MATLAB WITH PYTHON

Yann Debray

V2 – 2024-09-15

TABLE OF CONTENTS

1. Preface 7

1.1. A brief history of scientific computing 7

1.1.1. The roots of numerical analysis 7

1.1.2. In a parallel universe 9

1.2. About the author 10

1.3. Open-source vs Commercial 11

1.4. Who is this book for? 11

2. End-to-end project with MATLAB & Python 13

2.1. Call Python from MATLAB 15

2.1.1. Check the Python installation 15

2.1.2. Call Python user-defined functions from MATLAB 15

2.1.3. Convert Python data to MATLAB data 16

2.1.4. Convert Python lists to MATLAB matrices 17

2.1.5. Explore graphically the Python data imported in MATLAB 18

2.1.6. Call a Machine Learning model in MATLAB 19

2.2. Call MATLAB from Python 20

2.3. Generate a Python package from a set of MATLAB functions 21

3. Set-up MATLAB and Python 25

3.1. Install Python 25

3.1.1. Install Python on Windows 25

3.2. Install Anaconda or other Python distribution 27

3.2.1. Install Miniconda from conda-forge 27

3.2.2. Install Micromamba for minimal footprint 28

3.3. Manage your PATH 29

3.4. Install additional Python packages 31

3.5. Set up a Python virtual environment 32

3.6. Set up a Python Development Environment 32

3.6.1. Jupyter Notebooks 33

3.6.2. MATLAB Integration for Jupyter 36

3.6.3. Visual Studio Code 37

3.7. Connect MATLAB to Python 38

3.8. Install the MATLAB Engine for Python 39

4. Call Python from MATLAB 41

4.1. Execute Python statements and files in MATLAB 41

4.2. Execute Python code in a MATLAB Live Task 42

4.3. Basic syntax of calling Python functions from MATLAB 44

4.4. Call Python User Defined Functions from MATLAB 46

4.5. Call Python community packages from MATLAB 55

4.6. Debug Python code called by MATLAB 59

4.6.1. Debug with Visual Studio Code 60

4.6.2. Debug with Visual Studio 62

4.7. Mapping data between Python and MATLAB 64

4.7.1. Scalars 64

4.7.2. Dictionaries and Lists 66

4.7.3. Arrays 69

4.7.4. Dataframes 70

5. Call Python AI libraries from MATLAB 73

5.1. Call Scikit-learn from MATLAB 73

5.2. Call TensorFlow from MATLAB 82

5.3. Import TensorFlow model into MATLAB 86

6. Call MATLAB from Python 99

6.1. Getting started with the MATLAB Engine API for Python 99

6.2. Facilitate AI development by using MATLAB Apps 101

6.2.1. Data Cleaner App 101

6.2.2. Regression and Classification Learner Apps 107

6.2.3. Image Labeler App 111

6.3. Leverage the work from the MATLAB community 114

7. Simulink with Python 123

7.1. Bring Python code into Simulink as a library for co-execution 123

7.1.1. Python Importer for Simulink 124

7.1.2. MATLAB function and system objects 127

7.2. Integrate TensorFlow and PyTorch models for both simulation and code generation 129

7.2.1. Import your deep learning model in MATLAB directly 131

7.2.2. Integrate deep learning models from TensorFlow Lite (TFLite) 133

7.3. Simulate a Simulink model directly from Python 134
7.4. Export a Simulink model as a Python package for deployment 139

8. Resources 143
8.1. Getting Python packages on MATLAB Online 143
8.2. Generate this book with Quarto and Pandoc 144

1. PREFACE

Engineers and scientists that I meet every day think about MATLAB & Python as MATLAB **vs** Python. The goal of this book is to prove to them that it is possible to think about it as MATLAB **with** Python.

Python recently became the most used programming language[1]. It is general purpose by nature, and it is particularly used for scripting, web development and Artificial Intelligence (Machine Learning & Deep Learning).

MATLAB is mostly seen as a programming language for technical computing, and a development environment for engineers and scientists. But MATLAB also provides flexible two-way integration with many programming languages including Python.

MATLAB works with common python distributions. For this book I will be using Python 3.10 (downloaded on Python.org) and MATLAB 2024b.

1.1. A BRIEF HISTORY OF SCIENTIFIC COMPUTING

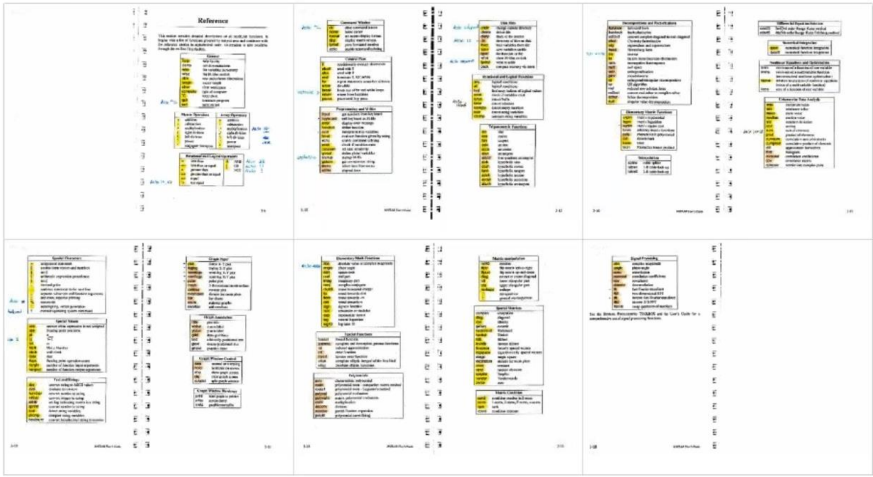
1.1.1. THE ROOTS OF NUMERICAL ANALYSIS

In the 1970s, Cleve Moler took actively part in the development of Fortran libraries called EISPACK[2] (to compute eigenvalues) and LINPACK[3] (for linear algebra). As he was professor of Mathematics at the University of New Mexico, he wanted to make those libraries accessible to student while sparing them the need to write Fortran wrapper code, compile it, debug it, compile again, run, …

So he created an interactive interpreter in Fortran for matrix computation, called MATLAB (short for MATrix LABoratory, nothing to do with the Matrix movie, that came out 30 years later). This first version was based on a few routines from EISPACK and LINPACK and only contained 80 functions.

This photo of a MATLAB manual at the time, shows the scope of the software in its early days.

[1] Python ranking #1 in the TIOBE index in 2021: https://www.tiobe.com/tiobe-index/
[2] EISPACK – computes the eigenvalues and eigenvectors of matrices: https://en.wikipedia.org/wiki/EISPACK
[3] LINPACK – Linear algebra package in Fortran: https://en.wikipedia.org/wiki/LINPACK



At that time MATLAB was not yet a programming language. It had no file extension (m-scripts), no toolboxes. The only available datatype was matrices. The graphic capabilities were asterisks drawn on the screen.

In order to add a function, you had to modify the Fortran source code and recompile everything. So the source code was open, because it needed to be (open-source only appeared in the 80s, with Richard Stallman and the Free Software movement).

After a course on numerical analysis that Cleve Moler gave at Stanford University in California, an MIT trained engineer came to him: “I introduced myself to Cleve”. This is the way Jack Little tells the story about their first encounter. Jack Little had anticipated the possible use of MATLAB on PC, and rewritten it in C. He knew, like Steve Jobs and Bill Gates that Personal Computing would win over the mainframe server business of IBM. He also added the ability to write program files to extend the capabilities of the software, and toolboxes that would become a well architectured, modular and scalable business model. In 1984, he created the company (The) MathWorks to commercialize MATLAB.

Read more about the origins of MATLAB:

- A history of MATLAB – published in June 20202 - https://dl.acm.org/doi/10.1145/3386331
- The Origins of MATLAB https://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html
- Cleve's Corner – History of MATLAB Published by the ACM https://blogs.mathworks.com/cleve/2020/06/13/history-of-matlab-published-by-the-acm/

1.1.2. In a parallel universe

In the 1980s, Guido van Rossum was working at the Centrum Wiskunde & Informatica (abbr. CWI; English: "National Research Institute for Mathematics and Computer Science") on a language called ABC.

"ABC was intended to be a programming language that could be taught to intelligent computer users who were not computer programmers or software developers in any sense. During the late 1970s, ABC's main designers taught traditional programming languages to such an audience. Their students included various scientists—from physicists to social scientists to linguists—who needed help using their very large computers. Although intelligent people in their own right, these students were surprised at certain limitations, restrictions, and arbitrary rules that programming languages had traditionally set out. Based on this user feedback, ABC's designers tried to develop a different language."

In 1986 Guido van Rossum moved to a different project at CWI, the Amoeba project. Amoeba was a distributed operating system. By the late 1980s, they realized that they needed a scripting language. With the freedom he was given inside this project, Guido van Rossum started his own "mini project".

In December 1989, Van Rossum had been looking for a "'hobby' programming project that would keep [him] occupied during the week around Christmas" as his office was closed when he decided to write an interpreter for a "new scripting language [he] had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers". He attributes choosing the name "Python" to "being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)".[4]

[4] Foreword for "Programming Python" Guido van Rossum (1996): https://www.python.org/doc/essays/foreword/

He wrote a simple virtual machine, a simple parser and a simple runtime. He created a basic syntax, using indentation for statement grouping. And he developed a small number of datatypes: dictionaries, lists, strings and numbers. Python was born. In Guido's opinion, his most innovative contribution to Python's success was making it easy to extend.

Main milestones of the Python language:

- 1991: Python 0.9.0 published to alt.sources by Guido Van Rossum
- 1994: Python 1.0. include functional programming (lambda's map, filter, reduce)
- 2000: Python 2.0 introduces list comprehension and garbage collection
- 2008: Python 3 fixes fundamental design flaws and is not backward compatible
- 2022: Python 2 is end of life, last version 2.7.18 released

Read more about Python:

- The Making of Python - A Conversation with Guido van Rossum, Part I https://www.artima.com/articles/the-making-of-python
- Microsoft Q&A with Guido van Rossum, Inventor of Python https://www.youtube.com/watch?v=aYbNh3NS7jA
- The Story of Python, by Its Creator, Guido van Rossum https://www.youtube.com/watch?v=J0Aq44Pze-w
- Python history timeline infographics https://python.land/python-tutorial/python-history

1.2. About the author

My name is Yann Debray, and I work for MathWorks, as a MATLAB Product Manager. You will probably think that I am biased, in that I am trying to sell you MATLAB. That's not wrong. But to better understand my motivations, you need to look a little deeper into my background.

