SOP: Running the GEP Model in Jupyter Notebook

1. Introduction

This document provides step-by-step instructions to run the GEP model in Jupyter Notebook. The objective is to simulate energy dispatch scenarios, calculate shadow prices, and generate visualizations for further analysis.

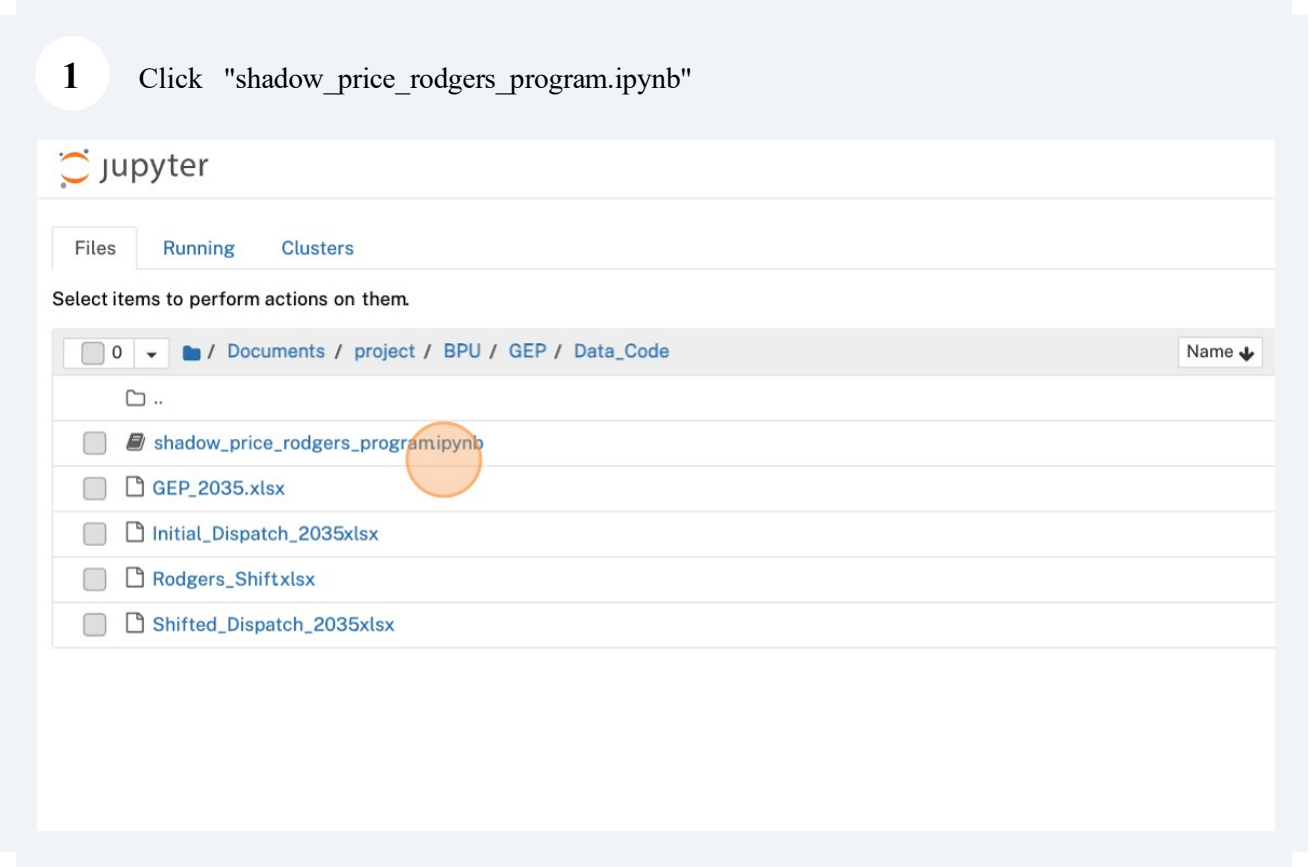
2. Preparing the Environment

1) Install Python and Required Libraries:
   a) Ensure Python 3.9 or above is installed.
   b) Install required libraries.
2) Set Up the Working Directory:
   Place the following data files in the same directory as the Jupyter Notebook:
   a) GEP_2035.xlsx: Input data file for GEP parameters.
   b) Initial_Dispatch_2035.xlsx: Initial energy dispatch data.
   c) Shifted_Dispatch_2035.xlsx: Shifted energy demand profiles.

3. Step-by-Step Instructions

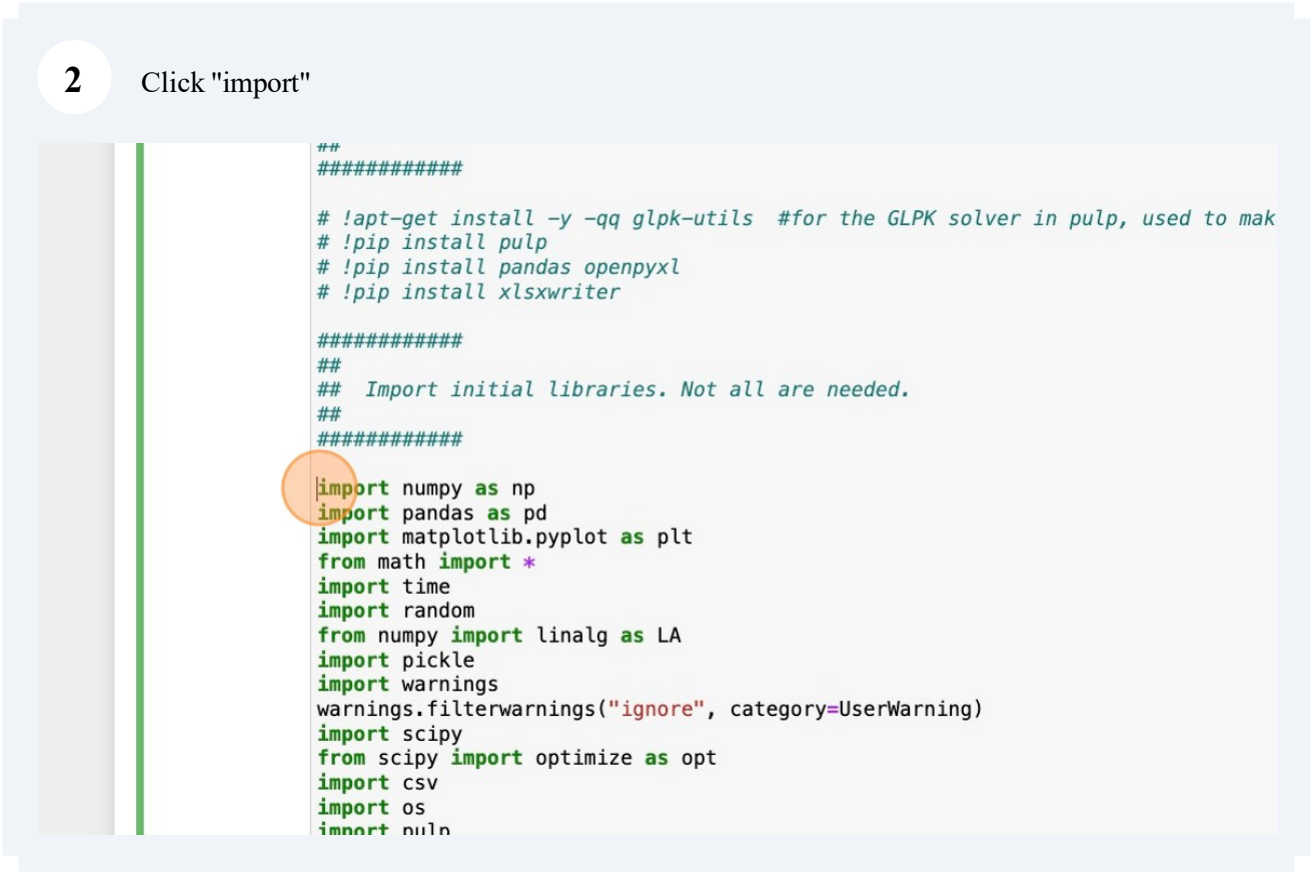
Step 1: Open the Jupyter Notebook

Locate the file shadow_price_rodgers_program.ipynb and open it in Jupyter Notebook.



Step 2: Import Required Libraries

Click the cell labeled import and run it to load all dependencies.

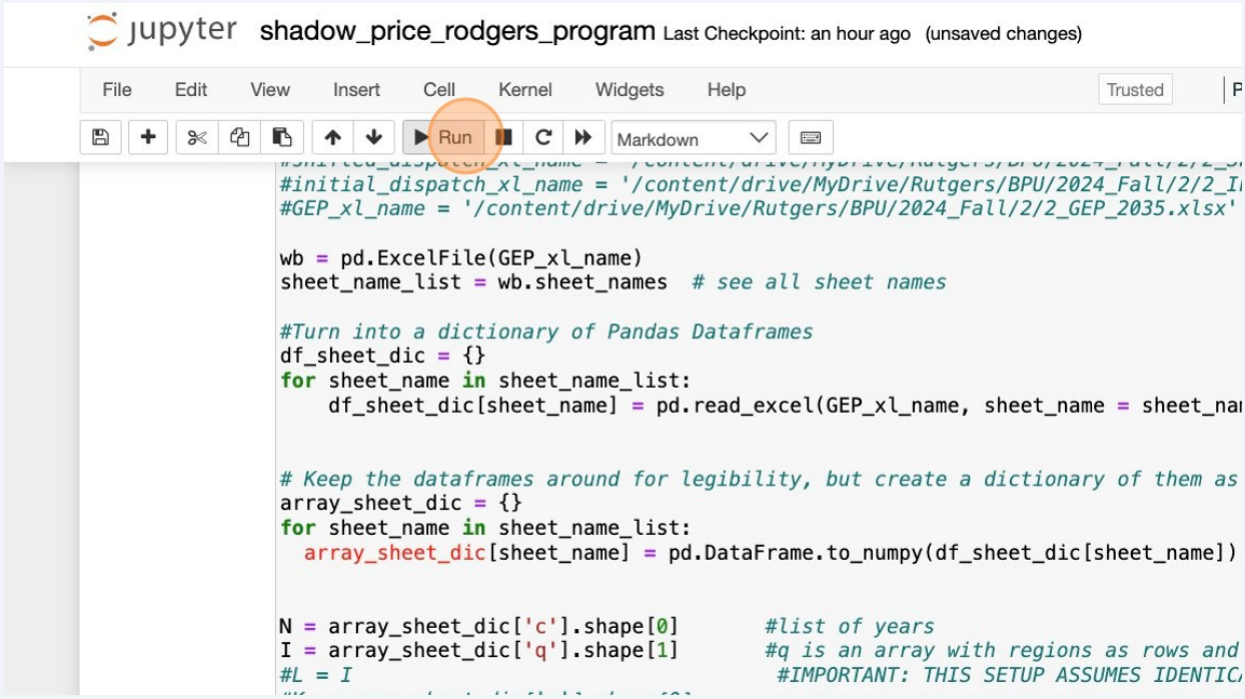


Step 3: This step loads the Excel input files required for the GEP model. The files include:

