

**NURO Analysis Code Manual**

Table of Contents

**Requirements2**

**File Setup3**

**Code Execution4**

Locating the Code4

Running Local App5

Deploying To Server5

**Updating Tests6**

Editing Code Manually6

Using modifyTests.py7

Changing JSON Files9

**Adding Data Columns11**

**Available Data12**

**Potential Future Improvements14**

**Commands and Terminology15**

**Resources16**

**Requirements**

**Languages**: Python (version 3.9.6), JavaScript, CSS, HTML

**Virtual Environment/Server**: use virtual environment to download all libraries when running app locally; see resources on more information on virtual environments and Azure servers

**Libraries**: Flask, Werkzeug, Requests, Pandas, meteostat\*\*

**Minimum Usage**: 30 users at one time; average size of GET request is 330,000 bytes

**Version 1.0:**

* Status: deployed; tested and debugged
* Latest Update: added Help Page
* Contents: new UI, no weather data, hard-coded tests
* Link: <https://github.com/elizabethpursell/Boiler-Web-App-Deploy>

**Version 2.0:**

* Status: not deployed; current version; working consistently
* Latest Update: added weather data and Test 14; used modifyTests.py to dynamically update tests
* Contents: weather data using meteostat library + mapquest API, Test 14, daily weather chart; modifyTests.py; test execution using JSON files
* Link: <https://github.com/elizabethpursell/Boiler-Web-App-Current>

**API Credentials\*\***: store in config.py for security

* MapQuest: gets coordinates for weather data; large usage limit
  + Request URL (coordsAPI): “<https://www.mapquestapi.com/geocoding/v1/batch>”
* Visual Crossing: gets weather data; unused; replaced by meteostat because usage limits
  + Request URL: “[https://weather.visualcrossing.com/VisualCrossingWebServices/ rest/services/timeline/](https://weather.visualcrossing.com/VisualCrossingWebServices/%20rest/services/timeline/)”
* Tomorrow.io: gets weather data; unused; replaced by meteostat because usage limits
  + Request URL: “<https://api.tomorrow.io/v4/historical>”

**\*\* Only available for Version 2.0**

**File Setup**

**NuroConnect**: functions to get data from NURO Connect API

* **\_\_init\_\_.py**: creates NuroConnect object to get and store data from API

**Static**:

* **CSS**: each page is styled by its own css file; mainly used for positioning, coloring, sizing
* **IMG**: all images are stored here, specifically for the home and help pages
* **JS**: each page is run by its own JavaScript file; used to display data to webpage, generate graphs, execute button functions
  + **analyzeData.js**: creates HTML for each test
  + **generateGraph.js**: generates test graph when its button is pressed
  + **graphFunctions.js**: functions to create info popups
  + **graphOptions.js**: functions for all graph features
  + **availableBoilers.js**: creates HTML for each boiler
* **PDF**: holds NURO Analysis Guide
* **PY**: all Python files that are used for running tests are stored here
  + **analyzeRecentData.py**: runs each test
  + **createTest.py\*\***: gets error/warning sets and other test setup options
  + **executeTest.py\*\***: gets graph data points and styles for each test
* **TXT**: files for boiler models, states, and statuses

**Templates**: each page is setup by its own HTML file; base.html is the parent for all pages

**App.py**: sets up all flask app routes to render HTML for each page and get data from API

**Requirements.txt**: holds all libraries that need to be downloaded to run app; created using pip freeze > requirements.txt; download libraries using pip install –r requirements.txt

**Config.py\*\***: stores API keys and request URLs; use variables in \_\_init\_\_.py for API calls

**ModifyTests.py\*\***: generates JSON files for sets and tests based on user input; eliminates hardcoded tests and sets

**setOptions.json\*\***: file generated by modifyTests.py to hold available sets for analysis; structure explained in setOptionsStructure.txt

**testOptions.json\*\***: file generated by modifyTests.py to hold all tests options; app reads test data from this file to execute correct tests; structure explained in testOptionsStructure.txt