I joined MathWorks in June 2020 (in the middle of the COVID-19 pandemic). Prior to that, I spent 6 years working on a project called Scilab[5]. Scilab is an open-source alternative to MATLAB. This experience translates my appetite for open-source and scientific computing.

[5] Scilab: https://scilab.org/

My first professional encounter with numerical computing after college was in December 2013, when I met Claude Gomez[6]. He was the CEO of Scilab Enterprises back then, and the one who had turned Scilab from a research project to a company. The business model was inspired from Red Hat selling services around Linux.

I know very well the challenge of making open-source a sustainable model in scientific computing, and that is the reason why I believe in an equilibrium in the force, between open-source and proprietary software. Not every software can be free. Based on the expertise required in fields like simulation – requiring decades of investments – we will still observe entire markets of engineering software driven by intellectual property for the years to come.

1.3. Open-source vs Commercial

One of the early questions around this book was: *Do I commercialize it, or do I make it open-source?*

I had an idealized view of what it would mean to write a book. The fame and the glamour. But pragmatically, I know it is not going to sell a lot, as it is quite niche. My best estimate for a target audience is around 30% of the 5 million users of MATLAB, that may also be interested in Python.

Beyond my idealism on open-source, I felt like I needed concrete motivation to see this project through. Hence my initial idea to sell a hard copy of this book. But my dear colleague and good friend Mike Croucher advised me against what he calls "dead wood". Hinting to the fact that the printed content would quickly become obsolete with every new version of MATLAB (twice a year).

Finally, I've decided that open-sourcing the content does not conflict with releasing a paid version of the book. In fact, when I buy technical books, I often decide for those who apply an open-source license.

1.4. Who is this book for?

If you recognize yourself in the following scenario, this book is for you:

You are an engineer or a researcher using MATLAB, and you are increasingly hearing about Python. This comes up particularly in subjects related to data science & artificial intelligence. When searching for code online, you might stumble on

[6] Interview with Claude Gomez, former CEO of Scilab Enterprises: https://www.d-booker.fr/content/81-interview-with-claude-gomez-ceo-of-scilab-enterprises-about-scilab-and-its-development

interesting scripts or packages written in Python. Or when working with colleagues that are using Python, you may be looking for ways to integrate their work:

I would like to **integrate Python** code **from the community** or **from my colleagues** in my MATLAB analysis

You are (or want to become) a Data Scientist, and you are working on scientific / engineering data (wireless, audio, video, radar/lidar, autonomous driving,…). You will probably be using Python for some of your daily operations related to data processing, but you may want to consider MATLAB for the engineering part of your AI workflow (especially if this intelligence will be integrated on embedded systems). If this part is covered by engineer colleagues, you might simply want to be able to run the models and scripts that they share with you:

Can I use MATLAB capabilities in my Python **Data Science & Artificial Intelligence** applications?

2. END-TO-END PROJECT WITH MATLAB & PYTHON

When I joined MathWorks, I met Heather. She had developed a really good demo to illustrate the use of MATLAB with Python. In this chapter, I'll show the **Weather Forecasting app** she developed. You can find the code on her GitHub repo: https://github.com/hgorr/weather-matlab-python

Start by retrieving the code by downloading a zip or cloning the repository:

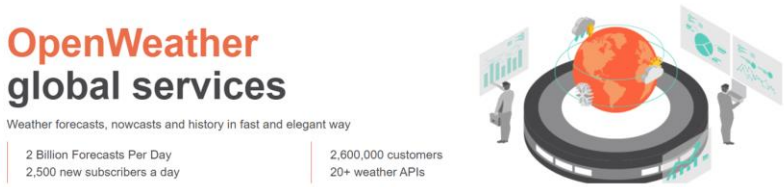
```
!git clone https://github.com/hgorr/weather-matlab-python
cd weather-matlab-python\
```

The resulting application will look like this:

We will work in steps to:

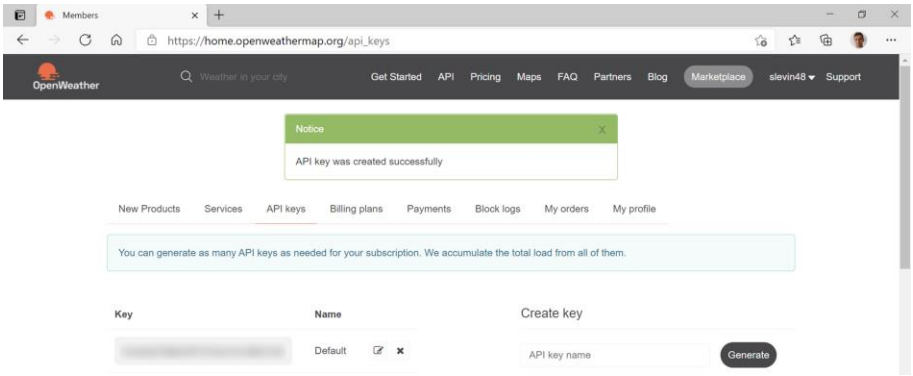
1. Call Heather's python code to retrieve the weather data
2. Integrate a MATLAB model predicting the air quality
3. Deploy the resulting application made out of MATLAB + Python

In this example we will be using data from a web service at openweathermap.org



In order to access live data, you will need to register[7] to the free tier offering. You will then have the option to generate an API key:

https://home.openweathermap.org/api_keys



This key will be necessary for each call of the web service. For instance, requesting the current weather[8] will be performed by calling the following address:

`api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}`

Save your API key in a text file called accessKey.txt.

```
% apikey = fileread("accessKey.txt");
```

Alternatively you can use the sample API key (as demonstrated in this script)

```
appid ='b1b15e88fa797225412429c1c50c122a1';
```

[7] Register to OpenWeatherMap.org: https://home.openweathermap.org/users/sign_up
[8] OpenWeatherMap current weather API: https://openweathermap.org/current

2.1. Call Python from MATLAB

Heather has created a module called weather.py that reads from the web service and parses the JSON data it returns. Of course, we can do this in MATLAB, but let's use this module as an example of accessing data from Python.

2.1.1. Check the Python installation

First connect to the Python environment using the pyenv[9] command. For more details on how to set-up MATLAB and Python, look at the next chapter. MATLAB can call python functions and create python objects from base Python, from packages you have installed and from your own Python code.

```
pyenv % Use pyversion for MATLAB versions before R2019b
```

```
ans =
  PythonEnvironment with properties:

         Version: "3.10"
      Executable: "C:\Users\...\python-3.10.4.amd64\python.exe"
         Library: "C:\Users\...\python-3.10.4.amd64\python310.dll"
            Home: "C:\Users\...\python-3.10.4.amd64"
          Status: NotLoaded
   ExecutionMode: OutOfProcess
```

2.1.2. Call Python user-defined functions from MATLAB

Now let's see how to use my colleague's weather module. We will start by getting the data for today. The get_current_weather function in the weather module gets the current weather conditions in Json format. The parse_current_json function then returns that data as a python dictionary.

```
jsonData =
py.weather.get_current_weather("London","UK",appid,api='samples')
```

```
jsonData =
  Python dict with no properties.
```

[9] Change default environment of Python interpreter
https://www.mathworks.com/help/matlab/ref/pyenv.html

```
    {'coord': {'lon': -0.13, 'lat': 51.51}, 'weather': [{'id': 300,
'main': 'Drizzle', 'description': 'light intensity drizzle', 'icon':
'09d'}], 'base': 'stations', 'main': {'temp': 280.32, 'pressure':
1012, 'humidity': 81, 'temp_min': 279.15, 'temp_max': 281.15},
'visibility': 10000, 'wind': {'speed': 4.1, 'deg': 80}, 'clouds':
{'all': 90}, 'dt': 1485789600, 'sys': {'type': 1, 'id': 5091,
'message': 0.0103, 'country': 'GB', 'sunrise': 1485762037, 'sunset':
1485794875}, 'id': 2643743, 'name': 'London', 'cod': 200}
```

```
weatherData = py.weather.parse_current_json(jsonData)
```

```
weatherData =
  Python dict with no properties.

    {'temp': 280.32, 'pressure': 1012, 'humidity': 81, 'temp_min':
279.15, 'temp_max': 281.15, 'speed': 4.1, 'deg': 80, 'lon': -0.13,
'lat': 51.51, 'city': 'London', 'current_time': '2023-03-15
16:04:38.427888'}
```

2.1.3. CONVERT PYTHON DATA TO MATLAB DATA

Let's convert the Python dictionary[10] into a MATLAB structure[11]:

```
data = struct(weatherData)
```

```
data = struct with fields:
            temp: 280.3200
        pressure: [1×1 py.int]
        humidity: [1×1 py.int]
        temp_min: 279.1500
        temp_max: 281.1500
           speed: 4.1000
             deg: [1×1 py.int]
             lon: -0.1300
             lat: 51.5100
            city: [1×6 py.str]
    current_time: [1×26 py.str]
```

[10] Python Dictionary: https://docs.python.org/3/tutorial/datastructures.html#dictionaries

[11] MATLAB Structure: https://www.mathworks.com/help/matlab/ref/struct.html

Most of the data gets automatically converted. Only some fields did not find an obvious equivalent:

- `pressure` & `humidity` remain as a `py.int` object in MATLAB.
- `city` and `current_time` remain as a `py.str` object in MATLAB.