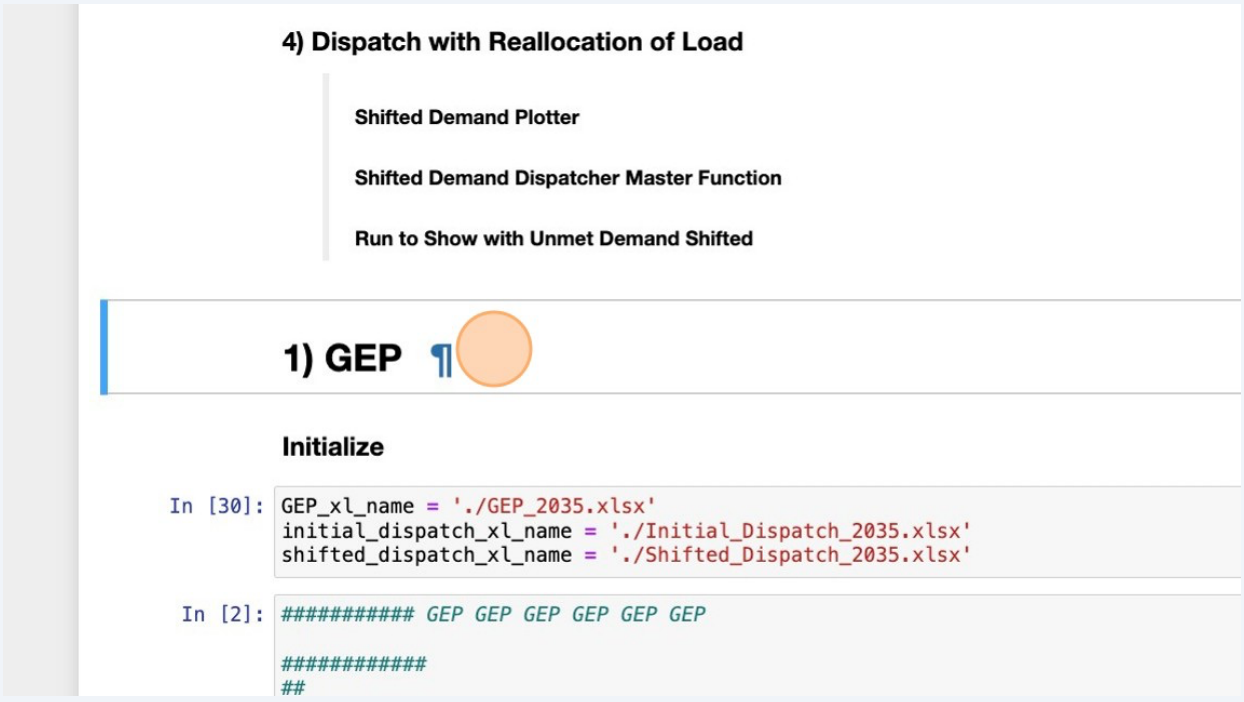
i. `GEP_2035.xlsx`: Contains parameters for the GEP model.
ii. `Initial_Dispatch_2035.xlsx`: Initial energy dispatch schedule.
iii. `Shifted_Dispatch_2035.xlsx`: Data for shifted demand.

Ensure the files are located in the specified directory and have no missing fields.

3 Click "Run"



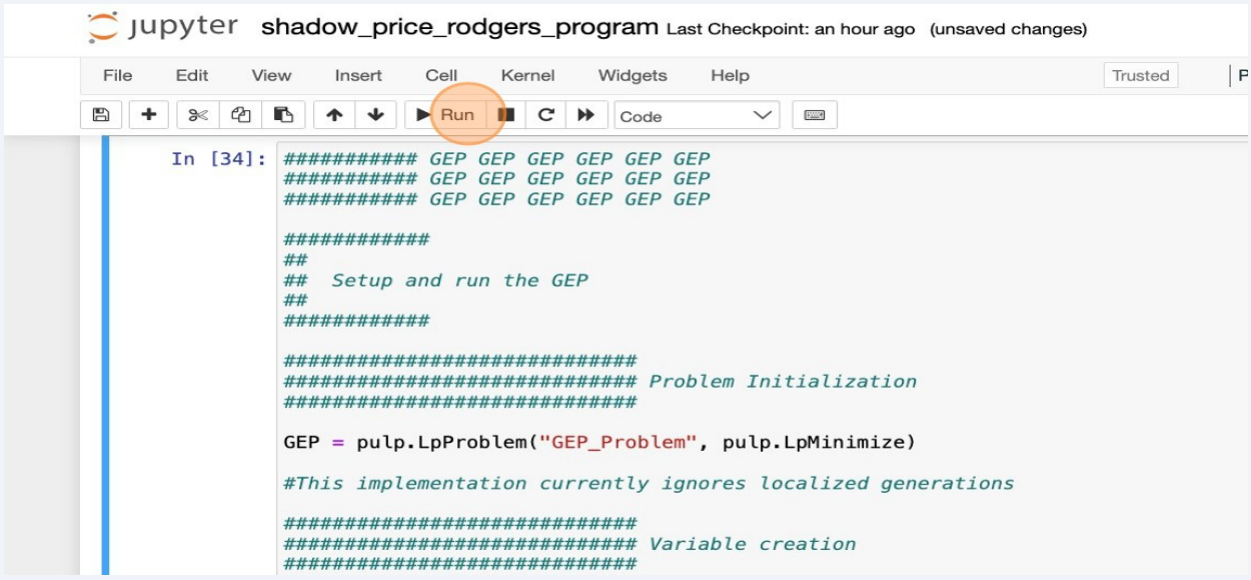
4 Click "1) GEP"



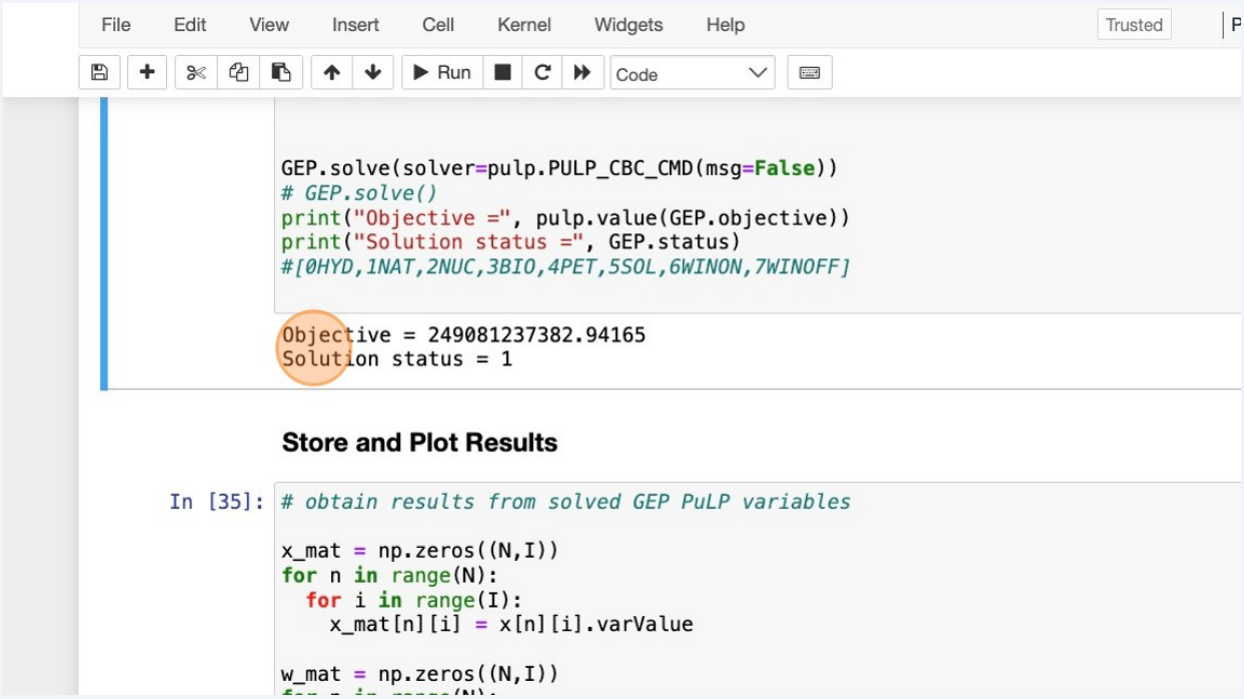
Step 4: Model Optimization Results
After running the optimization model, the key results are displayed directly in the output cell. These results include:

- **Objective Value**: The total cost or value the model aims to optimize.
- **Solution Status**: Indicates whether the optimization was successfully completed (1 = success). These outputs are critical for validating the model's execution and ensuring its feasibility.

