EPSILON – European Platform for Data Science: Incubation, Learning, Operations and Network

Training Material for Teaching and Self-Learning

Selected Use Cases

Module 4/6

Hochschule Harz
Harz University of Applied Sciences
EPSILON
Funded by the European Union

Agenda

- Project 1: Where to build new bicycle parking spots in Paris?
- Project 2: Predicting Long-Term Unemployment in Portugal
- Project 3: COVID-19 mortality surveillance platform
- Project 4: Domestic Violence Data Observatory

Where to build new bicycle parking spots in Paris?

Supporting data-driven decision-making with open data

Where to build new bicycle parking spots in Paris?

Project Team:

Volunteers from CorrelAid

10 people divided into a Data and a Research Team

Stakeholder:

City of Paris Mobility Department

Problem Statement:

The City of Paris is currently in transition to “Green Mobility” which includes strengthening bicycle infrastructure. Until now, streets have been mainly used by cars as well as parking spots. At the same time multiple national and regional French governments started publishing public data.

In 2020, government officials published the “Plan Velo 2021-2026“, which includes plans for a massive improvement in cycling infrastructure.

The goal of this project is to support the Green Mobility Transition by providing a compelling visualization of where bicycle parking spots are needed.

Where to build new bicycle parking spots in Paris?

Detailed Information

DFG organization:
CorrelAid

Partner type:
Governmental agency

Partner name:
City of Paris Mobility Department

Sustainable Development Goal (SDG):
11

Type of interaction:
Short-term project

Type of analytics:
Modeling

Type of data:
Government data

All data is available here

Where to build new bicycle parking spots in Paris?

Internal & External Data

Most of the data was collected by the government and made available on open data platforms.

- Bicycle parking spaces
  - collected by Île-de-France Mobilités
- Visitor numbers at train stations
  - collected by SNCF (Société nationale des chemins de fer français)
- Incoming annual traffic volume per station of the rail network 2021
  - collected by RATP (Régie autonome des transports Parisiens)
- Green spaces and similar
  - surveyed by Mairine de Paris
- Parking on public roads - parking spaces
  - surveyed by Mairine de Paris

Methods used for data processing

- Aggregation
- Normalization
- Formation of weighted average

Since the aim of the project was to provide graphical representation of parking space demand by geographical unit in Paris, a data processing process only took place to a limited extent. During data processing, the individual data frames from different data sources were aggregated and summarized into an overall data set. In addition, the individual variables considered and collected were **normalized** using the available bicycle parking spaces in order to finally determine the actual demand per level of the **smallest statistical spatial unit used in France (IRIS)**. In the subsequent presentation of the data, however, this was converted to the respective users or residents and the respective influence on the parking space requirements for bicycles and graphically displayed using a color scale.

Where to build new bicycle parking spots in Paris?

Data Sources

Bicycle Parking Spots
Data on already available parking spots in the city area and in train stations from two separate sources.

Census Data
Census provides current data on population density.

Green Spaces
Location and size of Parks and other green spaces.

Public Transport
Location and traffic through train, Metro and RER stations in Paris. Multiple data sources needed to be combined to obtain a full picture of public transport.

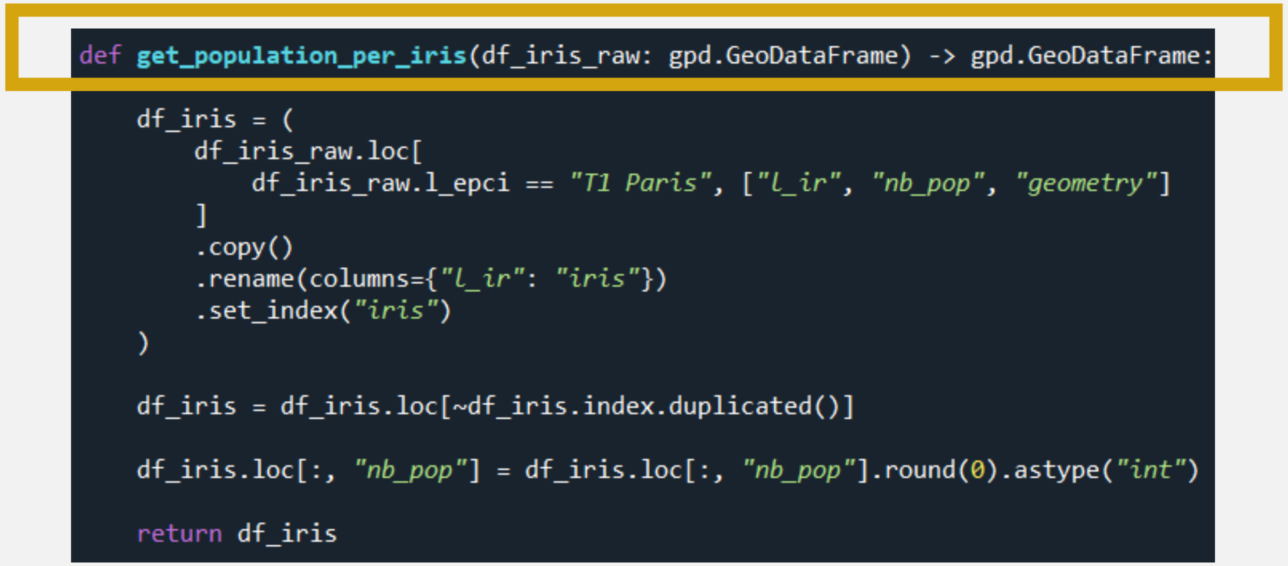
Museum Data
Location and attendance figures for museums in the city area.

Schools
Location and total capacity of schools.

Shop
Location of Shops.

Code Example

Cleaning Input Data

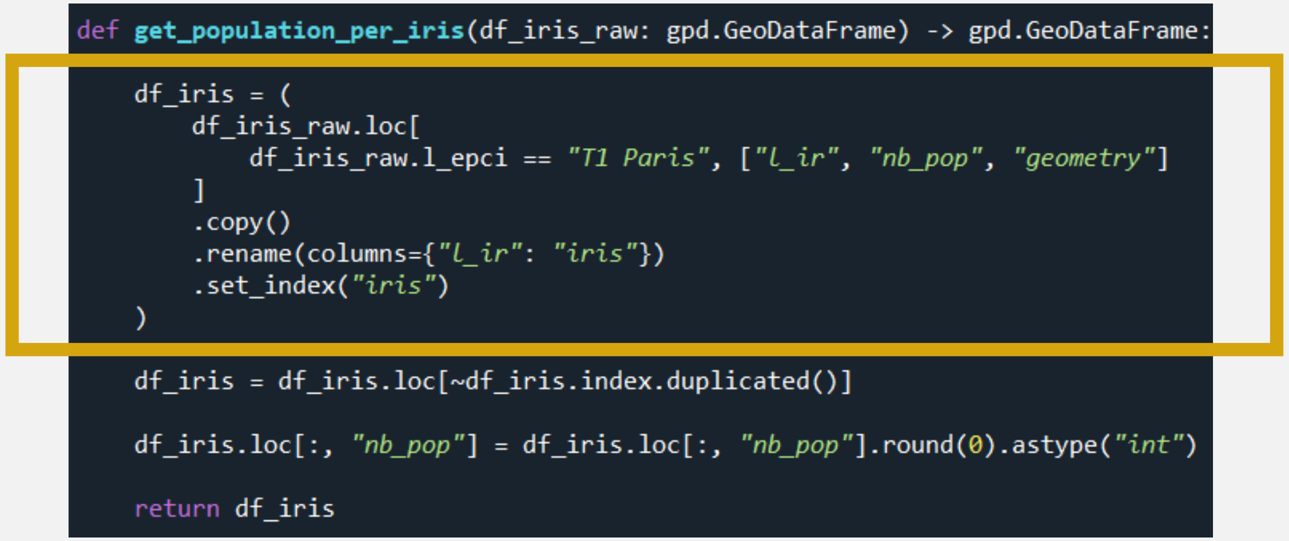


```python
def get_population_per_iris(df_iris_raw: gpd.GeoDataFrame) -> gpd.GeoDataFrame:

    df_iris = (
        df_iris_raw.loc[
            df_iris_raw.l_epci == "T1 Paris", ["l_ir", "nb_pop", "geometry"]
        ]
        .copy()
        .rename(columns={"l_ir": "iris"})
        .set_index("iris")
    )

    df_iris = df_iris.loc[~df_iris.index.duplicated()]

    df_iris.loc[:, "nb_pop"] = df_iris.loc[:, "nb_pop"].round(0).astype("int")

    return df_iris
```

The project team used the **Python** typing notation to clearly indicate what the required input and the expected output of a function is.

Code Example

Cleaning Input Data



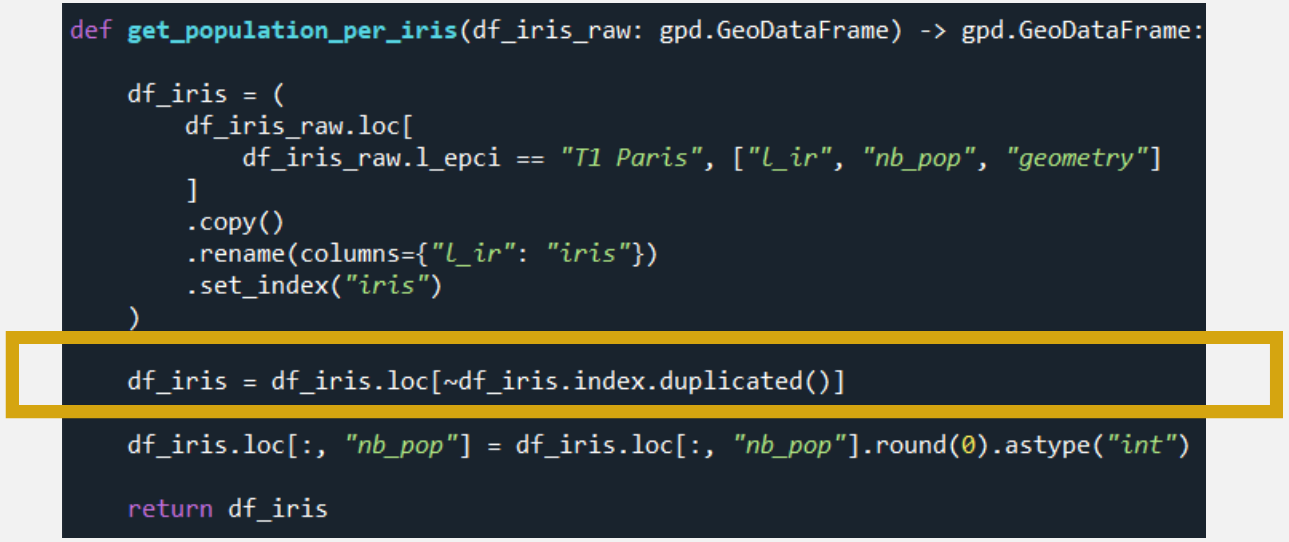
```python
def get_population_per_iris(df_iris_raw: gpd.GeoDataFrame) -> gpd.GeoDataFrame:

    df_iris = (
        df_iris_raw.loc[
            df_iris_raw.l_epci == "T1 Paris", ["l_ir", "nb_pop", "geometry"]
        ]
        .copy()
        .rename(columns={"l_ir": "iris"})
        .set_index("iris")
    )

    df_iris = df_iris.loc[~df_iris.index.duplicated()]

    df_iris.loc[:, "nb_pop"] = df_iris.loc[:, "nb_pop"].round(0).astype("int")

    return df_iris
```

The .geojson file contains multiple neighboring regions of France besides Paris, which is why this line of code reduces the data down to the region of Paris.

At the same time, irrelevant columns are dropped, and the index is reset to IRIS (the smallest spatial unit used for statistics in France).

Code Example

Cleaning Input Data

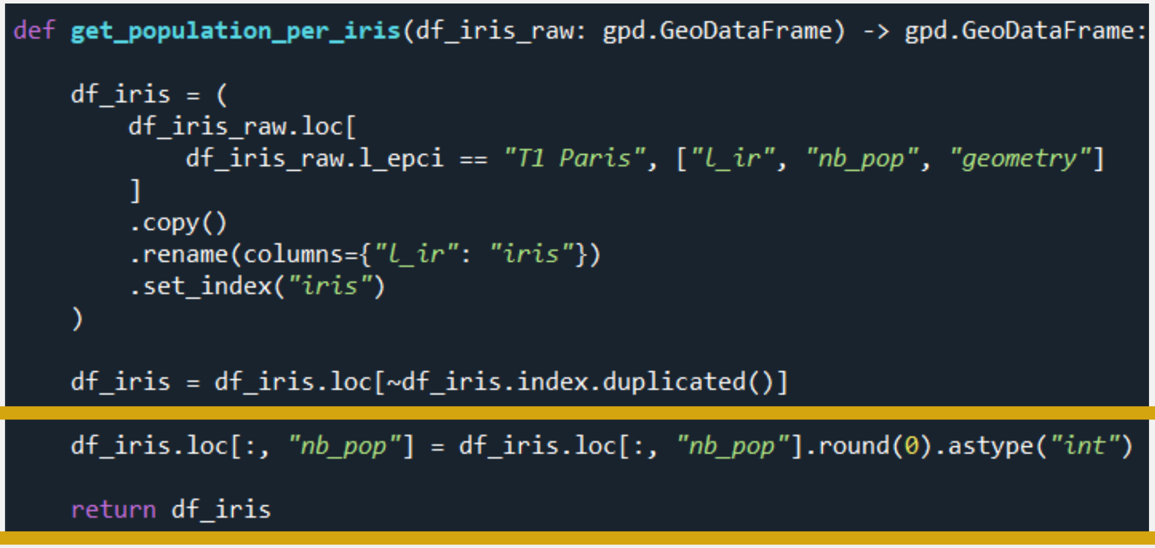


```
def get_population_per_iris(df_iris_raw: gpd.GeoDataFrame) -> gpd.GeoDataFrame:

    df_iris = (
        df_iris_raw.loc[
            df_iris_raw.l_epci == "T1 Paris", ["l_ir", "nb_pop", "geometry"]
        ]
        .copy()
        .rename(columns={"l_ir": "iris"})
        .set_index("iris")
    )

    df_iris = df_iris.loc[~df_iris.index.duplicated()]

    df_iris.loc[:, "nb_pop"] = df_iris.loc[:, "nb_pop"].round(0).astype("int")

    return df_iris
```

Through the previous process, some IRIS were duplicated in the data. These duplicates are now dropped.

Code Example

Cleaning Input Data

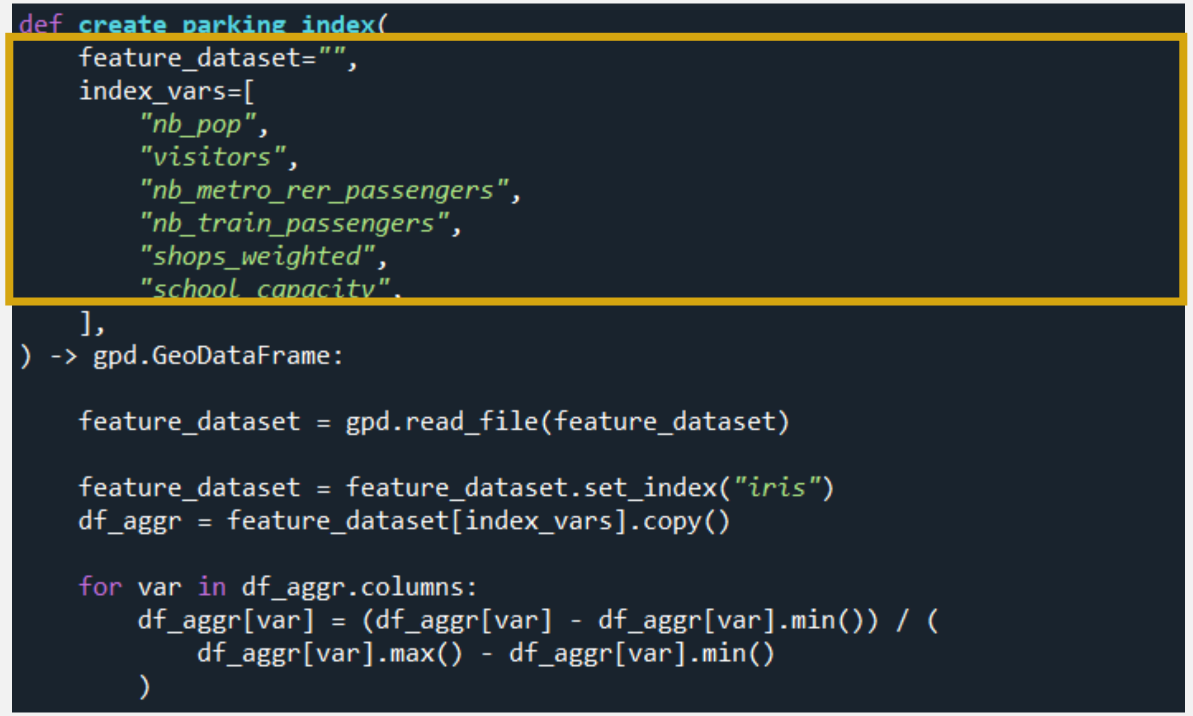


```
def get_population_per_iris(df_iris_raw: gpd.GeoDataFrame) -> gpd.GeoDataFrame:

    df_iris = (
        df_iris_raw.loc[
            df_iris_raw.l_epci == "T1 Paris", ["l_ir", "nb_pop", "geometry"]
        ]
        .copy()
        .rename(columns={"l_ir": "iris"})
        .set_index("iris")
    )

    df_iris = df_iris.loc[~df_iris.index.duplicated()]

    df_iris.loc[:, "nb_pop"] = df_iris.loc[:, "nb_pop"].round(0).astype("int")

    return df_iris
```

Finally, the relevant data (Population per IRIS) is converted from Float to Integer and the DataFrame is returned.

Code Example

Calculating Index

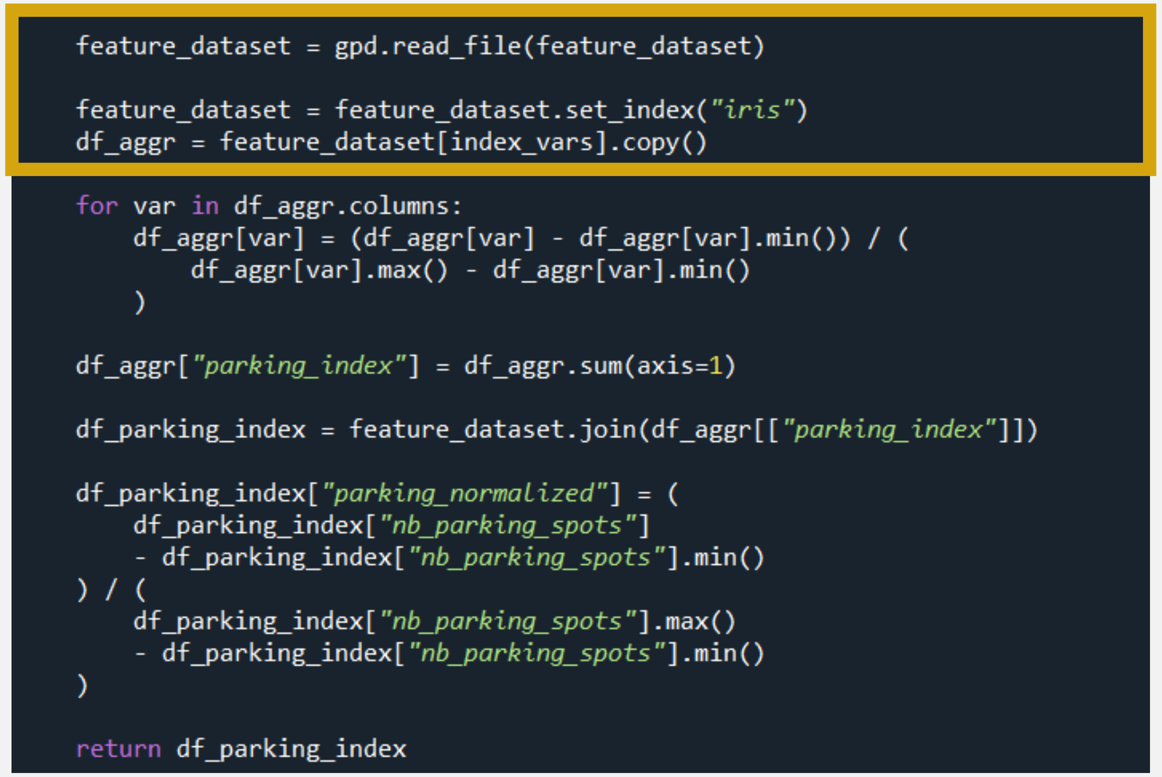


```
def create_parking_index(
    feature_dataset="",
    index_vars=[
        "nb_pop",
        "visitors",
        "nb_metro_rer_passengers",
        "nb_train_passengers",
        "shops_weighted",
        "school_capacity",
    ],
) -> gpd.GeoDataFrame:

    feature_dataset = gpd.read_file(feature_dataset)

    feature_dataset = feature_dataset.set_index("iris")
    df_aggr = feature_dataset[index_vars].copy()

    for var in df_aggr.columns:
        df_aggr[var] = (df_aggr[var] - df_aggr[var].min()) / (
            df_aggr[var].max() - df_aggr[var].min()
        )
```

As this function is only used to clarify what is happening in the code, it already contains the expected features as a default value.

Code Example

Calculating Index

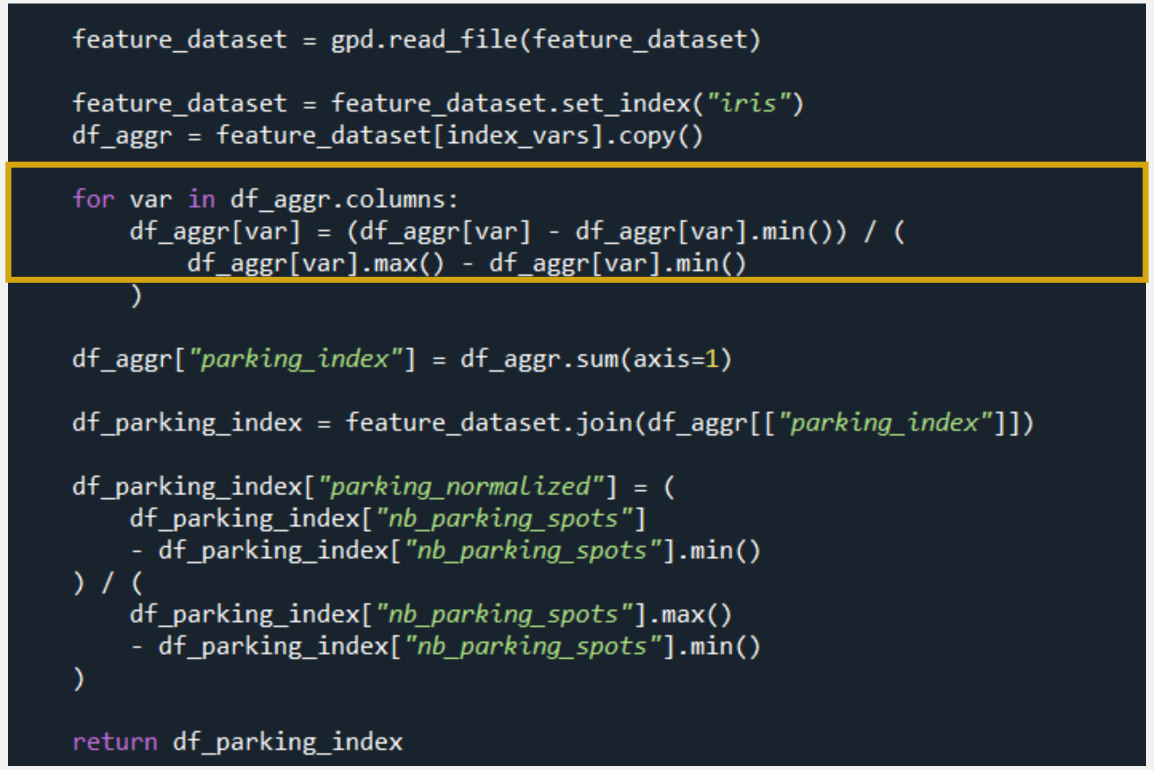


```python
feature_dataset = gpd.read_file(feature_dataset)

feature_dataset = feature_dataset.set_index("iris")
df_aggr = feature_dataset[index_vars].copy()

for var in df_aggr.columns:
    df_aggr[var] = (df_aggr[var] - df_aggr[var].min()) / (
        df_aggr[var].max() - df_aggr[var].min()
    )

df_aggr["parking_index"] = df_aggr.sum(axis=1)

df_parking_index = feature_dataset.join(df_aggr[["parking_index"]])

df_parking_index["parking_normalized"] = (
    df_parking_index["nb_parking_spots"]
    - df_parking_index["nb_parking_spots"].min()
) / (
    df_parking_index["nb_parking_spots"].max()
    - df_parking_index["nb_parking_spots"].min()
)

return df_parking_index
```

As a first step, the function loads the dataset and sets the index to the IRIS. Then the variables, which will be used in the index, are extracted and copied into a separate dataframe.

Code Example

Calculating Index



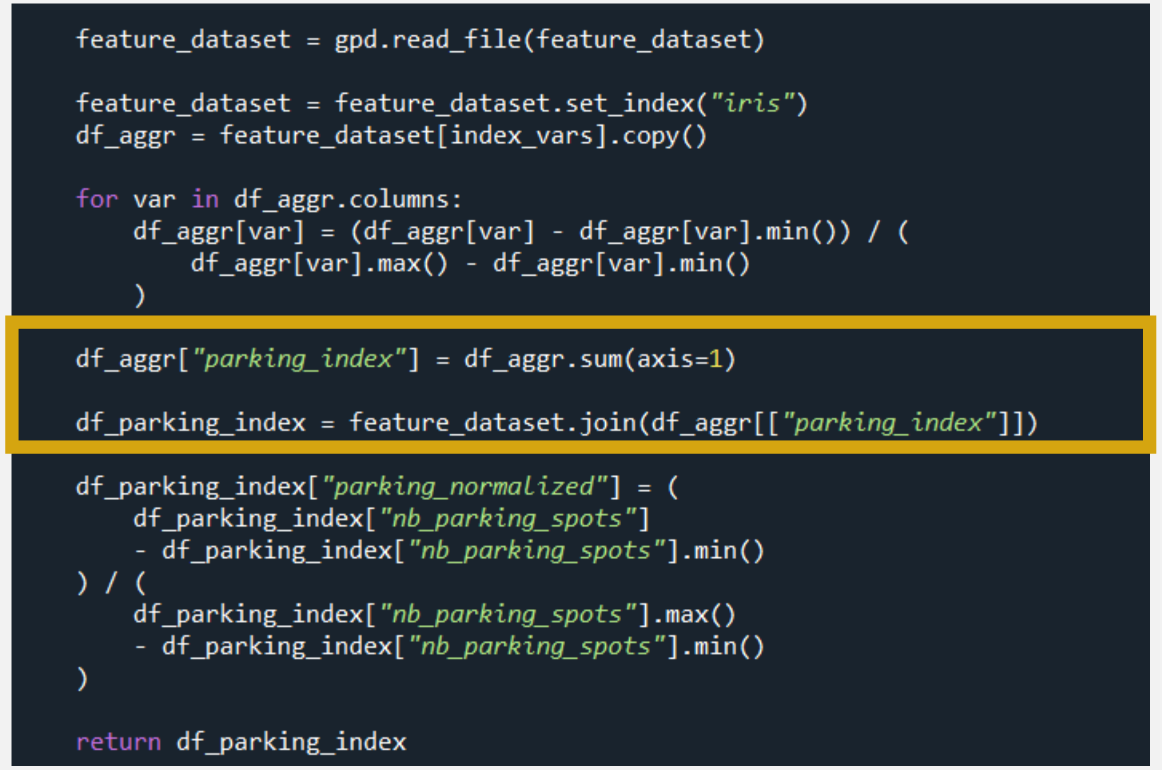
```python
feature_dataset = gpd.read_file(feature_dataset)

feature_dataset = feature_dataset.set_index("iris")
df_aggr = feature_dataset[index_vars].copy()

for var in df_aggr.columns:
    df_aggr[var] = (df_aggr[var] - df_aggr[var].min()) / (
        df_aggr[var].max() - df_aggr[var].min()
    )

df_aggr["parking_index"] = df_aggr.sum(axis=1)

df_parking_index = feature_dataset.join(df_aggr[["parking_index"]])

df_parking_index["parking_normalized"] = (
    df_parking_index["nb_parking_spots"]
    - df_parking_index["nb_parking_spots"].min()
) / (
    df_parking_index["nb_parking_spots"].max()
    - df_parking_index["nb_parking_spots"].min()
)

return df_parking_index
```

To make all the variables, which are using different scales, **comparable**, they are scaled.

The method used here is Min/Max Scaling. As a result, each datapoint gets transformed to be between 0 and 1, while still maintaining the relative distance between them inside the feature and correlations across features.