**\*\* Only available for Version 2.0**

**Code Execution**

**Locating the Code**: follow this tutorial before running locally or deploying to server

**\*\* Requirements**: Git, GitHub

1. Open the Command Prompt/Terminal using the search bar on your computer.
2. Navigate to where you want to copy the code to or where the code is on your computer using the command cd foldername. (ex. cd Desktop would go to the Desktop folder)
3. If needed, copy the code to your current folder.
   1. Using Git:
      1. Open the link to the GitHub Repository that has the desired code.
      2. Press the green <>Code dropdown.
      3. Under the Local tab, select the HTTPS tab and copy the URL.
      4. Back in the terminal, run the command git clone GitHubURL, where GitHubURL is the URL that you copied from GitHub.
   2. Using GitHub Zip File:
      1. Open File Explorer and navigate to the folder where you want the code.
      2. Open the link to the GitHub Repository that has the desired code.
      3. Press the green <>Code dropdown.
      4. Under the Local tab, press the Download ZIP button.
      5. Drag the Zip file from Downloads to the folder where you want the code.
      6. Extract the Zip file.

**Running Local App**: see resources for more information if needed

**\*\* Requirements**: Python

1. Locate the code on your computer.
2. Create a virtual environment. Follow tutorial in resources if needed.
   1. Install virtual environment using pip install –user virtualenv.
   2. Create a virtual environment using python –m virtualenv venv.
   3. Navigate to the venv/Scripts folder using cd venv and cd Scripts.
   4. Activate the virtual environment using activate.bat.
3. Download necessary libraries from requirements.txt using pip install –r requirements.txt in the root directory.
4. Run the app using python app.py.
5. Open the local link that is given after executing the command.

**Deploying to Server**: see resources for more information

**\*\* Requirements**: Python, Git, Azure access

1. Locate the code on your computer and navigate to the app’s root directory/folder in the Terminal/Command Prompt using cd foldername.
2. Login to the Azure Portal here: <https://portal.azure.com/>.
3. On the home page, under Resources, select the NURO Analysis app.
4. Press the Deployment Center button on the sidebar.
5. Copy the Git Clone Uri link.
6. Back in the Terminal, use the command git remote add azure GitCloneUri, where GitCloneUri is the link copied from the Azure Portal, to connect the code to Azure.
7. Use the command git push azure main:master to send the code to Azure.
   1. If it asks for credentials, in the Azure Deployment Center, under the Local Git/FTPS Credentials tab, find the section Application Scope.
   2. Use the local Git username and password as the credentials to finalize the deployment.

**Updating Tests**

**Editing Code Manually\***

1. Open analyzeRecentData.py, located in the static/py folder, using an IDE or Notepad.
2. Scroll to the bottom of the file. Find and copy the last test function.
3. Paste the function at the end of the file and rename it to be relevant to the new test.
4. To create errorSet, which identifies error points, use the same format as the previous. Change the column names, which are surrounded by quotes, as needed. Check the list of available data for possible selections. Use & (and) or | (or) to combine conditions.
   1. Ex) To create error points when oda > 65 and fan > 2000, errorSet = data[(data[“oda”] > 65) & (data[“fan”] > 2000)]
   2. If a duration check needed, use the checkDuration function and pass the errorSet, data, duration in minutes, and overLimit/underLimit based on if you want to trigger an error point if over or under the given duration.
      1. Ex) To create error points when the error condition is met for over 30 minutes at a time, errorSet = checkDuration(errorSet, data, 30, overLimit)
5. To create warningSet, which classifies warning points, use the same procedure as step 5.
6. Assign title to be a description of the error conditions.
7. Use the sets list to hold the column names for the sets to be graphed for the test.
8. To create testOptions, use the same structure as the previous test. Each set in the sets list is a key in the dictionary. Assign a title, axisID, thresholdLabel, thresholdType, thresholdFill, warnVal, thresholdData as needed.
   1. title: legend label for the dataset (str)
   2. axisID: assign left or right axis (str)
   3. thresholdLabel: describe the threshold condition for the dataset info popup (str)
   4. thresholdType: default, custom, exists, or abs (str)
   5. thresholdFill: where the threshold should be filled (str)
      1. if thresholdType default or custom, then above, below, or none
      2. if thresholdType exists, then all
      3. if thresholdType abs, then between (abs<warnVal) or split (abs>warnVal)
   6. warnVal: value that triggers warning point; not used with custom type (float)
   7. thresholdData: data points for threshold; see test1; only for custom type (list)
9. Assign axis labels with axisTitles. Set the left/right keys to their corresponding titles.
10. Set errorMsg to the causes for the test failure. This will display on the error summary.
11. Save the changes to analyzeRecentData.py.
12. Open app.py, located in the app’s route directory, in an IDE or Notepad, and find the function analyzeDataPost has all the test function calls.
13. Using the same format as the rest of the tests, add the function call to the new test.
14. Save the changes and test the program locally.