5 Click "Run"



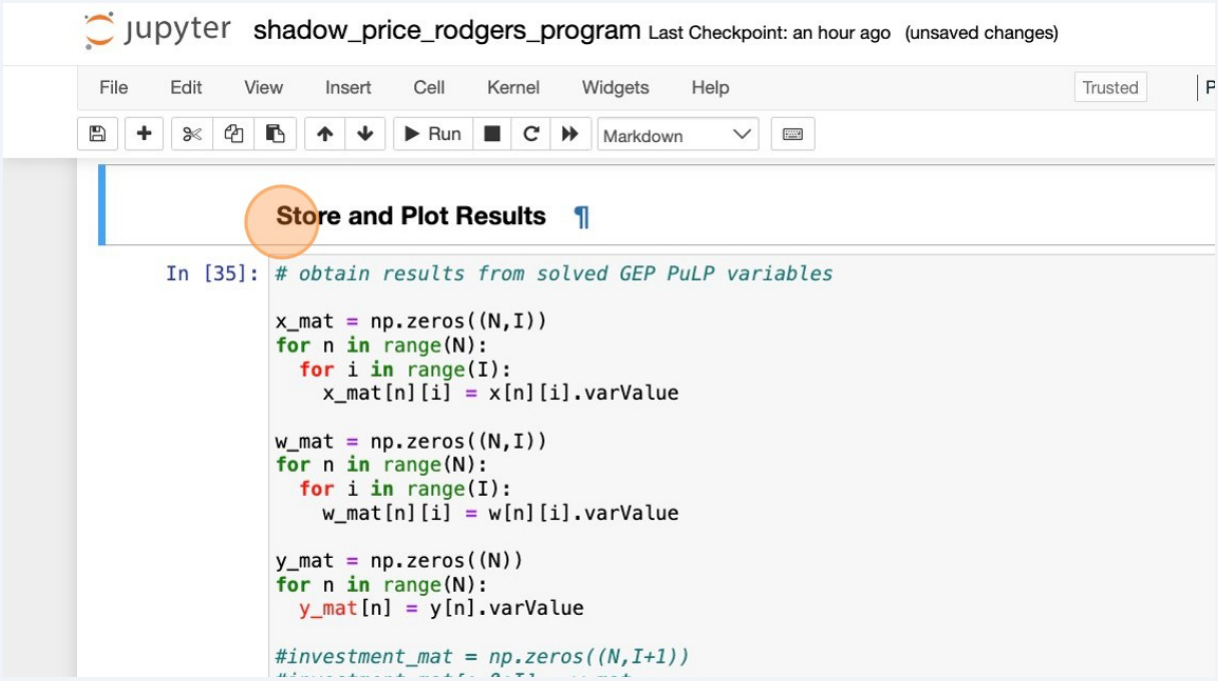
6



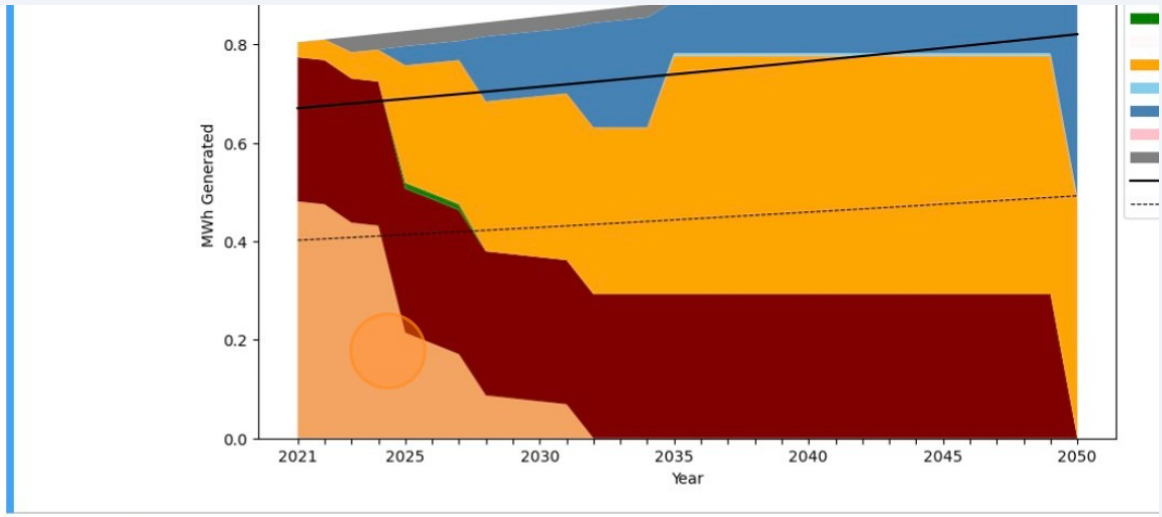
Step 5: Visualization - Generation Timeline
This visualization provides an overview of the power generation timeline broken down by different energy sources.

- The chart shows how each energy source contributes to the total generation over the years.
- It helps identify trends and assess the feasibility of energy policies.

7 Click "Store and Plot Results¶"



8



In [37]:

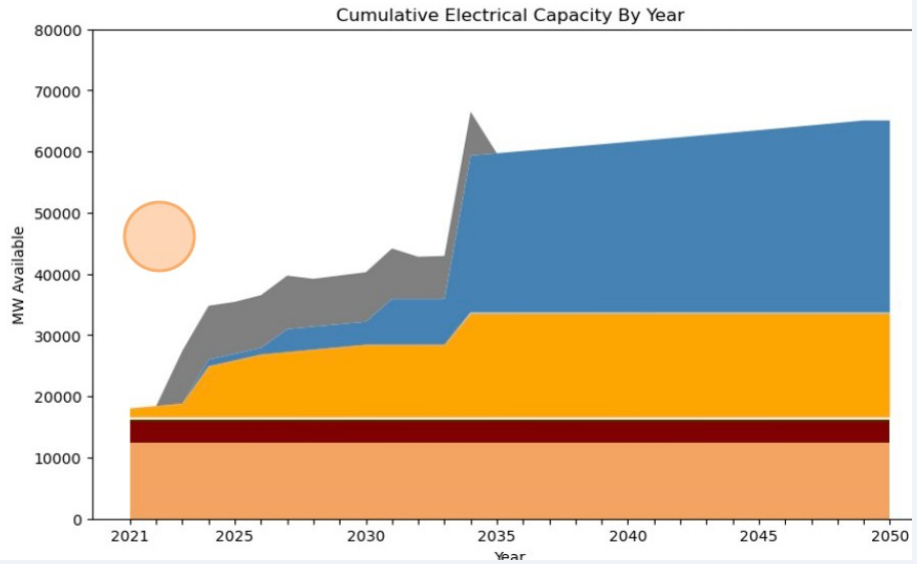
```
############
##
##  Plot the cumulative investments
##
############
```

Step 6: Visualization - Cumulative Capacity
This chart displays the cumulative electrical capacity by year for different energy sources.

- It aggregates the installed capacity over time, providing insights into the overall growth of the energy system.
- Different colors represent various energy sources, offering a clear breakdown of contributions.

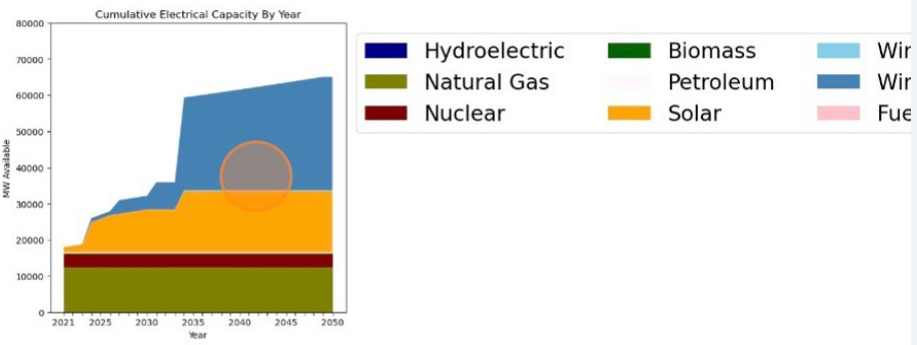
9 Click this image.

```
plt.show()
```



10 Click this image.

```
tech_list = ["Hydroelectric","Natural Gas","Nuclear","Biomass","Petroleum","Sol
#plt.legend(tech_list)
plt.legend(tech_list, loc='upper left', bbox_to_anchor=(1, 1), ncol = 3, prop =
plt.show()
```



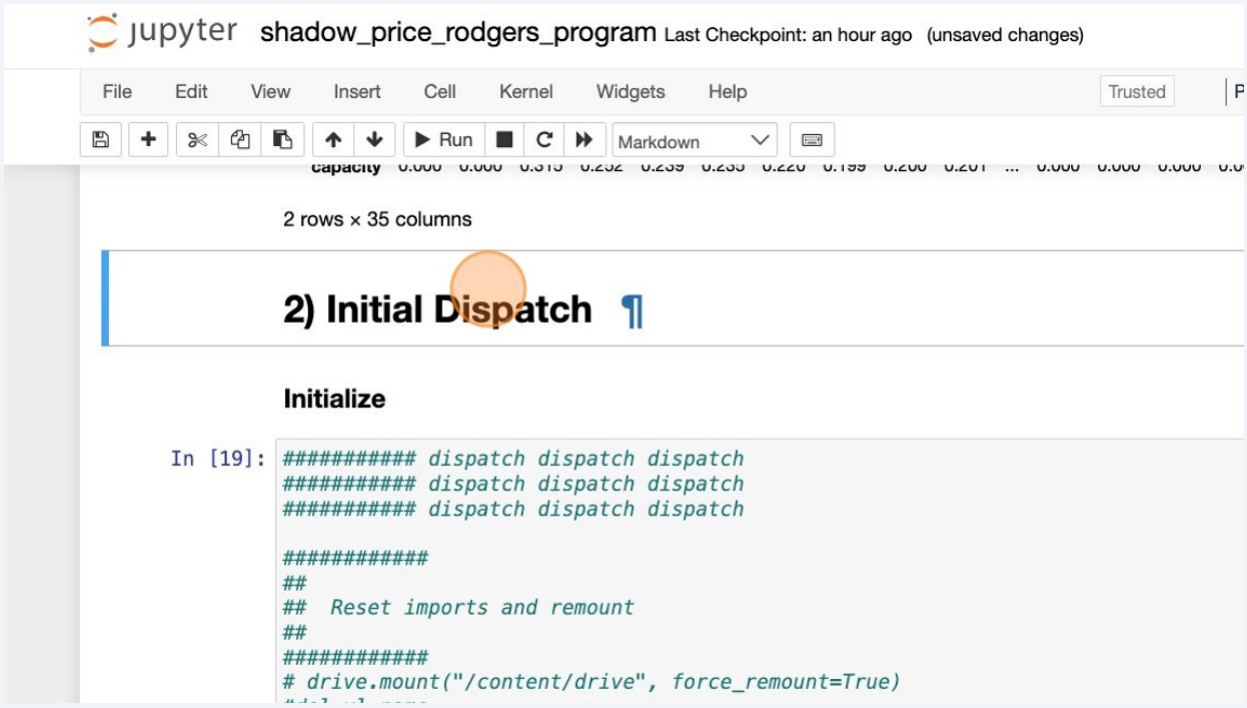

```
In [40]: sum(dic['alpha'][n][i]*w[n][i].varValue for i in range(I) for n in range(N))
Out[40]: 130595512345.50388

In [41]: # Obtain and store proportion of demand recommended to be met by external purch
```

Step 7: Initial Dispatch Module

- The Initial Dispatch module initializes data and parameters for the energy dispatch model.
- This step prepares the required inputs for the GEP model's optimization.
- It ensures all data files are correctly loaded and pre-processed.