Code Example

Calculating Index



```
feature_dataset = gpd.read_file(feature_dataset)

feature_dataset = feature_dataset.set_index("iris")
df_aggr = feature_dataset[index_vars].copy()

for var in df_aggr.columns:
    df_aggr[var] = (df_aggr[var] - df_aggr[var].min()) / (
        df_aggr[var].max() - df_aggr[var].min()
    )

df_aggr["parking_index"] = df_aggr.sum(axis=1)

df_parking_index = feature_dataset.join(df_aggr[["parking_index"]])

df_parking_index["parking_normalized"] = (
    df_parking_index["nb_parking_spots"]
    - df_parking_index["nb_parking_spots"].min()
) / (
    df_parking_index["nb_parking_spots"].max()
    - df_parking_index["nb_parking_spots"].min()
)

return df_parking_index
```

All the scaled variables are then summed up to form a complete demand index, which is then added to the original dataframe.

Code Example

Calculating Index

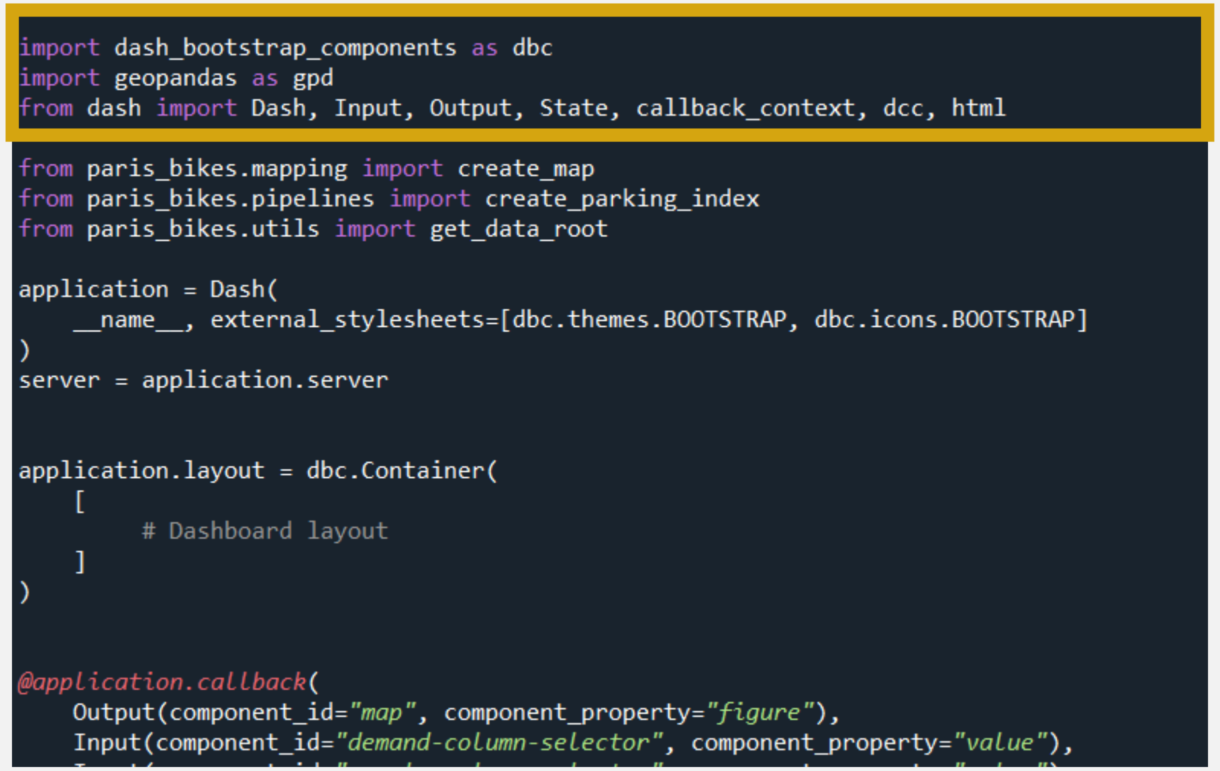
```
feature_dataset = gpd.read_file(feature_dataset)

feature_dataset = feature_dataset.set_index("iris")
df_aggr = feature_dataset[index_vars].copy()

for var in df_aggr.columns:
    df_aggr[var] = (df_aggr[var] - df_aggr[var].min()) / (
        df_aggr[var].max() - df_aggr[var].min()
    )

df_aggr["parking_index"] = df_aggr.sum(axis=1)

df_parking_index = feature_dataset.join(df_aggr[["parking_index"]])

df_parking_index["parking_normalized"] = (
    df_parking_index["nb_parking_spots"]
    - df_parking_index["nb_parking_spots"].min()
) / (
    df_parking_index["nb_parking_spots"].max()
    - df_parking_index["nb_parking_spots"].min()
)

return df_parking_index
```

Finally, the supply metric (how many parking spots are available at each IRIS) is also scaled using the same method as before.

Hochschule Harz
Harz University of Applied Sciences
EPSILON
Funded by the European Union

Code Example

Build a Geo-Dashboard



```python
import dash_bootstrap_components as dbc
import geopandas as gpd
from dash import Dash, Input, Output, State, callback_context, dcc, html
from paris_bikes.mapping import create_map
from paris_bikes.pipelines import create_parking_index
from paris_bikes.utils import get_data_root

application = Dash(
    __name__, external_stylesheets=[dbc.themes.BOOTSTRAP, dbc.icons.BOOTSTRAP]
)
server = application.server


application.layout = dbc.Container(
    [
         # Dashboard layout
    ]
)


@application.callback(
    Output(component_id="map", component_property="figure"),
    Input(component_id="demand-column-selector", component_property="value"),
```

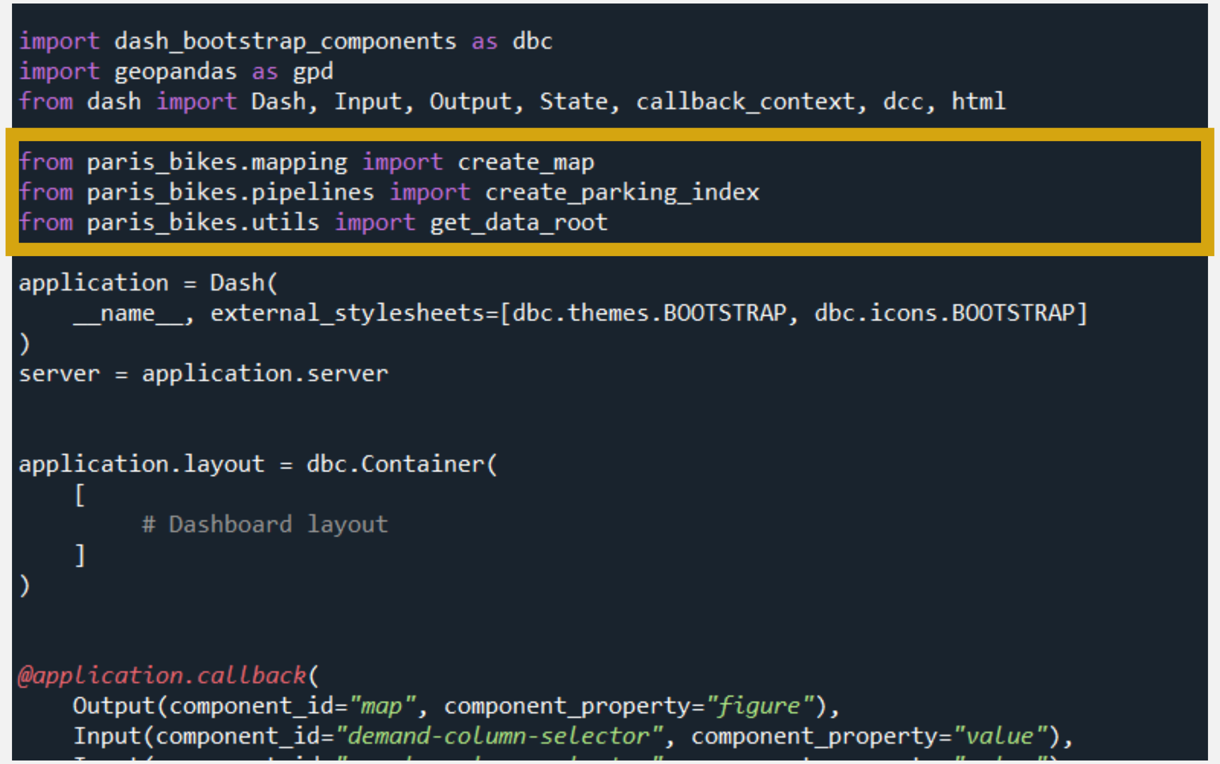
First, the relevant packages are imported.

Besides **geopandas** for data handling, this project uses dash and the dbc package to make dashboard creation as easy as possible.

Hochschule Harz
Harz University of Applied Sciences
EPSILON
Funded by the European Union

Code Example

Build a Geo-Dashboard



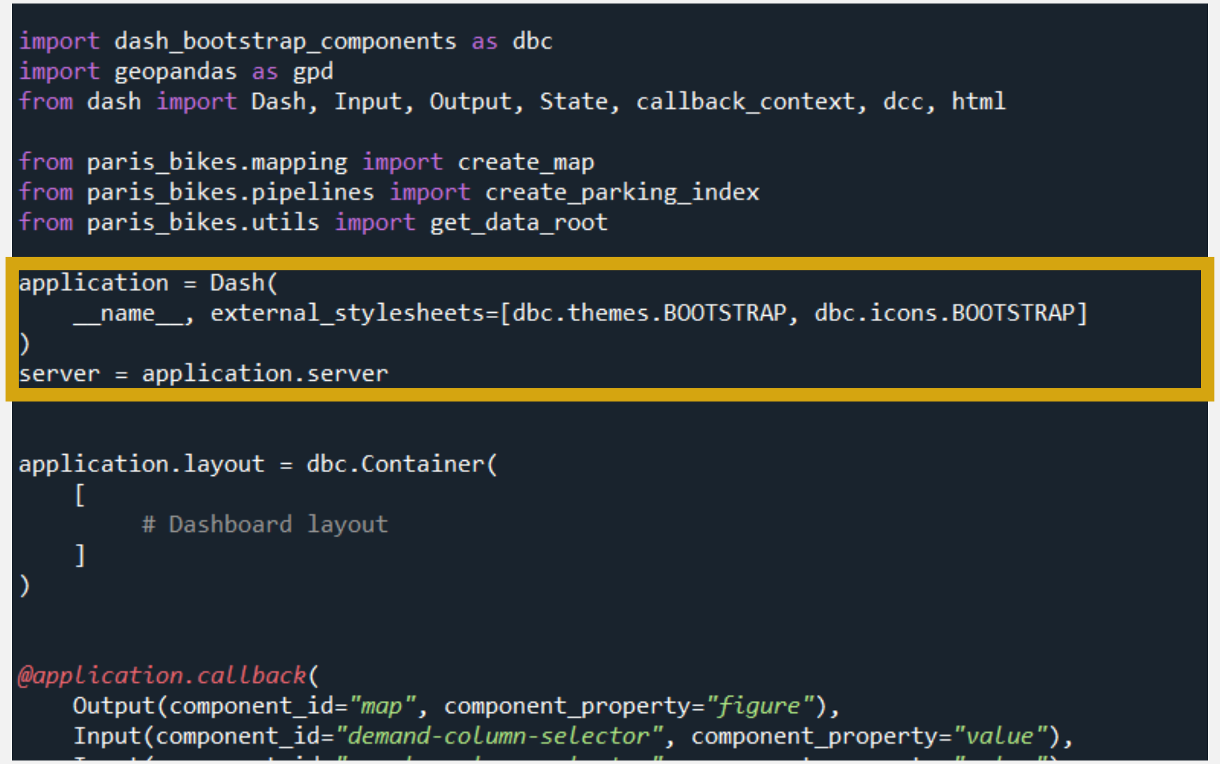
```python
import dash_bootstrap_components as dbc
import geopandas as gpd
from dash import Dash, Input, Output, State, callback_context, dcc, html

from paris_bikes.mapping import create_map
from paris_bikes.pipelines import create_parking_index
from paris_bikes.utils import get_data_root

application = Dash(
    __name__, external_stylesheets=[dbc.themes.BOOTSTRAP, dbc.icons.BOOTSTRAP]
)
server = application.server


application.layout = dbc.Container(
    [
        # Dashboard layout
    ]
)


@application.callback(
    Output(component_id="map", component_property="figure"),
    Input(component_id="demand-column-selector", component_property="value"),
```

To make the code as comprehensible as possible, a module was created to hide specific functionalities.

The relevant functions are then imported into the dashboard code.

Code Example

Build a Geo-Dashboard

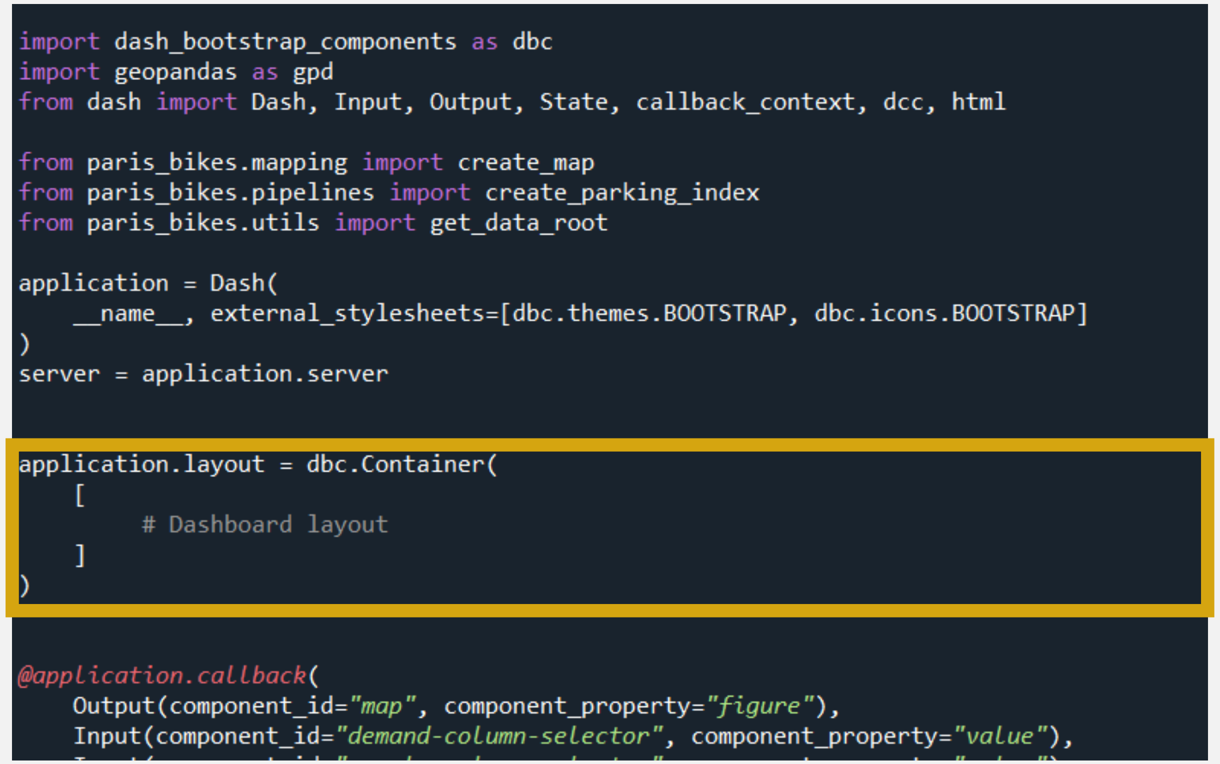


```python
import dash_bootstrap_components as dbc
import geopandas as gpd
from dash import Dash, Input, Output, State, callback_context, dcc, html

from paris_bikes.mapping import create_map
from paris_bikes.pipelines import create_parking_index
from paris_bikes.utils import get_data_root

application = Dash(
    __name__, external_stylesheets=[dbc.themes.BOOTSTRAP, dbc.icons.BOOTSTRAP]
)
server = application.server


application.layout = dbc.Container(
    [
        # Dashboard layout
    ]
)


@application.callback(
    Output(component_id="map", component_property="figure"),
    Input(component_id="demand-column-selector", component_property="value"),
```

Next, the dash app is initialized and the style is specified.

Code Example

Build a Geo-Dashboard

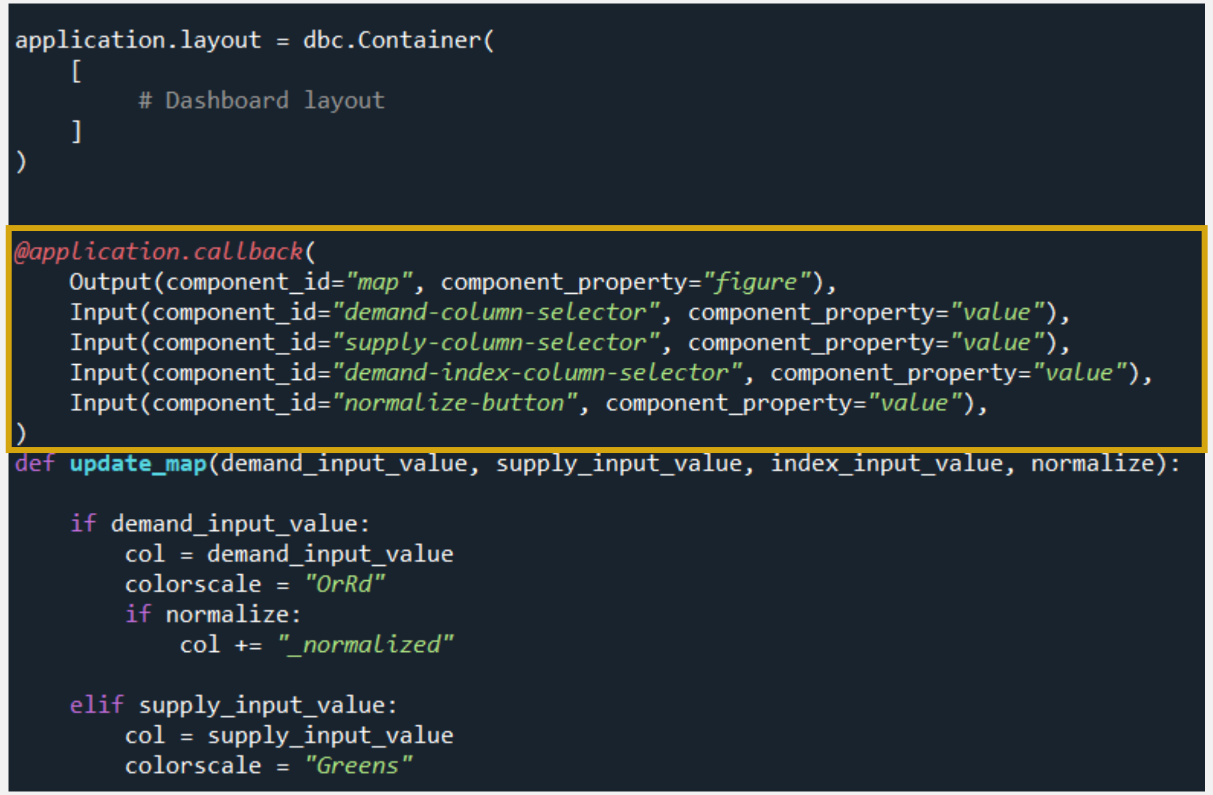


```python
import dash_bootstrap_components as dbc
import geopandas as gpd
from dash import Dash, Input, Output, State, callback_context, dcc, html

from paris_bikes.mapping import create_map
from paris_bikes.pipelines import create_parking_index
from paris_bikes.utils import get_data_root

application = Dash(
    __name__, external_stylesheets=[dbc.themes.BOOTSTRAP, dbc.icons.BOOTSTRAP]
)
server = application.server


application.layout = dbc.Container(
    [
        # Dashboard layout
    ]
)


@application.callback(
    Output(component_id="map", component_property="figure"),
    Input(component_id="demand-column-selector", component_property="value"),
```

In this step, the app layout is defined. To see the full layout process, you can visit the projects' GitHub repository.

Code Example

Build a Geo-Dashboard



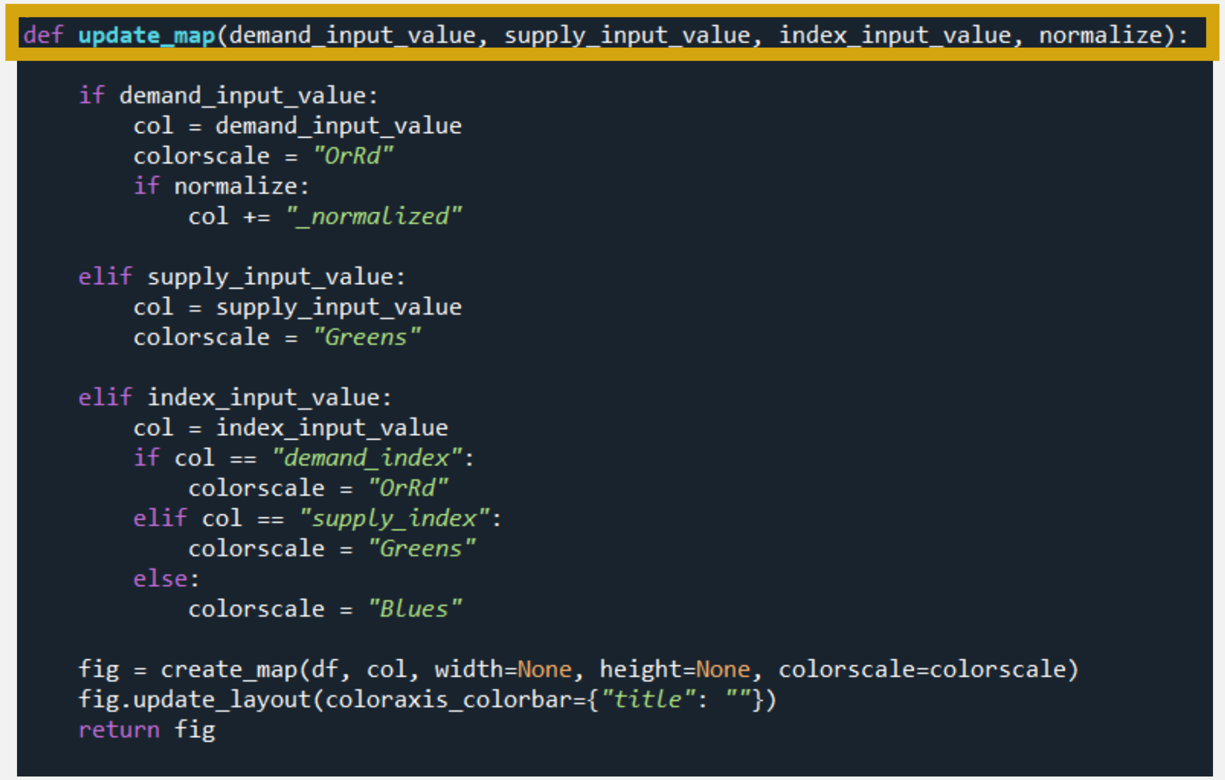
```
application.layout = dbc.Container(
    [
        # Dashboard layout
    ]
)


@application.callback(
    Output(component_id="map", component_property="figure"),
    Input(component_id="demand-column-selector", component_property="value"),
    Input(component_id="supply-column-selector", component_property="value"),
    Input(component_id="demand-index-column-selector", component_property="value"),
    Input(component_id="normalize-button", component_property="value"),
)
def update_map(demand_input_value, supply_input_value, index_input_value, normalize):

    if demand_input_value:
        col = demand_input_value
        colorscale = "OrRd"
        if normalize:
            col += "_normalized"

    elif supply_input_value:
        col = supply_input_value
        colorscale = "Greens"
```

This part of the code defines a callback function in a Dash application. The purpose of this callback is to update the figure of a map component based on user inputs.

The **decorator** ensures that the function will be triggered, if the input components (defined in application.layout) are changed.

Code Example

Build a Geo-Dashboard

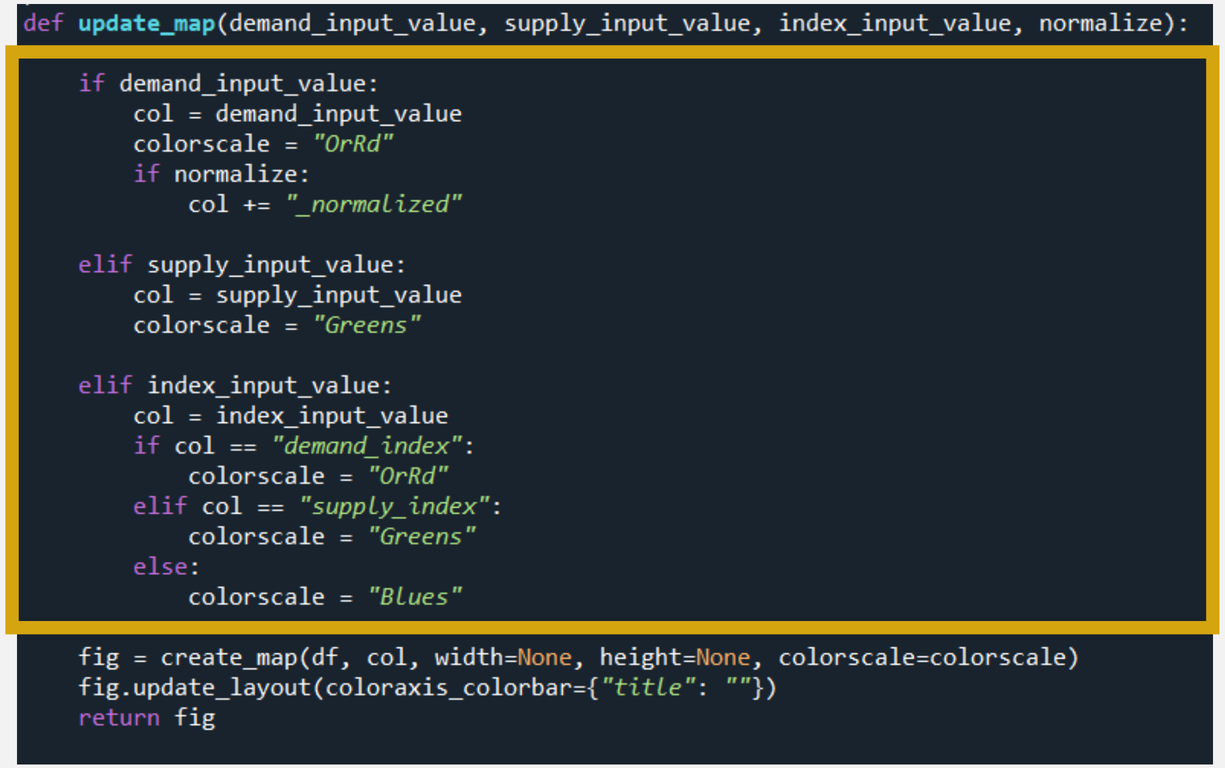


```python
def update_map(demand_input_value, supply_input_value, index_input_value, normalize):

    if demand_input_value:
        col = demand_input_value
        colorscale = "OrRd"
        if normalize:
            col += "_normalized"

    elif supply_input_value:
        col = supply_input_value
        colorscale = "Greens"

    elif index_input_value:
        col = index_input_value
        if col == "demand_index":
            colorscale = "OrRd"
        elif col == "supply_index":
            colorscale = "Greens"
        else:
            colorscale = "Blues"

    fig = create_map(df, col, width=None, height=None, colorscale=colorscale)
    fig.update_layout(coloraxis_colorbar={"title": ""})
    return fig
```

This function is called each time one of the user inputs is changed.

