displaying the value of station with a solara.info component.

5.2 Create a forecasting dashboard

In this section we will work on an actual dashboard to see the ensemble forecasting for TAHMO station locations.

5.2.1 Create a map with IPyleaflet

The first step is to create a map so we can see the location of the weather station. This will use the IPyleaflet module. Let's start by creating a dictionary with station data:

```python
station_list = ["TA00134", "TA00252", "TA00567"]

# station "TA00134" is empty, therefore we remove it from the list
station_list.remove("TA00134")

station_data = {}

for station in station_list:
    station_data[station] = api.getStations()[station]
```

[23] ✓ 0.9s Python

```
API request: services/assets/v2/stations
API request: services/assets/v2/stations
```

Note that station "TA00134" has not been recording precipitation data for the last months, so we exclude this station from our list.

Below we create our state variables and the selection menu we created earlier.

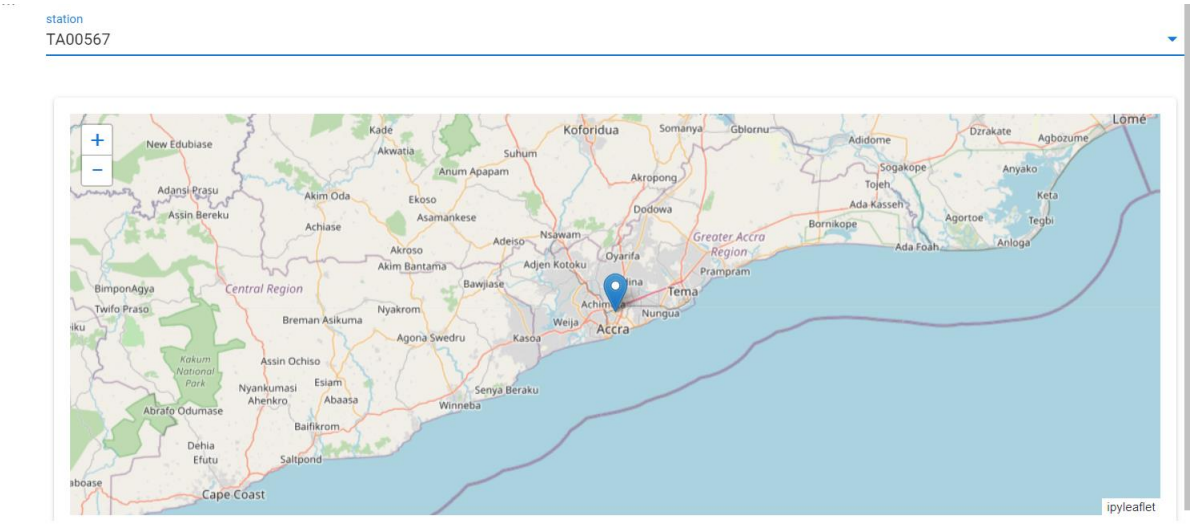
```python
import solara
import ipyleaflet
import TAHMO
from ipywidgets import HTML

# Create a TAHMO API wrapper and set credentials
api = TAHMO.apiWrapper()
api.setCredentials('demo', 'DemoPassword1!')

station_default = 'TA00252'
center_default = (station_data[station_default]['location']['latitude'], station_data[station_default]['location']['longitude'])
zoom_default = 9

# Define reactive variables for station data
station = solara.reactive(station_default)
zoom = solara.reactive(zoom_default)
center = solara.reactive(center_default)

def set_station(value):
    station.value = value
    center.value = (station_data[value]['location']['latitude'], station_data[value]['location']['longitude'])

@ solara.component
def StationSelect():
    """Solara component for a station selection dropdown."""
    solara.Select(label="station", values=station_list, value=station.value, on_value=set_station, style={"z-index": "10000"})

```

We add a “view” component that displays the station on the maps and automatically zooms to another station when selected from the dropdown menu.

```python
@solara.component
def View():
    """Solara component for displaying a map view with a marker for the selected station."""
    ipyleaflet.Map.element(center=center.value,
                           zoom=9,
                           on_center=center.set,
                       scroll_wheel_zoom=True,
                       layers=[ipyleaflet.TileLayer.element(url=ipyleaflet.basemaps.OpenStreetMap.Mapnik.build_url())] + [ipyleaflet.Marker.eleme
                       )

@solara.component
def Page():
    """Solara component for a page with two cards: View and StationSelect."""
    with solara.Column(style={"min-width": "500px", "height": "500px"}):
        with solara.Row():
            StationSelect()
        with solara.Card():
            View()

Page()
```