11 Click "2) Initial Dispatch"

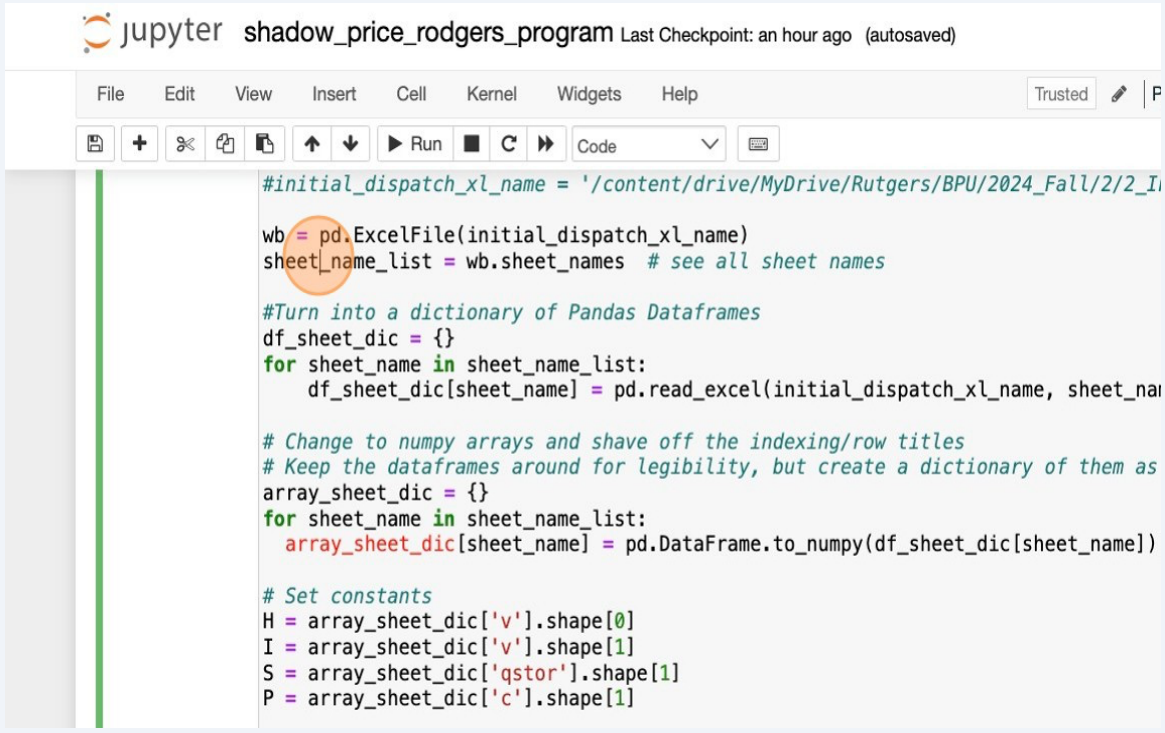


Step 8: Loading Excel Sheets

The model reads Excel sheets containing energy dispatch and demand data.

- The sheet_name_list variable stores the names of all sheets in the workbook.
- These sheets are converted into a dictionary of pandas DataFrames for easier manipulation.

12 Click "sheet_name_list"

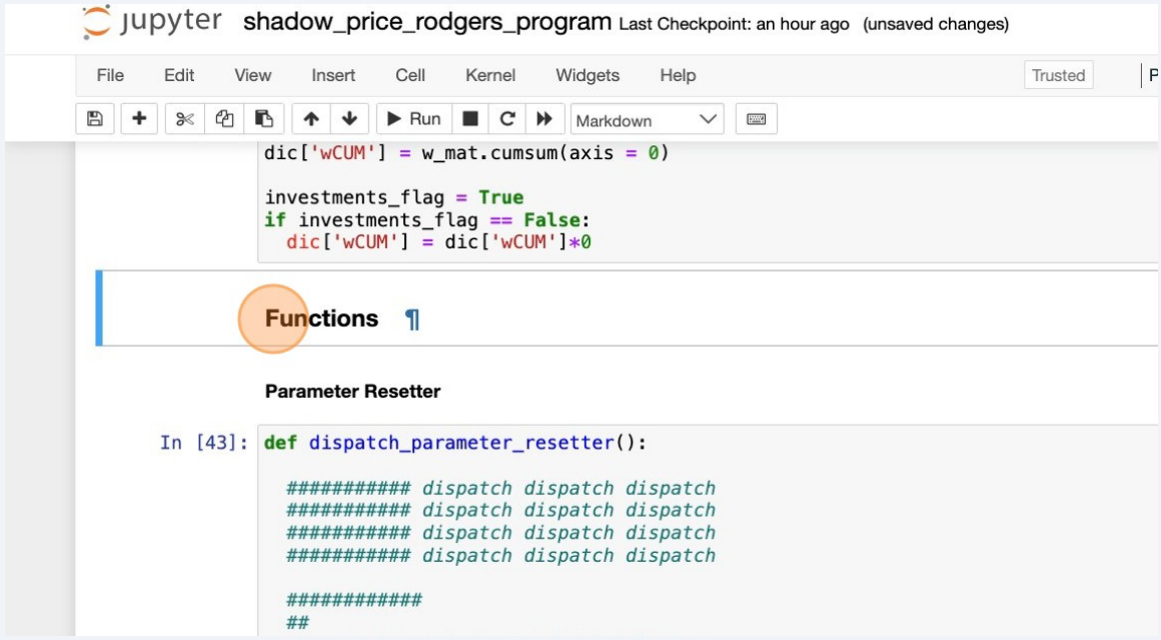


Step 9: Function Initialization

The notebook includes several pre-defined functions to streamline the execution of the GEP model.

- dispatch_parameter_resetter: Resets all model parameters to ensure consistency.
- Additional functions handle data pre-processing and result storage.
- These sheets are converted into a dictionary of pandas DataFrames for easier manipulation.

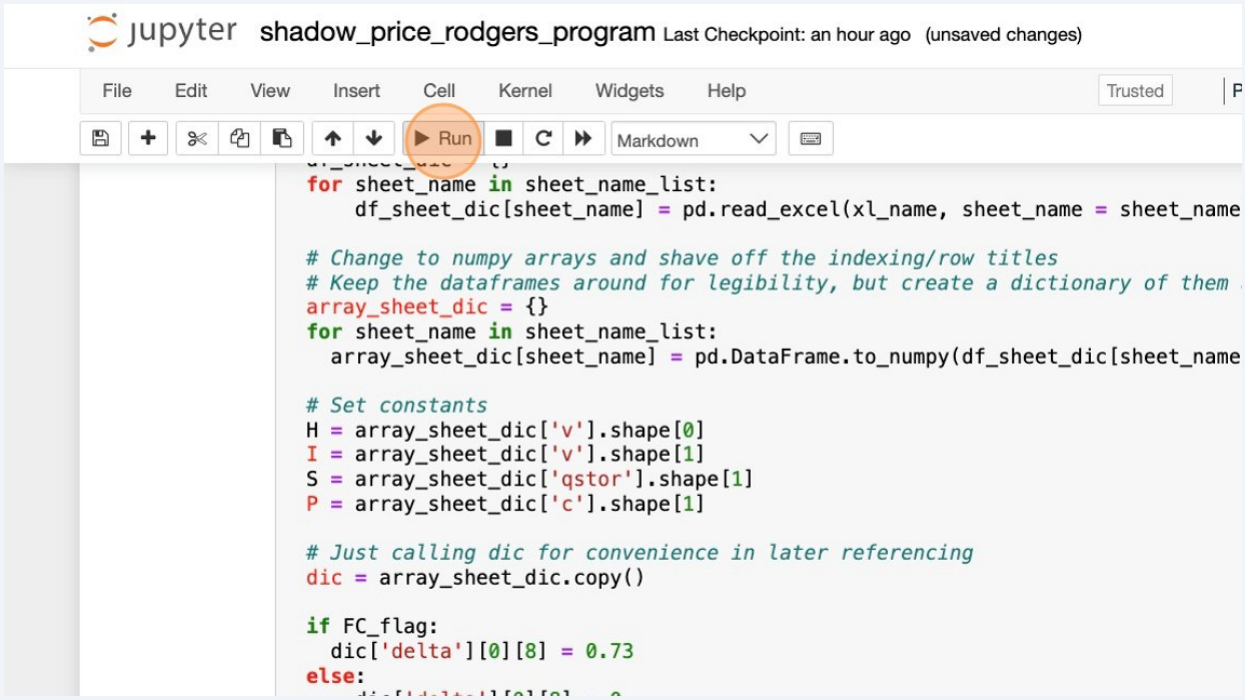
13 Click "Functions¶"



Step 10: Final Module Execution

This step involves executing the final module of the GEP model.
All functions and data have been prepared in the previous steps.
The final module integrates these components and runs the optimization model to completion.