Code Example

Build a Geo-Dashboard



```python
def update_map(demand_input_value, supply_input_value, index_input_value, normalize):

    if demand_input_value:
        col = demand_input_value
        colorscale = "OrRd"
        if normalize:
            col += "_normalized"

    elif supply_input_value:
        col = supply_input_value
        colorscale = "Greens"

    elif index_input_value:
        col = index_input_value
        if col == "demand_index":
            colorscale = "OrRd"
        elif col == "supply_index":
            colorscale = "Greens"
        else:
            colorscale = "Blues"

    fig = create_map(df, col, width=None, height=None, colorscale=colorscale)
    fig.update_layout(coloraxis_colorbar={"title": ""})
    return fig
```

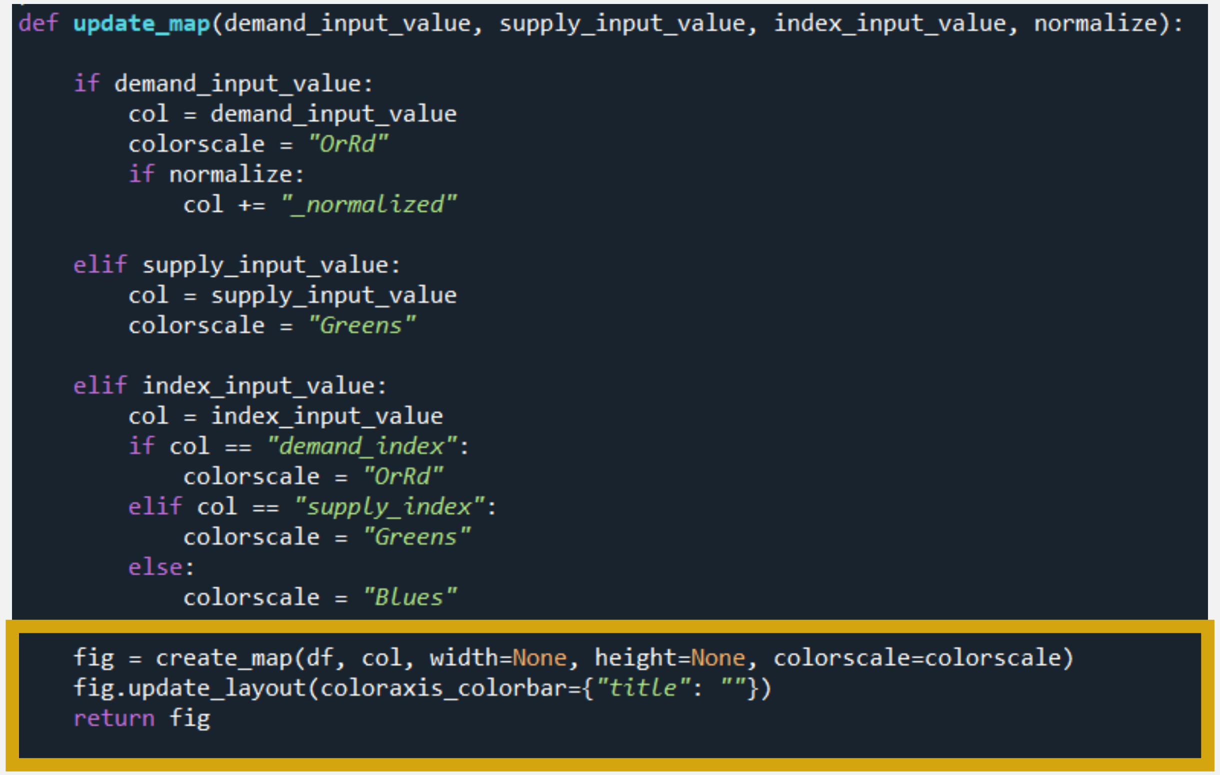
Depending on the user's input, a different color scale is used.

- **Demand variables** are displayed in **Orange and Red**
- **Supply variables** are displayed in **Green**
- The **calculated index** is displayed in **Blue**

The code **dynamically selects** the map display column from a prepared DataFrame, accommodating both raw and normalized data. If users opt for normalized data, the variable name is adjusted to access the corresponding "variable_normalized".

Code Example

Build a Geo-Dashboard



```python
def update_map(demand_input_value, supply_input_value, index_input_value, normalize):

    if demand_input_value:
        col = demand_input_value
        colorscale = "OrRd"
        if normalize:
            col += "_normalized"

    elif supply_input_value:
        col = supply_input_value
        colorscale = "Greens"

    elif index_input_value:
        col = index_input_value
        if col == "demand_index":
            colorscale = "OrRd"
        elif col == "supply_index":
            colorscale = "Greens"
        else:
            colorscale = "Blues"

    fig = create_map(df, col, width=None, height=None, colorscale=colorscale)
    fig.update_layout(coloraxis_colorbar={"title": ""})
    return fig
```

Finally, the function creates the map using the user input. The figure is then automatically updated.

Code Example

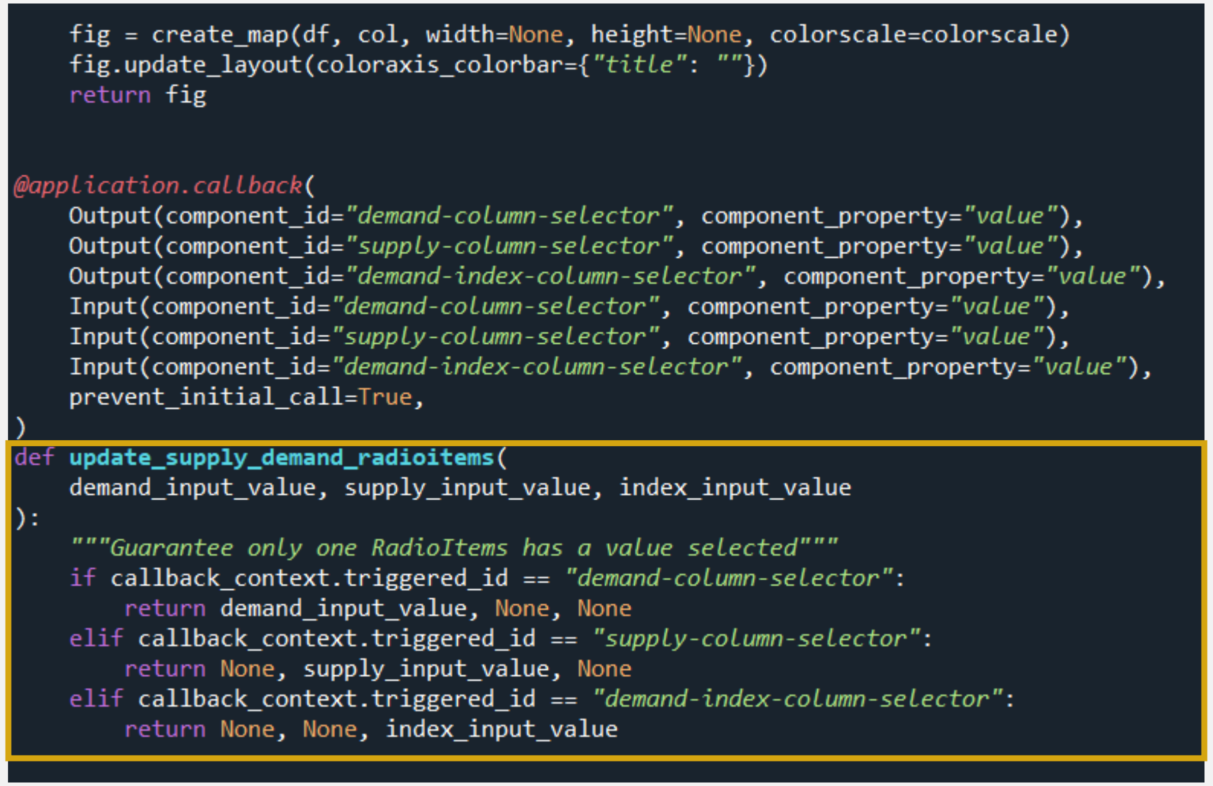
Build a Geo-Dashboard

```python
    fig = create_map(df, col, width=None, height=None, colorscale=colorscale)
    fig.update_layout(coloraxis_colorbar={"title": ""})
    return fig


@application.callback(
    Output(component_id="demand-column-selector", component_property="value"),
    Output(component_id="supply-column-selector", component_property="value"),
    Output(component_id="demand-index-column-selector", component_property="value"),
    Input(component_id="demand-column-selector", component_property="value"),
    Input(component_id="supply-column-selector", component_property="value"),
    Input(component_id="demand-index-column-selector", component_property="value"),
    prevent_initial_call=True,
)
def update_supply_demand_radioitems(
    demand_input_value, supply_input_value, index_input_value
):
    """Guarantee only one RadioItems has a value selected"""
    if callback_context.triggered_id == "demand-column-selector":
        return demand_input_value, None, None
    elif callback_context.triggered_id == "supply-column-selector":
        return None, supply_input_value, None
    elif callback_context.triggered_id == "demand-index-column-selector":
        return None, None, index_input_value
```

This is a decorator that tells Dash to treat the following function ('update_supply_demand_radioitems') as another callback function.

Code Example

Build a Geo-Dashboard



```python
    fig = create_map(df, col, width=None, height=None, colorscale=colorscale)
    fig.update_layout(coloraxis_colorbar={"title": ""})
    return fig


@application.callback(
    Output(component_id="demand-column-selector", component_property="value"),
    Output(component_id="supply-column-selector", component_property="value"),
    Output(component_id="demand-index-column-selector", component_property="value"),
    Input(component_id="demand-column-selector", component_property="value"),
    Input(component_id="supply-column-selector", component_property="value"),
    Input(component_id="demand-index-column-selector", component_property="value"),
    prevent_initial_call=True,
)
def update_supply_demand_radioitems(
    demand_input_value, supply_input_value, index_input_value
):
    """Guarantee only one RadioItems has a value selected"""
    if callback_context.triggered_id == "demand-column-selector":
        return demand_input_value, None, None
    elif callback_context.triggered_id == "supply-column-selector":
        return None, supply_input_value, None
    elif callback_context.triggered_id == "demand-index-column-selector":
        return None, None, index_input_value
```

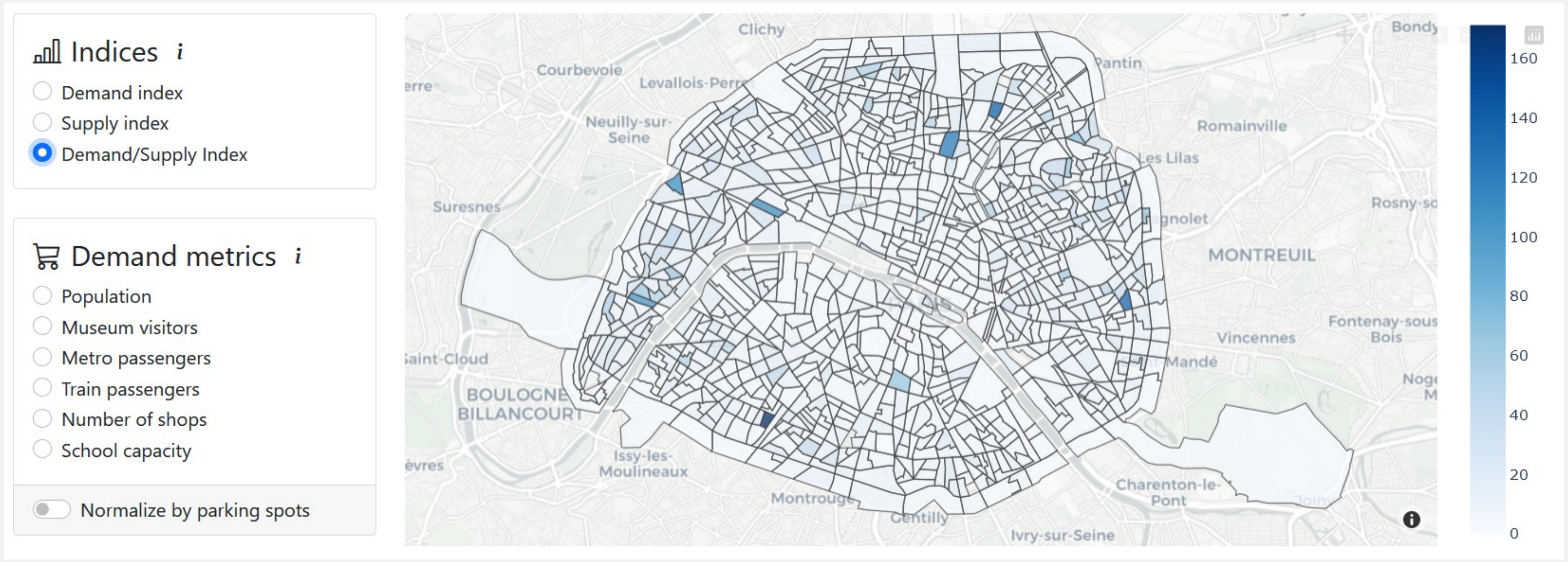
The function itself sets each value **besides the current user selection** to “none”.

This ensures that only **one** variable is displayed on the map.

Where to build new bicycle parking spots in Paris?

Solution – Project Deliverables

1) Demo of a Dashboard displaying the various Supply/Demand Metrics
2) Publicly available code repository on GitHub


Indices

- Demand index
- Supply index
- [✓ Demand/Supply Index]

Demand metrics

- Population
- Museum visitors
- Metro passengers
- Train passengers
- Number of shops
- School capacity

Normalize by parking spots

This dashboard provides an easy visual access to otherwise complex data

Predicting Long-Term Unemployment in Portugal

Predicting Long-Term Unemployment in Portugal

Project Team:

Data Science for Social Good Portugal (DSSG)

Data Science Knowledge Center @ Nova SBE

3 Data Scientists, 1 Technical Mentor, 1 Project Manager

Stakeholder:

IEFP, the institute of employment and vocational training in Portugal

Problem Statement:

The partner organization operates over 80 job centers nationwide. Within these centers, job counselors are assisting job seekers by suggesting interventions such as training courses and aiding in the job application process. Before this project, counselors were drafting personal action plans, picking from a multitude of available interventions.

This project had two main goals:

1. Improve on the process of identifying people at high risk of becoming long-term unemployed
2. Develop an intervention recommender system for job counselors, providing personalized recommendations to individual job seekers taking into account their personal profile

Detailed Information

DFG organization:
DSSG and Data Science Knowledge Center at NovaSBE

Partner type:
Governmental agency

Partner name:
IEFP

Sustainable Development Goal (SDG):
8

Type of interaction:
Short-term project

Type of analytics:
Modeling

Type of data:
Government data

All data is available here

Internal & External Data

The IEFP provided the project team with data on people registered in their unemployment programs.

This data consists of 12 years of transactional data with currently 3.1 million registered individuals.

Features:

- ►Demographic information
- ►Professional background
- ►Past history with IEFP
- ►Trainings attended, by type and outcome
- ►Job offers replied to, by outcome
- ►Numbers of times summoned by IEFP, by outcome

Predicting Long-Term Unemployment in Portugal

Methods used for data processing

1) Classification

2) Recommender System

1) Classification

Objective: Identify high-risk individuals for long-term unemployment

Method: Machine learning on historical data

Features Analyzed: Education, employment history, skills, demographics

Outcome: Proactively identify and prioritize high-risk individuals for job counseling

2) Recommender System

Objective: Improve suggested interventions using machine learning

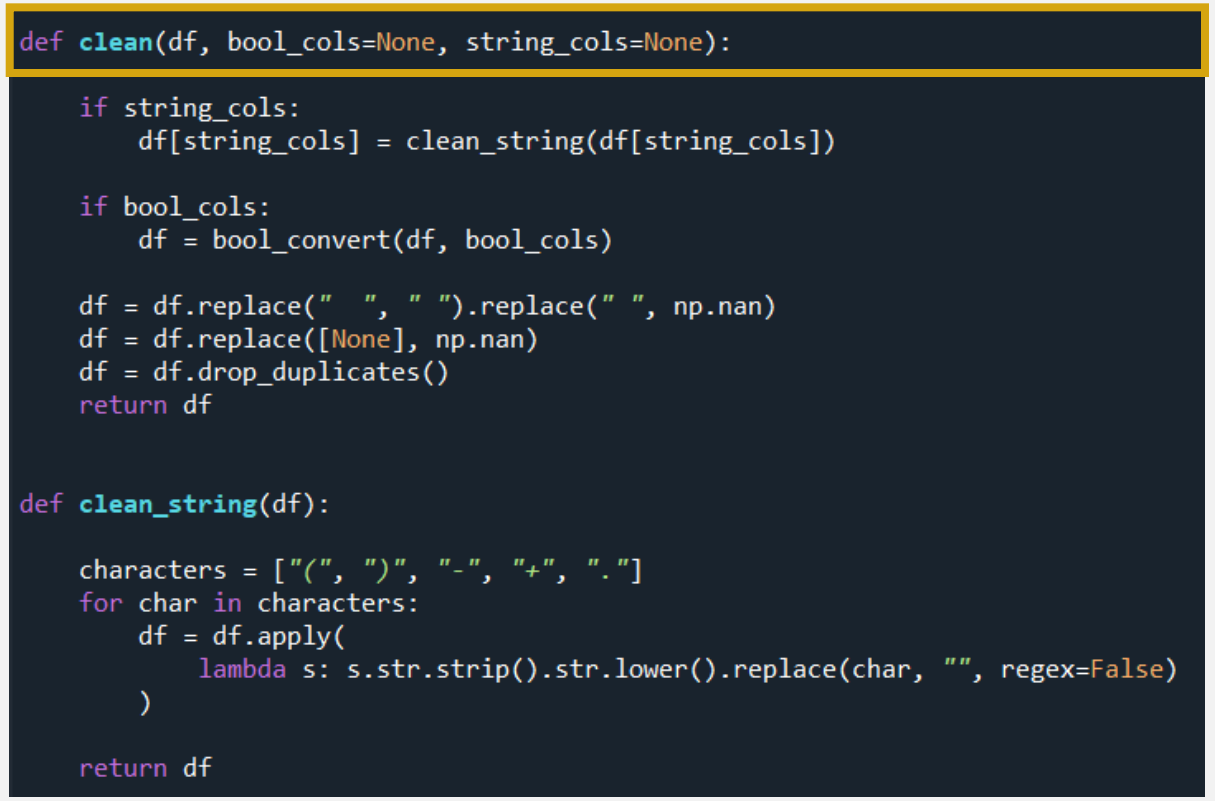
Method: Predict success probability of interventions using client history

Features Analyzed: Predicted success vs. average success of a reference group (e.g. age)

Outcome: Provide error measure to aid counselors in drafting intervention plans

Code Example

Data Cleaning



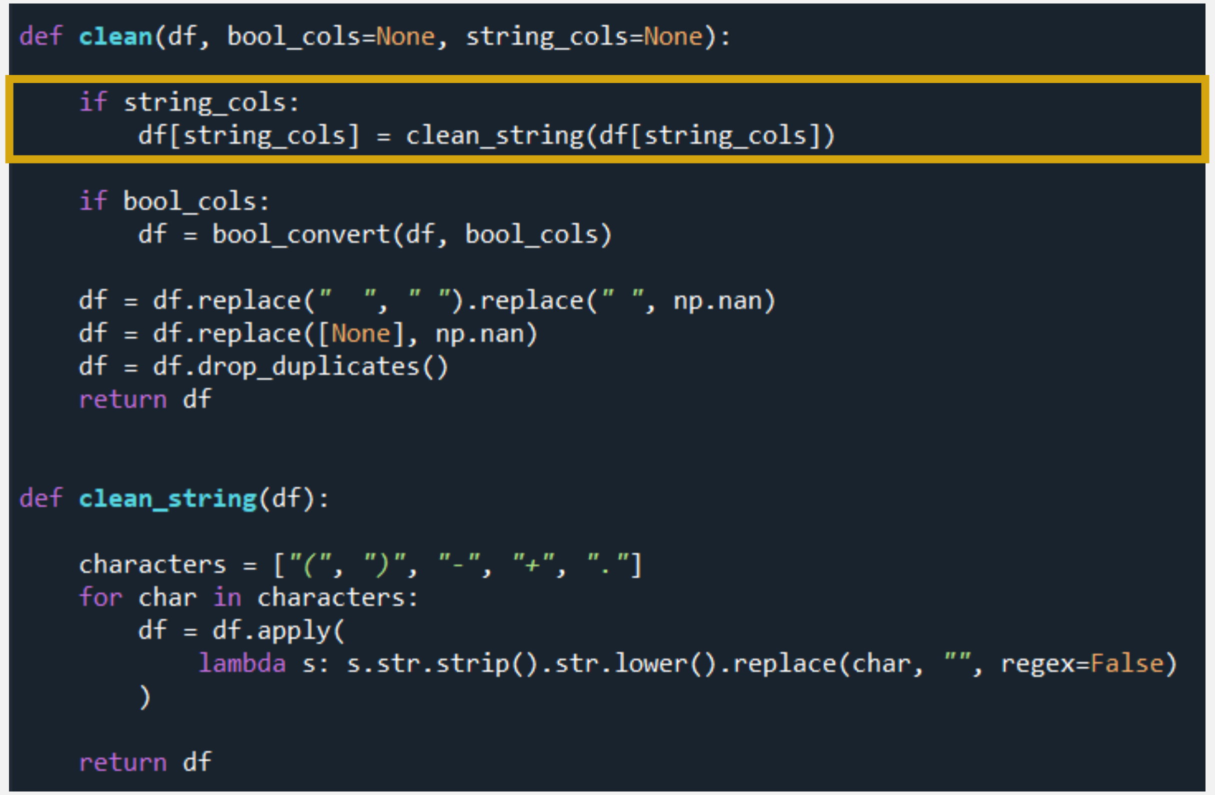
```
def clean(df, bool_cols=None, string_cols=None):

    if string_cols:
        df[string_cols] = clean_string(df[string_cols])

    if bool_cols:
        df = bool_convert(df, bool_cols)

    df = df.replace("  ", " ").replace(" ", np.nan)
    df = df.replace([None], np.nan)
    df = df.drop_duplicates()
    return df


def clean_string(df):

    characters = ["(", ")", "-", "+", "."]
    for char in characters:
        df = df.apply(
            lambda s: s.str.strip().str.lower().replace(char, "", regex=False)
        )

    return df
```

To use the provided data in modeling, it needs to be brought into a consistent format first.

First, the function “**clean**” is defined, taking a dataframe and two lists as input. The lists contain column names.

Code Example

Data Cleaning

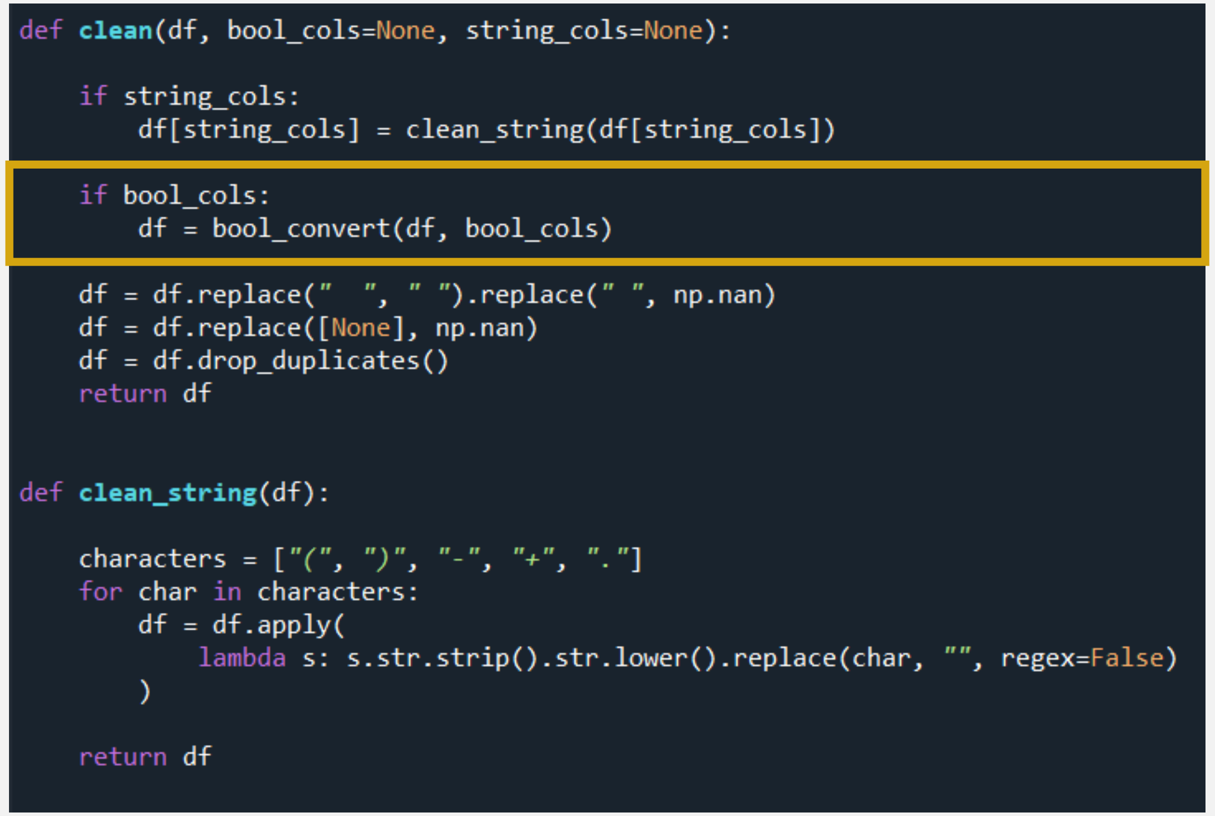


```
def clean(df, bool_cols=None, string_cols=None):

    if string_cols:
        df[string_cols] = clean_string(df[string_cols])

    if bool_cols:
        df = bool_convert(df, bool_cols)

    df = df.replace("  ", " ").replace(" ", np.nan)
    df = df.replace([None], np.nan)
    df = df.drop_duplicates()
    return df


def clean_string(df):

    characters = ["(", ")", "-", "+", "."]
    for char in characters:
        df = df.apply(
            lambda s: s.str.strip().str.lower().replace(char, "", regex=False)
        )

    return df
```

If the list “**string_cols**” is provided, the subset of the complete dataframe containing the named columns is passed to the function “**clean_string**”.

Code Example

Data Cleaning

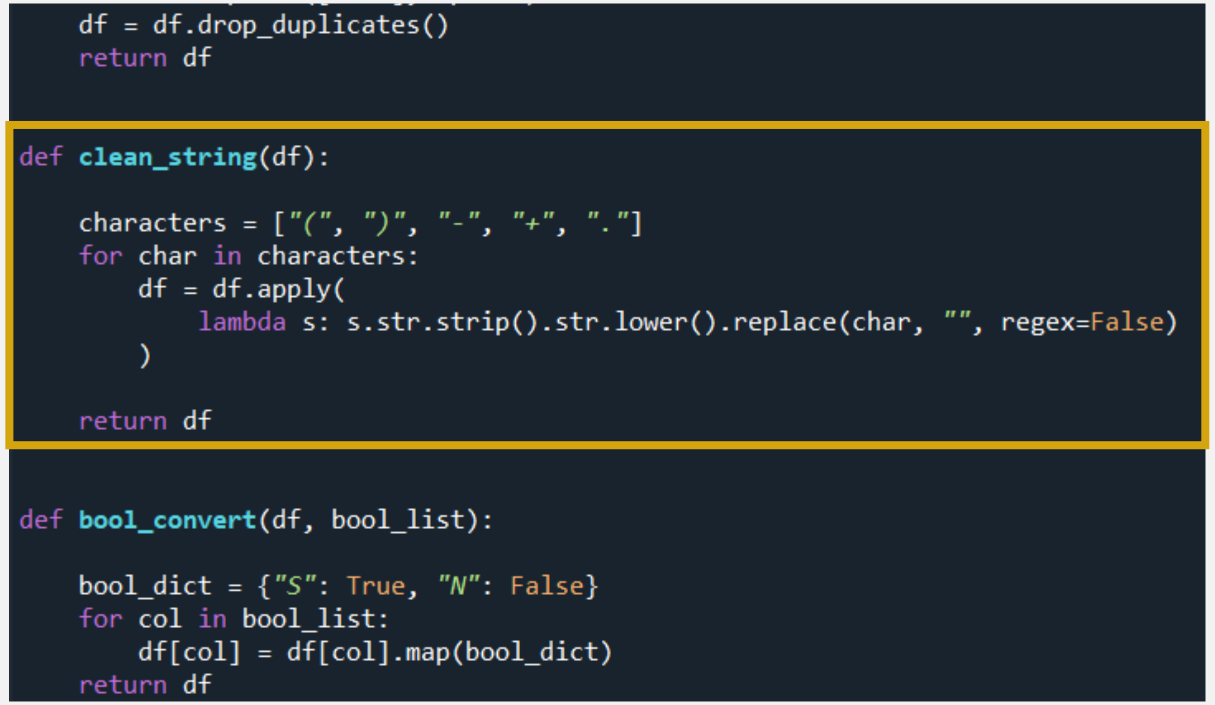


```python
def clean(df, bool_cols=None, string_cols=None):

    if string_cols:
        df[string_cols] = clean_string(df[string_cols])

    if bool_cols:
        df = bool_convert(df, bool_cols)

    df = df.replace("  ", " ").replace(" ", np.nan)
    df = df.replace([None], np.nan)
    df = df.drop_duplicates()
    return df


def clean_string(df):

    characters = ["(", ")", "-", "+", "."]
    for char in characters:
        df = df.apply(
            lambda s: s.str.strip().str.lower().replace(char, "", regex=False)
        )

    return df
```

If the list "**bool_cols**" is provided, the subset of the complete dataframe containing the named columns is passed to the function "**bool_converter**".