**\* Only available for Version 1.0**

**Using modifyTests.py\*\***

\*\* At all times, press CTRL + Fn + Pause Break/PgUp or CTRL + C to terminate the program

1. Using the Command Prompt/Terminal and the cd folderName command, navigate to the app’s code folder.
2. Once in the root directory of the app, use the command python modifyTests.py to run the program. This opens the main menu to update test information.
3. Option 1: Add a Test
   1. To select add a test, enter 1 or add into the command line.
   2. Input the information for the new test as prompted and verify as you go.
   3. See testOptionsStructure.txt for more information on the data needed.
      1. Title: description of the error conditions
      2. Sets: choose from the available sets shown; select sets by entering the keywords found in the parentheses next to the set label; the first set entered will be assigned the left axis during the auto generation of axes; input done when the set list is complete.
      3. Duration: checking the duration of error/warning conditions is optional
      4. Error Conditions:
         1. Select a set to initialize a new error condition.
         2. Choose the comparison type for the condition (value compares the set to a number, set compares the set to another set, custom compares the set to another set that is adjusted by a constant).
         3. Enter the values and sets as prompted to create the condition.
            1. For value types, you can decide to compare the absolute value of the set to the value.
            2. For set types, you can select any other set for comparison.
            3. For custom types, you can select any other set, choose the value to adjust the chosen set by, and enter the operation used for the adjustment (+, -, \*, /).

Ex) To make the condition outlet > hx + 5 , select outlet as the initial set, hx as the comparison set, 5 as the adjustment, + as the adjustment operation, and > as the comparison operator.

* + - 1. Input the comparison operator for how you want to compare the two sets/values (<, <=, >, >=, ==, !=).
      2. Verify the final condition. Decide if you want another condition.
      3. If are adding another condition, select if you want to AND or OR the two conditions together. AND creates an error point only if both conditions are true. OR creates an error point if at least one is true. Repeat the error conditions steps for the new condition.
      4. Once all error conditions are added, verify the final condition.
      5. If you chose to check duration, enter the number of minutes for which the error condition needs to occur to create an error point. Then, select if the condition should check for duration that is over or under the value. Verify the final duration condition,
    1. Warning Conditions: follow the same procedure as error conditions
    2. Dataset Labels: verify auto generated labels or enter correct labels
    3. Axes: verify auto generated axes or enter correct axis assignments/titles
    4. Causes: causes for test failure; shown on error summary and help page
    5. Importance: need for fixing test failure; shown on help page
  1. Data is saved to testOptions.json.

1. Option 2: Remove a Test
   1. To select remove a test, enter 2 or remove into the command line.
   2. Enter the number of the test that you want to delete.
   3. Verify the correct removal of the test.
   4. Data is saved to testOptions.json.
2. Option 3: Update Sets
   1. See the Adding Data Columns section.
3. Test the program locally.

**\*\* Only available for Version 2.0**

**Changing JSON Files\*\***

\*\* If field is not required, leave as “”