14 Click "Run"

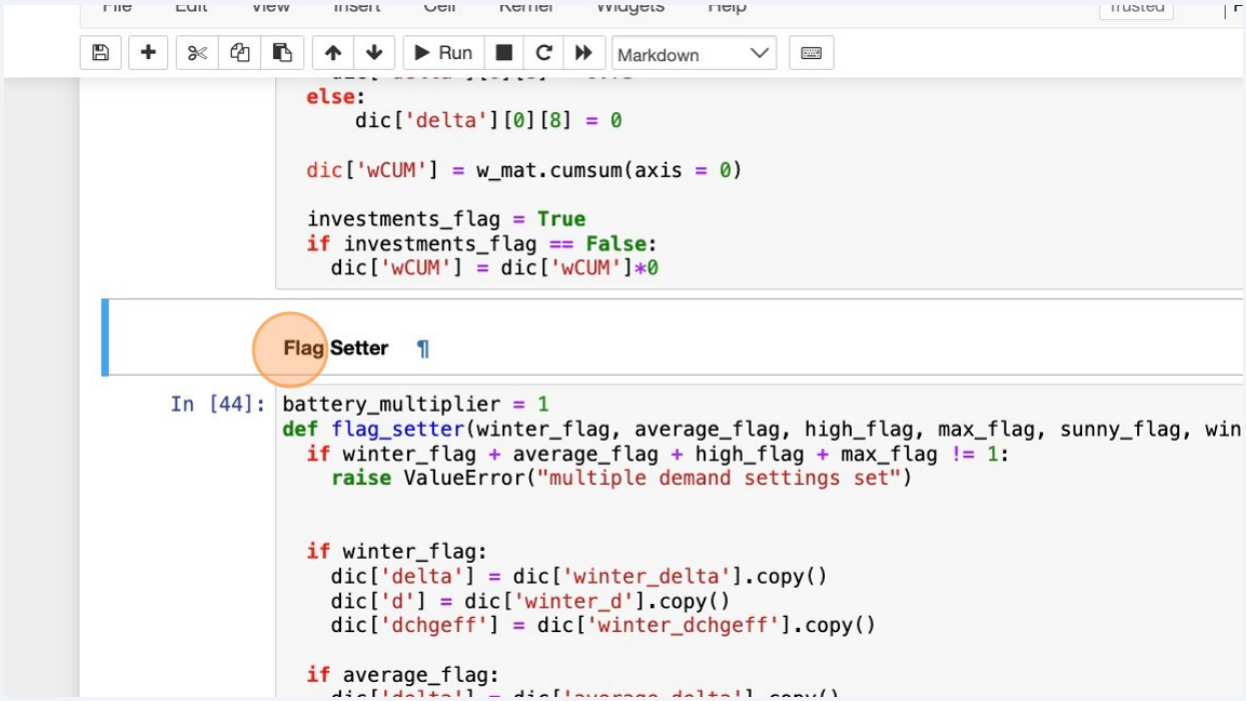


Step 11: Flag Setter

This module defines flags to control different scenarios for the GEP model.

- Flags like winter_flag, sunny_flag, and windy_flag allow users to test specific conditions.
- These flags are used throughout the model to modify constraints or adjust calculations.

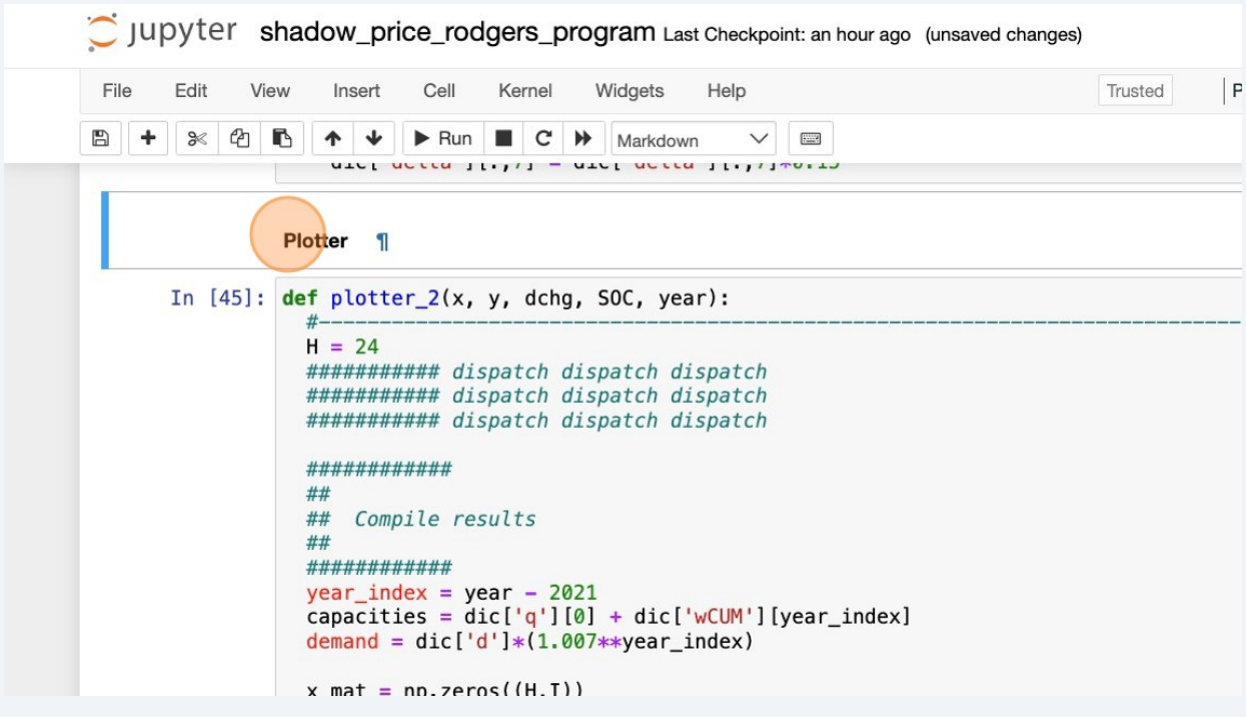
15 Click "Flag Setter¶"



Step 12: Plotter Function
This function is used to generate visualizations of model outputs.

- It plots key metrics, such as dispatch timelines and cumulative capacity, based on the results.

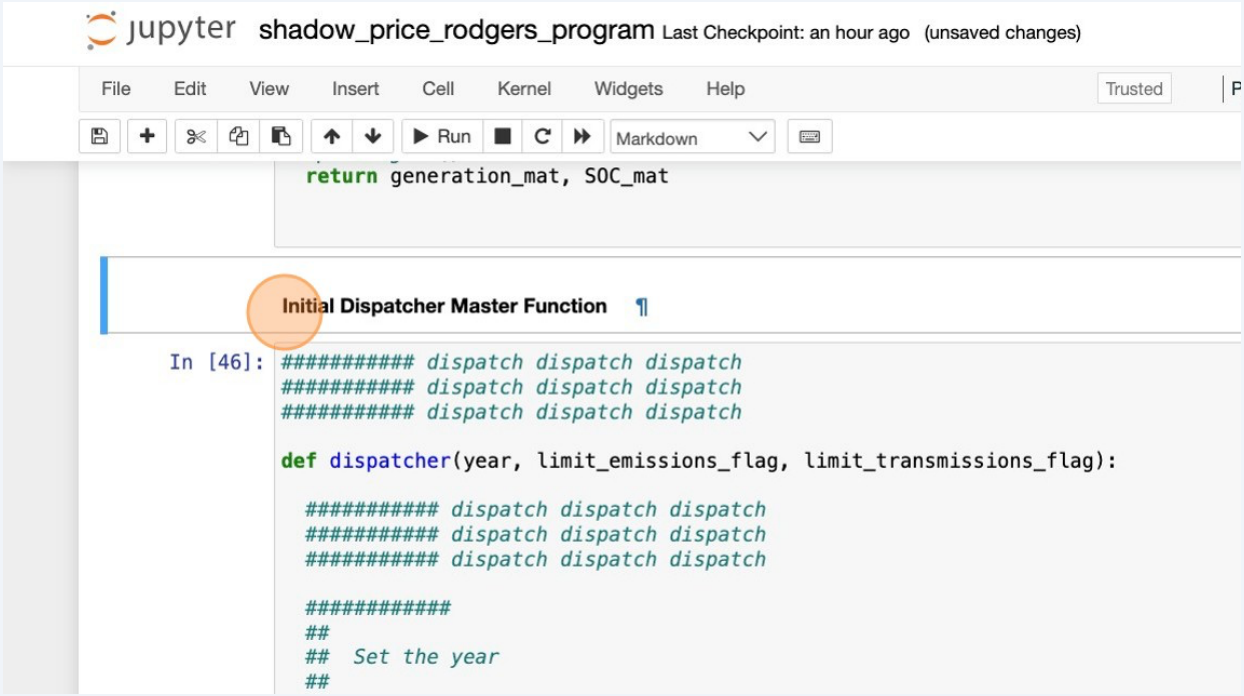
16 Click "Plotter¶"



Step 13: Dispatcher Master Function
This function serves as the main controller for running the energy dispatch model.

- It initializes key parameters and calls sub-functions to compute results.
- The output includes optimized dispatch schedules and shadow prices.

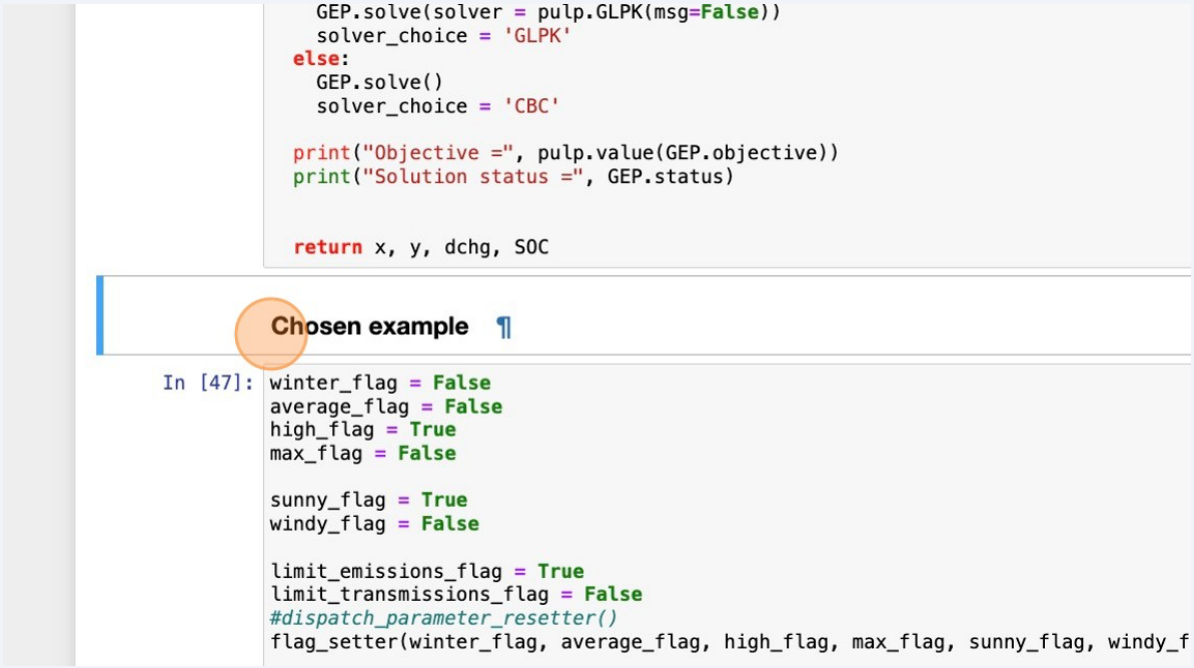
17 Click "Initial Dispatcher Master Function¶"



Step 14: Chosen Example Setup
In this step, the model allows you to set specific conditions and flags for testing scenarios.