Code Example

Data Cleaning



```
    df = df.drop_duplicates()
    return df


def clean_string(df):

    characters = ["(", ")", "-", "+", "."]
    for char in characters:
        df = df.apply(
            lambda s: s.str.strip().str.lower().replace(char, "", regex=False)
        )

    return df


def bool_convert(df, bool_list):

    bool_dict = {"S": True, "N": False}
    for col in bool_list:
        df[col] = df[col].map(bool_dict)
    return df
```

The function “**clean_string**“ reformats all columns containing strings by applying the methods:

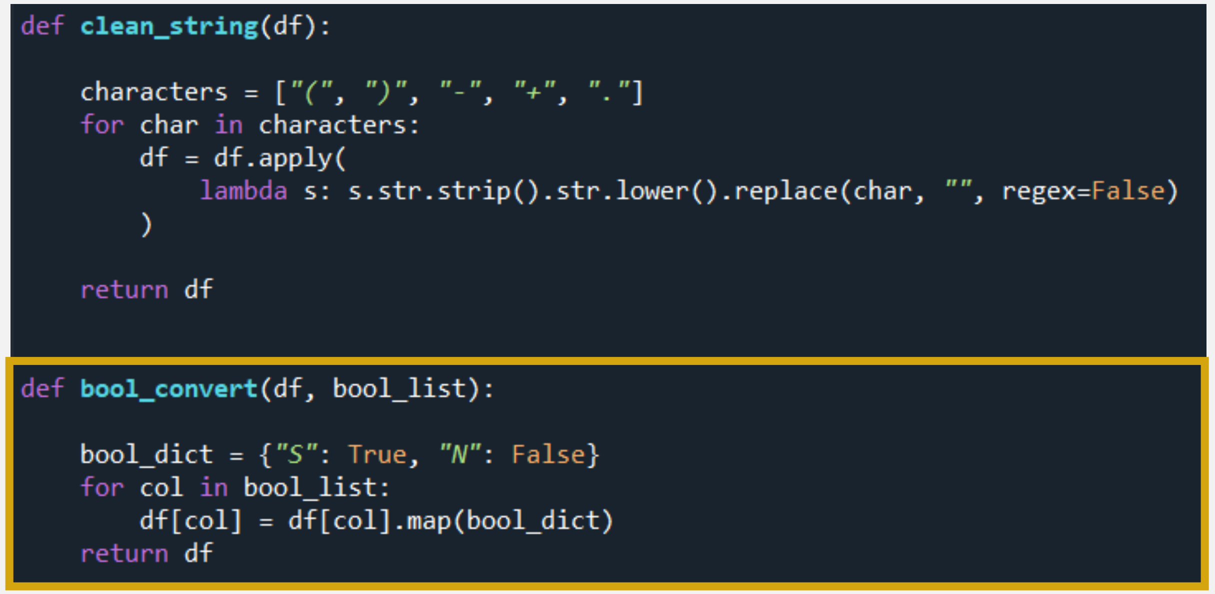
- “**strip**“ removes white spaces before and after the string
- “**lower**“ switches uppercase characters to lowercase
- “**replace**“ removes the specified characters

Then, the transformed columns are returned.

Funded by the European Union

Code Example

Data Cleaning



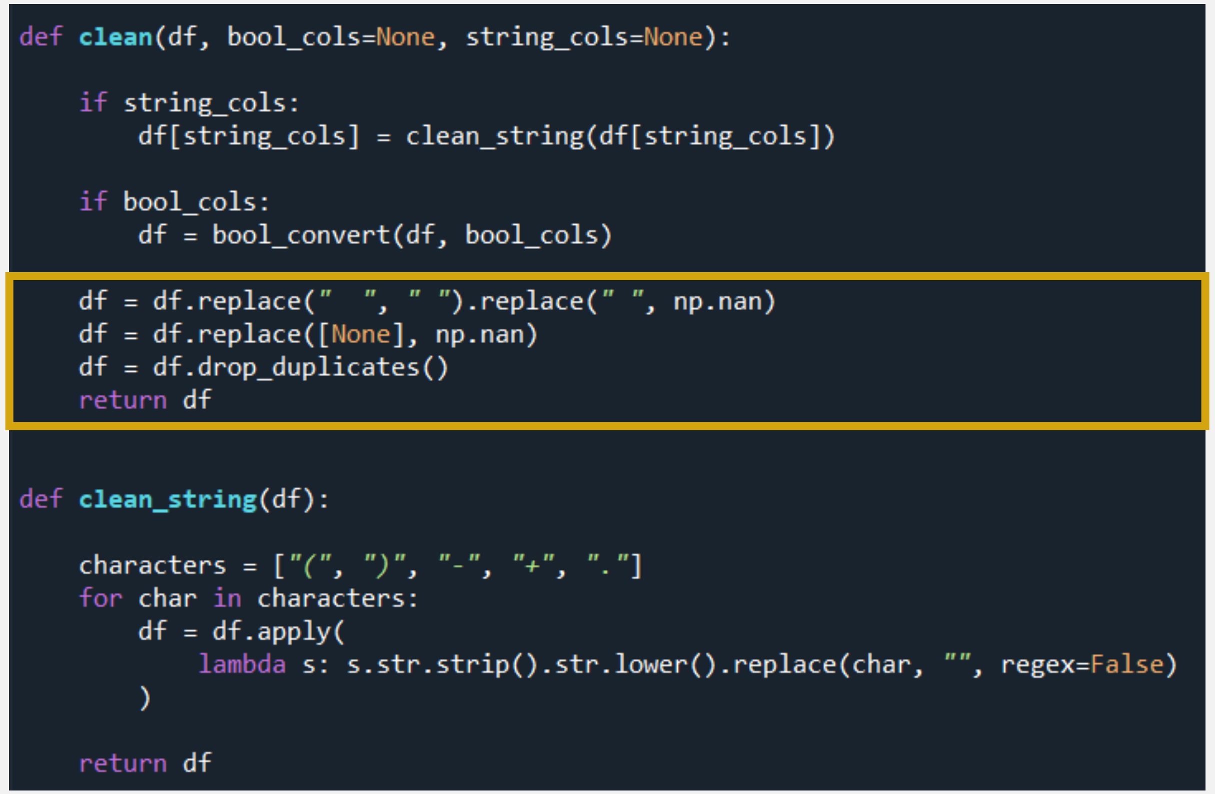
```python
def clean_string(df):

    characters = ["(", ")", "-", "+", "."]
    for char in characters:
        df = df.apply(
            lambda s: s.str.strip().str.lower().replace(char, "", regex=False)
        )

    return df

def bool_convert(df, bool_list):

    bool_dict = {"S": True, "N": False}
    for col in bool_list:
        df[col] = df[col].map(bool_dict)
    return df
```

The provided data contained the letters **“S”** for “sim/yes” and **“N”** for “não/no”.

The function “bool_converter” transforms those characters into boolean values **“True/False”**.

Code Example

Data Cleaning



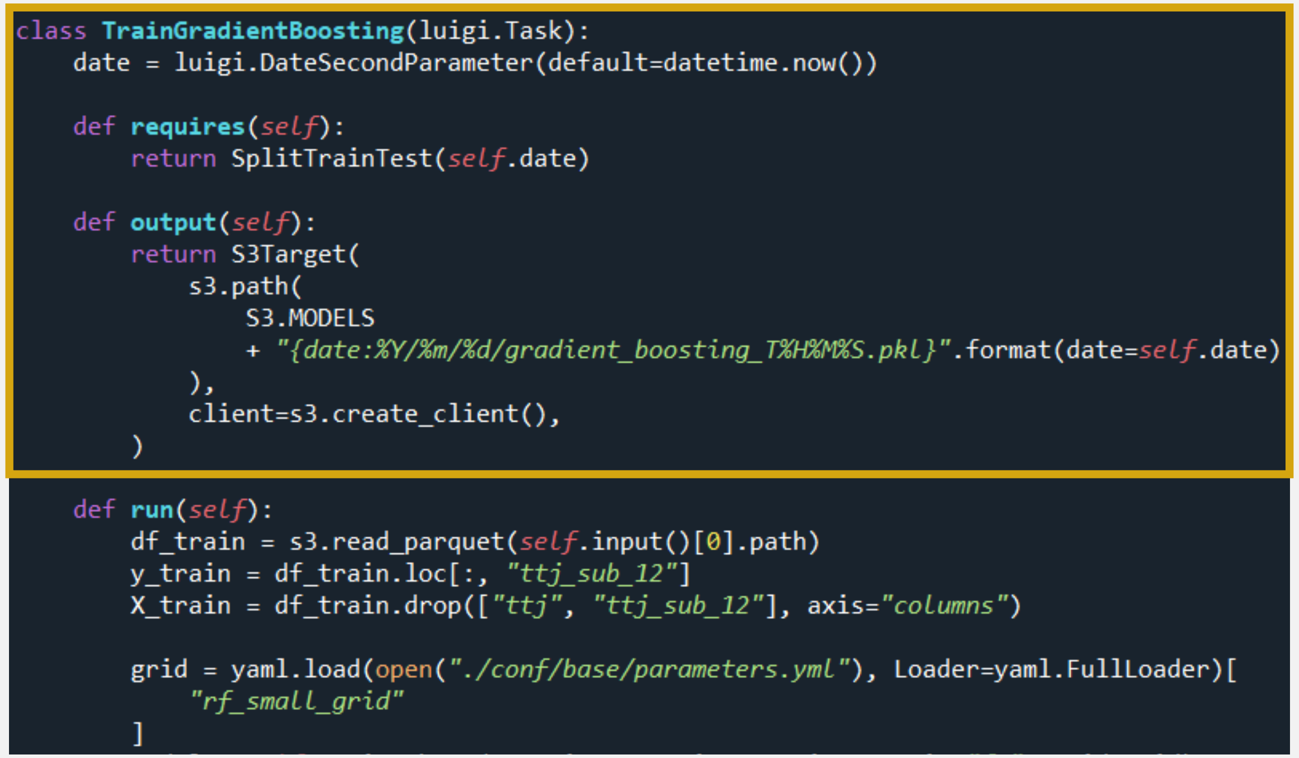
```python
def clean(df, bool_cols=None, string_cols=None):

    if string_cols:
        df[string_cols] = clean_string(df[string_cols])

    if bool_cols:
        df = bool_convert(df, bool_cols)

    df = df.replace("  ", " ").replace(" ", np.nan)
    df = df.replace([None], np.nan)
    df = df.drop_duplicates()
    return df


def clean_string(df):

    characters = ["(", ")", "-", "+", "."]
    for char in characters:
        df = df.apply(
            lambda s: s.str.strip().str.lower().replace(char, "", regex=False)
        )

    return df
```

Lastly, all empty values are replaced with the **NumPy** “not a number” value.

Duplicate rows are then dropped and the finished dataframe is returned.

Code Example

Classification



```
class TrainGradientBoosting(luigi.Task):
    date = luigi.DateSecondParameter(default=datetime.now())

    def requires(self):
        return SplitTrainTest(self.date)

    def output(self):
        return S3Target(
            s3.path(
                S3.MODELS
                + "{date:%Y/%m/%d/gradient_boosting_T%H%M%S.pkl}".format(date=self.date)
            ),
            client=s3.create_client(),
        )

    def run(self):
        df_train = s3.read_parquet(self.input()[0].path)
        y_train = df_train.loc[:, "ttj_sub_12"]
        X_train = df_train.drop(["ttj", "ttj_sub_12"], axis="columns")

        grid = yaml.load(open("./conf/base/parameters.yml"), Loader=yaml.FullLoader)[
            "rf_small_grid"
        ]
```

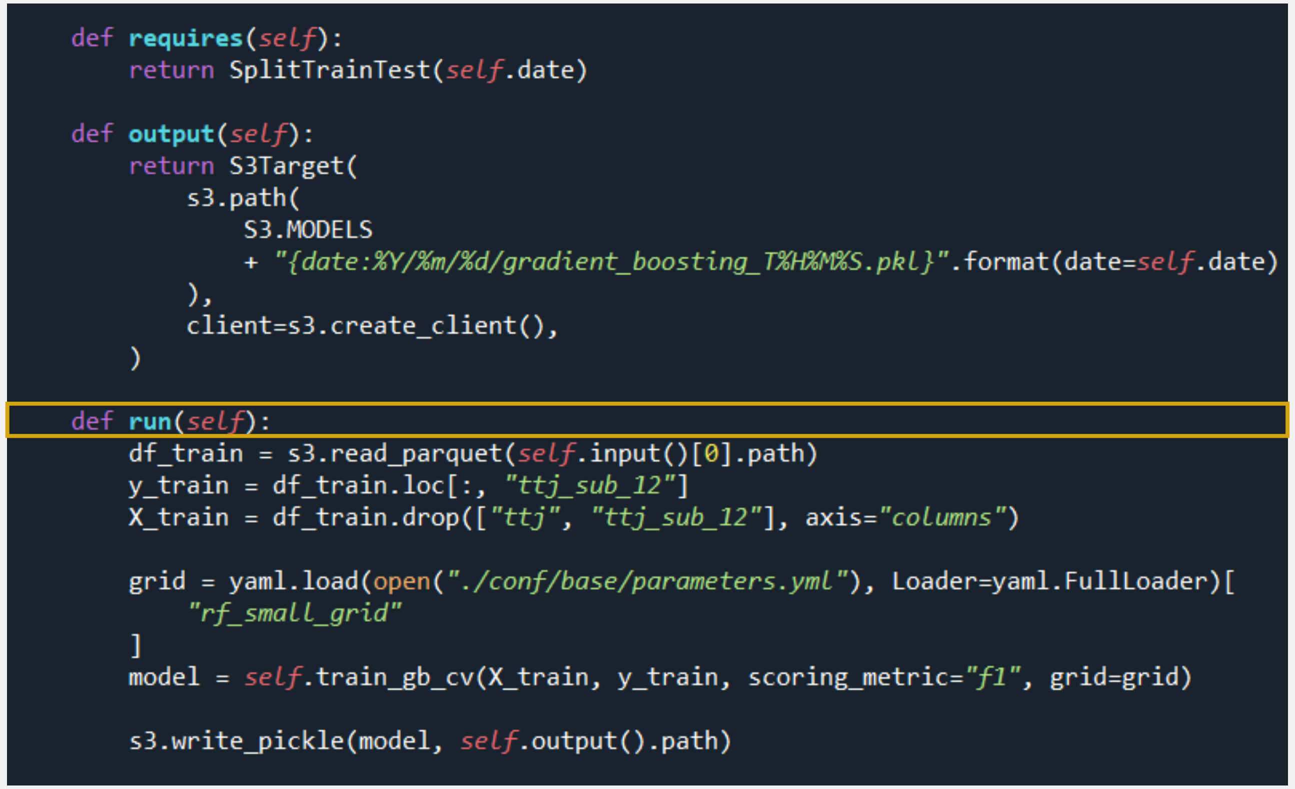
This class inherits from the **"luigi-Task"** class.

"requires" is a function defined to signal which tasks are required to run before this task. In this case, a "Train-Test Split" has to be done before running the training step.

"output" defines where the task's output will be saved.

Code Example

Classification

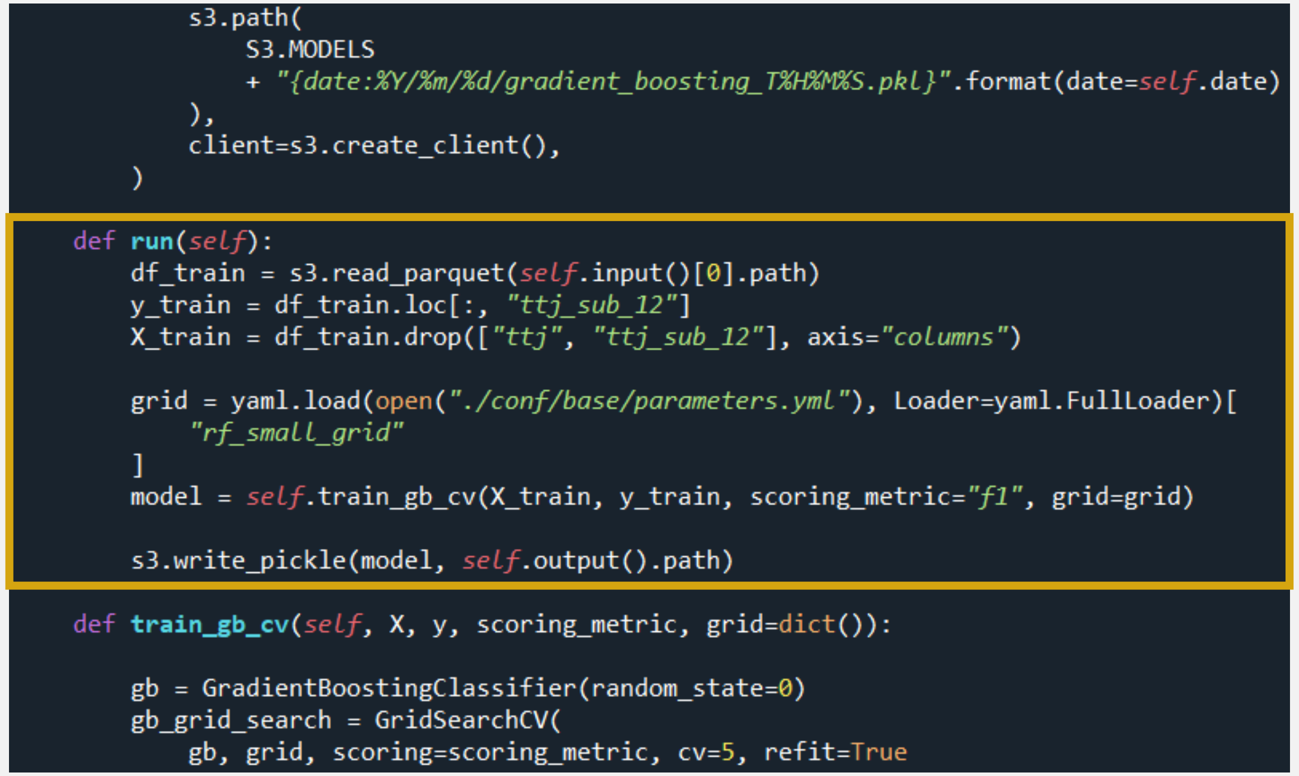


```
    def requires(self):
        return SplitTrainTest(self.date)

    def output(self):
        return S3Target(
            s3.path(
                S3.MODELS
                + "{date:%Y/%m/%d/gradient_boosting_T%H%M%S.pkl}".format(date=self.date)
            ),
            client=s3.create_client(),
        )

    def run(self):
        df_train = s3.read_parquet(self.input()[0].path)
        y_train = df_train.loc[:, "ttj_sub_12"]
        X_train = df_train.drop(["ttj", "ttj_sub_12"], axis="columns")

        grid = yaml.load(open("./conf/base/parameters.yml"), Loader=yaml.FullLoader)[
            "rf_small_grid"
        ]
        model = self.train_gb_cv(X_train, y_train, scoring_metric="f1", grid=grid)

        s3.write_pickle(model, self.output().path)
```

The function “**run**” is the actual logic of the task. As soon as the task scheduler running in the background starts this task, the code in “**run**” is executed.

Code Example

Classification



```python
            s3.path(
                S3.MODELS
                + "{date:%Y/%m/%d/gradient_boosting_T%H%M%S.pkl}".format(date=self.date)
            ),
            client=s3.create_client(),
        )

    def run(self):
        df_train = s3.read_parquet(self.input()[0].path)
        y_train = df_train.loc[:, "ttj_sub_12"]
        X_train = df_train.drop(["ttj", "ttj_sub_12"], axis="columns")

        grid = yaml.load(open("./conf/base/parameters.yml"), Loader=yaml.FullLoader)[
            "rf_small_grid"
        ]
        model = self.train_gb_cv(X_train, y_train, scoring_metric="f1", grid=grid)

        s3.write_pickle(model, self.output().path)

    def train_gb_cv(self, X, y, scoring_metric, grid=dict()):

        gb = GradientBoostingClassifier(random_state=0)
        gb_grid_search = GridSearchCV(
            gb, grid, scoring=scoring_metric, cv=5, refit=True
```

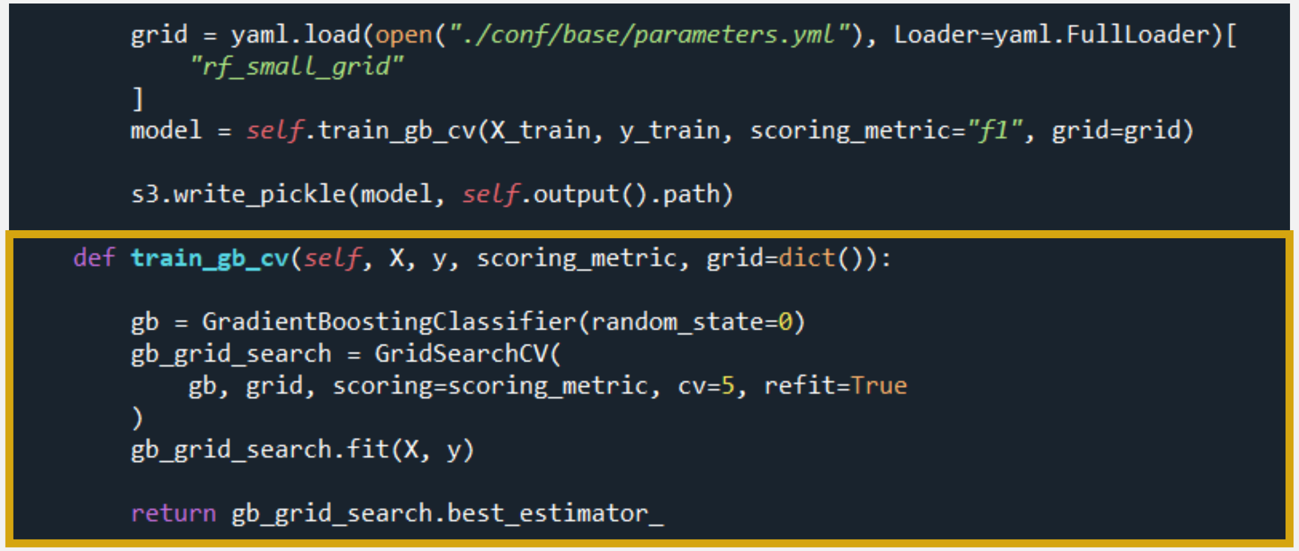
In this case, data is loaded and then the labels and features are saved in separate variables **“y_train”** and **“X_train”**.

In the variable **“grid”** previously defined parameters for a grid search are loaded. A grid search is a way to find optimal hyperparameters.

Then a function is called training a “GradientBoostingClassifier” utilizing a grid search and the model with the highest f1-score is saved at the output path.

Code Example

Classification



```
        grid = yaml.load(open("./conf/base/parameters.yml"), Loader=yaml.FullLoader)[
            "rf_small_grid"
        ]
        model = self.train_gb_cv(X_train, y_train, scoring_metric="f1", grid=grid)

        s3.write_pickle(model, self.output().path)

    def train_gb_cv(self, X, y, scoring_metric, grid=dict()):

        gb = GradientBoostingClassifier(random_state=0)
        gb_grid_search = GridSearchCV(
            gb, grid, scoring=scoring_metric, cv=5, refit=True
        )
        gb_grid_search.fit(X, y)

        return gb_grid_search.best_estimator_
```

The training function itself uses the **scikit-learn** package to fit a GradientBoostingClassifier.

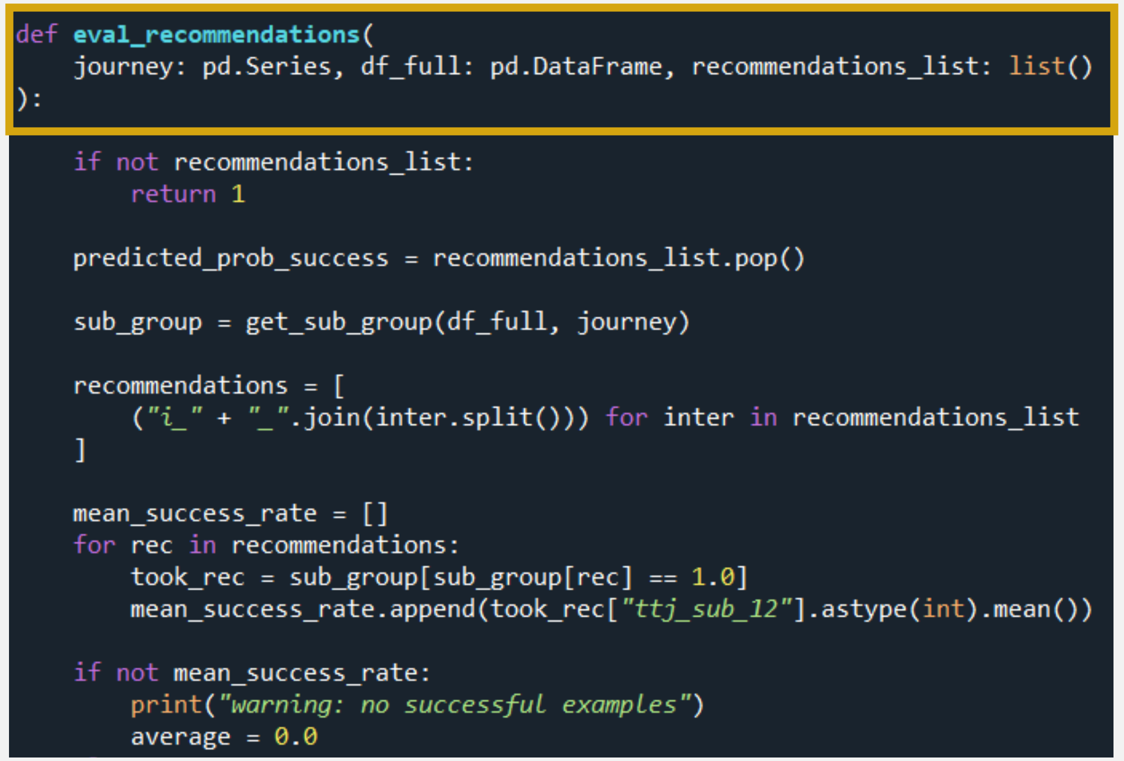
First, the Classifier and the Grid Search objects are initialized.

Then the **“fit”** method is called, which performs the grid search.

Lastly, the model with the highest f1-score is returned.

Code Example

Recommender Error Calculation



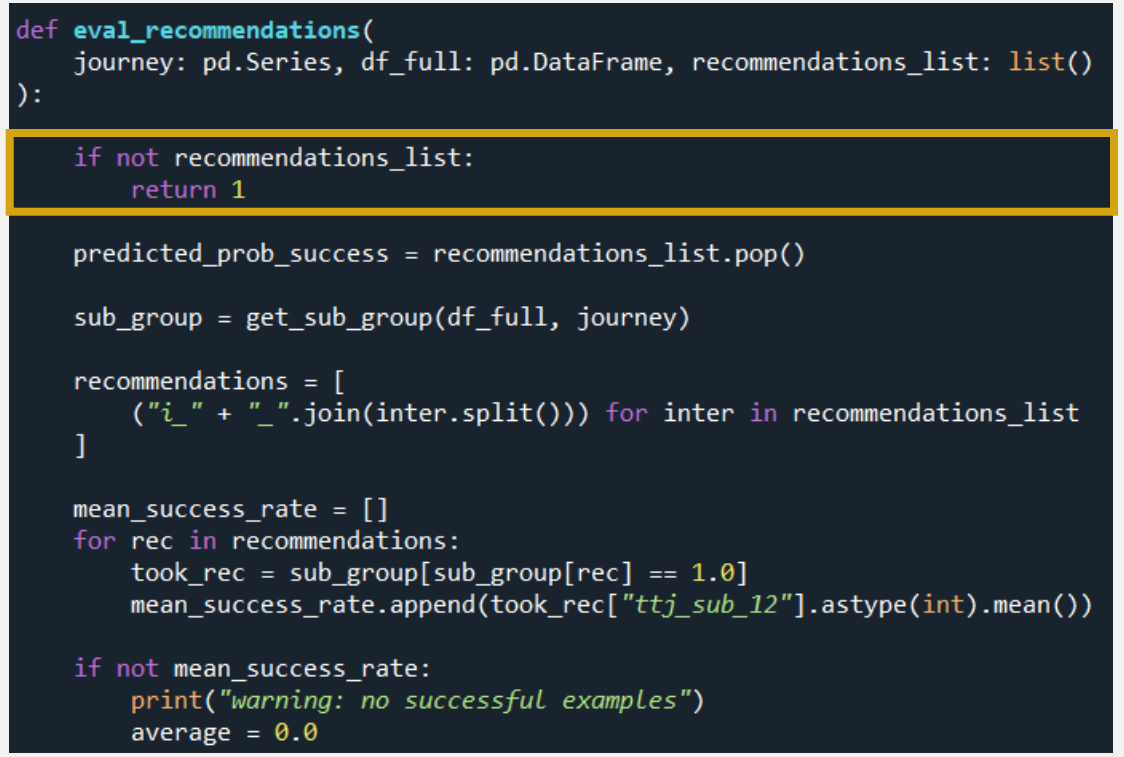
```python
def eval_recommendations(
    journey: pd.Series, df_full: pd.DataFrame, recommendations_list: list()
):

    if not recommendations_list:
        return 1

    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
```

After recommendations are given, it is important to estimate how certainly this recommendation is correct. To achieve this, this estimated success probability is **compared** to a reference group of people in the data set.