We can convert them explicitly using standard MATLAB functions like double[12], string [13]and datetime[14]:

```
data.pressure = double(data.pressure);
data.humidity = double(data.humidity);
data.deg = double(data.deg);
data.city = string(data.city);
data.current_time = datetime(string(data.current_time))
```

```
data = struct with fields:
            temp: 280.3200
        pressure: 1012
        humidity: 81
        temp_min: 279.1500
        temp_max: 281.1500
           speed: 4.1000
             deg: 80
             lon: -0.1300
             lat: 51.5100
            city: "London"
    current_time: 15-Mar-2023 16:04:38
```

2.1.4. Convert Python lists to MATLAB matrices

Now let's call the get_forecast function which returns a series of predicted weather conditions over the next few days. We can see that the fields of the structure are returned as Python list[15]:

```
jsonData =
py.weather.get_forecast('Muenchen','DE',appid,api='samples');
forecastData = py.weather.parse_forecast_json(jsonData);
forecast = struct(forecastData)
```

```
forecast = struct with fields:
```

[12] Double-precision arrays: https://www.mathworks.com/help/matlab/ref/double.html
[13] Characters and Strings: https://www.mathworks.com/help/matlab/characters-and-strings.html
[14] Dates and Time: https://www.mathworks.com/help/matlab/date-and-time-operations.html
[15] Python List: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

```
    current_time: [1×36 py.list]
            temp: [1×36 py.list]
             deg: [1×36 py.list]
           speed: [1×36 py.list]
        humidity: [1×36 py.list]
        pressure: [1×36 py.list]
```

Lists containing only numeric data can be converted into doubles (since MATLAB R2022a):

```
forecast.temp = double(forecast.temp) - 273.15; % Kelvin to Celsius
forecast.temp
```

```
ans = 1×36
   13.5200   12.5100    3.9000   -0.3700    0.1910    2.4180
3.3280 ⋯
```

Lists containing text can be transformed to strings, and further processed into specific data types like datetime:

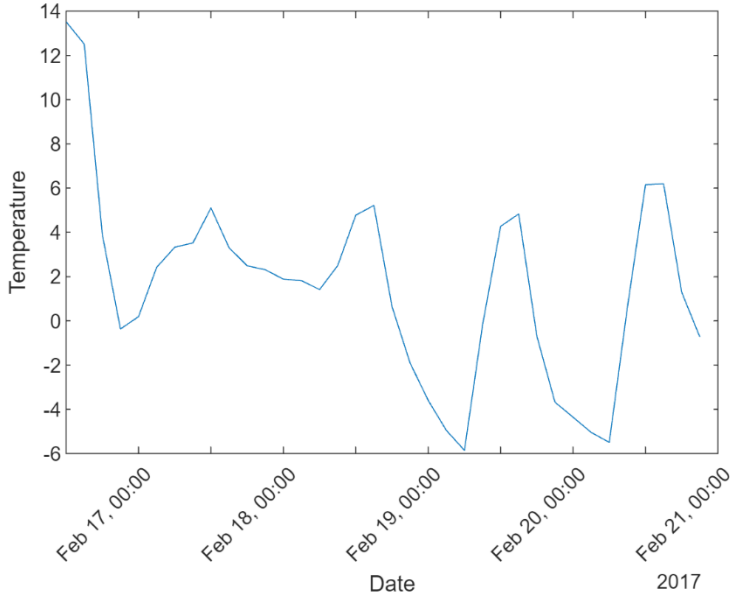
```
forecast.current_time = string(forecast.current_time);
forecast.current_time = datetime(forecast.current_time);
forecast.current_time
```

```
ans = 1×36 datetime
16-Feb-2017    12:00:0016-Feb-2017    15:00:0016-Feb-2017
18:00:0016-Feb-2017 21:00: ⋯
```

Read more about mapping data between Python and MATLAB (section 4.7)

2.1.5. Explore graphically the Python data imported in MATLAB

```
plot(forecast.current_time,forecast.temp)
xtickangle(45)
xlabel('Date')
ylabel('Temperature')
```



2.1.6. CALL A MACHINE LEARNING MODEL IN MATLAB

Now let's suppose we have used some historical data to create a machine learning model that takes a set of weather conditions and returns a prediction of the air quality. My Python colleague wants to make use of my model in her Python code.

First, let's see how the air quality prediction works. There are three steps:

- Load the model from a .mat file
- Convert the current weather data from openweathermap.org to the format expected by the model
- Call the predict method of the model to get the expected air quality for that day

```
load airQualModel.mat model
testData = prepData(data);
airQuality = predict(model,testData)
```

```
airQuality = categorical
     Good
```

To give this to my colleague, I'm going to pack up these steps into a single function called predictAirQuality:

```
function airQual = predictAirQual(data)
% PREDICTAIRQUAL Predict air quality, based on machine learning
model
%
%#function CompactClassificationEnsemble

% Convert data types
currentData = prepData(data);

% Load model
mdl = load("airQualModel.mat");
model = mdl.model;

% Determine air quality
airQual = predict(model,currentData);

% Convert data type for use in Python
airQual = char(airQual);

end
```

This function does the same three steps as above – loads the model, converts the data, and calls the model's predict method.

However, it has to do one other thing. The model returns a MATLAB categorical value which doesn't have a direct equivalent in Python, so we convert it to a character array.

Now that we have our MATLAB function that uses the air quality prediction model, let's see how to use it in Python.

2.2. Call MATLAB from Python

Here we will demonstrate calling MATLAB from Python inside of a simple Python shell (>>>).

The first step is to use the engine API to start a MATLAB running in the background for Python to communicate with (we will assume here that you have installed it already – else check section 3.8).

```
>>> import matlab.engine
>>> m = matlab.engine.start_matlab()
```

Once the MATLAB is running, we can call any MATLAB function on the path.

```
>>> m.sqrt(42.0)
6.48074069840786
```

We need to access the key from the txt file:

```
>>> with open("accessKey.txt") as f:
...   apikey = f.read()
```

Now we can use the `get_current_weather` and the `parse_current_json` functions from the weather module just like we did in MATLAB to get the current weather conditions.

```
>>> import weather
>>> json_data = weather.get_current_weather("Boston","US",apikey)
>>> data = weather.parse_current_json(json_data)
>>> data
{'temp': 62.64, 'feels_like': 61.9, 'temp_min': 58.57, 'temp_max':
65.08, 'pressure': 1018, 'humidity': 70, 'speed': 15.01, 'deg': 335,
'gust': 32.01, 'lon': -71.0598, 'lat': 42.3584, 'city': 'Boston',
'current_time': '2022-05-23 11:28:54.833306'}
```

Then we can call the MATLAB function `predictAirQuality` to get the predicted result.

```
>>> aq = m.predictAirQuality(data)
>>> aq
Good
```

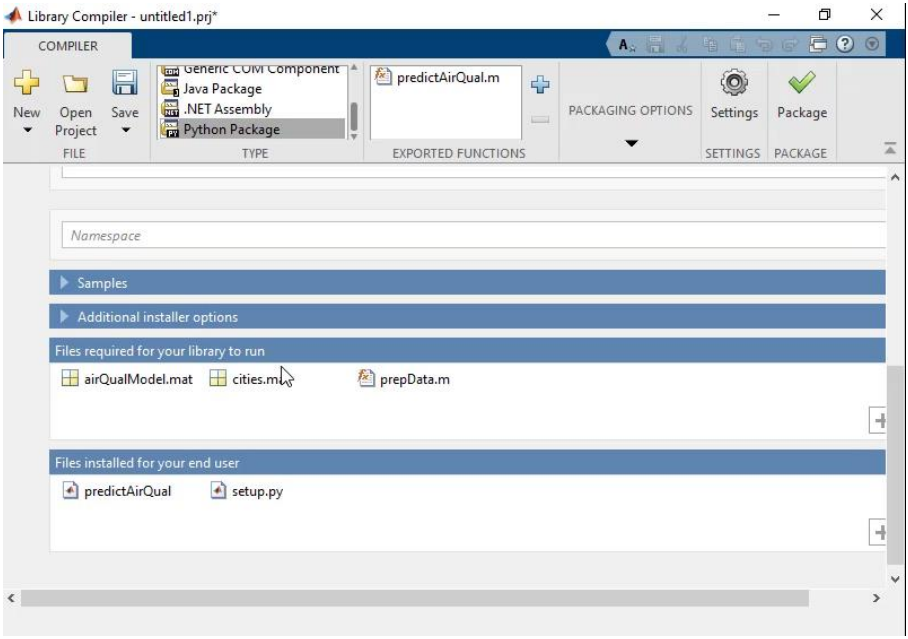
The last step is to shutdown the MATLAB started by the engine API at the beginning of our notebook.

```
>>> m.exit()
```

However, your Python colleague might not have access to MATLAB. The next two sections will target this use case.

2.3. Generate a Python package from a set of MATLAB functions

For this, you will need to use a dedicated toolbox called MATLAB Compiler SDK[16]. You can select the Library Compiler in the Apps ribbon, or enter in the Command Window (`libraryCompiler`):



Simply select the MATLAB function(s) that you want to turn them into Python functions. The dependencies will be automatically added to the Python package (in this case, the Air Quality Model, the list of cities, and the pre-processing function).

This packages the files we need and creates a *setup.py* and *readme.txt* file with instructions for the Python steps. To learn more on how to set-up the generated package read the section 7.1.

Then we need to import and initialize the package and can call the functions, like so:

```
>>> import AirQual
>>> aq = AirQual.initialize()
>>> result = aq.predictAirQual()
```

When we're finished, wrap things up by terminating the process:

```
>>> aq.terminate()
```

[16] MATLAB Compiler SDK: https://www.mathworks.com/help/compiler_sdk/

We can go one step further in sharing the MATLAB functionality to be used as a web service (and potentially accessed by many users at once). In this case, MATLAB Production Server[17] can be used for load balancing and the MATLAB code can be accessed through a RESTful API[18] or Python client[19].