- Examples of flags include:
  - winter_flag, sunny_flag: Used to simulate seasonal variations.
  - limit_emissions_flag: Applies constraints to limit emissions.
- The chosen example combines these flags to create a specific scenario for testing the model.

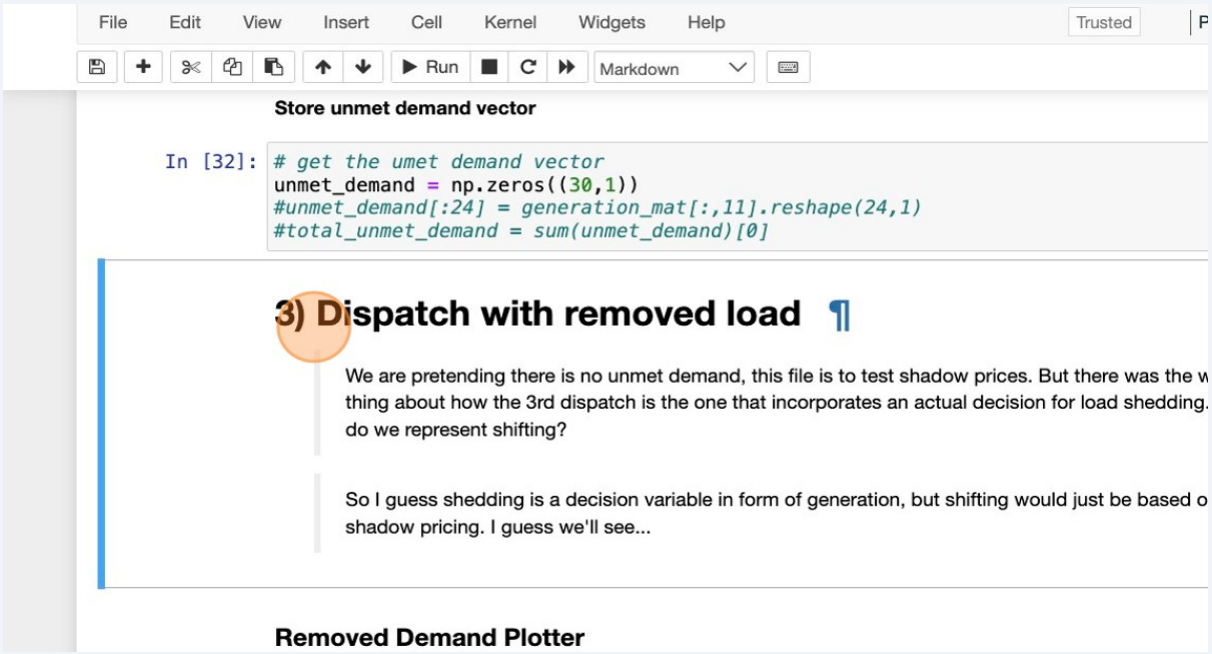
18 Click "Chosen example¶"



Step 15: Dispatch with Removed Load
This module tests the model's response to load removal scenarios.

- It calculates the unmet demand vector, simulating cases where load shedding is necessary.
- The output includes dispatch schedules adjusted to meet these conditions while minimizing unmet demand.

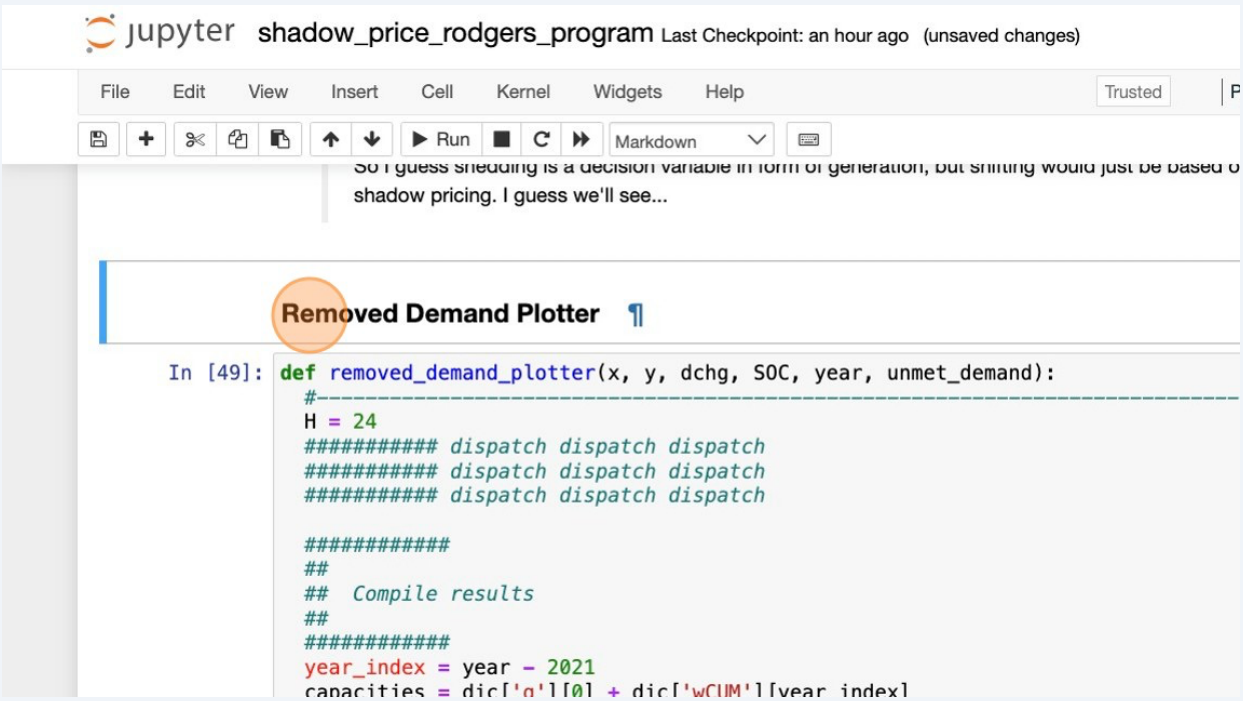
19 Click "3) Dispatch with removed load"



Step 16: Removed Demand Plotter Definition
This function defines the process for visualizing dispatch schedules with removed demand.

- It computes the unmet demand and generates plots showing dispatch changes.
- The function ensures visual representation of how demand is met under specific constraints.

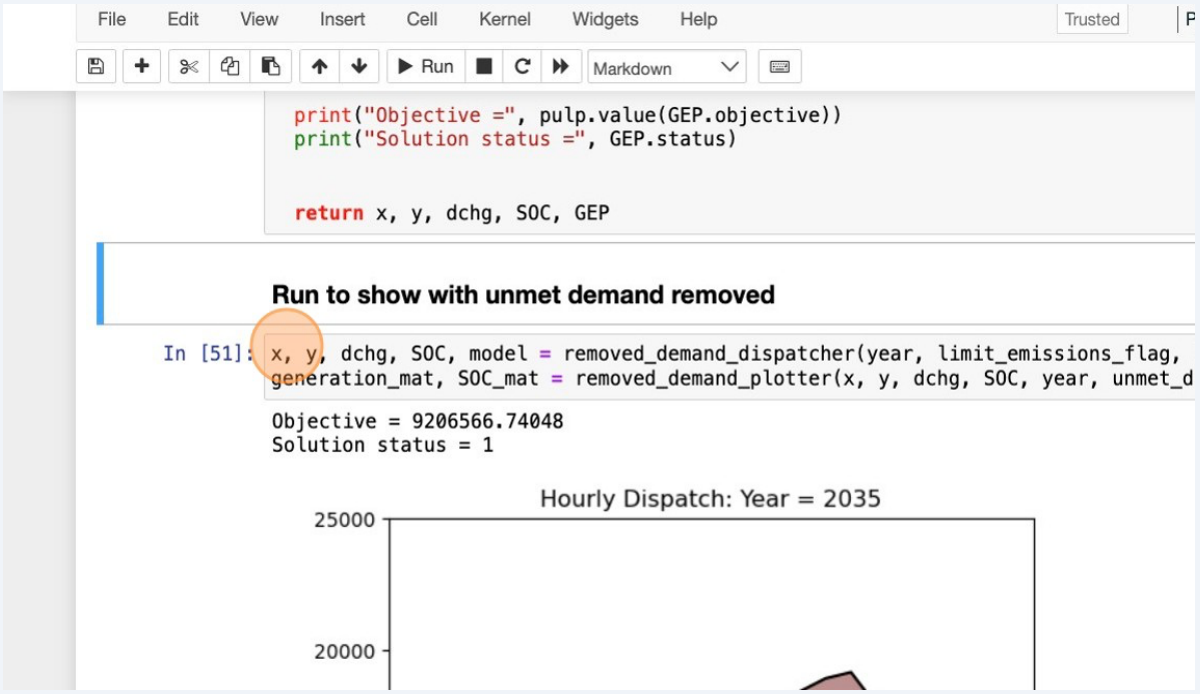
20 Click "Removed Demand Plotter¶"