As an input, the function expects the journey for which the recommendations were calculated, the full remaining data set and the calculated recommendations.

Code Example

Recommender Error Calculation

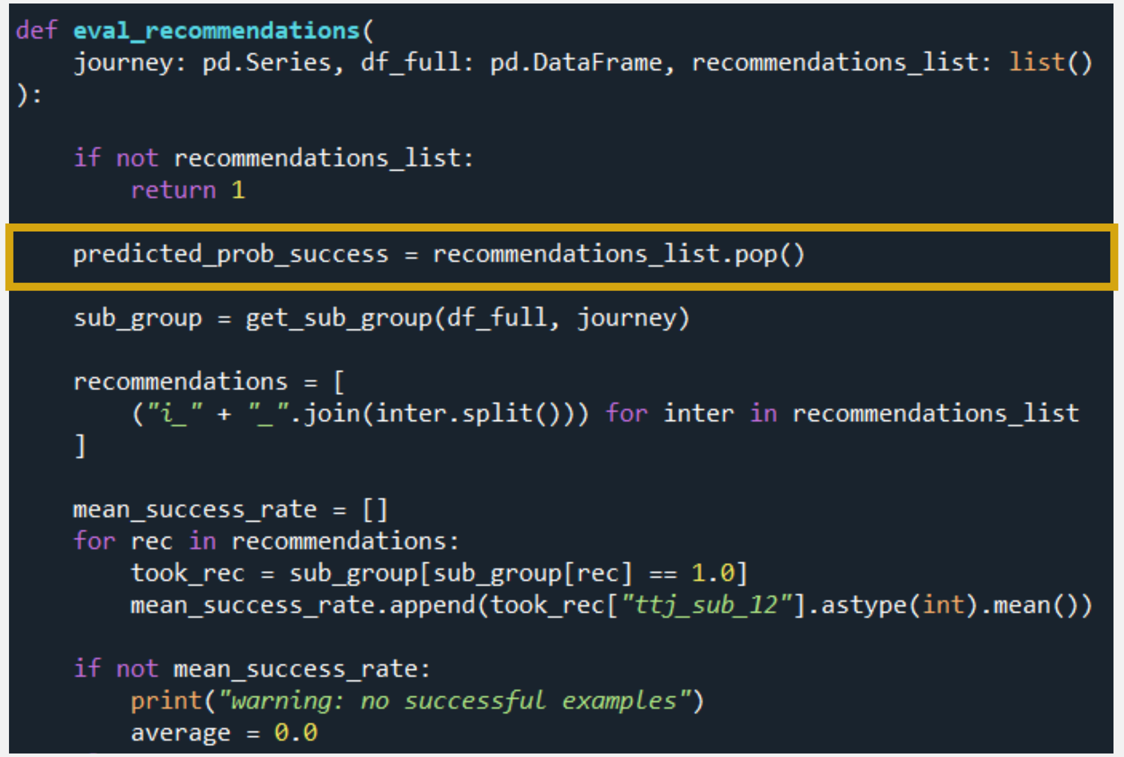


```python
def eval_recommendations(
    journey: pd.Series, df_full: pd.DataFrame, recommendations_list: list()
):

    if not recommendations_list:
        return 1

    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
```

If **no** recommendations were passed, the maximum error value of 1 is returned.

Code Example

Recommender Error Calculation

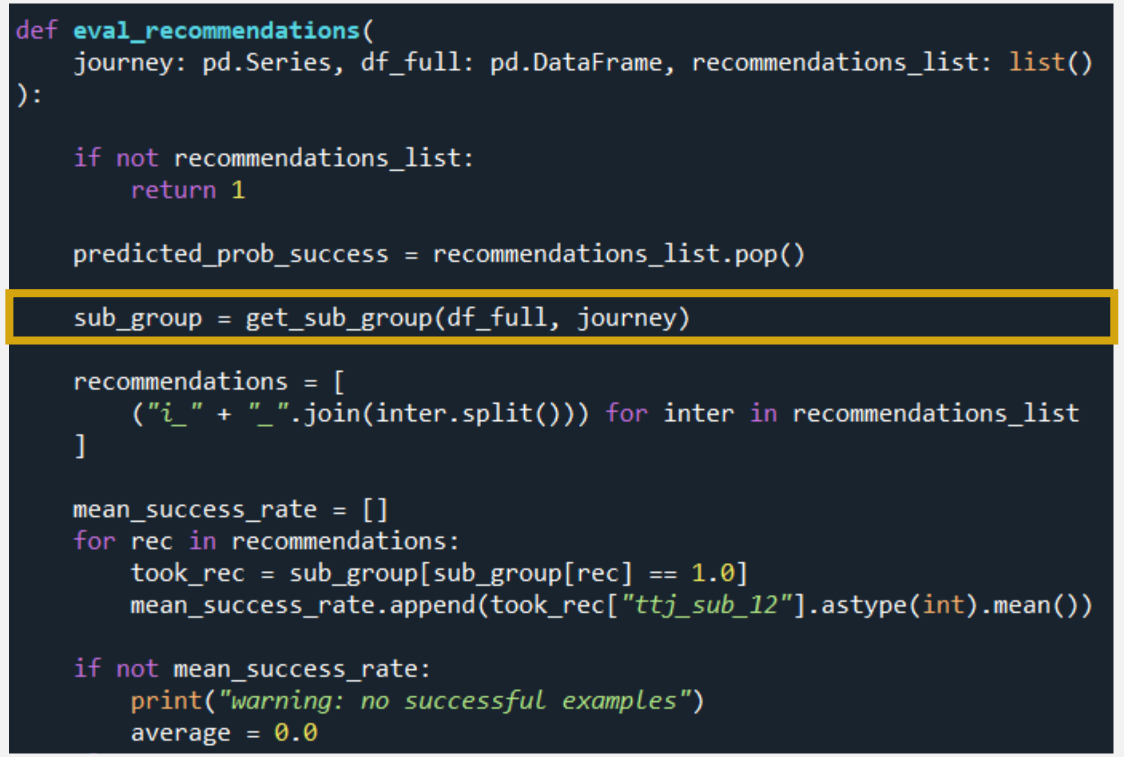


```python
def eval_recommendations(
    journey: pd.Series, df_full: pd.DataFrame, recommendations_list: list()
):

    if not recommendations_list:
        return 1

    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
```

The predicted probability is the last item of the passed list, so we can use **.pop()** to remove it from the list and store it in a separate variable.

Code Example

Recommender Error Calculation

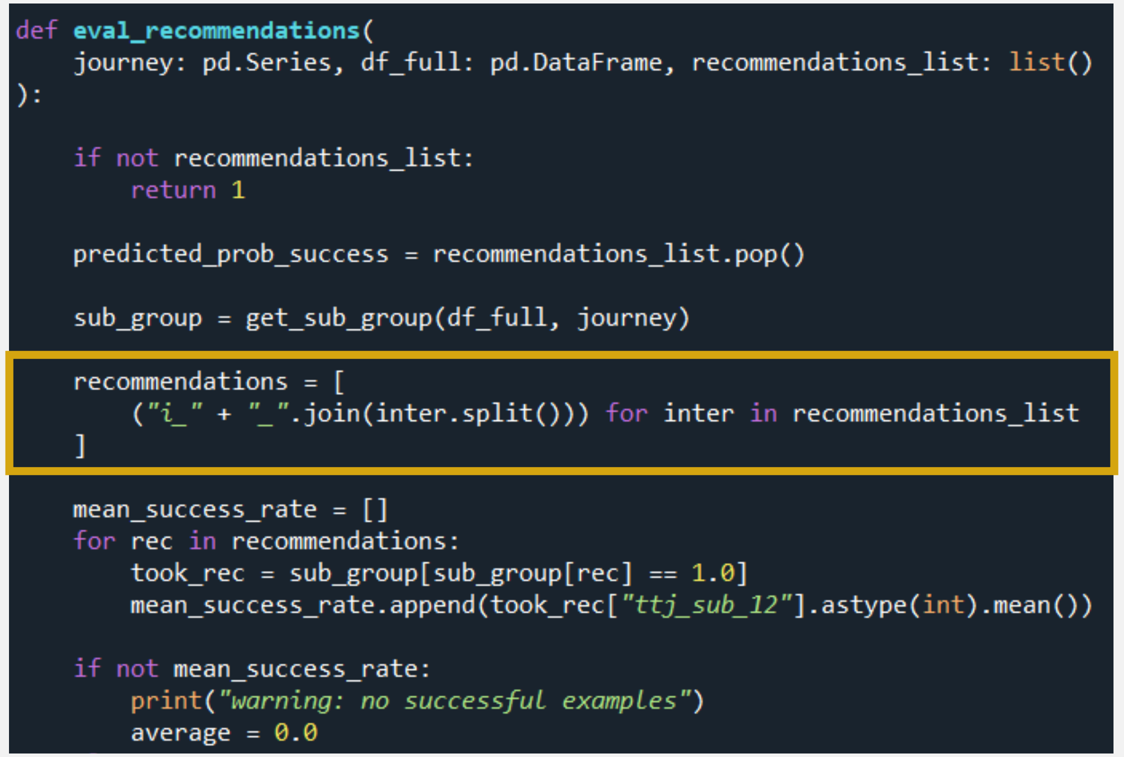


```
def eval_recommendations(
    journey: pd.Series, df_full: pd.DataFrame, recommendations_list: list()
):

    if not recommendations_list:
        return 1

    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
```

Then, a subset of other persons is selected from the full data set.

Code Example

Recommender Error Calculation

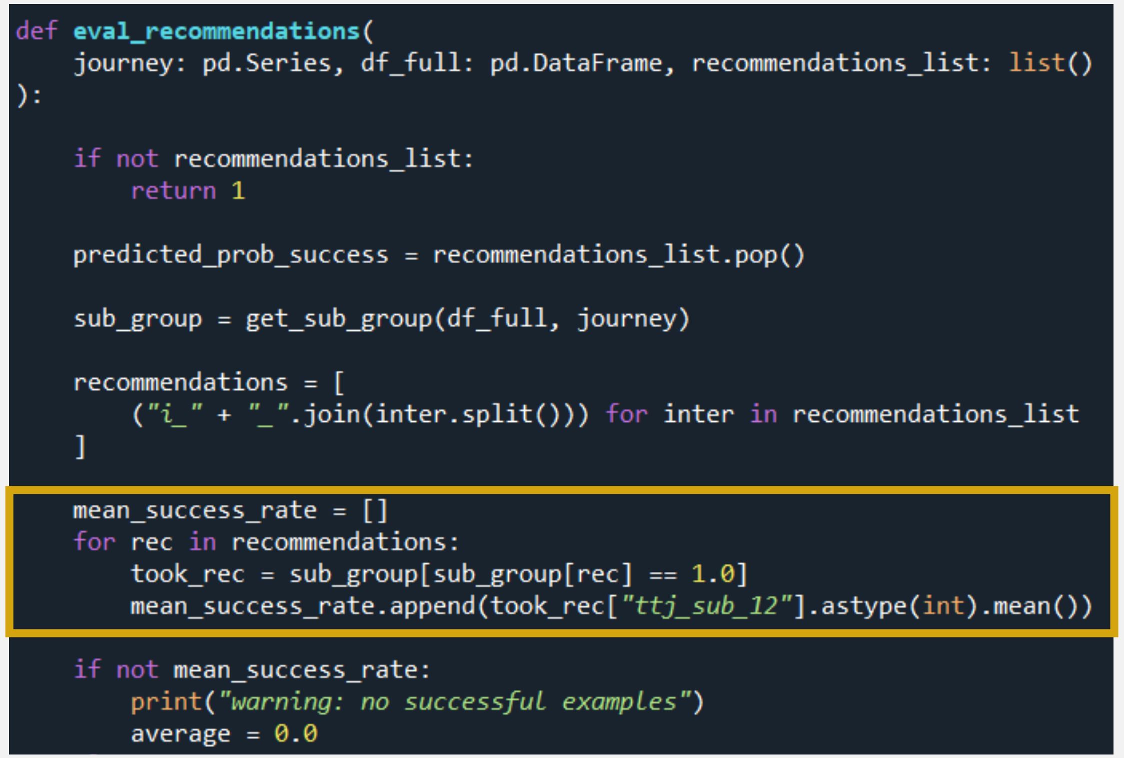


```python
def eval_recommendations(
    journey: pd.Series, df_full: pd.DataFrame, recommendations_list: list()
):

    if not recommendations_list:
        return 1

    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
```

Then, the recommendations in the list are transformed to match the variable names in the data set.

Code Example

Recommender Error Calculation



```python
def eval_recommendations(
    journey: pd.Series, df_full: pd.DataFrame, recommendations_list: list()
):

    if not recommendations_list:
        return 1

    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
```

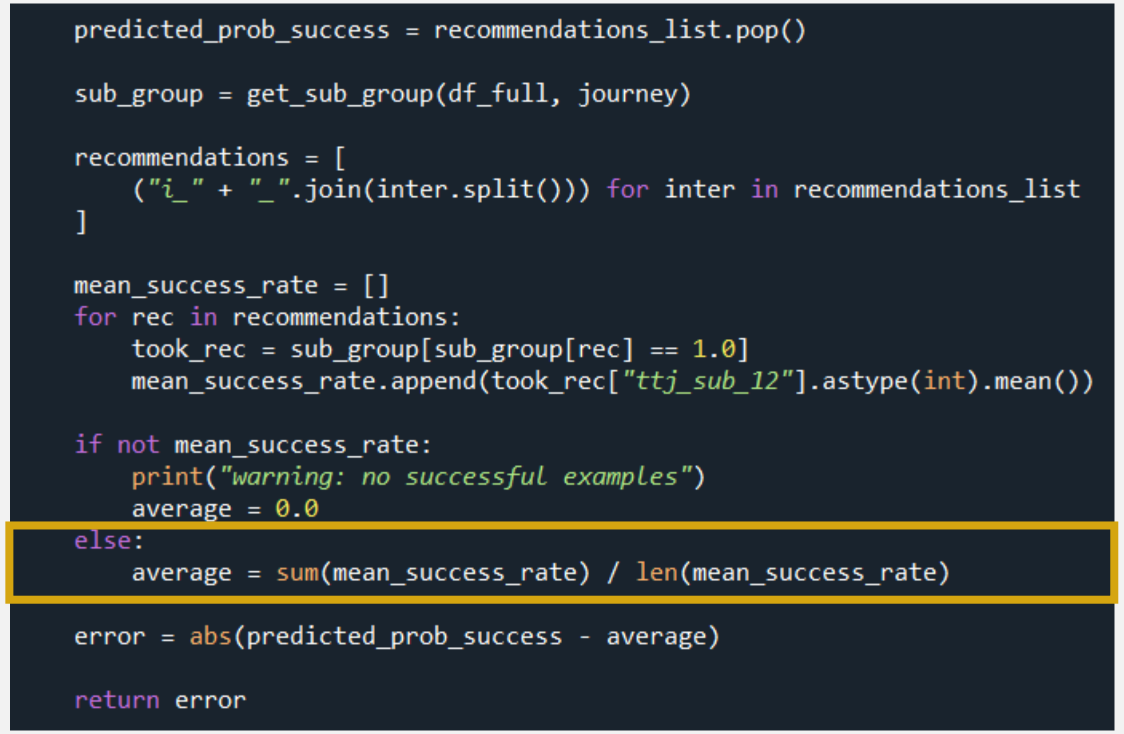
For **each** recommendation, another subsample of people in the reference group is created.

It contains the names of the people who also participated in this recommendation.

For those subsamples, the success of the intervention is calculated by calculating the percentage of people who are still unemployed **afterwards**.

Code Example

Recommender Error Calculation

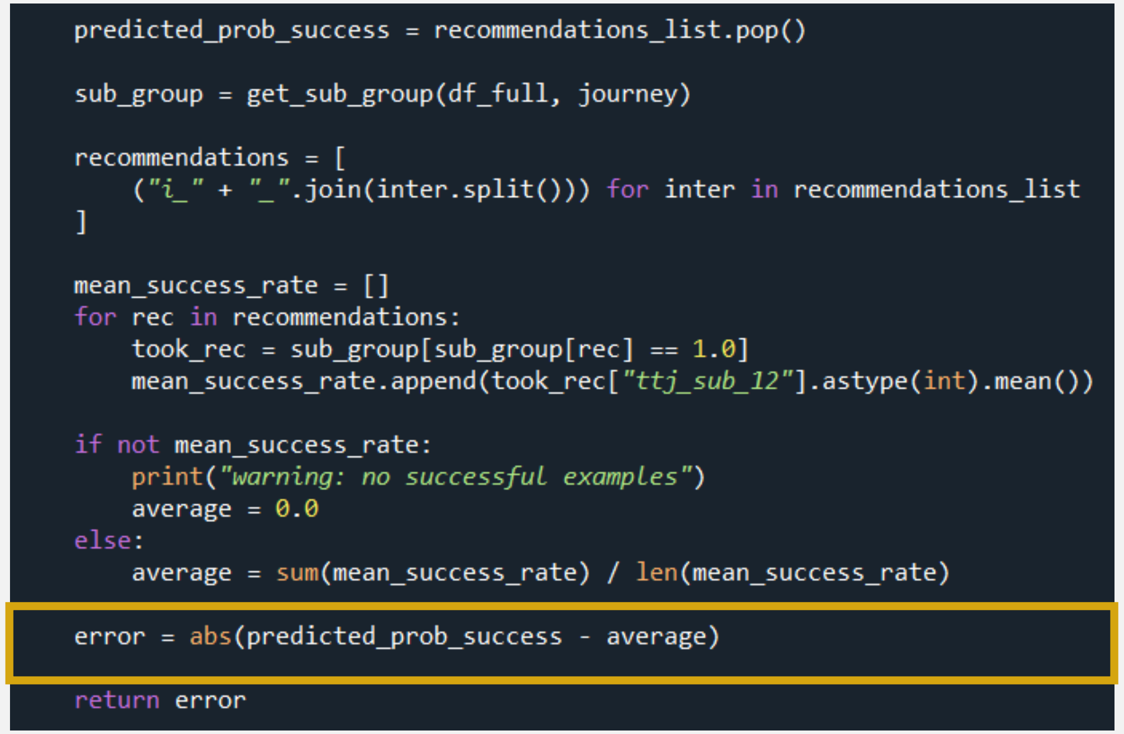


```
predicted_prob_success = recommendations_list.pop()

sub_group = get_sub_group(df_full, journey)

recommendations = [
    ("i_" + "_".join(inter.split())) for inter in recommendations_list
]

mean_success_rate = []
for rec in recommendations:
    took_rec = sub_group[sub_group[rec] == 1.0]
    mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

if not mean_success_rate:
    print("warning: no successful examples")
    average = 0.0
else:
    average = sum(mean_success_rate) / len(mean_success_rate)

error = abs(predicted_prob_success - average)

return error
```

All success rates are then summed up and averaged to be in the same range as the probability.

Code Example

Recommender Error Calculation



```python
    predicted_prob_success = recommendations_list.pop()

    sub_group = get_sub_group(df_full, journey)

    recommendations = [
        ("i_" + "_".join(inter.split())) for inter in recommendations_list
    ]

    mean_success_rate = []
    for rec in recommendations:
        took_rec = sub_group[sub_group[rec] == 1.0]
        mean_success_rate.append(took_rec["ttj_sub_12"].astype(int).mean())

    if not mean_success_rate:
        print("warning: no successful examples")
        average = 0.0
    else:
        average = sum(mean_success_rate) / len(mean_success_rate)

    error = abs(predicted_prob_success - average)

    return error
```

Lastly, the error of the prediction is calculated by comparing it with the average success rate of the recommendations in the subgroup.

Solution

The project team improved job counselors' performance by developing two optimized models using historical unemployment data:

Risk Classification: A predictive model was created to identify clients at risk of long-term unemployment. It was tested, rolled out, and refined based on feedback from counselors.

Intervention Recommendation: A second model used data on past interventions to predict which actions are most likely to help individuals return to work. This provided personalized recommendations and a quantitative metric for counselors to assess success, reducing reliance on their experience alone.

COVID-19 mortality surveillance platform

COVID-19 mortality surveillance platform

Project Team:

Data Science for Social Good Portugal (DSSG)

3 data scientists

Stakeholder:

This project was initiated **without** being commissioned by a public institution

Problem Statement:

During the first years of the COVID-19 pandemic, mortality data played a central role in estimating the severity of the disease. As a result, it also proved to be a major factor in the decision-making process on measures taken to contain the disease.

The initiators of this project noticed problems with the publicly available data. Mainly, it was without structure and proper documentation.

The goal of this project was therefore to create a pipeline for improving the quality of the data and making it available in an accessible format.

COVID-19 mortality surveillance platform

Detailed Information

DFG organization:
DSSG PT

Partner type:
Project developed using open Protugues-government data

Partner name:
-

Sustainable Development Goal (SDG):
3, 17

Type of interaction:
Short-term project

Type of analytics:
Data consulting

Type of data:
Web scraping of open Portuguese-government data

All data is available here

Internal & External Data & Methods

Prior to the project mortality data was only publicly available on SICO – eVM, a portal documenting this data in Portugal.

On the website, multiple tables and graphs are provided, but the raw data is **not** readily available in a downloadable format.

Web Scraping:

To make data usable for visualization, machine learning, or other tasks, it is usually beneficial to provide it in a database or in an easy to handle file format like .csv. If interesting data is **only** available in form the of content on a website, **web scrapers** are a way to obtain the data in a usable format. When web scraping scripts are created to access the website of interest automatically and save the data in the format best suitable for further work.

Transforming Data:

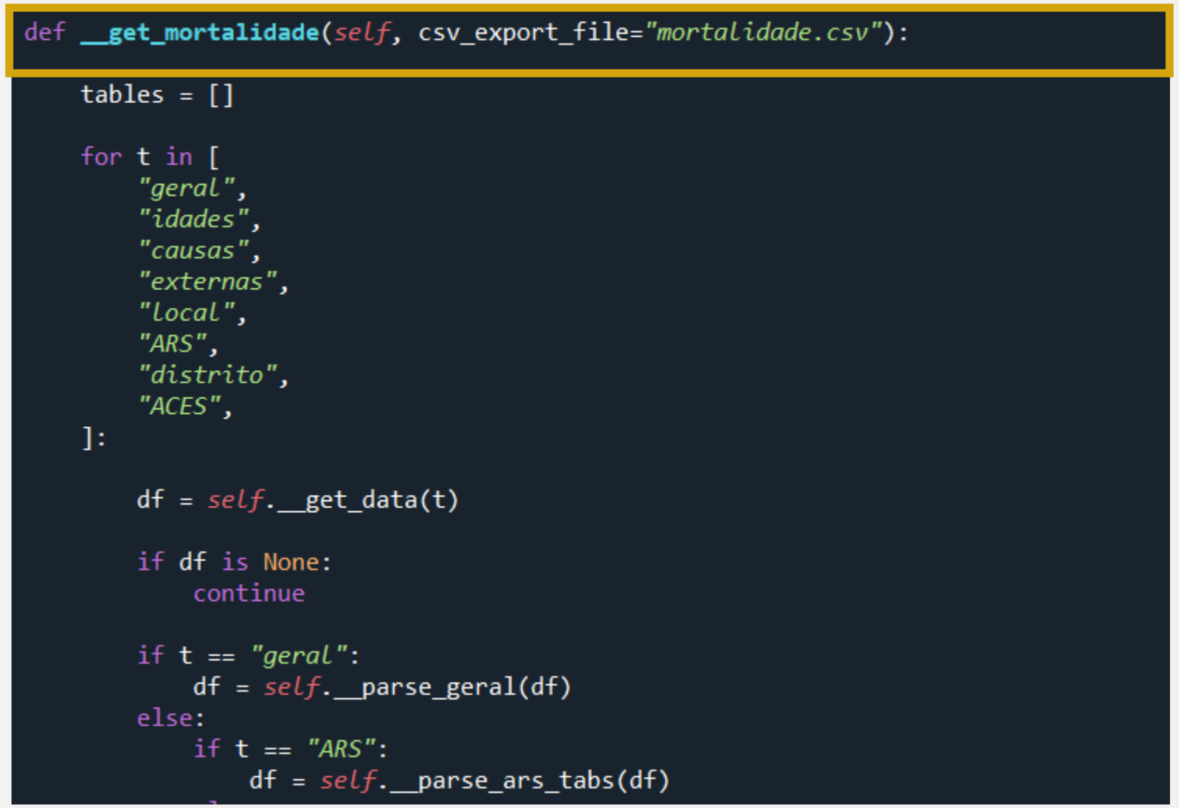
Data in **JSON form** is provided by websites in a previously defined format. The project team analyzed the structure of the provided data and developed an automated script to format the data into an .csv.

Reporting:

Web scrapers are prone to changes made to the target website. Owners of a website may change the accessibility or the way information is displayed. In this case it is important to be aware of how data is provided by having a report system. This should include automated checks on whether or not data is still being obtained.

Code Example

Data Extraction



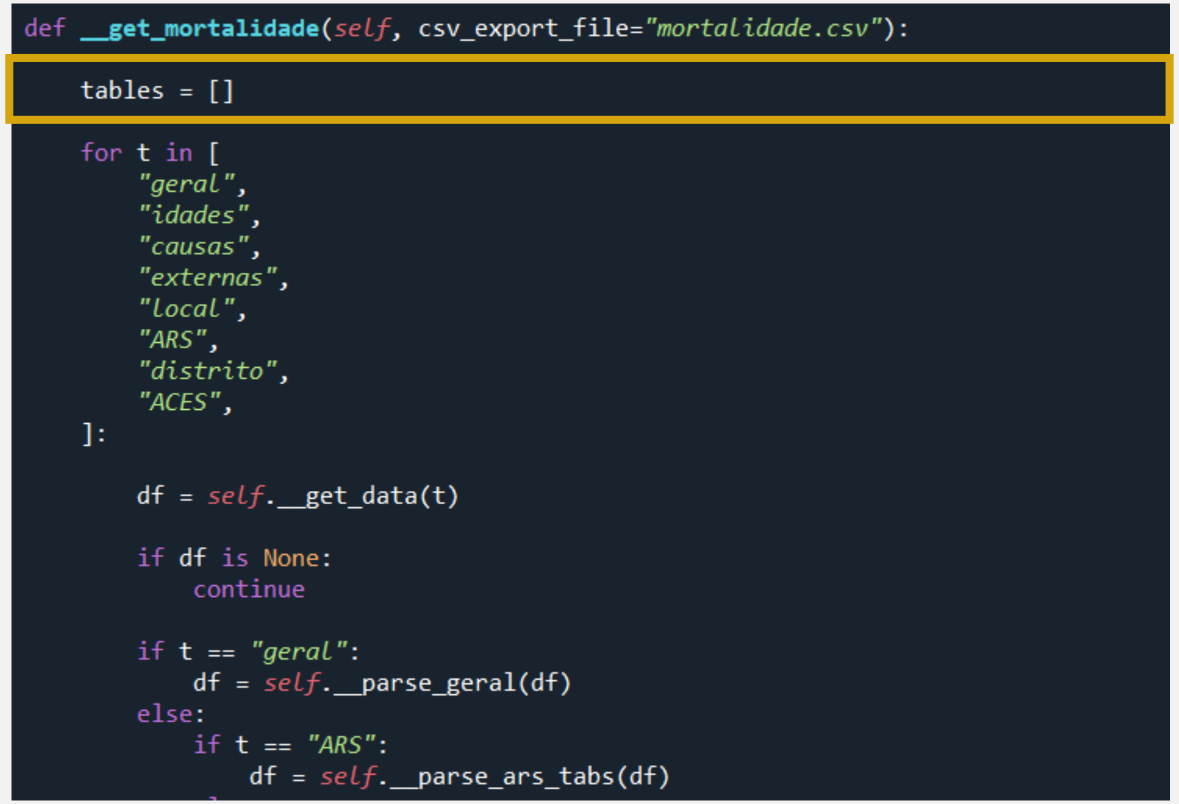
```
def __get_mortalidade(self, csv_export_file="mortalidade.csv"):
    tables = []

    for t in [
        "geral",
        "idades",
        "causas",
        "externas",
        "local",
        "ARS",
        "distrito",
        "ACES",
    ]:

        df = self.__get_data(t)

        if df is None:
            continue

        if t == "geral":
            df = self.__parse_geral(df)
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
```

This following code example is used to scrape the SICO-eVM portal.

This function is used inside the “MortalityScraping” class. Another function calls it and other functions to start the scraping.