1. From the root directory, open testOptions.json. It may also be helpful to open testOptionsStructure.txt for further explanation of the JSON structure.
2. To add a new test, copy an existing test, starting from the test number to the end of field called “importance”, including the closing curly brace and the comma.
3. Add a comma after the last test and paste the copied test.
4. See testOptionsStructure.txt for more information on the data needed.
   1. Test Number: change the test number to one more than the last test
   2. Title: description of the error conditions for the new test
   3. Type: default, duration, daily weather, hourly weather, or all weather, depending on what you are testing and what data you are using
   4. Sets: list of the set/column names needed for the test
   5. Error: list of error conditions to create error points; connect multiple conditions using “and” (both conditions are true) or “or” (at least one is true).
      1. Type: set, value, custom, or abs based on what is being compared
         1. Set: compares two sets
         2. Value: compares a set to a value
         3. Custom: compares one set to another that is adjusted by a value
         4. Abs: compares the absolute value of a set to a value
      2. Set1: the set/column name of the first set in the comparison
      3. Set2: compared to set1; only use required fields; leave others as “”
         1. Name: required for set and custom types; set/column name of the set that set1 is compared to
         2. Value: required for value and custom types; for value types, the number that set1 is compared to; for custom types, the constant that set2 is adjusted by
         3. Operation: required for custom types; operation that is used to adjust set2 by value
      4. Compare: <, <=, >, >=, ==, or != based on how set1 and set2 are compared
   6. ErrorDuration: required if duration type
      1. Value: duration to check error condition for
      2. Compare: > or < based on checking if duration is over or under value
   7. Warning/warningDuration: use the same structure as error/errorDuration
   8. Options: dictionary to hold settings for each set; each set in sets gets its own key
      1. Title: legend label for the dataset
      2. AxisID: left or right based on the axis the set should use
      3. Units: the units for the dataset; use “\u00b0” for a degrees symbol
      4. ThresholdLabel: description of threshold shown on dataset info popup
      5. ThresholdType: value, none, set, custom, exists, or abs based on the comparison type; use exists if there is no warning condition for the set
      6. ThresholdFill: above (>), below (<), none, all (exists type), split (abs>val), or between (abs<val) based on warning condition
   9. AxisTitles: assign the left and right axis titles
   10. Causes: causes for test failure; shown on error summary and help page
   11. Importance: need for fixing test failure; shown on help page
5. Save the changes and test the program locally.

**\*\* Only available for Version 2.0**

**Adding Data Columns**

**Updating analyzeRecentData.py**

1. Open analyzeRecentData.py, located in the static/py folder, in an IDE or Notepad.
2. Go to the first function named cleanRawData.
3. Here is where you can add new data columns. All of the relevant data columns are added already, but you can add additional difference columns.
4. Use print(data, data.columns, flush=True) to display the data frame and columns in your terminal when the app is run locally.

**Updating Weather Data\*\***

1. Open \_\_init\_\_.py, which is located in the NuroConnect folder, in an IDE or Notepad.
2. Go to the functions named getDailyWeather and getHourlyWeather, either one can be updated. You can add columns by doing calculation, like Heating Degree Days, or you can add columns to find changes/differences.
3. Save the changes and see the section Applying Changes Using modifyTests.py.

**Applying Changes Using modifyTests.py\*\***

\*\* At all times, press CTRL + Fn + Pause Break/PgUp or CTRL + C to terminate the program

1. Using Command Prompt/Terminal and cd folderName, navigate to the root directory.
2. Execute python modifyTests.py to run the program and open the test update menu.
3. Select Update Sets by entering 3 or update into the command line.
4. Before using this menu, follow the procedure under the section Updating analyzeRecentData.py or Updating Weather Data.
5. Enter the set keyword/column name that you added so the test menu can be updated.
6. Input the legend label for the set.
7. Select the dataset (data, dailyWeather, hourlyWeather) that you added the column to.
8. Choose the measurement type for the column. Default uses imperial units. Setpoint uses imperial units and is used for any column that stores a boiler setpoint. Difference uses imperial units and is used for any column that finds the difference between two columns or the change of one column. Metric uses metric units, typically used with weather data. Metric differences uses metric units, typically used with weather data.
9. Enter the type of units that should be used. Temperature, current, frequency, pressure, height, degrees, percent, speed, or other for custom unit types.
10. If other is chosen, input the custom unit for the column
11. Verify that the units are correct.
12. Data is saved to setOptions.json

**\*\* Only available for Version 2.0**

**Available Data**

\*\* Column name / set keyword in parentheses

**Data Dataset**

* **Temperature**
  + Heat Exchanger Temperature (hx)
  + Outlet Temperature (supply)
  + Inlet Temperature (inlet)
  + Exhaust Temperature (stack)
  + Outdoor Air Temperature (oda)
  + Header Temperature (header)
  + Domestic Hot Water Temperature (dhw)
* **Temperature Set Points**
  + Central Heating Set Point (chsetpoint)
  + Domestic Hot Water Set Point (dhwsetpoint)
  + Domestic Hot Water Tank Set Point (dhwtanksetpoint)
* **Temperature Differences**
  + Inlet/Outlet Temperature Difference (difference)
  + Heat Exchanger Temperature Difference (hx\_diff)
  + Exhaust Temperature Difference (stack\_diff)
* **Current**
  + Flame Signal (flame)
  + Firing Rate (firerate)
* **Current Differences**
  + Flame Signal Difference (flame\_diff)
  + Firing Rate Difference (firerate\_diff)
* **Frequency/Fan**
  + Fan Speed (fan)
  + Fan Speed Difference (fan\_diff)