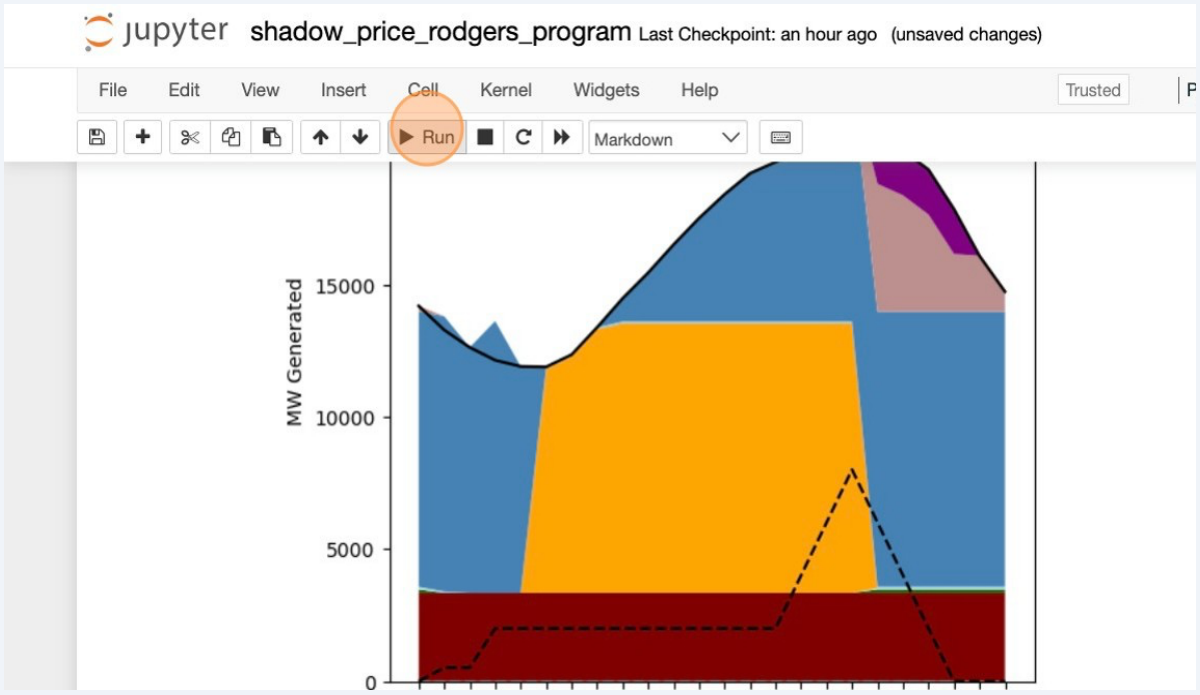
Step 17: Results with Unmet Demand Removed
This step runs the model to analyze scenarios where unmet demand is removed.

- The output includes the optimized objective value and solution status.
- Additionally, a visualization of hourly dispatch results for the year 2035 is generated.

21



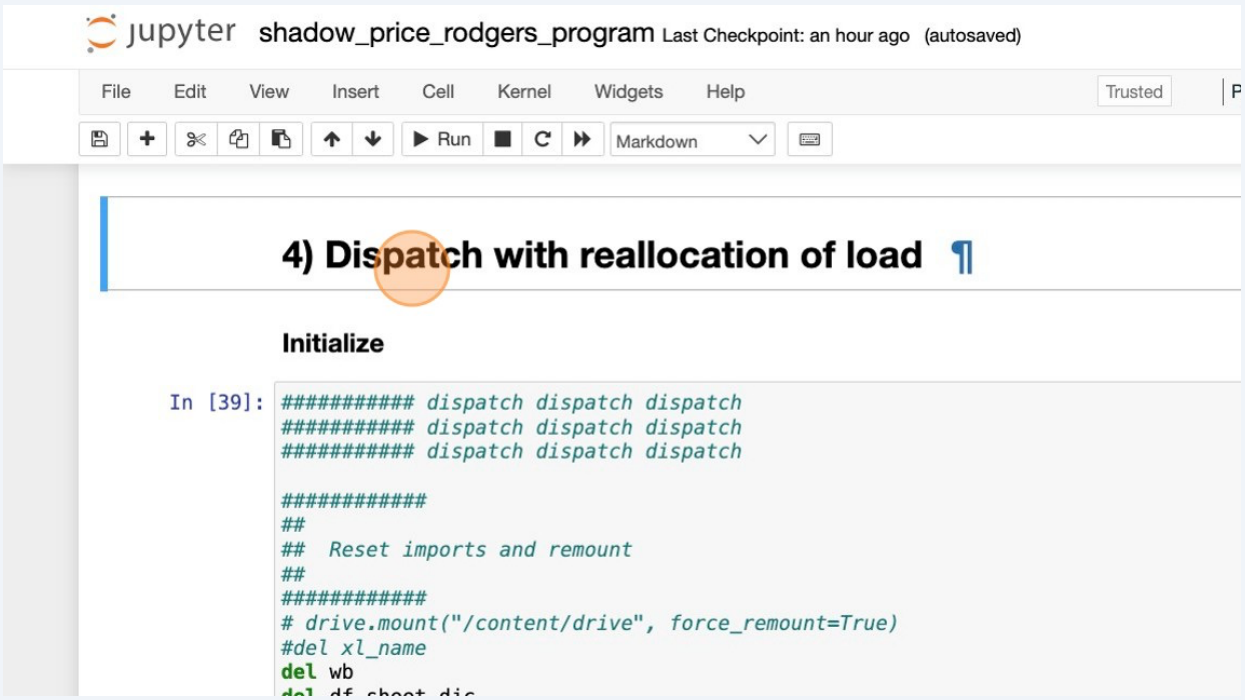
22 Click "Run"



Step 18: Dispatch with Reallocation of Load
This module tests the reallocation of energy dispatch under shifted demand conditions.

- The reallocation process adjusts dispatch schedules to minimize unmet demand.
- This step prepares the model for further analysis and result generation.

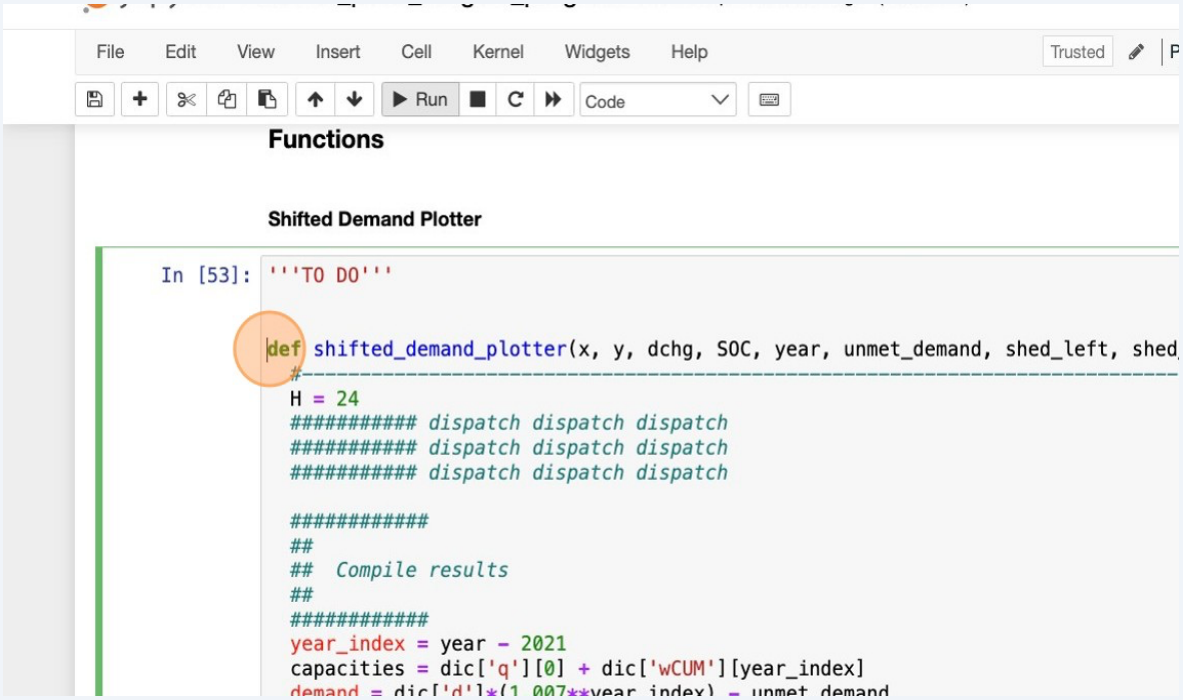
23 Click "4) Dispatch with reallocation of load"



Step 19: Shifted Demand Plotter Function
This function (shifted_demand_plotter) is designed to visualize the impact of shifted demand on energy dispatch.

- It takes key parameters such as unmet demand, shifted load, and system constraints as inputs.
- The output includes plots that represent adjusted dispatch schedules and other critical metrics.

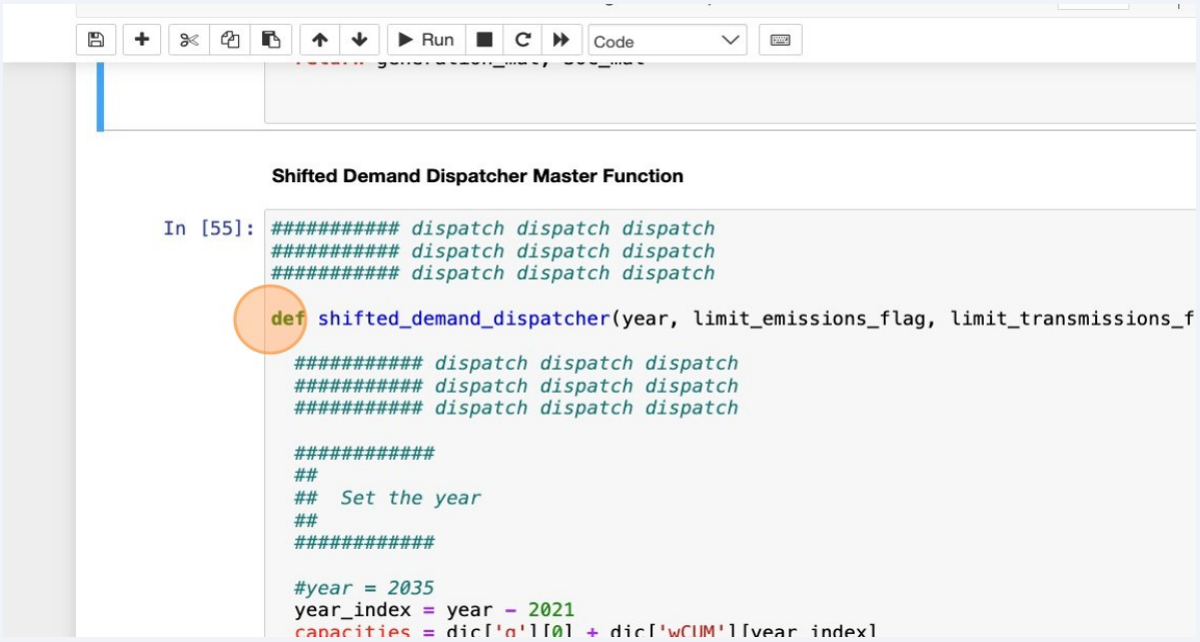
24 Click "def"



Step 20: Shifted Demand Dispatcher Function
This function (shifted_demand_dispatcher) is responsible for handling the reallocation of demand in the dispatch model.