Code Example

Data Extraction

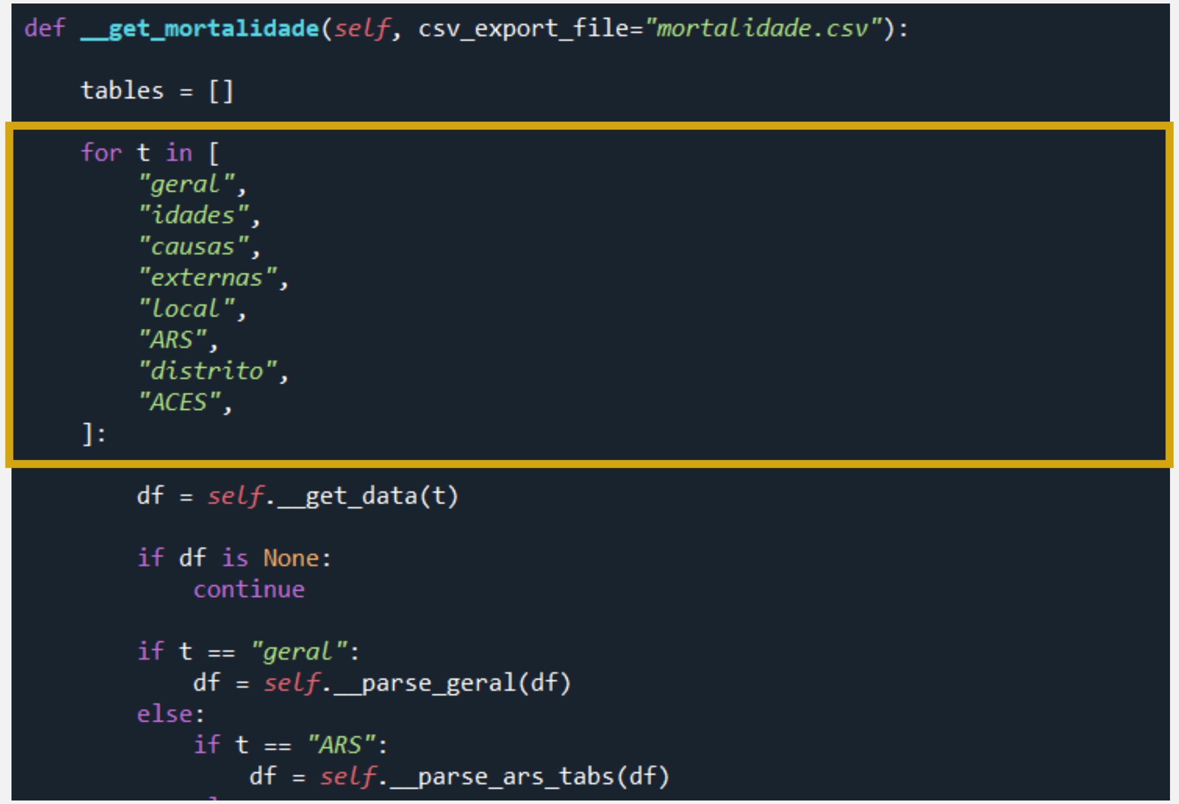


```
def __get_mortalidade(self, csv_export_file="mortalidade.csv"):
    tables = []

    for t in [
        "geral",
        "idades",
        "causas",
        "externas",
        "local",
        "ARS",
        "distrito",
        "ACES",
    ]:

        df = self.__get_data(t)

        if df is None:
            continue

        if t == "geral":
            df = self.__parse_geral(df)
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
```

This variable is initialized to later store the dataframes containing the scraped data.

Code Example

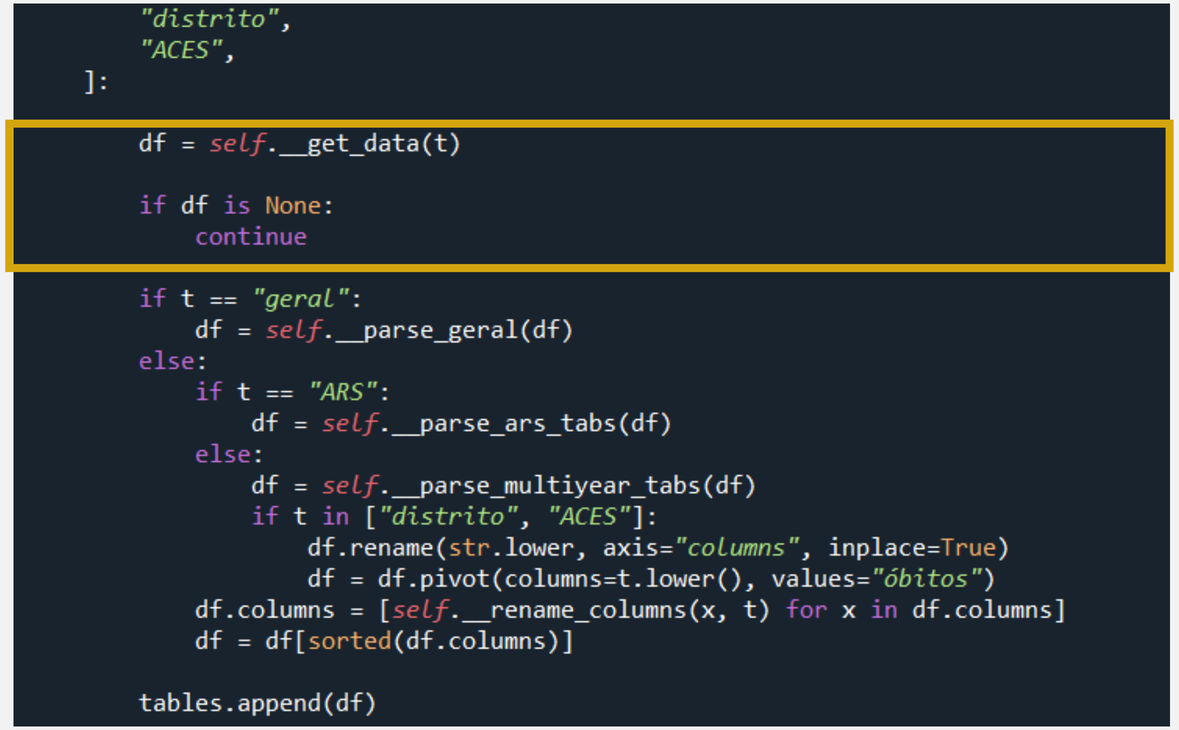
Data Extraction



```python
def __get_mortalidade(self, csv_export_file="mortalidade.csv"):

    tables = []

    for t in [
        "geral",
        "idades",
        "causas",
        "externas",
        "local",
        "ARS",
        "distrito",
        "ACES",
    ]:

        df = self.__get_data(t)

        if df is None:
            continue

        if t == "geral":
            df = self.__parse_geral(df)
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
```

Here, a loop started iterating through all the names of relevant tables.

Code Example
Data Extraction

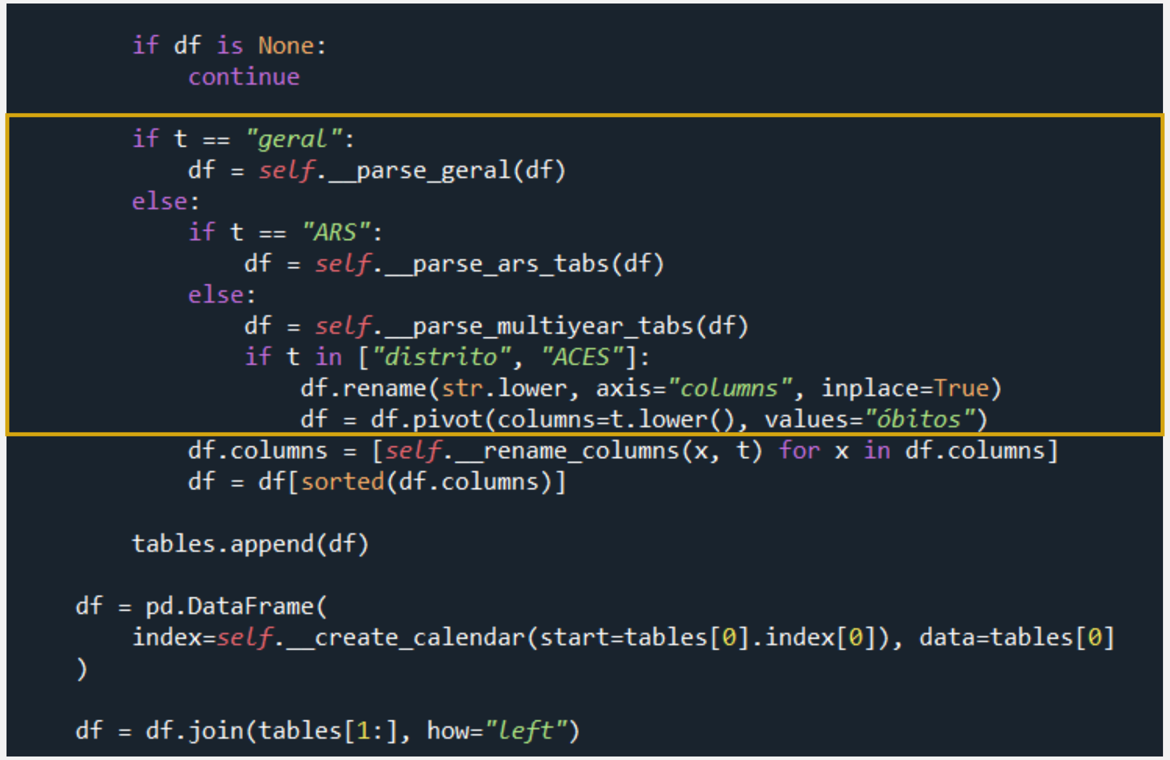


```
        "distrito",
        "ACES",
    ]:

        df = self.__get_data(t)

        if df is None:
            continue

        if t == "geral":
            df = self.__parse_geral(df)
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
            else:
                df = self.__parse_multiyear_tabs(df)
                if t in ["distrito", "ACES"]:
                    df.rename(str.lower, axis="columns", inplace=True)
                    df = df.pivot(columns=t.lower(), values="óbitos")
            df.columns = [self.__rename_columns(x, t) for x in df.columns]
            df = df[sorted(df.columns)]

        tables.append(df)
```

The function **"_get_data"** is called to send a request to the website and extract the data as raw text. This function returns **"None"** if the request fails. If so, the remaining code is skipped and the next table is scraped.

Code Example

Data Extraction



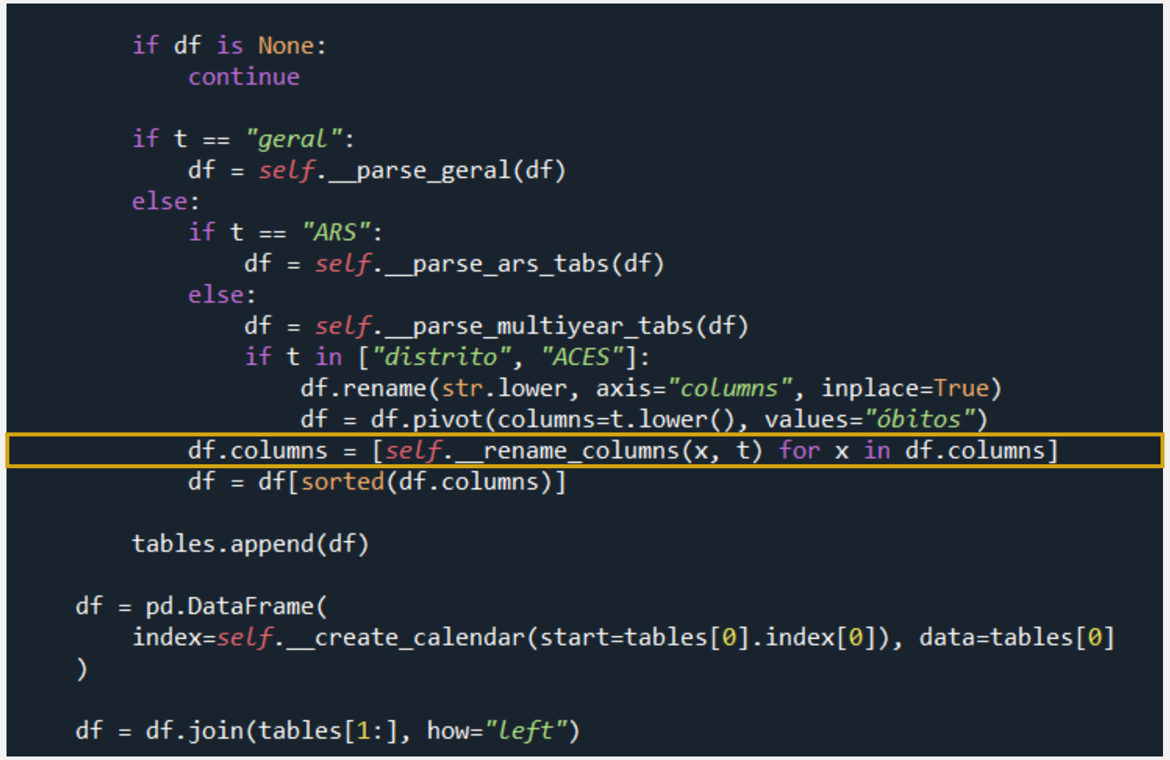
```
        if df is None:
            continue

        if t == "geral":
            df = self.__parse_geral(df)
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
            else:
                df = self.__parse_multiyear_tabs(df)
                if t in ["distrito", "ACES"]:
                    df.rename(str.lower, axis="columns", inplace=True)
                    df = df.pivot(columns=t.lower(), values="óbitos")
            df.columns = [self.__rename_columns(x, t) for x in df.columns]
            df = df[sorted(df.columns)]

        tables.append(df)

    df = pd.DataFrame(
        index=self.__create_calendar(start=tables[0].index[0]), data=tables[0]
    )

    df = df.join(tables[1:], how="left")
```

Not all tables have the same structure. This is why separate functions handle the structures of the tables.

The functions extract the raw data and save it into a dataframe.

Code Example

Data Extraction



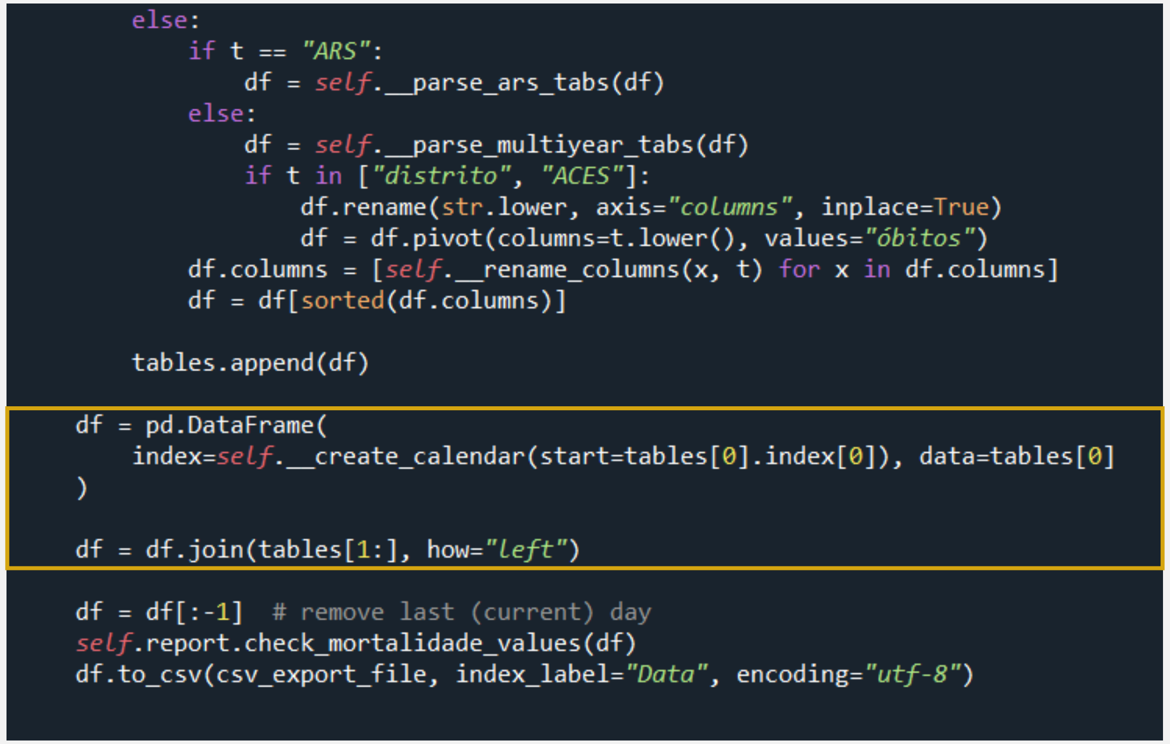
```
        if df is None:
            continue

        if t == "geral":
            df = self.__parse_geral(df)
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
            else:
                df = self.__parse_multiyear_tabs(df)
                if t in ["distrito", "ACES"]:
                    df.rename(str.lower, axis="columns", inplace=True)
                    df = df.pivot(columns=t.lower(), values="óbitos")
            df.columns = [self.__rename_columns(x, t) for x in df.columns]
            df = df[sorted(df.columns)]

        tables.append(df)

    df = pd.DataFrame(
        index=self.__create_calendar(start=tables[0].index[0]), data=tables[0]
    )

    df = df.join(tables[1:], how="left")
```

In the next step, the columns are renamed to be **consistent** with the current format.

EPSILON
Funded by the European Union

Code Example

Data Extraction



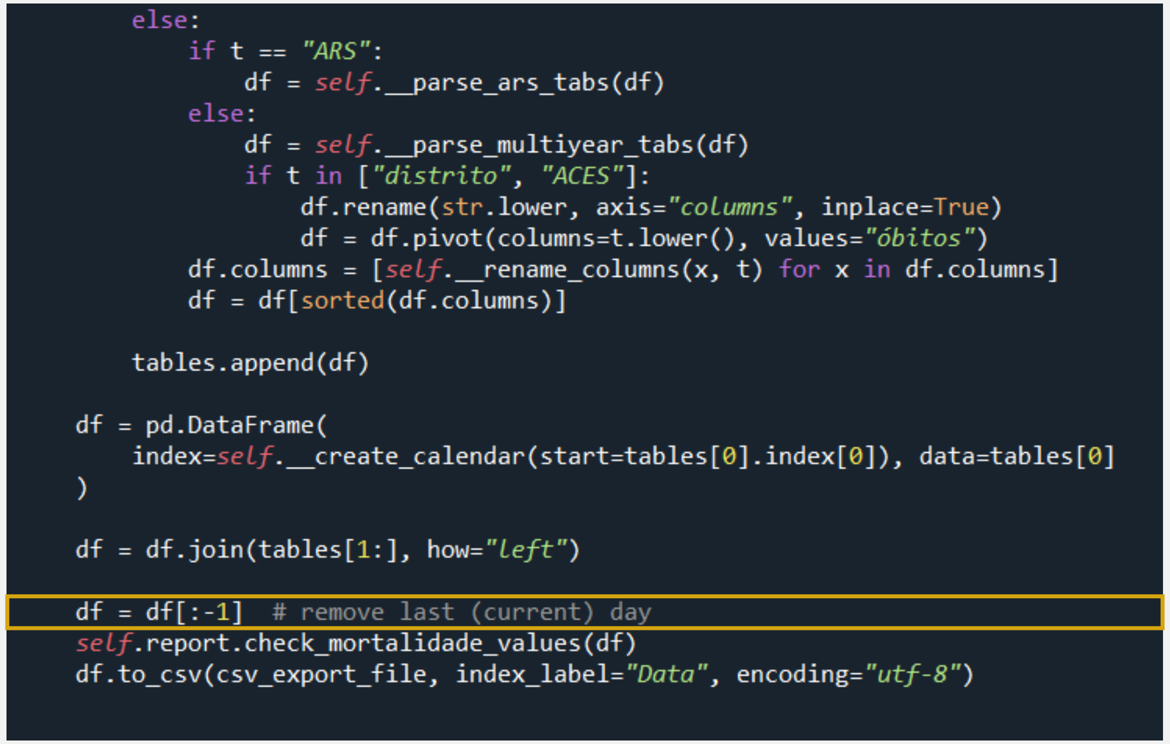
```
    else:
        if t == "ARS":
            df = self.__parse_ars_tabs(df)
        else:
            df = self.__parse_multiyear_tabs(df)
            if t in ["distrito", "ACES"]:
                df.rename(str.lower, axis="columns", inplace=True)
                df = df.pivot(columns=t.lower(), values="óbitos")
        df.columns = [self.__rename_columns(x, t) for x in df.columns]
        df = df[sorted(df.columns)]

    tables.append(df)

df = pd.DataFrame(
    index=self.__create_calendar(start=tables[0].index[0]), data=tables[0]
)

df = df.join(tables[1:], how="left")

df = df[:-1]  # remove last (current) day
self.report.check_mortalidade_values(df)
df.to_csv(csv_export_file, index_label="Data", encoding="utf-8")
```

Here, the function **"__create_calendar"** creates a dataframe and uses a consecutive series of dates as an index going from the first available date to today.

Then all remaining dataframes are joined onto the dataframe using the date as the index.

Code Example

Data Extraction

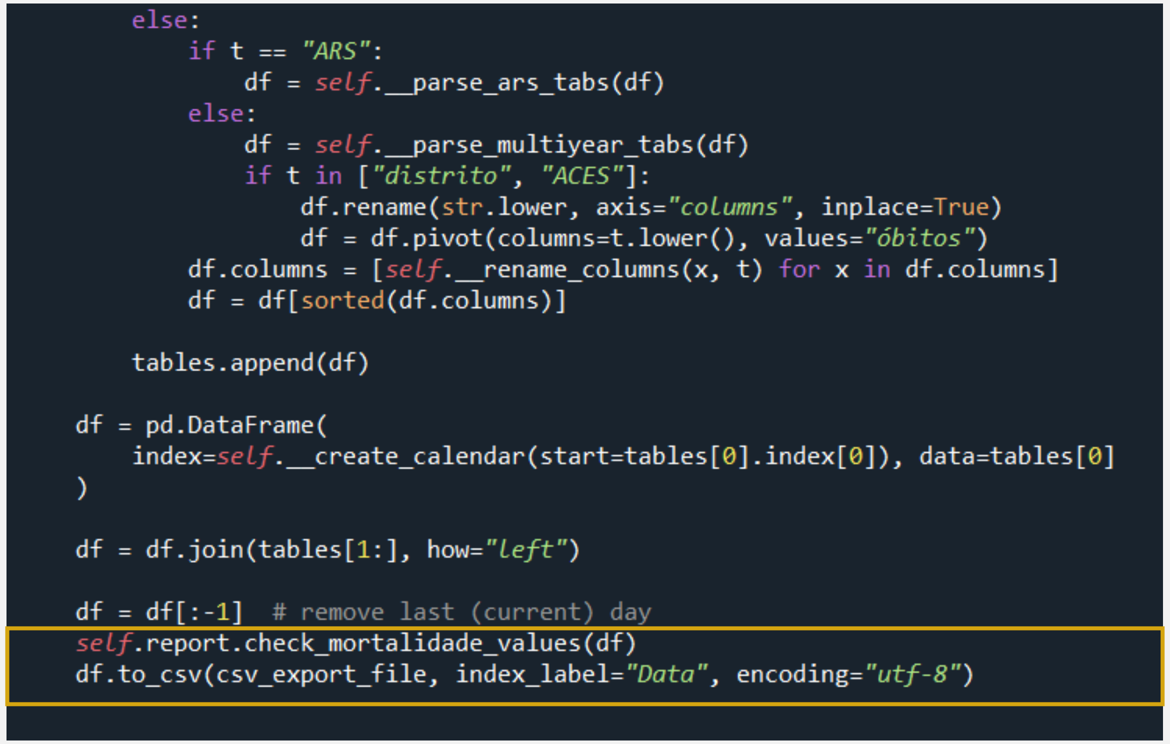


```
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
            else:
                df = self.__parse_multiyear_tabs(df)
                if t in ["distrito", "ACES"]:
                    df.rename(str.lower, axis="columns", inplace=True)
                    df = df.pivot(columns=t.lower(), values="óbitos")
            df.columns = [self.__rename_columns(x, t) for x in df.columns]
            df = df[sorted(df.columns)]

        tables.append(df)

    df = pd.DataFrame(
        index=self.__create_calendar(start=tables[0].index[0]), data=tables[0]
    )

    df = df.join(tables[1:], how="left")

    df = df[:-1]  # remove last (current) day
    self.report.check_mortalidade_values(df)
    df.to_csv(csv_export_file, index_label="Data", encoding="utf-8")
```

As a last step of data processing, the current date is removed, because the data is not complete yet.

Code Example

Data Extraction



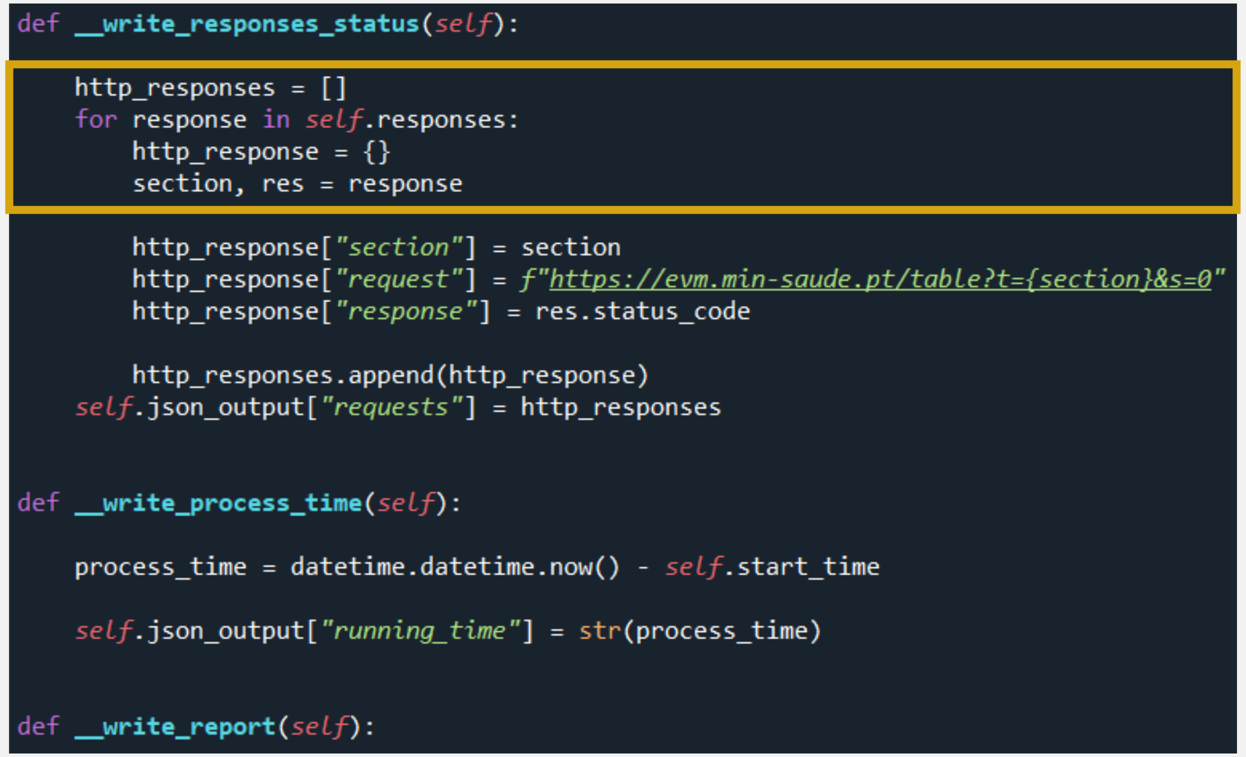
```
        else:
            if t == "ARS":
                df = self.__parse_ars_tabs(df)
            else:
                df = self.__parse_multiyear_tabs(df)
                if t in ["distrito", "ACES"]:
                    df.rename(str.lower, axis="columns", inplace=True)
                    df = df.pivot(columns=t.lower(), values="óbitos")
            df.columns = [self.__rename_columns(x, t) for x in df.columns]
            df = df[sorted(df.columns)]

        tables.append(df)

    df = pd.DataFrame(
        index=self.__create_calendar(start=tables[0].index[0]), data=tables[0]
    )

    df = df.join(tables[1:], how="left")

    df = df[:-1]  # remove last (current) day
    self.report.check_mortalidade_values(df)
    df.to_csv(csv_export_file, index_label="Data", encoding="utf-8")
```

Finally, the web scraping report is updated with information on how many missing values are documented.

To save the data, it is written into a .csv file.