[17] MATLAB Production Server: https://www.mathworks.com/help/mps/
[18] Create client programs using the RESTful API: https://www.mathworks.com/help/mps/restful-api-and-json.html
[19] Create a MATLAB Production Server Python Client: https://www.mathworks.com/help/mps/python/create-a-matlab-production-server-python-client.html

3. SET-UP MATLAB AND PYTHON

3.1. INSTALL PYTHON

You can simply go to www.python.org/downloads and select a version of Python compatible with your MATLAB version[20]. For instance, this is the list of versions compatible with the latest releases:

MATLAB Version	Compatible Versions of Python 3
R2024b	3.10, 3.11, 3.12
R2024a	3.9, 3.10, 3.11
R2023b	3.9, 3.10, 3.11
R2023a	3.8, 3.9, 3.10
R2022b	2.7, 3.8, 3.9, 3.10

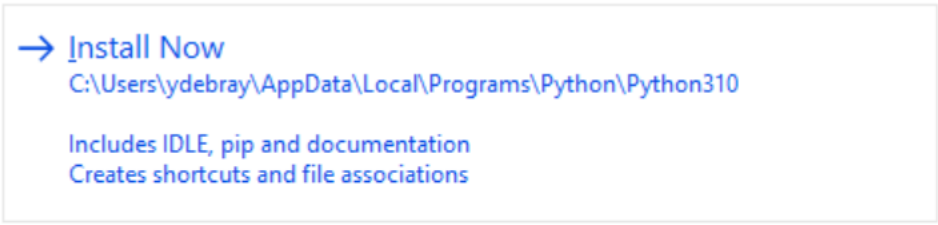
3.1.1. INSTALL PYTHON ON WINDOWS

If you are running on Windows, download the Windows installer (64-bit)[21]: the file python-3.10.10-amd64.exe is only 28Mo. Just run this executable (you can uncheck the admin privileges):

[20] Versions of Python Compatible with MATLAB Products by Release:
https://www.mathworks.com/support/requirements/python-compatibility.html
[21] Windows installer (64-bit): https://www.python.org/ftp/python/3.10.10/python-3.10.10-amd64.exe

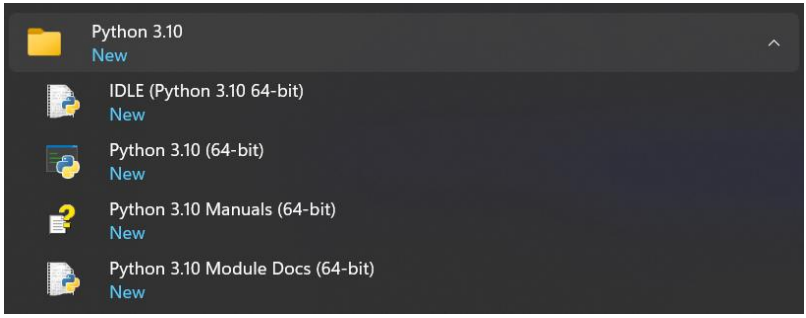
By default, the checkbox “Add python.exe to PATH” isn’t checked. I would advise you to select it (Otherwise, you will have to add Python to your PATH manually):

Select “→ Install Now”:



It should only take about a minute to get everything installed on your machine.

The following applications have been installed and are accessible from your Start menu:

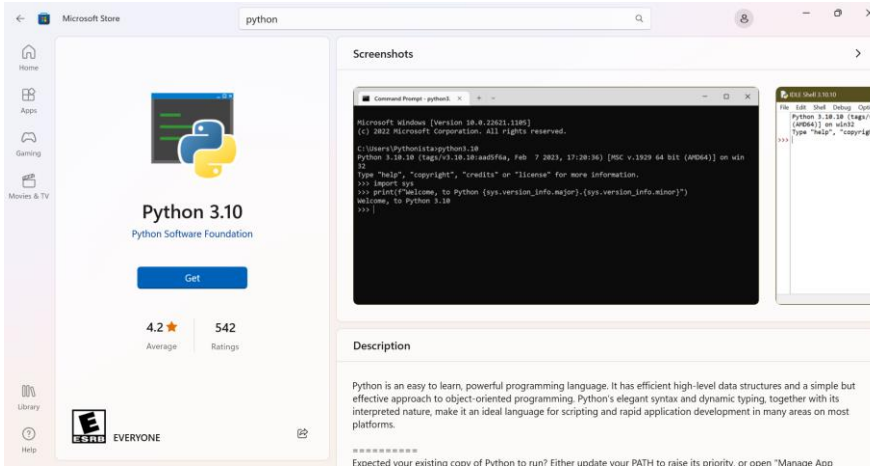


To check that you have Python installed and available to your PATH, open a command prompt:

```
C:\Users\ydebray>where python
C:\Users\ydebray\AppData\Local\Programs\Python\Python310\python.exe
C:\Users\ydebray\AppData\Local\Microsoft\WindowsApps\python.exe
```

If you have several versions of Python installed, it will return each of them, in the order listed in your PATH, plus the last one that isn’t actually installed: `C:\Users\ydebray\AppData\Local\Microsoft\WindowsApps\python.exe`

This is a link to a version packaged on the Microsoft Store. If you run it, you'll be redirect to the Store:



3.2. Install Anaconda or other Python distribution

With the previous version installed, you only have the base Python language. No numerical packages, or Development Environment (unlike MATLAB that ships all of those features by default). To get a set of curated data science packages pre-installed, you can download a distribution, like Anaconda:

Be aware of the fact that you now need to comply with Anaconda's terms of services[22] (since September 2020): You can only use the open-source Anaconda Distribution[23] professionally for free if you are not part of an organization with more than 200 employees. Otherwise, you will need to buy a license of Anaconda Professional[24].

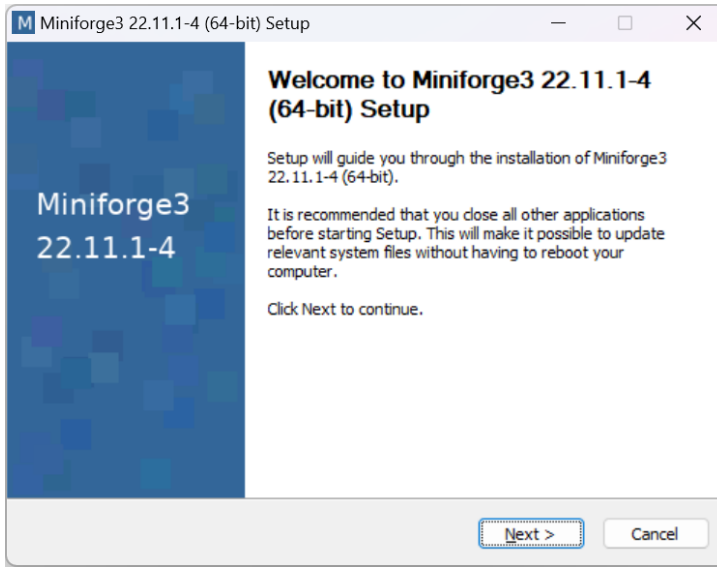
If you are searching for an alternative distribution to Anaconda, I would recommend WinPython[25] on Windows. If you are running on Linux, I believe you don't need a distribution and can manage packages yourself.

3.2.1. Install Miniconda from conda-forge

[22] Anaconda's terms of services: https://www.anaconda.com/terms-of-service
[23] open-source Anaconda Distribution: https://www.anaconda.com/products/distribution
[24] Anaconda Professional license: https://www.anaconda.com/products/professional
[25] WinPython: https://winpython.github.io/

Conda-forge[26] provides installers of the conda[27] package manager that point by default to the community channel, to remain in compliance with the terms of use of the Anaconda repo, even for “commercial activities”.

Download and run the installer of miniforge (55 MB):



3.2.2. INSTALL MICROMAMBA FOR MINIMAL FOOTPRINT

micromamba[28] is a 4 MB pure-C++ drop-in replacement for the conda package manager. Unlike pip or conda, it is not written in Python, so you don’t need to get Python to get it, and it can retrieve python:

```
(base) $ mamba install python
Looking for: ['python']

conda-forge/noarch                                  11.6MB @
3.6MB/s  3.7s
conda-forge/linux-64                                30.3MB @
3.7MB/s  9.4s
```

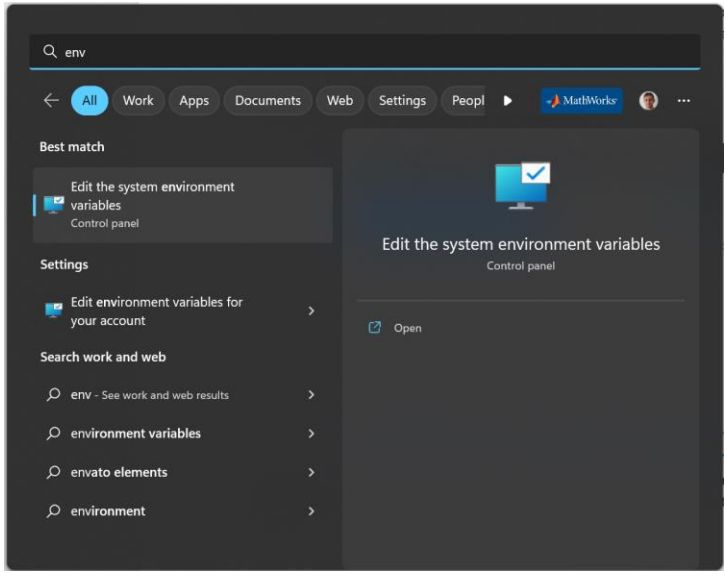
[26] Conda-forge: https://conda-forge.org/
[27] Miniconda installer from Conda-forge: https://github.com/conda-forge/miniforge
[28] Micromamba doc: https://mamba.readthedocs.io/ - on Linux : `curl micro.mamba.pm/install.sh | bash`

3.3. MANAGE YOUR PATH

When you have several versions of Python installed, the command python returns the version that is higher up in your PATH. To check which version of Python is used by default:

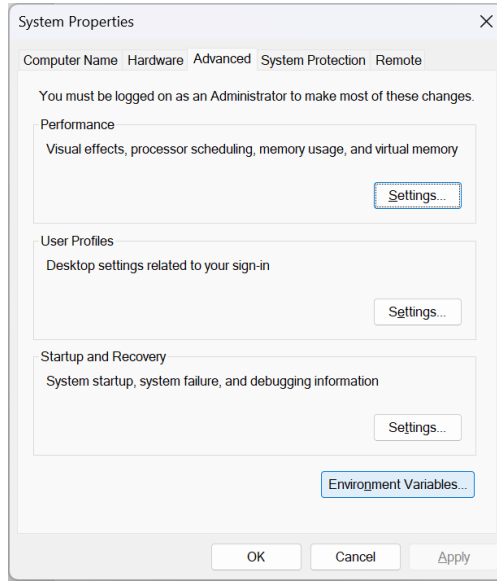
```
C:\Users\ydebray>python --version
Python 3.10.10
```

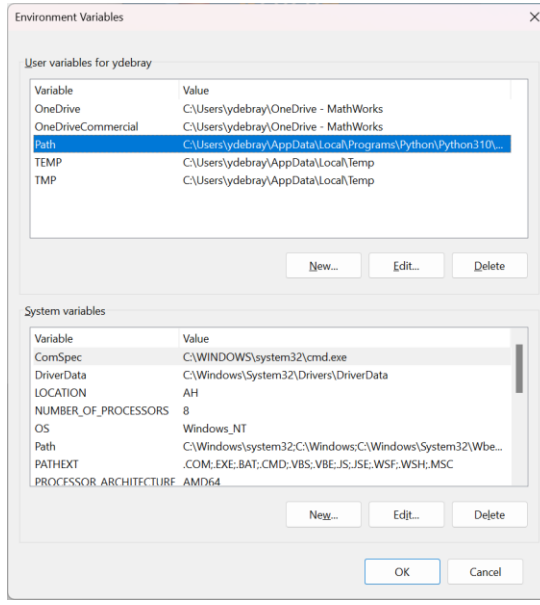
To change this, you will need to modify your PATH[29]