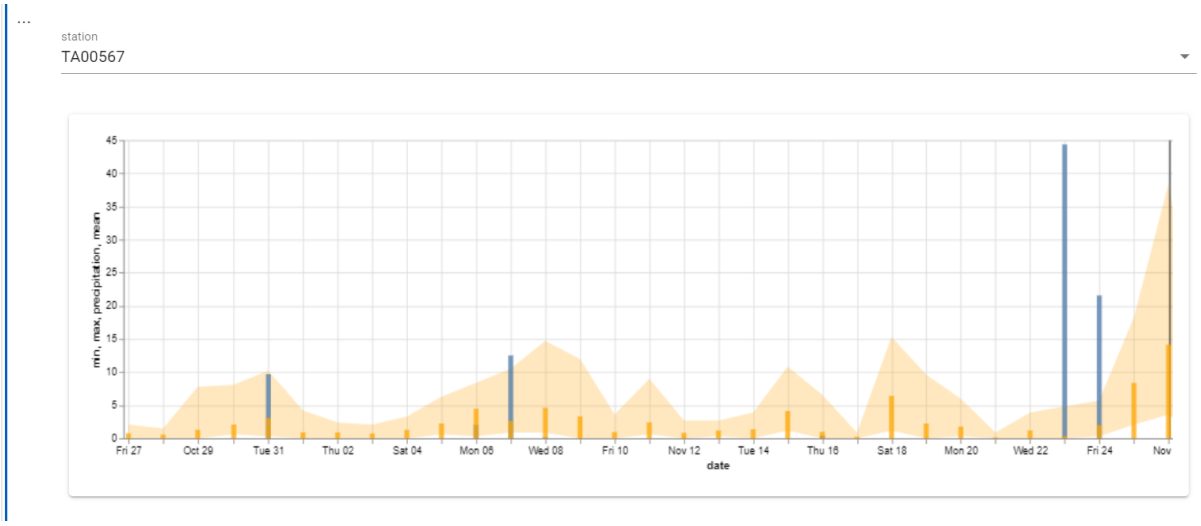
1.2s Python

The output of this code block is shown below. Select another station and the center of the map will the new location.



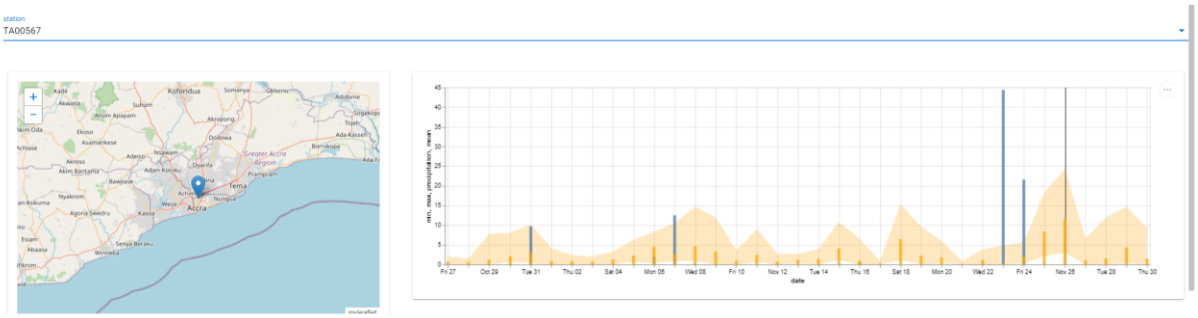
5.2.2 Create a timeseries plot with precipitation measurements and forecasts

The second figure presents the precipitation measurements at the TAHMO station (in blue) and the mean of the ensemble forecast (in orange). The orange band shows the minimum and maximum daily precipitation of the ensembles. Run the code in the Notebook and see if you can get the same result.

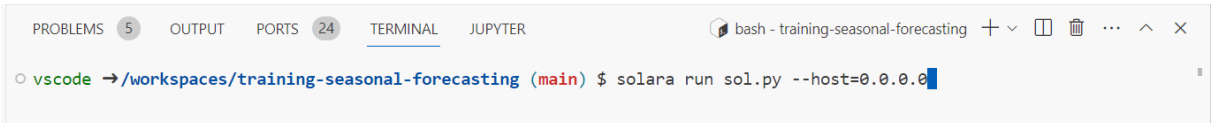


5.2.3 Creating a final dashboard

In the notebook, we combine the map and the timeseries plot. When you select another station, both will update, a request is made to get the ensemble forecasts for the new station.



You can either create a dashboard inside the notebook, such as is done in this training, but it is also possible to run a python script using Solara server. This is done by opening a new terminal and using the following command. The use of the –host argument is was needed on my local computer but it may work without this in the Codespaces environment. Try it out.



This will return a URL that you can open in your own browser!

HKV

Location Lelystad
Botter 11-29
8232 JN Lelystad
The Netherlands

Location Delft
Informaticalaan 8
2628 ZD Delft
The Netherlands

Location Amersfoort
Berkenweg 7
3818 LA Amersfoort
The Netherlands

0320 294242
info@hkv.nl
www.hkv.nl