Code Example

Reporting



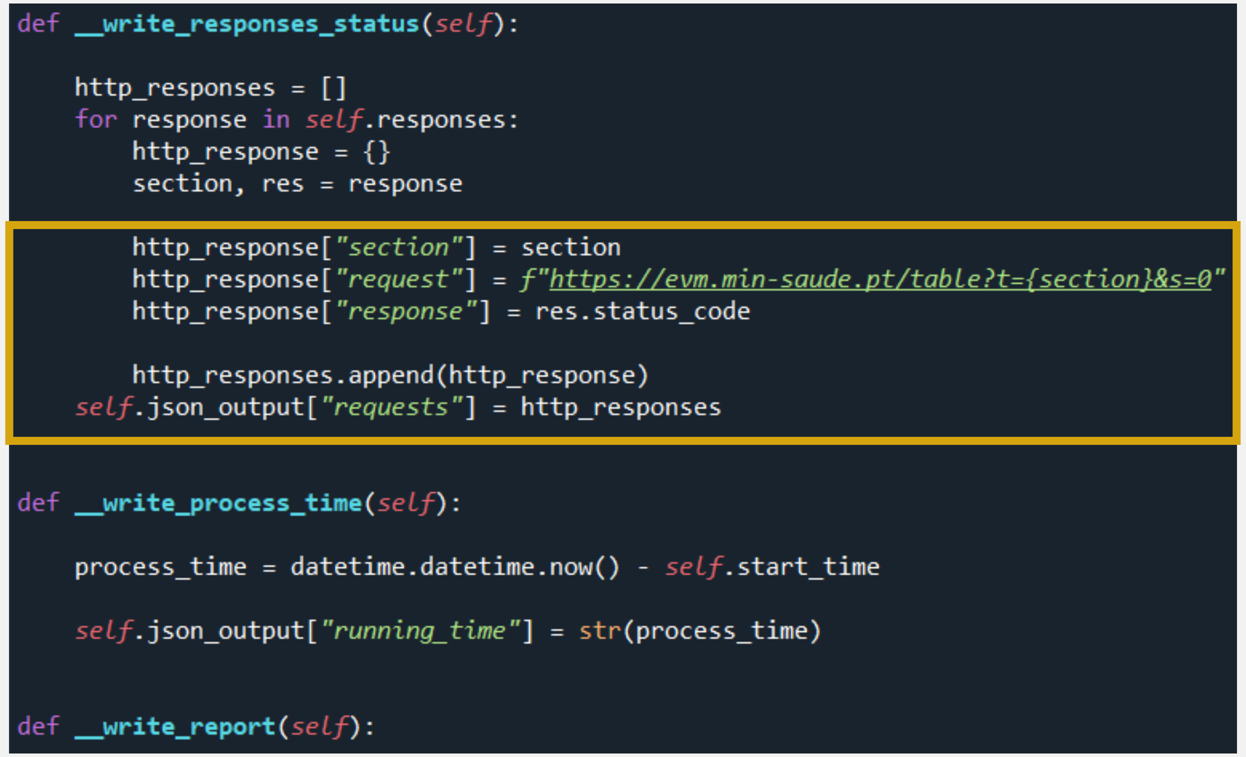
```python
def __write_responses_status(self):

    http_responses = []
    for response in self.responses:
        http_response = {}
        section, res = response

        http_response["section"] = section
        http_response["request"] = f"https://evm.min-saude.pt/table?t={section}&s=0"
        http_response["response"] = res.status_code

        http_responses.append(http_response)
    self.json_output["requests"] = http_responses


def __write_process_time(self):

    process_time = datetime.datetime.now() - self.start_time

    self.json_output["running_time"] = str(process_time)


def __write_report(self):
```

The following code is used inside a class automating the creation of a web scraping report. This report is used for monitoring of the task.

This function starts by iterating through **all** the requests made by the web scraper.

Code Example

Reporting

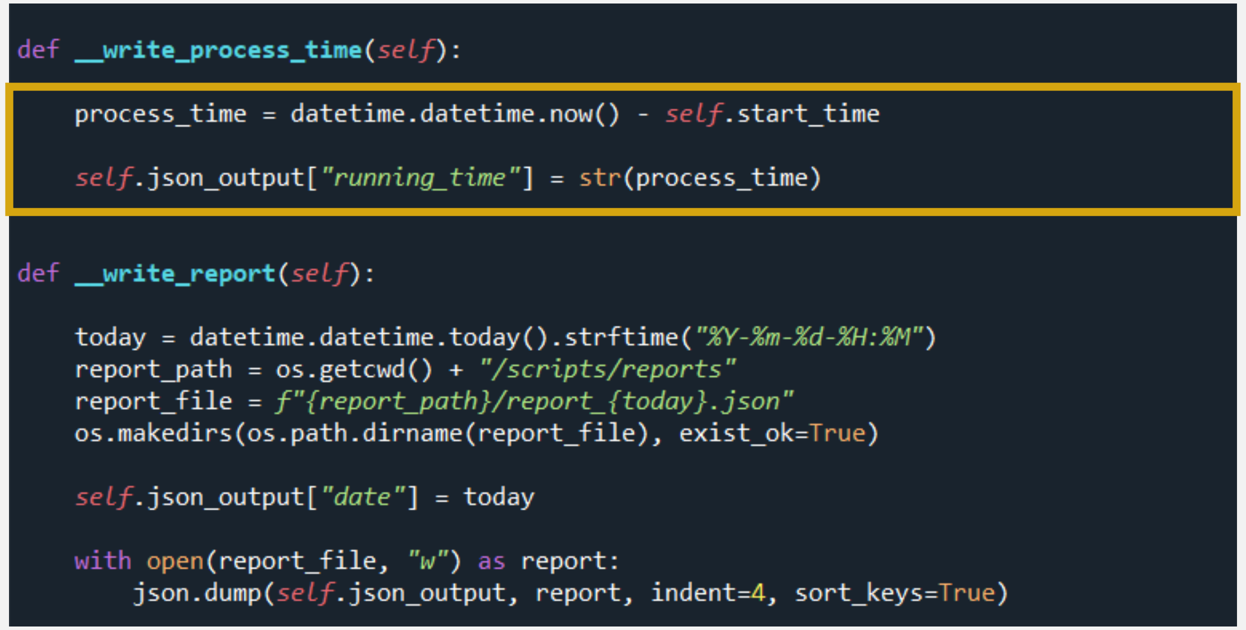


```python
def __write_responses_status(self):

    http_responses = []
    for response in self.responses:
        http_response = {}
        section, res = response

        http_response["section"] = section
        http_response["request"] = f"https://evm.min-saude.pt/table?t={section}&s=0"
        http_response["response"] = res.status_code

        http_responses.append(http_response)
    self.json_output["requests"] = http_responses


def __write_process_time(self):

    process_time = datetime.datetime.now() - self.start_time

    self.json_output["running_time"] = str(process_time)


def __write_report(self):
```

For each request, the name of the scraped table and the response code is saved.

Here, the information is saved into the output variable.

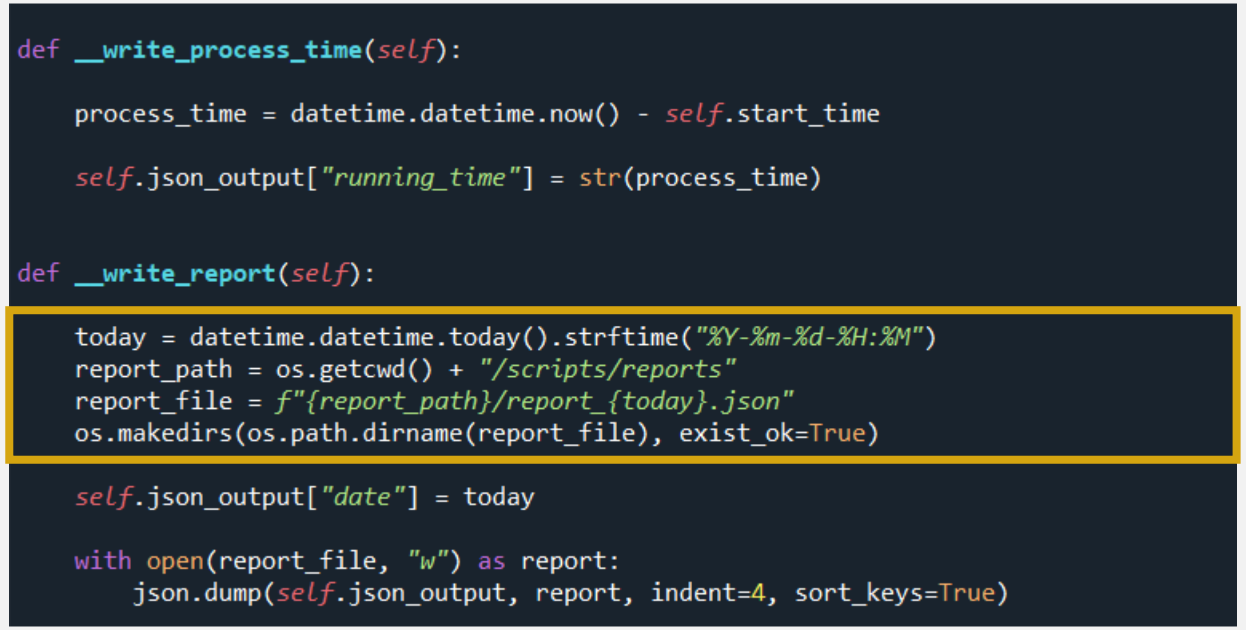
Code Example
Reporting



```python
def __write_process_time(self):

    process_time = datetime.datetime.now() - self.start_time

    self.json_output["running_time"] = str(process_time)


def __write_report(self):

    today = datetime.datetime.today().strftime("%Y-%m-%d-%H:%M")
    report_path = os.getcwd() + "/scripts/reports"
    report_file = f"{report_path}/report_{today}.json"
    os.makedirs(os.path.dirname(report_file), exist_ok=True)

    self.json_output["date"] = today

    with open(report_file, "w") as report:
        json.dump(self.json_output, report, indent=4, sort_keys=True)
```

This function checks how much time has passed since the web scraping started and again saves this information to be passed to the output later on.

Code Example
Reporting

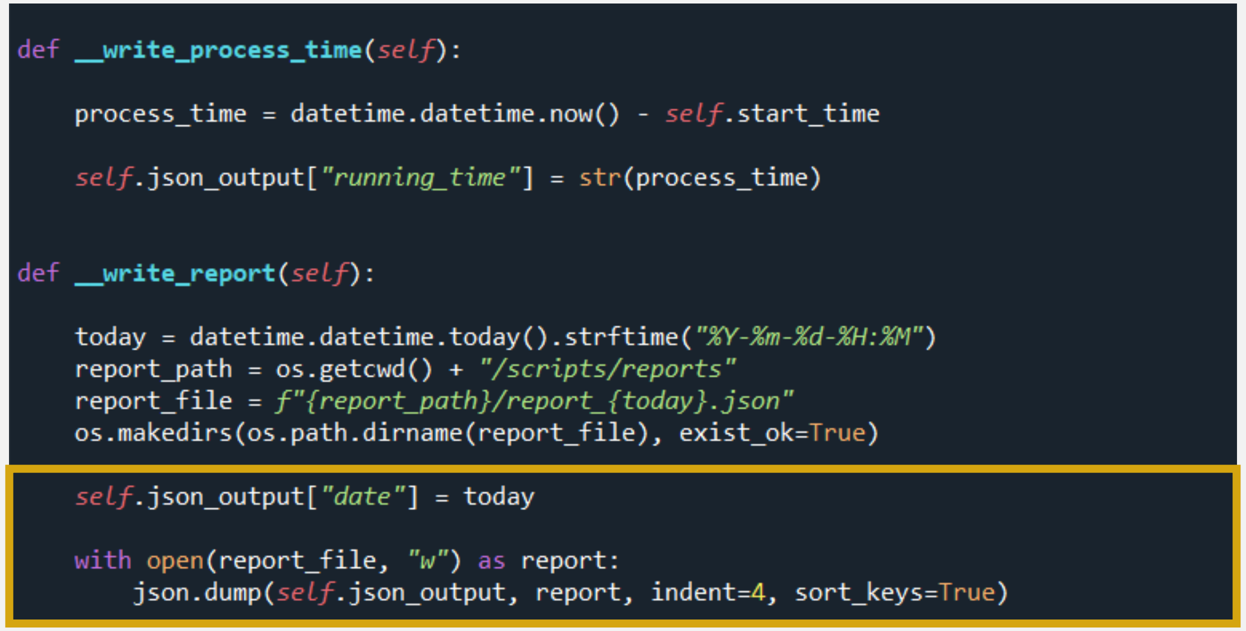


```
def __write_process_time(self):

    process_time = datetime.datetime.now() - self.start_time

    self.json_output["running_time"] = str(process_time)


def __write_report(self):

    today = datetime.datetime.today().strftime("%Y-%m-%d-%H:%M")
    report_path = os.getcwd() + "/scripts/reports"
    report_file = f"{report_path}/report_{today}.json"
    os.makedirs(os.path.dirname(report_file), exist_ok=True)

    self.json_output["date"] = today

    with open(report_file, "w") as report:
        json.dump(self.json_output, report, indent=4, sort_keys=True)
```

At first, this function gets the current date and a file is created in which the report is later going to be saved.

Code Example

Reporting



```
def __write_process_time(self):

    process_time = datetime.datetime.now() - self.start_time

    self.json_output["running_time"] = str(process_time)


def __write_report(self):

    today = datetime.datetime.today().strftime("%Y-%m-%d-%H:%M")
    report_path = os.getcwd() + "/scripts/reports"
    report_file = f"{report_path}/report_{today}.json"
    os.makedirs(os.path.dirname(report_file), exist_ok=True)

    self.json_output["date"] = today

    with open(report_file, "w") as report:
        json.dump(self.json_output, report, indent=4, sort_keys=True)
```

Next, the date is saved and the complete output (including the previously saved data) is written into the file.

COVID-19 mortality surveillance platform

Solution

The project team was able to create an automatic system for accessing the mortality data on a daily basis.

This helped the following publicly available services:

Data repository

A well-documented and comprehensive repository containing historical data on mortality from 2014 up to today.

In addition to data in multiple formats, a data dictionary was created, explaining how the data is stored.

Web Scraping Example

The code for this project is publicly available and can be a starting point to create other web scrapers with automatic reporting.

Domestic Violence Data Observatory

Domestic Violence Data Observatory

Project Team:

Data Science for Social Good Portugal (DSSG)

3 data scientists

Stakeholder:

This project was initiated without being commissioned by a public institution

Problem Statement:

Domestic violence is the second most commonly recorded crime in Portugal. Although there is a multitude of data sources on the subject, there is a lack of easily accessible dash boards and data in simple formats (e.g. .csv)

This project has two main goals:

1. Create a data repository making data on domestic violence available in a simple format
2. Visualize the data in a dashboard

Domestic Violence Data Observatory

Detailed Information

DFG organization:
DSSG PT

Partner type:
Project developed using open Protugues-government data

Partner name:
-

Sustainable Development Goal (SDG):
3, 5, 16

Type of interaction:
Short-term project

Type of analytics:
Data consulting

Type of data:
Web scraping of open Portuguese-government data and NGOs data in Portugal

All data is available here

Data Sources & Methods

This project relied mainly on the following data sources. The data was only available in form of .pdf files. The aggregation of those files into an .csv file was handled individually.

Data Sources:

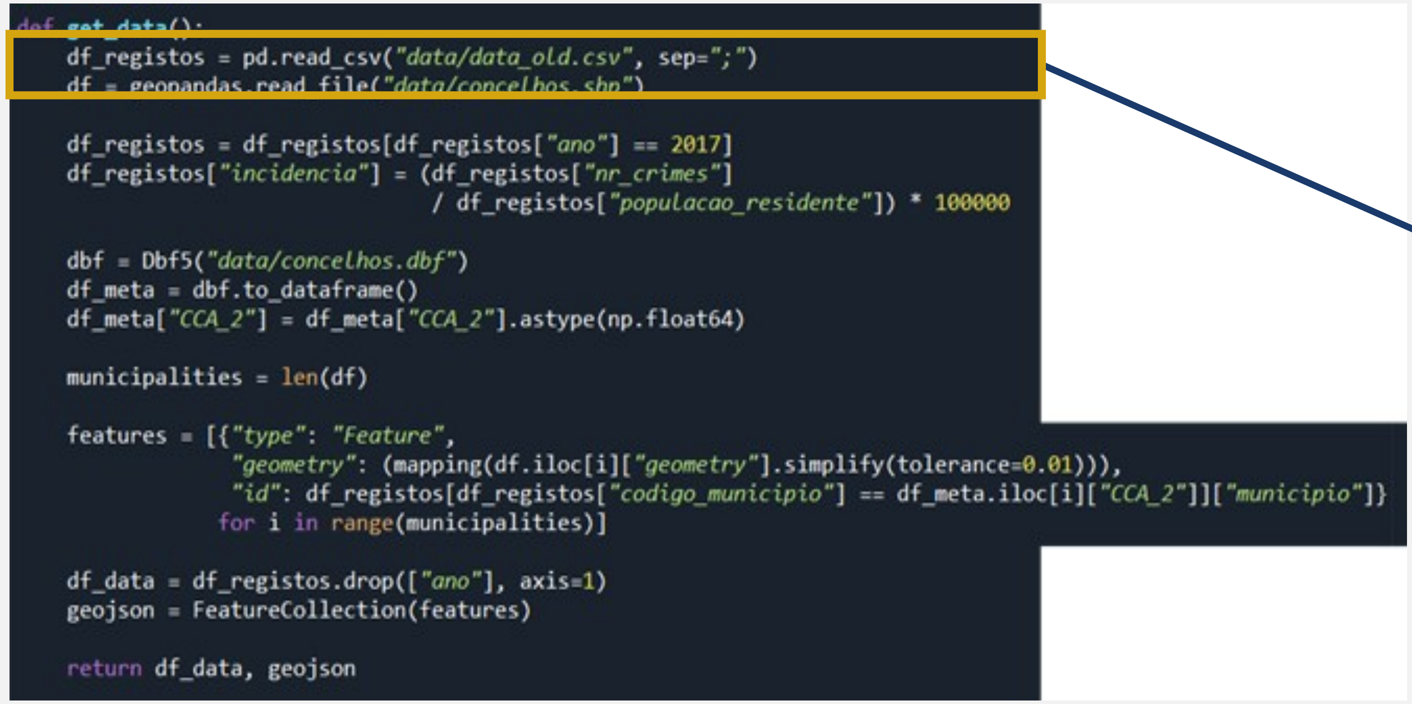
- APAV (Portuguese Association for Victim Support) Reports on Domestic Violence
- Quarterly Report on Domestic Violence provided by the Portuguese Government
- Yearly Report on Domestic Violence provided by Ministry of Internal Affairs
- Report on Victim Support Structures

Dashboard

- To provide a clear overview, data from multiple reports was aggregated geographically into a dashboard during this project.

Code Example

Data Loading



```
def get_data():
    df_registos = pd.read_csv("data/data_old.csv", sep=";")
    df = geopandas.read_file("data/concelhos.shp")

    df_registos = df_registos[df_registos["ano"] == 2017]
    df_registos["incidencia"] = (df_registos["nr_crimes"]
                                 / df_registos["populacao_residente"]) * 100000

    dbf = Dbf5("data/concelhos.dbf")
    df_meta = dbf.to_dataframe()
    df_meta["CCA_2"] = df_meta["CCA_2"].astype(np.float64)

    municipalities = len(df)

    features = [{"type": "Feature",
                 "geometry": (mapping(df.iloc[i]["geometry"].simplify(tolerance=0.01))),
                 "id": df_registos[df_registos["codigo_municipio"] == df_meta.iloc[i]["CCA_2"]]["municipio"]}
                for i in range(municipalities)]

    df_data = df_registos.drop(["ano"], axis=1)
    geojson = FeatureCollection(features)

    return df_data, geojson
```

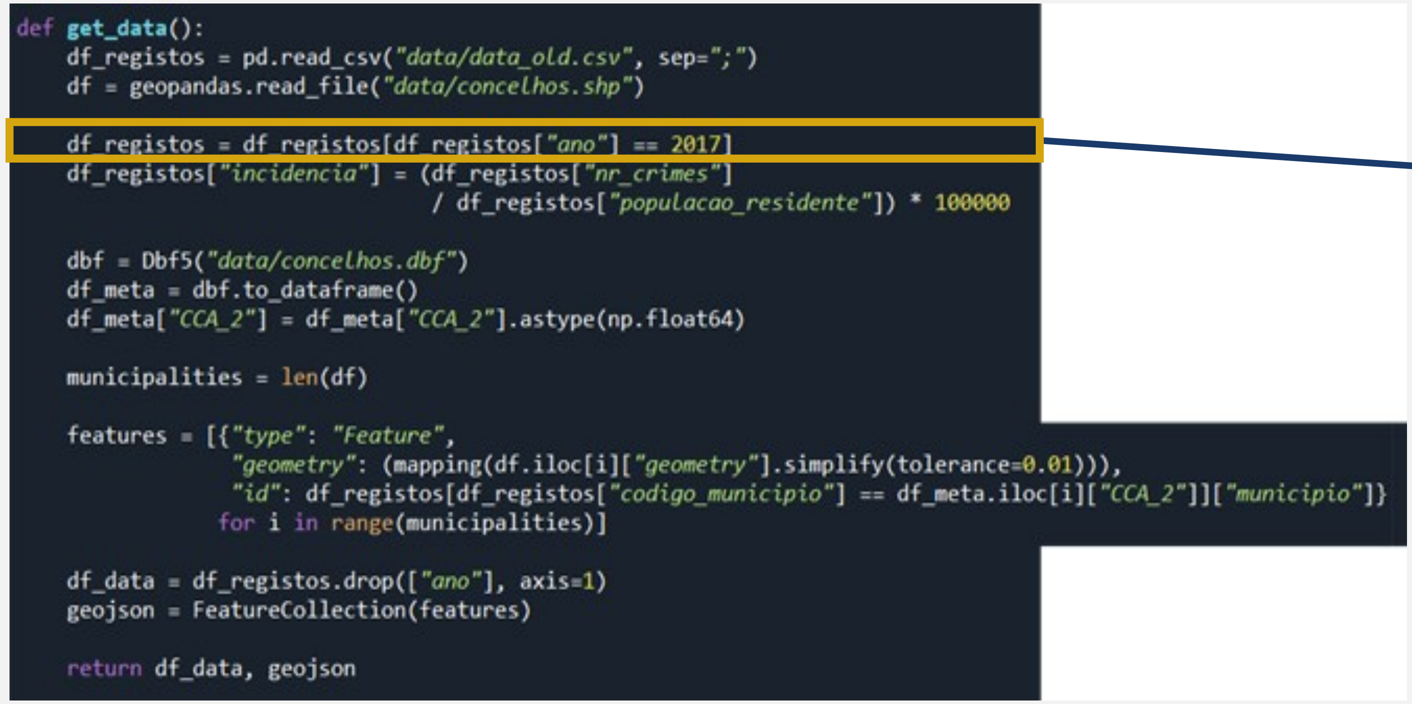
To prepare available data for the dashboard, it needs to be mapped to the geographic location.

First, the available data and geographical data on Portuguese municipalities are loaded.

Hochschule Harz
Harz University of Applied Sciences
EPSILON
Funded by the European Union

Code Example

Data Loading



```python
def get_data():
    df_registos = pd.read_csv("data/data_old.csv", sep=";")
    df = geopandas.read_file("data/concelhos.shp")

    df_registos = df_registos[df_registos["ano"] == 2017]
    df_registos["incidencia"] = (df_registos["nr_crimes"]
                                 / df_registos["populacao_residente"]) * 100000

    dbf = Dbf5("data/concelhos.dbf")
    df_meta = dbf.to_dataframe()
    df_meta["CCA_2"] = df_meta["CCA_2"].astype(np.float64)

    municipalities = len(df)

    features = [{"type": "Feature",
                 "geometry": (mapping(df.iloc[i]["geometry"].simplify(tolerance=0.01))),
                 "id": df_registos[df_registos["codigo_municipio"] == df_meta.iloc[i]["CCA_2"]]["municipio"]}
                for i in range(municipalities)]

    df_data = df_registos.drop(["ano"], axis=1)
    geojson = FeatureCollection(features)

    return df_data, geojson
```

Next, the data is filtered to only contain data from 2017.

This is done as the **final dashbaord** is supposed to be a prototype.

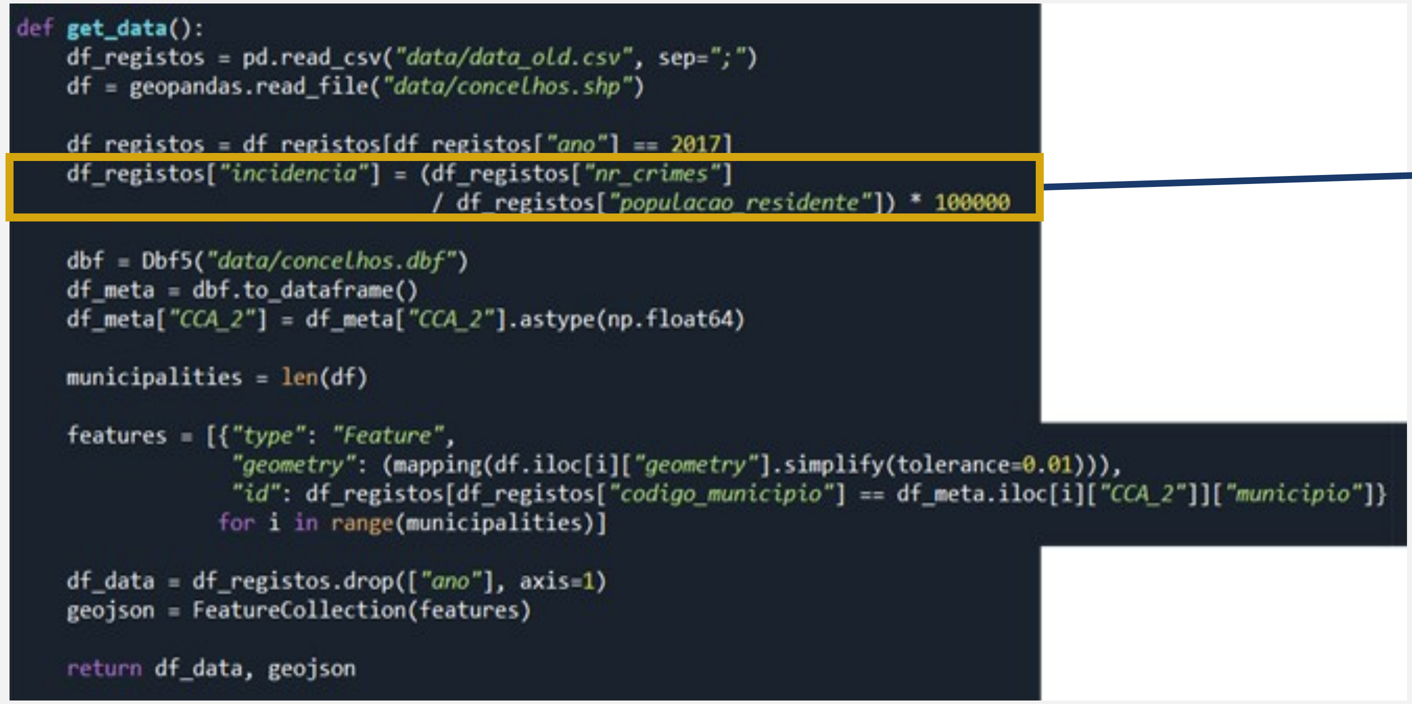
Hochschule Harz
Harz University of Applied Sciences

EPSILON

Funded by the European Union

Code Example

Data Loading

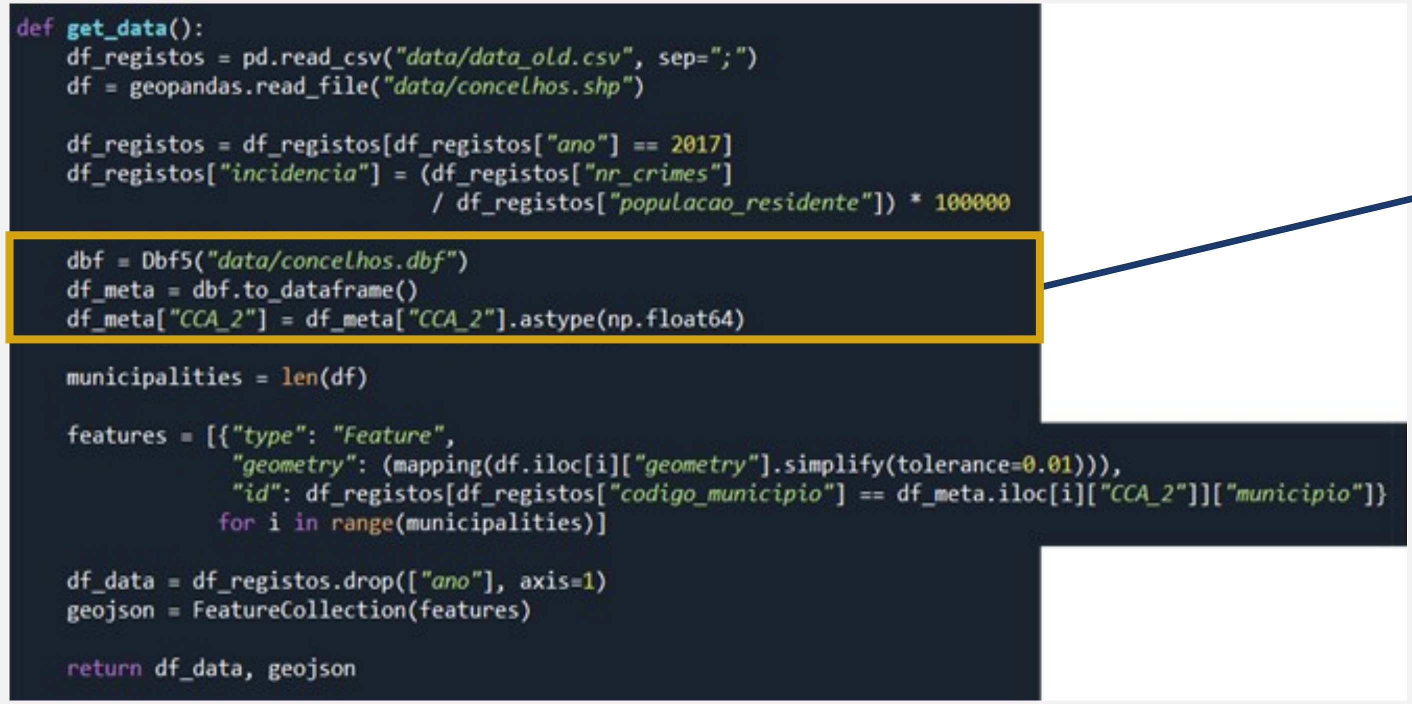


```python
def get_data():
    df_registos = pd.read_csv("data/data_old.csv", sep=";")
    df = geopandas.read_file("data/concelhos.shp")

    df_registos = df_registos[df_registos["ano"] == 2017]
    df_registos["incidencia"] = (df_registos["nr_crimes"]
                                 / df_registos["populacao_residente"]) * 100000

    dbf = Dbf5("data/concelhos.dbf")
    df_meta = dbf.to_dataframe()
    df_meta["CCA_2"] = df_meta["CCA_2"].astype(np.float64)

    municipalities = len(df)

    features = [{"type": "Feature",
                 "geometry": (mapping(df.iloc[i]["geometry"].simplify(tolerance=0.01))),
                 "id": df_registos[df_registos["codigo_municipio"] == df_meta.iloc[i]["CCA_2"]]["municipio"]}
                for i in range(municipalities)]

    df_data = df_registos.drop(["ano"], axis=1)
    geojson = FeatureCollection(features)

    return df_data, geojson
```

To make the number of crimes comparable between municipalities, it is scaled to represent the number of crimes per 100000 residents.