You can edit your PATH in **environment variables**, by tipping “path” in the search bar of your Windows start menu. Select the Path in the user variables (it will be written on top of the system variables):

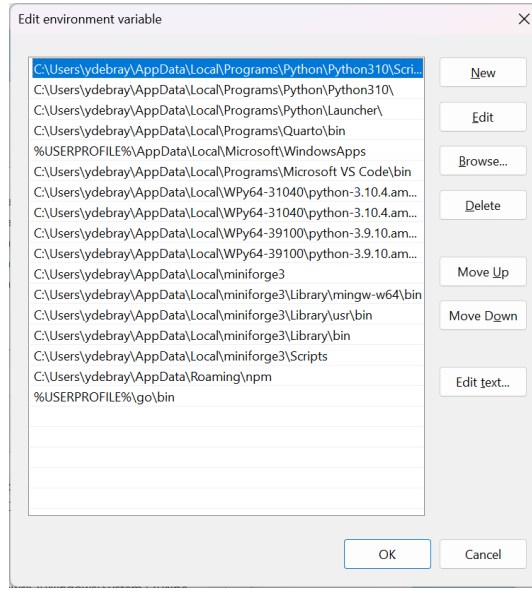
[29] PATH: https://en.wikipedia.org/wiki/PATH_(variable)





You can modify the order in which each version of Python is listed in the PATH. And in order to access pip (the Python default package manager), make sure to also list the Script folder in the PATH:

```
C:\Users\ydebray\AppData\Local\Programs\Python\Python310\Scripts
```



3.4. INSTALL ADDITIONAL PYTHON PACKAGES

In order to retrieve additional packages from the Python Package Index[30], use the pip command:

```
C:\Users\ydebray>pip install pandas
```

This will for instance install the famous pandas[31] package. It will also automatically retrieve its dependencies (in this case numpy, python-dateutil, pytz).

You can check if a package is installed by calling the method `pip show`. It will show information about this package:

```
C:\Users\ydebray>pip show pandas
Name: pandas
Version: 1.3.3
Summary: Powerful data structures for data analysis, time series, and statistics
Home-page: https://pandas.pydata.org
Author: The Pandas Development Team
Author-email: pandas-dev@python.org
```

[30] Python Package Index (PyPI) https://pypi.org/
[31] Pandas https://pandas.pydata.org

```
License: BSD-3-Clause

Location:
c:\users\ydebray\appdata\local\programs\python\python39\lib\site-
packages

Requires: numpy, python-dateutil, pytz

Required-by: streamlit, altair
```

To upgrade a package previously installed with a new version:

```
C:\Users\ydebray>pip install --upgrade pandas
```

3.5. Set up a Python virtual environment

If you have different projects leveraging different versions of the same package, or if you want a way to replicate your production environment in a clean space, use Python virtual environment[32]. It's a type of virtualization at the language level (like Virtual Machine at the Machine level, or Docker container at the Operating System level). This is the default way (shipped with base Python) to create a virtual environment called env:

```
C:\Users\ydebray>python -m venv env
```

You then need to activate it

- On Windows:

```
C:\Users\ydebray>.\env\Scripts\activate
```

- On Linux:

```
$ source env/bin/activate
```

Once you've done that you can install the libraries you want, for instance from a list of requirements:

```
C:\Users\ydebray>pip install -r requirements.txt
```

3.6. Set up a Python Development Environment

Once you've installed Python and the relevant packages for scientific computing, you still don't quite have the same experience as with the MATLAB Integrated Development Environment (IDE).

[32] Python virtual environment: https://docs.python.org/3/tutorial/venv.html

Two key open-source technologies are taking a stab at reshaping the tech computing landscape:

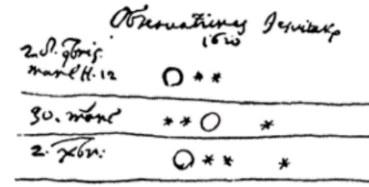
- Jupyter Notebooks
- Visual Studio Code

They are redefining the way *Languages* and *Development environments* are interacting. Based on open standards for interactive computing first with Jupyter. Adding richer interaction for multiple languages in the IDE, with the VS Code Language Server Protocol.

3.6.1. JUPYTER NOTEBOOKS

Jupyter Notebooks have become over the years, one of the most used and appreciated data science tools. They combine text (as Markdown), code, and output (numerical and graphical). Notebooks help data scientist to communicate goals, methods and results. It can be seen as an executable form of textbook or scientific paper.

Jupyter stands for Julia, Python and R, but it is also an homage to Galileo's notebooks recording the discovery of the moons of Jupiter. Those notebooks were probably one of the first instance of open science, data-and-narrative papers. When Galileo published the Sidereal Messenger in 1610 (one of the first scientific paper), he actually published his observations with code and data. It was a log of the dates and the states of the night. There was data and metadata, and there was a narrative.



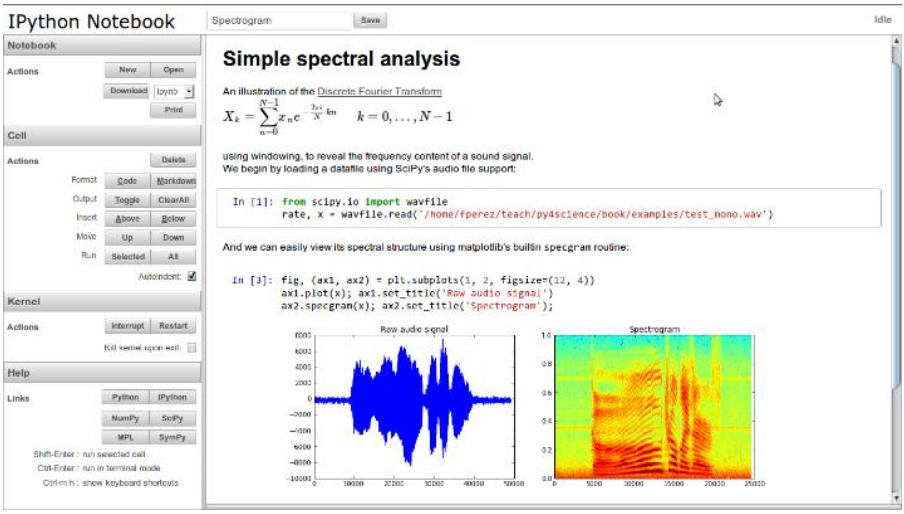
Jupyter is a project that spun off in 2014 [33] from IPython. IPython stands for Interactive Python and was created in 2001 by Fernando Perez. He drew his inspiration from Maple and Mathematica that both had notebook environments. He really liked the Python language, but he felt limited by the interactive prompt to do scientific computing. So he wrote a python startup file to provide the ability to hold state and capture previous results for reuse, and adding some nice features like loading the Numeric library and Gnuplot. 'ipython-0.0.1'[34] was born, a mere 259 lines to be loaded as $PYTHONSTARTUP.

Around 2006, the IPython project took some inspiration from another open-source project called Sage[35]. The Sage Notebook was taking the route of using the filesystem

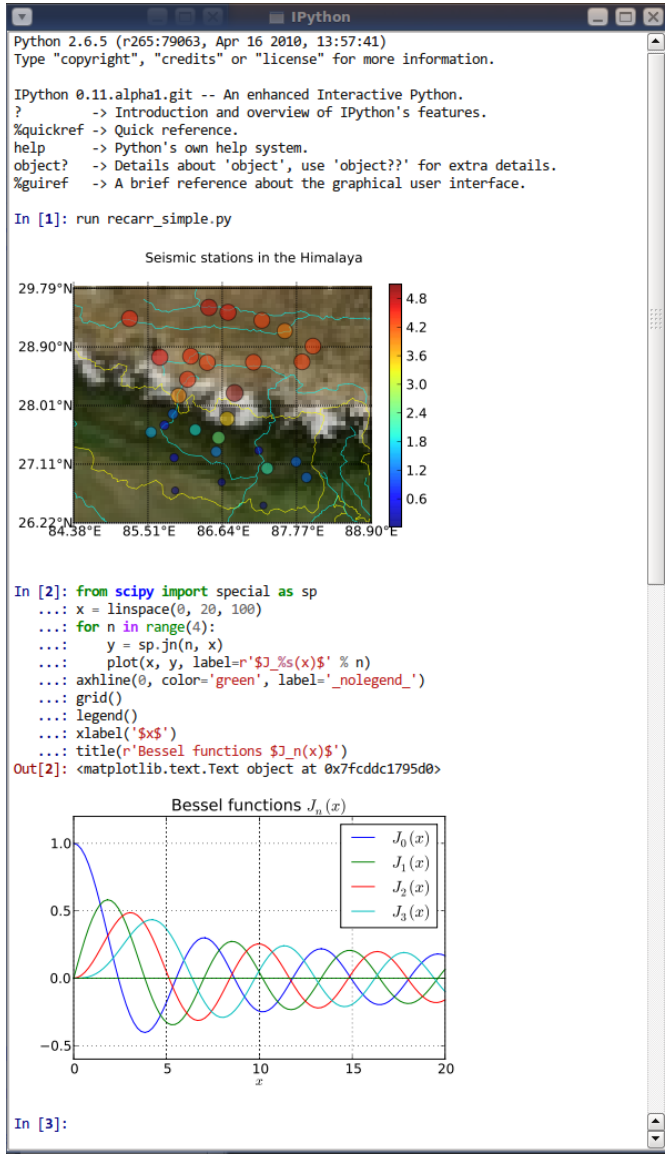
[33] Launch of Project Jupyter at SciPy 2014: https://speakerdeck.com/fperez/project-jupyter
[34] ipython-0.0.1 - https://gist.github.com/fperez/1579699
[35] SageMath mathematics software system licensed under the GPL: https://www.sagemath.org/

for notebook operations. You couldn't meaningfully list files with 'ls' or move around the filesystem by changing directory with 'cd'. Sage would execute your code in hidden directories with each cell actually being a separate subdirectory.

In 2010, the architecture of IPython evolved by separating the notebook front-end from the kernel executing Python code, and communicating between the two with the ZeroMQ protocol[36]. This design enabled the development of a Qt client, a Visual Studio extension, and finally a web frontend.