- It modifies the dispatch schedules based on the shifted demand conditions provided.
- The function integrates key constraints such as emissions limits and transmission constraints.

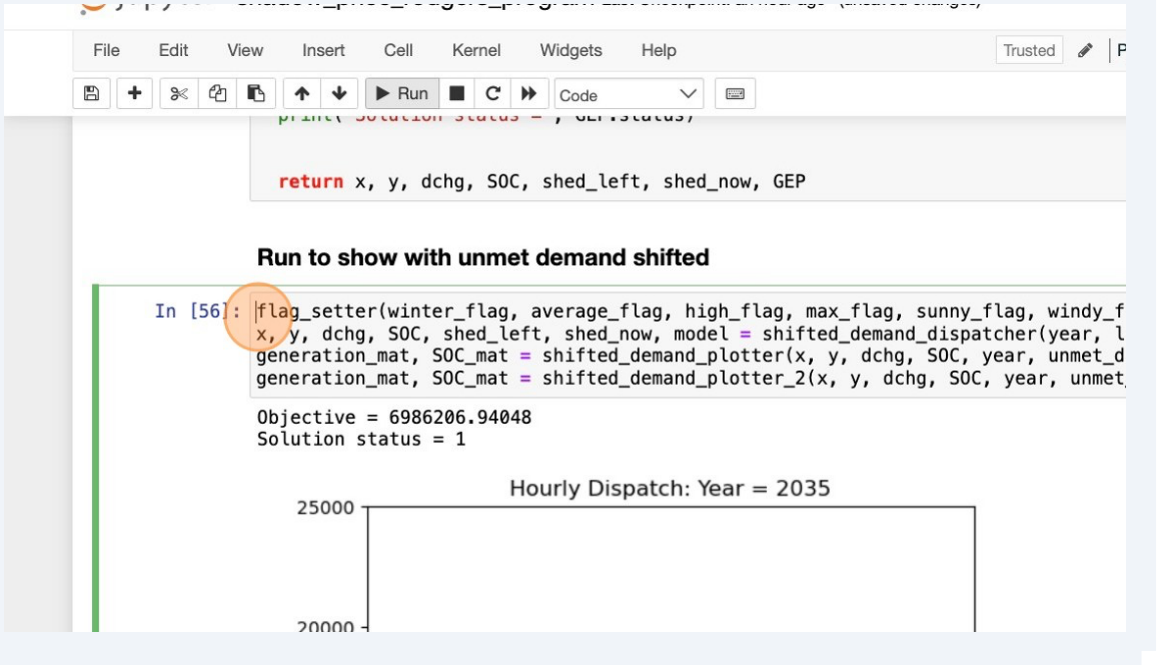
25 Click "def"



Step 21: Results with Shifted Demand Reallocation
This step generates the final results for the reallocated load scenario.

- The output includes the optimized objective value and solution status.
- Additionally, a visualization of hourly dispatch under the shifted demand condition is provided.

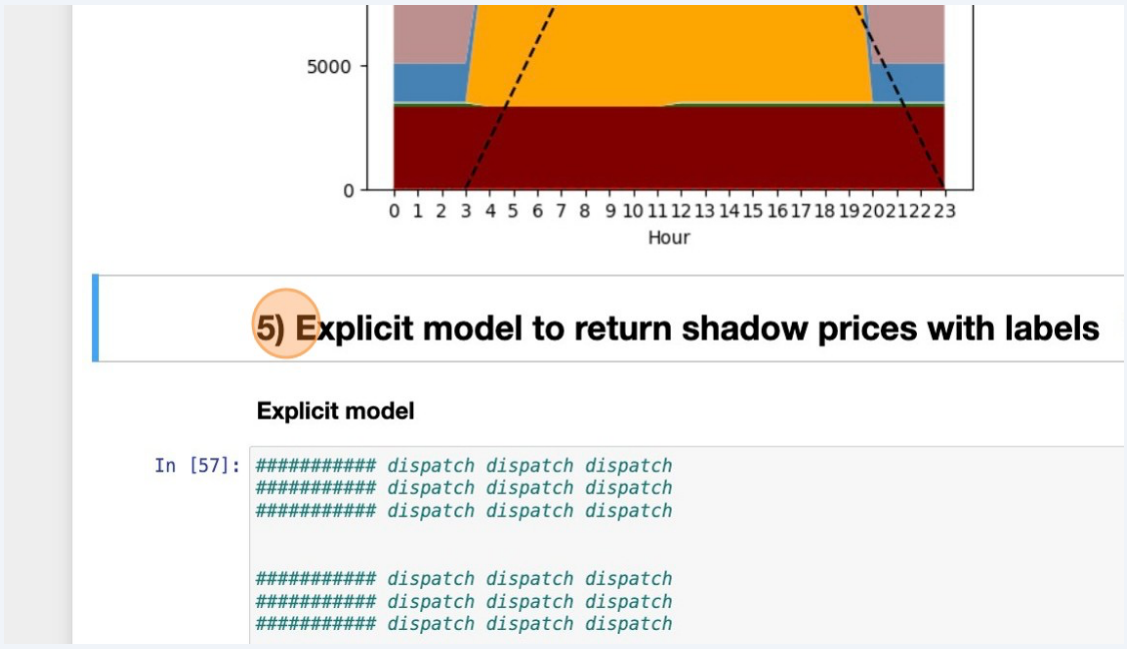
26 Click "flag_setter"



Step 22: Explicit Model for Shadow Prices
This module focuses on calculating shadow prices for various constraints in the dispatch model.

- Shadow prices help quantify the value of relaxing constraints in the optimization problem.
- The module outputs shadow prices labeled for easier interpretation and analysis.

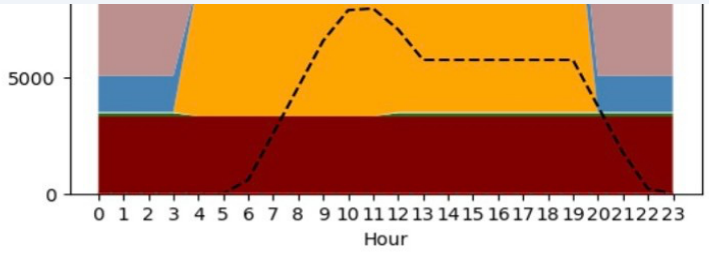
27 Click "5) Explicit model to return shadow prices with labels"



Step 23: Shadow Price Results
This step displays the computed shadow prices for constraints in the dispatch model.

- Each shadow price represents the marginal value of relaxing a specific constraint.
- These values are critical for understanding bottlenecks in the optimization process.

28 Click "run"




```
In [58]: for constraint in dispatch_model.constraints.values():
             print(f"{constraint.name}: Shadow price = {constraint.pi}")
```

```
balance_0: Shadow price = 100.0
balance_1: Shadow price = 100.0
balance_2: Shadow price = 100.0
balance_3: Shadow price = 70.0
balance_4: Shadow price = 0.0
balance_5: Shadow price = 0.0
balance_6: Shadow price = 0.0
balance_7: Shadow price = 0.0
balance_8: Shadow price = 0.0
balance_9: Shadow price = 0.0
balance_10: Shadow price = 0.0
balance_11: Shadow price = 0.0
```

Step 24: Calculation of Shadow Prices Over Time
This step involves creating a time-based array to compute shadow prices associated with balance constraints at each hour.

- time_array is used to index through 24 hours.
- balance_shadow_prices captures the shadow price of maintaining balance in the power grid for each hour.

29 Click "time_array"

```
1,2} + 5.06*x_{11,3} + 4.92*x_{11,4} + 0.62*x_{11,8} + 1.46*x_{12,0} + 1.96*x_{
*x_{12,2} + 5.06*x_{12,3} + 4.92*x_{12,4} + 0.62*x_{12,8} + 1.46*x_{13,0} + 1.9
2.48*x_{13,2} + 5.06*x_{13,3} + 4.92*x_{13,4} + 0.62*x_{13,8} + 1.46*x_{14,0} +
1} + 2.48*x_{14,2} + 5.06*x_{14,3} + 4.92*x_{14,4} + 0.62*x_{14,8} + 1.46*x_{15
_{15,1} + 2.48*x_{15,2} + 5.06*x_{15,3} + 4.92*x_{15,4} + 0.62*x_{15,8} + 1.46*
1.96*x_{16,1} + 2.48*x_{16,2} + 5.06*x_{16,3} + 4.92*x_{16,4} + 0.62*x_{16,8} +
0} + 1.96*x_{17,1} + 2.48*x_{17,2} + 5.06*x_{17,3} + 4.92*x_{17,4} + 0.62*x_{17
_{18,0} + 1.96*x_{18,1} + 2.48*x_{18,2} + 5.06*x_{18,3} + 4.92*x_{18,4} + 0.62*
1.46*x_{19,0} + 1.96*x_{19,1} + 2.48*x_{19,2} + 5.06*x_{19,3} + 4.92*x_{19,4} +
```

```
In [61]: #for constraint in dispatch_model.constraints.values():
         #    print(f"{constraint.name}: Shadow price = {constraint.pi}")

         time_array = np.array(range(0,24))

         balance_shadow_prices = np.zeros(24)
         for time in time_array:
           balance_shadow_prices[time] = dispatch_model.constraints[f'balance_{time}'].p

         dfer(balance_shadow_prices)
```

Out[61]:

	0
0	100.000
1	100.000
2	100.000