Code Example

Data Loading

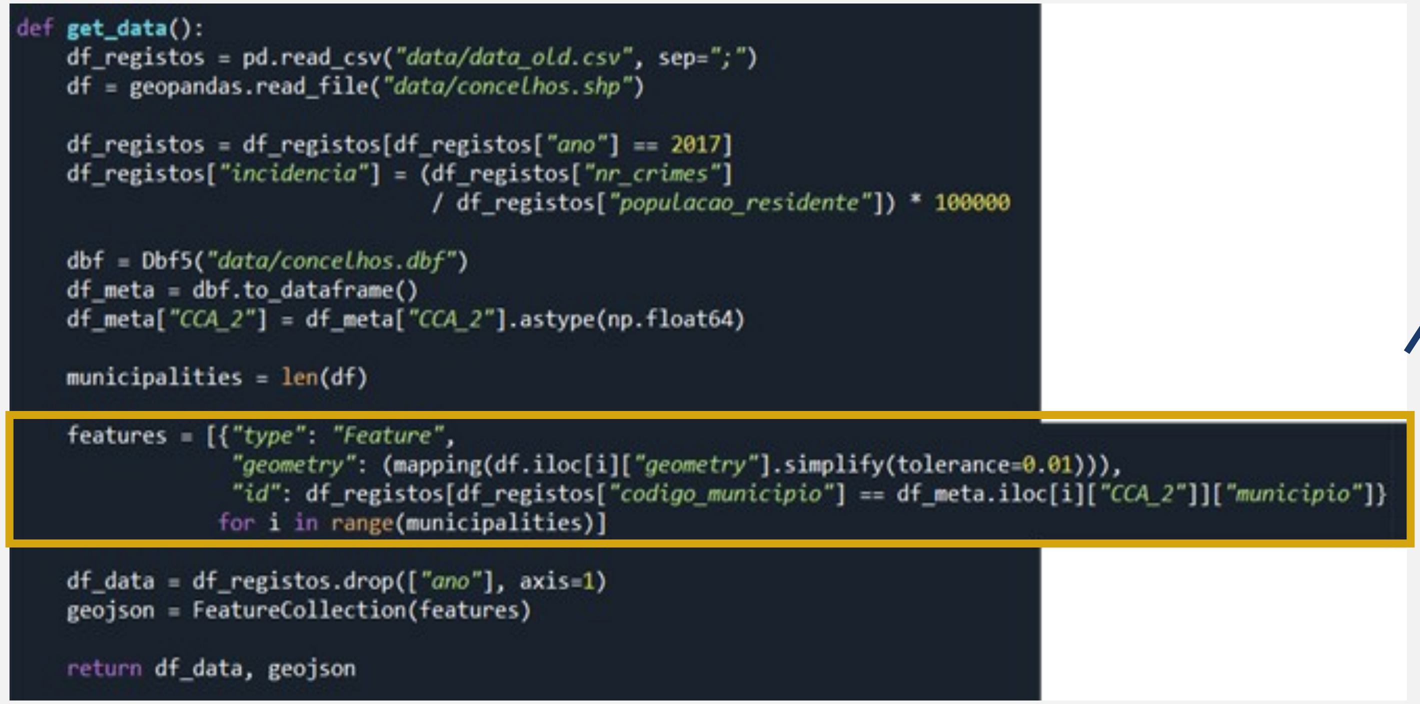


```
def get_data():
    df_registos = pd.read_csv("data/data_old.csv", sep=";")
    df = geopandas.read_file("data/concelhos.shp")

    df_registos = df_registos[df_registos["ano"] == 2017]
    df_registos["incidencia"] = (df_registos["nr_crimes"]
                                 / df_registos["populacao_residente"]) * 100000

    dbf = Dbf5("data/concelhos.dbf")
    df_meta = dbf.to_dataframe()
    df_meta["CCA_2"] = df_meta["CCA_2"].astype(np.float64)

    municipalities = len(df)

    features = [{"type": "Feature",
                 "geometry": (mapping(df.iloc[i]["geometry"].simplify(tolerance=0.01))),
                 "id": df_registos[df_registos["codigo_municipio"] == df_meta.iloc[i]["CCA_2"]]["municipio"]}
                for i in range(municipalities)]

    df_data = df_registos.drop(["ano"], axis=1)
    geojson = FeatureCollection(features)

    return df_data, geojson
```

Here, meta data on the municipalities is loaded. This is done to filter the municipalities later on.

Code Example

Data Loading



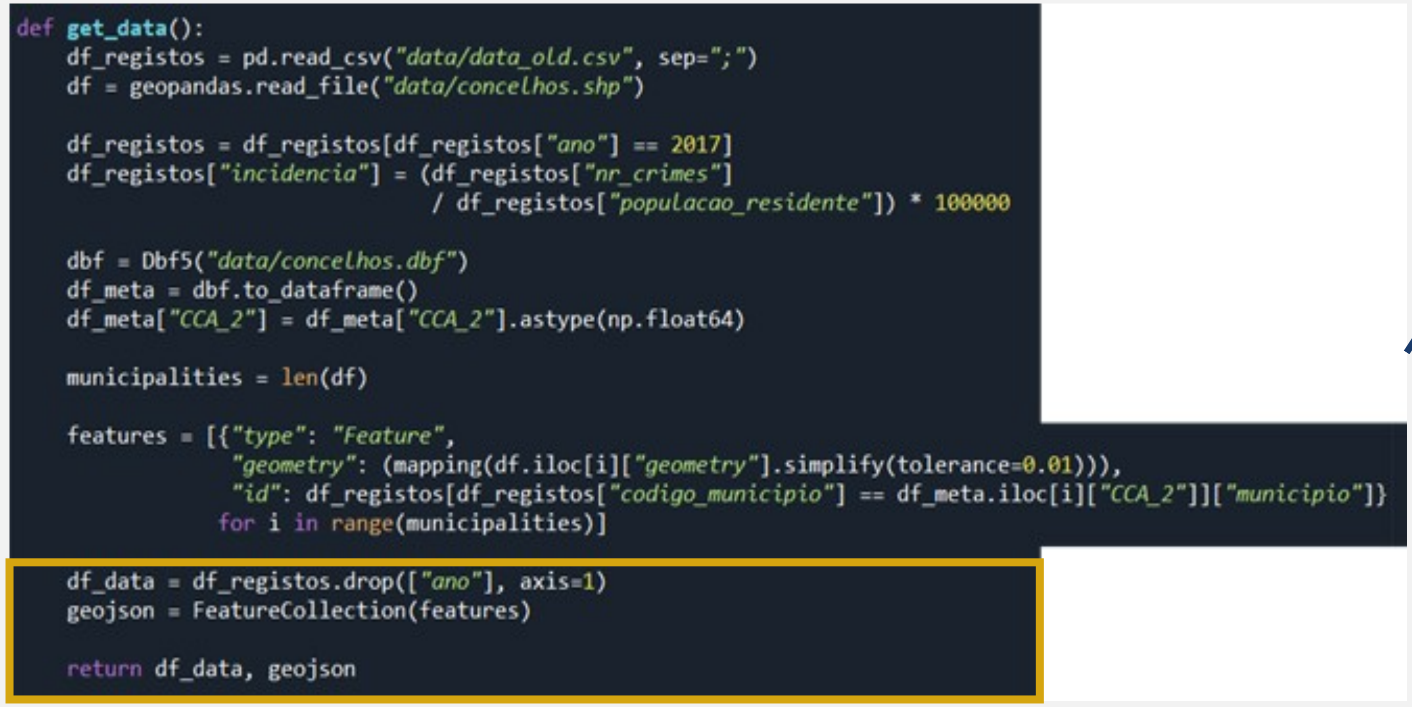
```python
def get_data():
    df_registos = pd.read_csv("data/data_old.csv", sep=";")
    df = geopandas.read_file("data/concelhos.shp")

    df_registos = df_registos[df_registos["ano"] == 2017]
    df_registos["incidencia"] = (df_registos["nr_crimes"]
                                 / df_registos["populacao_residente"]) * 100000

    dbf = Dbf5("data/concelhos.dbf")
    df_meta = dbf.to_dataframe()
    df_meta["CCA_2"] = df_meta["CCA_2"].astype(np.float64)

    municipalities = len(df)

    features = [{"type": "Feature",
                 "geometry": (mapping(df.iloc[i]["geometry"].simplify(tolerance=0.01))),
                 "id": df_registos[df_registos["codigo_municipio"] == df_meta.iloc[i]["CCA_2"]]["municipio"]}
                for i in range(municipalities)]

    df_data = df_registos.drop(["ano"], axis=1)
    geojson = FeatureCollection(features)

    return df_data, geojson
```

For each municipality, the geometric information is mapped together with an associated id.

This geometric information is later used to map the data on the dashboard.

Code Example

Data Loading

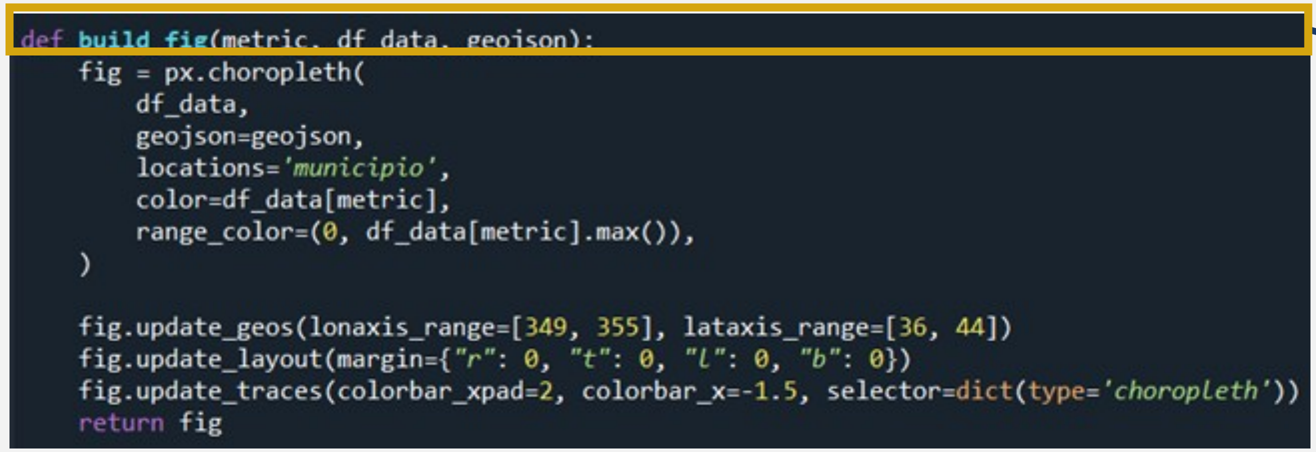


```python
def get_data():
    df_registos = pd.read_csv("data/data_old.csv", sep=";")
    df = geopandas.read_file("data/concelhos.shp")

    df_registos = df_registos[df_registos["ano"] == 2017]
    df_registos["incidencia"] = (df_registos["nr_crimes"]
                                / df_registos["populacao_residente"]) * 100000

    dbf = Dbf5("data/concelhos.dbf")
    df_meta = dbf.to_dataframe()
    df_meta["CCA_2"] = df_meta["CCA_2"].astype(np.float64)

    municipalities = len(df)

    features = [{"type": "Feature",
                 "geometry": (mapping(df.iloc[i]["geometry"].simplify(tolerance=0.01))),
                 "id": df_registos[df_registos["codigo_municipio"] == df_meta.iloc[i]["CCA_2"]]["municipio"]}
                for i in range(municipalities)]

    df_data = df_registos.drop(["ano"], axis=1)
    geojson = FeatureCollection(features)

    return df_data, geojson
```

In the last step, the year column is dropped from the data, and the geometric information together with the data is returned by the function.

Code Example

Plotting



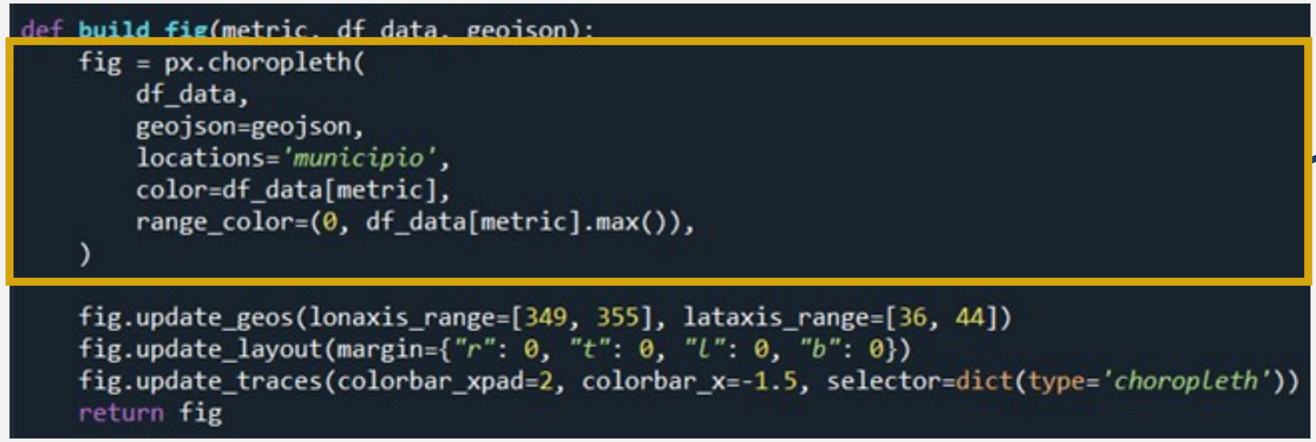
```python
def build_fig(metric, df_data, geojson):
    fig = px.choropleth(
        df_data,
        geojson=geojson,
        locations='municipio',
        color=df_data[metric],
        range_color=(0, df_data[metric].max()),
    )

    fig.update_geos(lonaxis_range=[349, 355], lataxis_range=[36, 44])
    fig.update_layout(margin={"r": 0, "t": 0, "l": 0, "b": 0})
    fig.update_traces(colorbar_xpad=2, colorbar_x=-1.5, selector=dict(type='choropleth'))
    return fig
```

This code utilizes the **pyplot express package (px)** to create a map displaying the data easily.

This function expects a string specifying which metric is to be visualized on the map. The preprocessed dataframe and the geometric information are contained in geojson format.

Code Example

Plotting



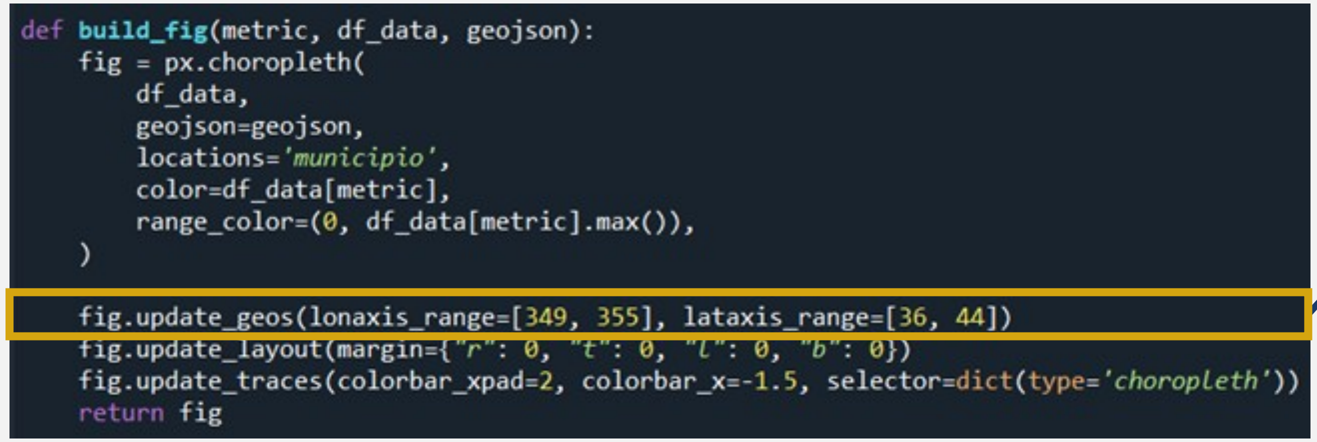
```
def build_fig(metric, df_data, geojson):
    fig = px.choropleth(
        df_data,
        geojson=geojson,
        locations='municipio',
        color=df_data[metric],
        range_color=(0, df_data[metric].max()),
    )

    fig.update_geos(lonaxis_range=[349, 355], lataxis_range=[36, 44])
    fig.update_layout(margin={"r": 0, "t": 0, "l": 0, "b": 0})
    fig.update_traces(colorbar_xpad=2, colorbar_x=-1.5, selector=dict(type='choropleth'))
    return fig
```

Here, the plotly package is provided with:

- Dataframe & geojson
- What locations are to be plotted (here municipalities)
- Which metric is to be displayed (color = …)

Code Example

Plotting

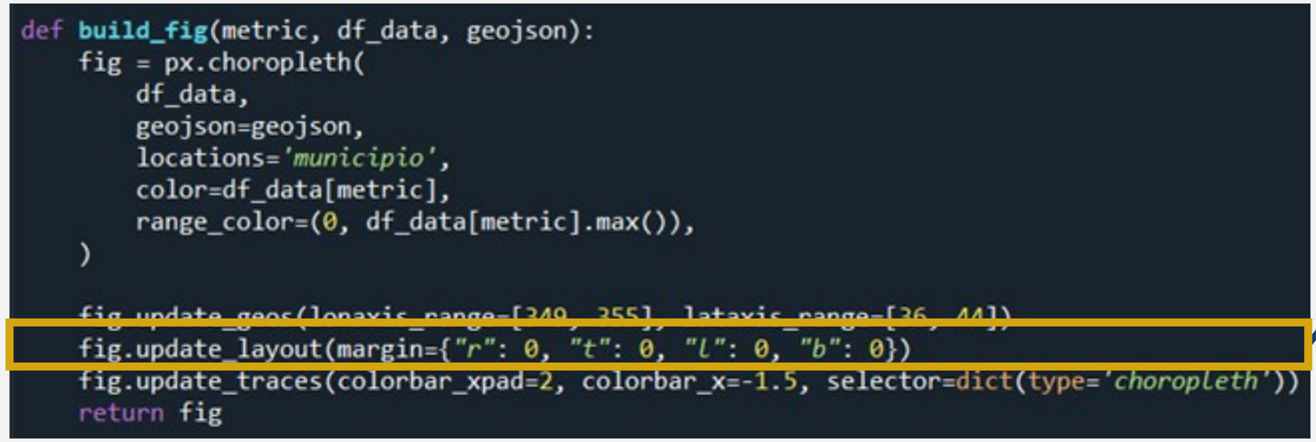


```
def build_fig(metric, df_data, geojson):
    fig = px.choropleth(
        df_data,
        geojson=geojson,
        locations='municipio',
        color=df_data[metric],
        range_color=(0, df_data[metric].max()),
    )

    fig.update_geos(lonaxis_range=[349, 355], lataxis_range=[36, 44])
    fig.update_layout(margin={"r": 0, "t": 0, "l": 0, "b": 0})
    fig.update_traces(colorbar_xpad=2, colorbar_x=-1.5, selector=dict(type='choropleth'))
    return fig
```

This line moves the boundaries of the plot on „**Portugal**".

Code Example

Plotting

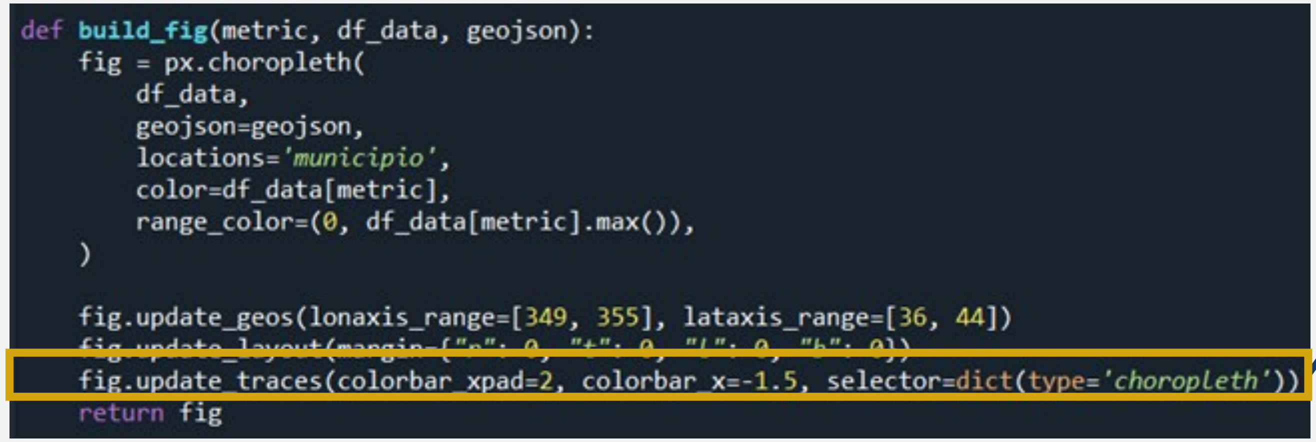


```python
def build_fig(metric, df_data, geojson):
    fig = px.choropleth(
        df_data,
        geojson=geojson,
        locations='municipio',
        color=df_data[metric],
        range_color=(0, df_data[metric].max()),
    )

    fig.update_geos(lonaxis_range=[349, 355], lataxis_range=[36, 44])
    fig.update_layout(margin={"r": 0, "t": 0, "l": 0, "b": 0})
    fig.update_traces(colorbar_xpad=2, colorbar_x=-1.5, selector=dict(type='choropleth'))
    return fig
```

To make the plot look cleaner, margins are removed.

Code Example

Plotting



```
def build_fig(metric, df_data, geojson):
    fig = px.choropleth(
        df_data,
        geojson=geojson,
        locations='municipio',
        color=df_data[metric],
        range_color=(0, df_data[metric].max()),
    )

    fig.update_geos(lonaxis_range=[349, 355], lataxis_range=[36, 44])
    fig.update_layout(margin={"r": 0, "t": 0, "l": 0, "b": 0})
    fig.update_traces(colorbar_xpad=2, colorbar_x=-1.5, selector=dict(type='choropleth'))
    return fig
```

Lastly, the position and appearance of the colorbar is adjusted.

Domestic Violence Data Observatory

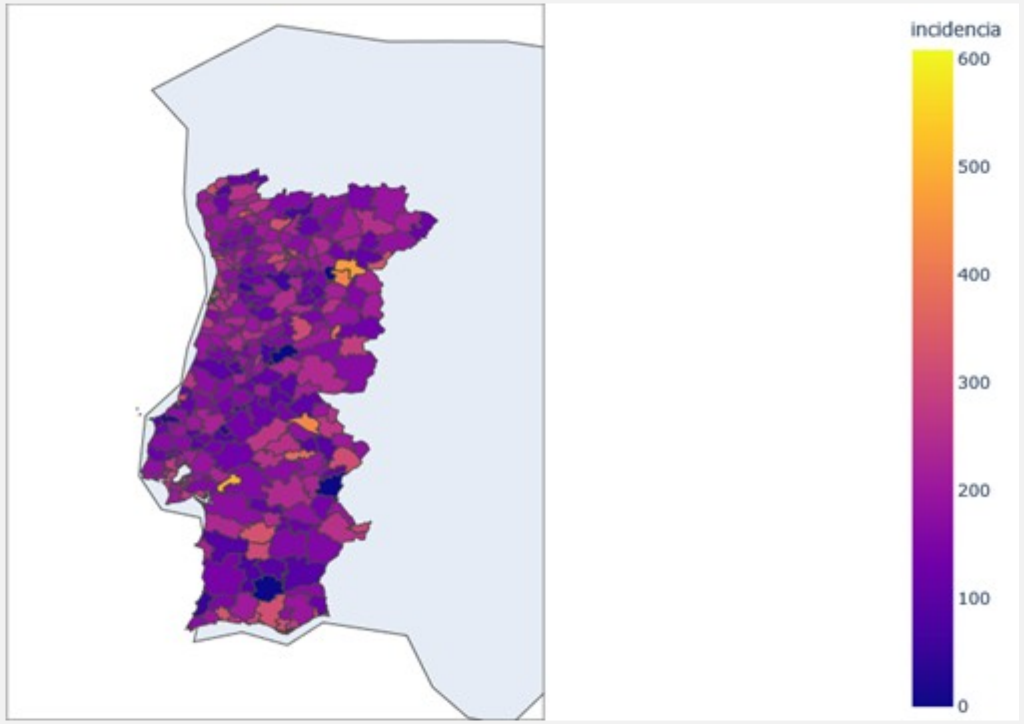
Solution

A dashboard displaying data on domestic violence in Portugal serves as a comprehensive and accessible tool for understanding key aspects of this social issue.

The dashboard integrates data from various sources to provide users with real-time insights, trends, and geographical patterns related to domestic violence.

A prototype of this project is displayed on the right.

The project is accessible here:
https://www.dssg.pt/en/projects/domestic-violence-data-observatory/



Additional Projects

Identifying Fraud & Collusion in International Development Projects

Content from: https://www.dssgfellowship.org/project/identifying-fraud-collusion-in- international-development-projects/

Paper: https://www.dssgfellowship.org/wp-content/uploads/2016/12/world_bank_fraud.pdf

Surveying target groups for interest in a good life for the elderly in rural areas?

Content from: https://www.correlaid.org/en/using-data/project-database/2020-03-DEN/

Identification of causes and optimization of waiting times for veterinary consultations at the AZP veterinary hospital

Content from: https://www.dssg.pt/en/projects/identification-of-causes-and-optimization-of-waiting-times-for-veterinary-consultations-at-the-azp-veterinary-hospital/

Sources I

- Kloppenburg & Moura, K. & Dietrich, J. (2023) CorrelAid/paris-bikes: *Where to build new bicycle parking spots in Paris? Supporting data-driven decision making with open data*. Available online at https://github.com/CorrelAid/paris-bikes/ (last accessed on 04.12.2023).
- Data Gouv France: Where to build new bycicle parking spots in Paris supporting data driven decision making with open data. Available online at https://www.data.gouv.fr/en/reuses/where-to-build-new-bicycle-parking-spots-in-parissupporting-data-driven-decision-making-with-open-data/ (last accessed on 04.12.2023)
- DSSG Portugal (a): Predicting long-term unemployment in Portugal. Available online at https://www.dssgfellowship.org/project/predicting-long-term-unemployment-in-continental- portugal/ (last accessed on 11.11.2024)
- DSSG Portugal (b): Mortality Surveillance. Available online at https://www.dssg.pt/en/projects/mortality-surveillance/ (last accessed on 11.11.2024)
- DSSG Portugal (c): Domestic Violence Data Observatory. Available online at https://www.dssg.pt/en/projects/domestic-violence-data-observatory/ (last accessed on 11.11.2024)

Open Educational Resources