[36] ZeroMQ Protocol: https://zeromq.org/



IPython gave turn to Jupyter, to become language agnostic. Jupyter supports execution environments (aka kernels) in several dozen languages among which are Julia, R, Haskell, Ruby, and of course Python (via the IPython kernel)… and MATLAB[37] (via a kernel maintained by MathWorks as of 2023).

[37] MATLAB Kernel for Jupyter: https://blogs.mathworks.com/matlab/2023/01/30/official-mathworks-matlab-kernel-for-jupyter-released/

To summarize, Jupyter provides 3 key *components* to the modern scientific computing stack:

Some of the testimonies of the pervasive success of Jupyter in data science are the development of additional capabilities from the ecosystem:

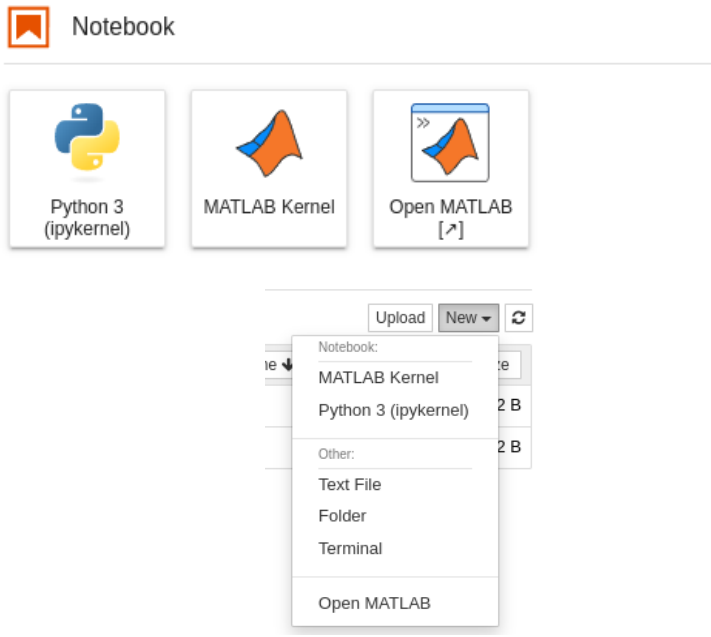
- Running Notebooks on Google Colab
- Render Notebooks on GitHub

Read more on Jupyter:

- The scientific paper is obsolete, by James Somers – The Atlantic – APRIL 5, 2018
  https://www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676/
- The IPython notebook: a historical retrospective
  http://blog.fperez.org/2012/01/ipython-notebook-historical.html
- A Brief History of Jupyter Notebooks
  https://ep2020.europython.eu/media/conference/slides/7UBMYed-a-brief-history-of-jupyter-notebooks.pdf
- The First Notebook War - Martin Skarzynski
  https://www.youtube.com/watch?v=QR7gR3njNWw

3.6.2. MATLAB Integration for Jupyter

MathWorks has released an official kernel for Jupyter in January 2023. In addition to this, you also have a way to integrate the MATLAB full environment as an app inside of a JupyterHub server installation. You can find this app easier in the 'New' menu, or if you are using JupyterLab, as an icon in the launcher:



To find out more about the MATLAB Integration for Jupyter:

- https://github.com/mathworks/jupyter-matlab-proxy
- https://www.mathworks.com/products/reference-architectures/jupyter.html

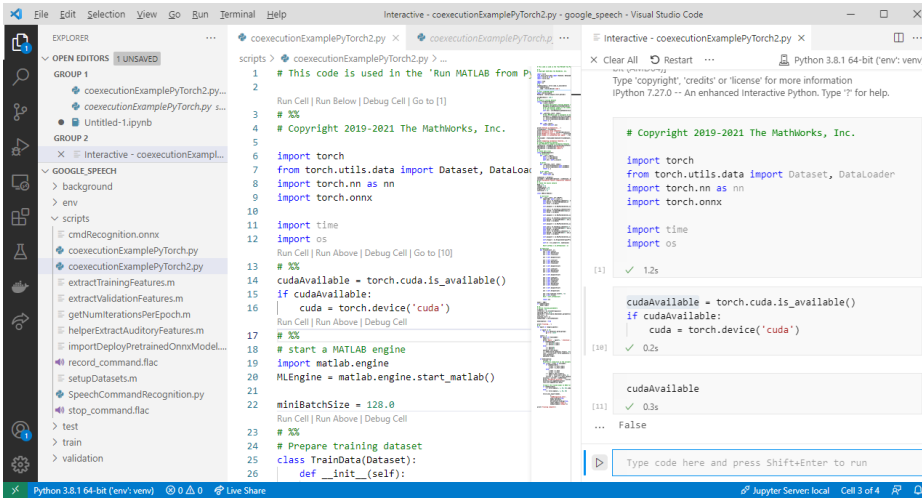
3.6.3. VISUAL STUDIO CODE

I adopted VS Code when I discovered that it was supporting Jupyter/IPython Notebook ipynb files.

As any other Integrated Development Environment, VS Code supports writing scripts and executing them in several languages (Python, Javascript, …)

As of early 2024, VSCode also has a MATLAB extension[38] that enables you to run MATLAB code in the editor by connecting to a MATLAB session running on your machine.

[38] MATLAB Extension for VSCode: https://blogs.mathworks.com/matlab/2024/03/05/matlab-extension-for-visual-studio-code-now-with-code-execution/



The big difference with the Eclipse[39] approach to componentization is the web standards adopted[40]. This enables to have richer interactions between the development tool and the language server.

And since it is all based on web technologies, you can access a web version at vscode.dev. Unlike for web languages like HTML/JS, this does not enable the execution of Python as it would require an interpreter running in the browser, or a server to connect to. Some hacks exist based on Pyodide[41] (a port of Python to WebAssembly).

3.7. CONNECT MATLAB TO PYTHON

You can connect your MATLAB session to Python using the pyenv command since 2019b. Before that, use pyversion (introduced in 2014b).

If you have multiple Python versions installed, you can specify which version to use, either with:

```
>> pyenv('Version','3.10')
```

or

[39] Eclipse IDE, famous for JAVA development: https://en.wikipedia.org/wiki/Eclipse_(software)
[40] Language Server Protocol: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/
[41] Pyodide: https://pyodide.org/en/stable/index.html

```
>> pyenv('Version','C:\Users\ydebray\AppData\Local\Programs\
Python\Python310\python.exe')
```

This is also the way to connect to Python virtual environments:

```
>> pyenv('Version','env\Scripts\python.exe')
```

In your project folder, where you created your virtual environment called env, you simply need to point to the Python executable that is contained in the Scripts subfolder.

Execution Mode:

By default, Python runs in the same process as MATLAB. On the plus side, it means that you don't have overhead for inter-process data exchange between the two systems. But it also means that if Python encounters an error and crashes, then MATLAB crashes as well. This can happen when MATLAB uses different versions of the same library than a given package. For this reason, the *Out-of-Process* execution mode has been introduced[42]:

```
>> pyenv("ExecutionMode","OutOfProcess")
```

Setup Tips:

- Ensure all code is on path (both on the MATLAB and Python side[43])
- Check environment settings, depending on how you set up Python
- In Out-of-Process Execution, you can terminate the Python process[44]

3.8. Install the MATLAB Engine for Python

Since the MATLAB Engine for Python has been added to the Python Package Index[45] (mid-April 2022), you can simply install it with the pip command:

```
C:\Users\ydebray>pip install matlabengine
```

Before that and for release prior to MATLAB R2022a, you had to install in manually[46]:

[42] Out-of-Process Execution of Python Functionality: https://www.mathworks.com/help/matlab/matlab_external/out-of-process-execution-of-python-functionality.html
[43] Add a script to the Python path: https://github.com/hgorr/matlab-with-python/blob/master/setUpPyPath.m
[44] Terminate process associated with Python interpreter: https://www.mathworks.com/help/matlab/ref/pythonenvironment.terminate.html
[45] MATLAB Engine for Python on PyPI: https://pypi.org/project/matlabengine/
[46] Install MATLAB Engine for Python: https://www.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html

```
cd "matlabroot\extern\engines\python"

python setup.py install
```

On Linux, you need to make sure that the default install location of MATLAB by calling `matlabroot` in a MATLAB Command Window. By default, Linux installs MATLAB at:

```
/usr/local/MATLAB/R2024b
```

4. Call Python from MATLAB

Why would you want to call Python from MATLAB. There could be a number of reasons.

First, as a single user. You might want to grab features available in Python. For instance, specialized libraries in fields like AI: Machine Learning with Scikit-Learn or XGBoost, Deep-Learning with TensorFlow or PyTorch, Reinforcement Learning with OpenAI Gym, …

Second, if you are working with colleagues that developed Python functions, that you would like to leverage as a MATLAB user, without the need to recode.

Third, if you are deploying your MATLAB Application in a Python-based environment, where some of the services, for instance for data access like in the case of the weather app from the first chapter, are written in Python.

4.1. Execute Python statements and files in MATLAB

Since R2021b, you can run Python statements directly from MATLAB with pyrun. This is convenient to simply run short snippets of Python code, without having to wrap it into a script.

```
pyrun("l = [1,2,3]")
pyrun("print(l)")
```

```
[1, 2, 3]
```

As you can see, the pyrun function is stateful, in that it maintains the variable defined in previous calls. You can retrieve the Python variable on the MATLAB side by entering it as a second argument:

```
pyrun("l2 = [k^2 for k in l]","l2")
```

```
ans =
  Python list with values:

    [3, 0, 1]

    Use string, double or cell function to convert to a MATLAB array.
```

You can retrieve the list of variables defined in the local scope with the function dir():

```
D = pyrun("d = dir()","d")
```

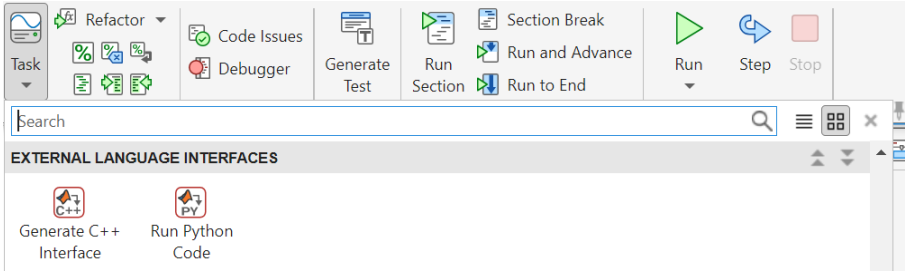
```
D =
  Python list with values:

    ['__builtins__', '__name__', 'l', 'l2']

    Use string, double or cell function to convert to a MATLAB array.
```

If it feels more convenient to paste your Python code snippet into a script, you can use pyrunfile.