**DailyWeather Dataset\*\***

* **Imperial Units**
  + Average Temperature (tavgF\_day)
  + Minimum Temperature (tminF\_day)
  + Maximum Temperature (tmaxF\_day)
  + Precipitation (prcpIN\_day)
  + Average Wind Speed (wspdMPH\_day)
  + Average Pressure (presPSI\_day)
* **Metric Units**
  + Average Temperature (tavg)
  + Minimum Temperature (tmin)
  + Maximum Temperature (tmax)
  + Precipitation (prcp)
  + Average Wind Speed (wspd)
  + Average Pressure (pres)
* **Other Data**
  + Heating Degree Days (hdd)
  + Average Wind Direction (wdir)

**HourlyWeather Dataset\*\***

* **Imperial Units**
  + Temperature (tempF\_hour)
  + Dew Point (dwptF\_hour)
  + Precipitation (prcpIN\_hour)
  + Average Wind Speed (wspdMPH\_hour)
  + Average Pressure (presPSI\_hour)
* **Metric Units**
  + Temperature (temp)
  + Dew Point (dwpt)
  + Precipitation (prcp)
  + Average Wind Speed (wspd)
  + Average Pressure (pres)
* **Other Data**
  + Relative Humidity (rhum)
  + Average Wind Direction (wdir)

**\*\* Only available for Version 2.0**

**Potential Future Improvements**

**Using Weather Data**

* Dynamically apply boiler settings based on weather (ie. If temp = 90 °F, then boiler off)
* Add more tests using weather data

**Integrating With NURO Connect**

* Add button in NURO Connect sidebar to open NURO Analysis

**Creating Mobile App**

* All sizing and functions work with mobile; make mobile app version available

**Website Efficiency**

* To speed up Available Boilers page load, create dropdown to select site to reduce amount of API calls. Currently, it iterates over each site and then each boiler.

**Adding Modify Menu to modifyTests.py**

* A modify menu would eliminate the need to manually edit the JSON file
* Function Ideas: change title, change threshold label, change warning value, change error value, change dataset label, change axis label, change duration, swap left and right axes, hide dataset threshold, renumber tests

**Test Error Checking**

* Currently, the error checking for individual test execution failures does not fully remove the graph. It is replaced by the next test’s graph. This is not a problem currently because it only occurs if the weather data cannot be retrieved, and that graph is last in the list.

**Commands and Terminology**

**Command Line**:

* Change Directory/Folder: cd folderName
* Execute Python Code: python filename
* List Files/Folders in Current Directory: dir (Windows), ls (Linux)
* Install Library: pip install libraryName

**Git Commands**

* Cloning GitHub Repository: git clone GitHubURL
* Add File/Folder To Repository: git add fileName
* Save Code Changes To Repository: git push

**Terminology**:

* Directory: folder
* Current Working Directory: the folder you are currently in
* Root Directory: folder where all the app’s code is stored

**Resources**

**Virtual Environment**: <https://stackoverflow.com/questions/48911582/virtualenv-to-path-on-windows-10>

**Running Flask Apps Locally**: <https://stackoverflow.com/questions/17309889/how-to-debug-a-flask-app>

**Windows Command Line**: <https://serverspace.us/support/help/windows-cmd-commands-cheat-sheet/>

**Deploying to Azure**: <https://learn.microsoft.com/en-us/azure/app-service/quickstart-python?tabs=flask%2Cwindows%2Cazure-portal%2Clocal-git-deploy%2Cdeploy-instructions-azportal%2Cterminal-powershell%2Cdeploy-instructions-zip-azcli>

**Bootstrap 5 Documentation**: <https://getbootstrap.com/docs/5.0/getting-started/introduction/>

**Meteostat Python Library**: <https://dev.meteostat.net/python/>

**MapQuest Geocoding API**: [https://developer.mapquest.com/documentation/ geocoding-api](https://developer.mapquest.com/documentation/%20geocoding-api)

**Visual Crossing Weather API**: <https://www.visualcrossing.com/resources/documentation/> weather-api/timeline-weather-api/

**Tomorrow.io Weather API**: <https://docs.tomorrow.io/reference/historical-overview/>

**Degree Days Information**: <https://www.degreedays.net/calculation>