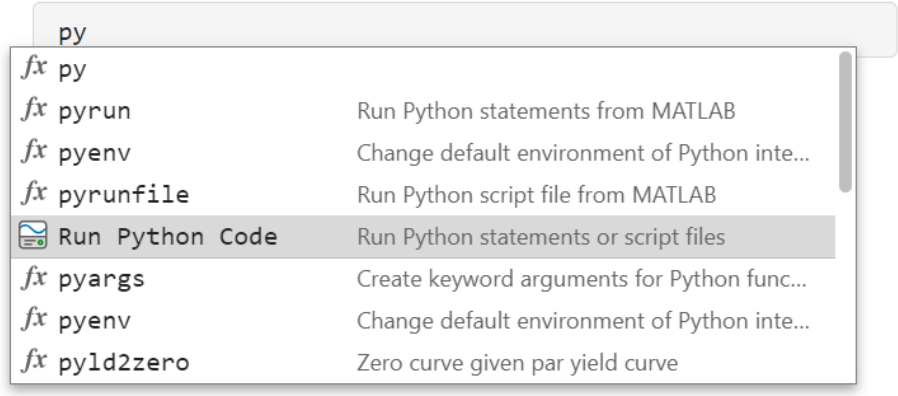
4.2. Execute Python code in a MATLAB Live Task

Since MATLAB 2022a, you can develop your own custom live tasks. So, in mid-2021, we started prototyping a Python live task with Lucas Garcia. The truth is: I made a first crappy version, and Lucas turned it into something awesome (Lucas should get all the credits for this). Based on this Minimal Viable Product, we engaged with the development teams, both the MATLAB editor team, and the Python interface team. We decided it would be best to release this prototype in open-source on GitHub to get early feedback. The code is available on https://github.com/mathworks/MATLAB-Live-Task-for-Python

As of 24a, you can find this live task directly in the product. To test it, create a new Live Script, and select Task in the Live Editor tab. You should see this icon under EXTERNAL LANGUAGE INTERFACES:



If you click on it, it will add the live task to your Live Script where the cursor is located. Alternatively, you can start typing “python” or “run” directly in your Live Script select the task:



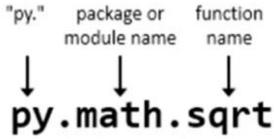
This is what the in-product version of this Python live task looks like:

You can try it out by inputting raw python code, that takes a MATLAB input, like:

```
import textwrap as tw

format = tw.TextWrapper(width=30)

wrapped = format.wrap(T)
```

4.3. Basic syntax of calling Python functions from MATLAB

All Python functions in MATLAB have the same basic syntax:



The basic example that I give to kick things off is usually calling the square root function from the math module[47], that is part of the Python standard library. It makes little sense to call mathematics functions in Python from MATLAB, but it is easy to compare the result with what you would expect directly from MATLAB:

In the MATLAB Command Window:

```
>> py.math.sqrt(42)
```

In a MATLAB Live Script:

```
py.math.sqrt(42)
```

```
ans = 6.4807
```

We can create Python data structures from within MATLAB:

```
py.list([1,2,3])
```

```
ans =
  Python list with values:

    [1.0, 2.0, 3.0]

    Use string, double or cell function to convert to a MATLAB array.
```

```
py.list({1,2,'a','b'})
```

```
ans =
  Python list with values:
```

[47] Python math module: https://docs.python.org/3/library/math.html

```
    [1.0, 2.0, 'a', 'b']

    Use string, double or cell function to convert to a MATLAB array.
```

```
s = struct('a', 1, 'b', 2)
```

```
s = struct with fields:
    a: 1
    b: 2
```

```
d = py.dict(s)
```

```
d =
  Python dict with no properties.

    {'a': 1.0, 'b': 2.0}
```

And we can run methods on those data structures from the MATLAB side:

```
methods(d)
```

```
Methods for class py.dict:

char        copy        eq          get         items       le
ne          popitem     struct      values
clear       dict        ge          gt          keys        lt
pop         setdefault  update

Static methods:

fromkeys

Methods of py.dict inherited from handle.
```

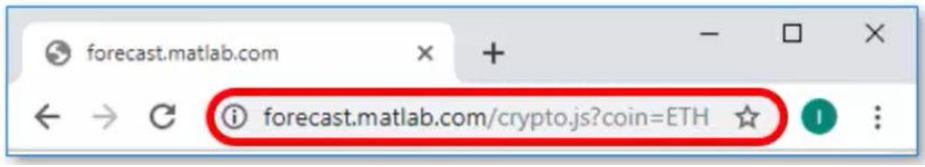
```
d.get('a')
```

```
ans = 1
```

4.4. Call Python User Defined Functions from MATLAB

In this chapter, we will leverage a demo developed by a Finance colleague. In this example, he is responsible for building enterprise web predictive analytics that other business critical applications can connect to as a web service. It follows the same structure as the weather example in chapter 2.

This web service is **forecasting the price of cryptocurrencies** [48] : forecast.matlab.com/crypto.js?coin=ETH



It returns data in the following form (JSON):

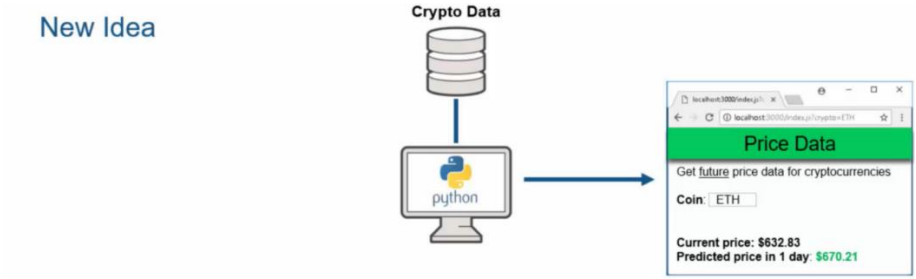
```
[{"Time":"2022-01-21T12:00:00Z","predictedPrice":2466.17},

...

{"Time":"2022-01-21T17:00:00Z","predictedPrice":2442.25}]
```

The first step is to develop an application that simply shows the historical price movement of a particular cryptocurrency:

[48] Forecasting the price of cryptocurrencies with Python and MATLAB https://www.mathworks.com/videos/integrating-python-with-matlab-1605793241650.html

This allows you to monitor the evolution of the price over the last 24 hours and take decisions to buy or sell your crypto assets based on this. Then one day, you manager comes to you and says:

"Hey, I have an idea. If we had access to the predicted forward-looking data as opposed to the historical data, we could make additional profit beyond what we're currently making, even if the prediction is 100% accurate. "



Let's assume the organization has a few quants that have extensive MATLAB expertise. And they know exactly how to build out such predictive models that the business users are looking for.

However, before we can get to that, our first challenge is to call the Python data scraping libraries and pull that data directly into MATLAB. Our first task at hand: Parse the cryptocurrency URL that we are connecting to, and just get out the domain name. For that, we want to use this function that's contained within the Python standard libraries and use it from within MATLAB. In this case, we are going to call a package `urllib`[49]. It contains a sub-module called `parse`, that contains in turn the function `urlparse`.

```
 startDate = '2022-01-21T12:00:00Z';
 stopDate = '2022-01-21T17:00:00Z';
 url = "https://api.pro.coinbase.com/products/ETH-
USD/candles?start="+startDate+"&end="+stopDate+"&granularity=60";
 urlparts = py.urllib.parse.urlparse(url)
```

```
urlparts =
  Python ParseResult with properties:

    fragment
    hostname
    netloc
    params
```

[49] urllib package from the Python standard library: https://docs.python.org/3/library/urllib.html

```
    password
    path
    port
    query
    scheme
    username

    ParseResult(scheme='https', netloc='api.pro.coinbase.com',
path='/products/ETH-USD/candles', params='', query='start=2022-01-
23T01:00:00Z&end=2022-01-23T06:00:00Z&granularity=60', fragment='')
```

```
domain = urlparts.netloc
```

```
domain =
  Python str with no properties.

    api.pro.coinbase.com
```

To avoid the unnecessary back and forth of intermediate data between MATLAB and Python, we write a **Python User Defined Module**[50], called `dataLib.py` with a few functions in it:

```
jsonData = py.dataLib.getPriceData("ETH", startDate, stopDate)
historicalPrices = py.dataLib.parseJson(jsonData, [0,4])
```

`dataLib.py` imports 1-minute bars from Coinbase Pro[51]. Note, the API does not fetch the first minute specified by the start date so the times span (start, stop]. To return data we are using a variety of data structures from Numpy arrays to lists and dictionaries, and even JSON.

This is what the function looks like:

```
def getPriceData(coin, start, stop):
# returns back a python list containing the historical data
# of the cryptocurrency 'product' for the period of time
# ('start', 'stop'].


    import urllib.request
```

[50] Call User-Defined Python Module: https://www.mathworks.com/help/matlab/matlab_external/call-user-defined-custom-module.html
[51] Coinbase Pro: https://pro.coinbase.com/

```
    import json
    import os

    # website we want to pull data from
    hostname = 'api.pro.coinbase.com'

    # all cryptocurrency products returned in USD currency
    product = coin + '-USD'

    # granularity is in seconds, so we are getting 1-minute bars
    granularity = '60'

    # returns back: [date, low, high, open, close, volume]
      url = 'https://' + hostname + '/products/' + product +
'/candles?start=' + start + '&end=' + stop + '&granularity=' +
granularity

    # execute call to the website
         urlRequest = urllib.request.Request(url, data=None,
headers={'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0.1916.47
Safari/537.36'})
    response = urllib.request.urlopen(urlRequest)
    html = response.read()

    # python 3.x requires decoding from bytes to string
    data = json.loads(html.decode())
    return data
```

This is how you would call this function from MATLAB:

```
product = "ETH";
startDate = '2022-01-21T12:00:00Z';
stopDate = '2022-01-21T17:00:00Z';
jsonData = py.dataLib.getPriceData(product, startDate, stopDate)
```

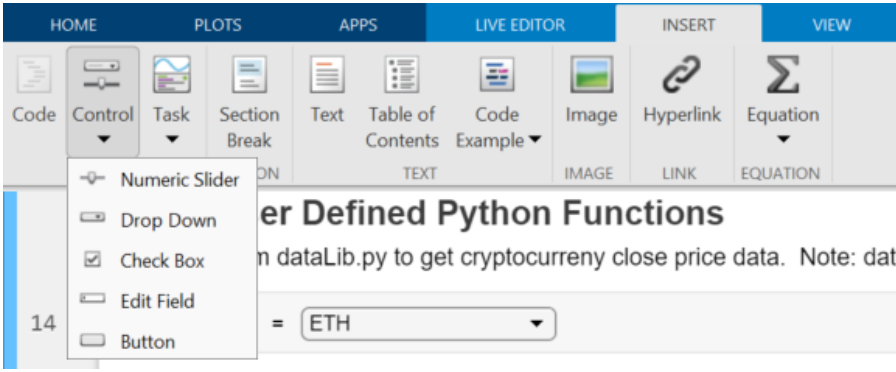
```
jsonData =
  Python list with no properties.
```

```
    [[1642917600, 2466.17, 2473.56, 2468.52, 2469.96, 258.02707836],
...
[1642899660, 2442.25, 2446.79, 2446.77, 2445.17, 276.4743004]]
```

If you want to add interactivity to your Live Script, you can add so called **Live Controls**[52]. This is helpful to point other people to areas where you may want to change parameters or select things to do scenario analysis.

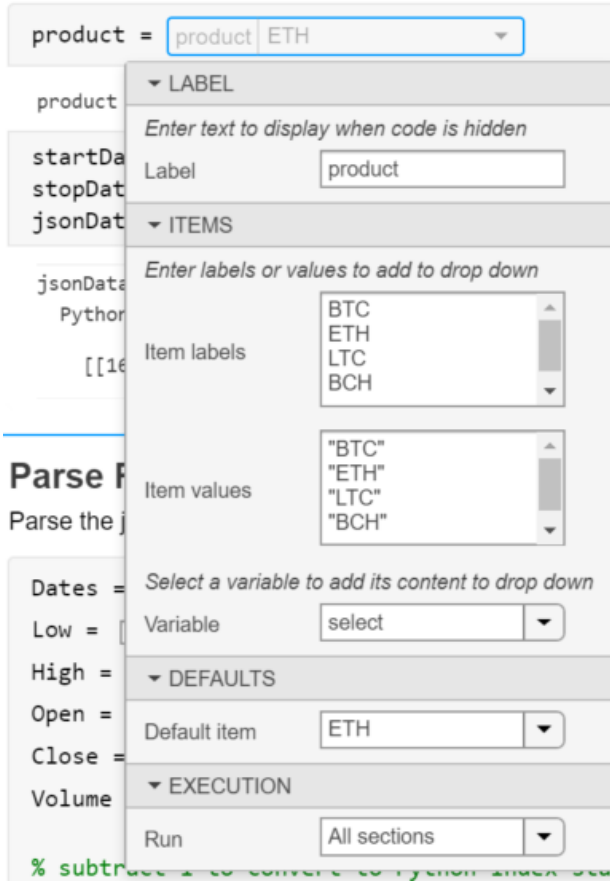

```
product = "ETH"
```

You can insert controls from the ribbon:



This is how you would parametrize the Live Control:

[52] Add Interactive Controls to a Live Script:
https://www.mathworks.com/help/matlab/matlab_prog/add-interactive-controls-to-a-live-script.html



Another type of Live Control that is useful here are simple checkboxes to select the information we want to return from the `parseJson` function:

```
Dates = ☑ ;
Low = ☐ ;
High = ☐ ;
Open = ☐ ;
Close = ☑ ;
Volume = ☐ ;

% subtract 1 to convert to Python index starting at 0
selectedColumns = find([Dates Low High Open Close Volume])-1
```

```
selectedColumns = 1×2
     0     4
```

Pay attention to the fact that we are subtracting 1 to the resulting array to adapt to Python indexing starting at 0.

```
% this function returns back two outputs as a tuple
data = py.dataLib.parseJson(jsonData, selectedColumns);
```

The last thing we will do in this part of the story is to convert the Python function outputs do MATLAB Data types (this will be covered in the last section of this chapter on mapping data between Python and MATLAB).

In this case, we have a complex tuple that has a Numpy arrays inside of it, as well as a list. We can easily split up the tuple by using syntax like this:

```
priceData = data{1}
```

```
priceData =
  Python ndarray:

   1.0e+09 *

    1.6429    0.0000
    1.6429    0.0000
       ...
    1.6429    0.0000
    1.6429    0.0000

    Use details function to view the properties of the Python object.

    Use double function to convert to a MATLAB array.
```

```
columnNames = data{2}
```

```
columnNames =
  Python list with no properties.

    ['Date', 'Close']
```

Then we can cast over the Numpy array on the right-hand side by just using the double command:

```
priceData = double(priceData)
```

```
priceData = 300×2
10^9 ×
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
    1.6429    0.0000
      ⋮
```

Likewise, we have a variety of commands for casting lists like string (or cell before R2022a):

```
columnNames = string(columnNames);
```

Once we have those data in MATLAB, we will convert it over to the MATLAB table, which is basically equivalent to Pandas data frames:

```
data = array2table(priceData, 'VariableNames', columnNames);
```

Like tables, timetable are built-in data constructs that appeared in MATLAB over the last couple of years to make our lives easy for doing simple types of tasks or even complex types of tasks. If I want to deal with time zones and convert the times – which are with respect to universal time zone – to a view of someone who is in New York, the command datetime[53] allows us to do that conversion:

```
data.Date = datetime(data.Date, 'ConvertFrom', 'posixtime',
'TimeZone', 'America/New_York')
```

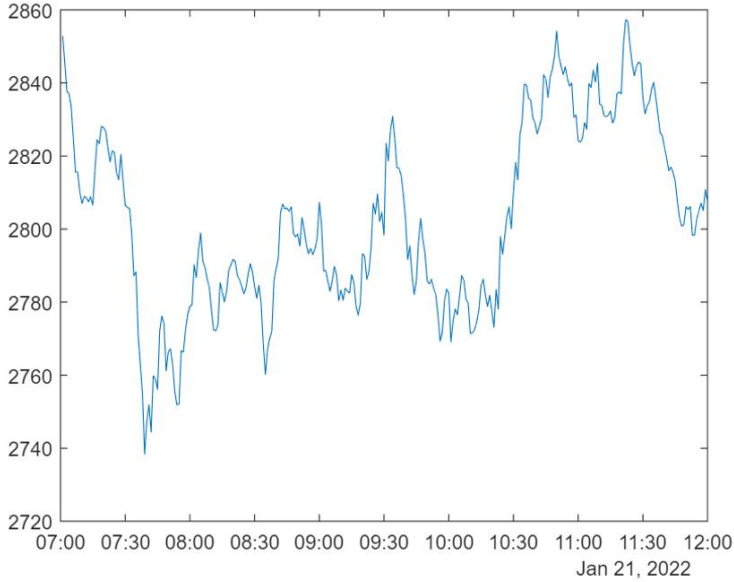
```
data = 300×2 table
```

	Date	Close
1	21-Jan-2022 12:00:00	2.8073e+03
2	21-Jan-2022 11:59:00	2.8108e+03
3	21-Jan-2022 11:58:00	2.8051e+03
4	21-Jan-2022 11:57:00	2.8071e+03

[53] MATLAB datetime: https://www.mathworks.com/help/matlab/ref/datetime.html#d123e298898

	Date	Close
5	21-Jan-2022 11:56:00	2.8051e+03
	⋮	

```
plot(data.Date, data.Close)
```



Reload Modified User-Defined Python Module[54]

What if you've made modifications to the functions inside of your `dataLib` module? You call those again from MATLAB, but you don't see any difference. It is because you need to reload the module:

```
mod = py.importlib.import_module('dataLib');
py.importlib.reload(mod);
```

You may need to unload the module first, by clearing the classes. This will delete all variables, scripts and classes in your MATLAB workspace.

```
clear classes
```

[54] Reload Modified User-Defined Python Module: https://www.mathworks.com/help/matlab/matlab_external/call-user-defined-custom-module.html#buuz303

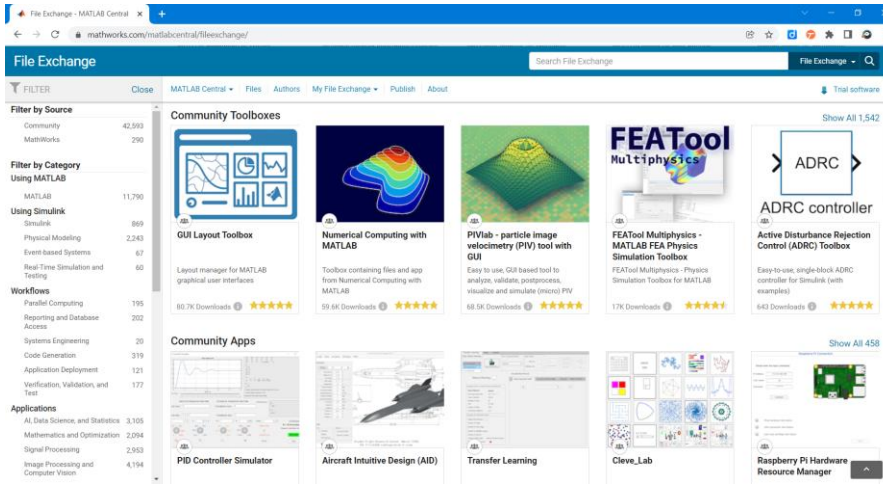
If you're running Python out-of-process, another approach is to simply terminate the process[55].

```
terminate(pyenv)
```

4.5. CALL PYTHON COMMUNITY PACKAGES FROM MATLAB

In some scientific fields like earth and climate sciences, we observe a growing Python community. But as programming skills may vary a lot in researchers and engineers, a MATLAB interface to Python community packages can open up some domain specific capabilities to the 5M+ MATLAB community.

One great example of this is the Climate Data Store Toolbox[56] developed by Rob Purser, a fellow MathWorker. Rob and I are part of the MathWorks Open Source Program. We are promoting open-source, both to support the use of open-source software in MathWorks products and to help for MathWorkers to contribute their work on GitHub and the MATLAB File Exchange[57].



In this section we will demonstrate with the Climate Data Store Toolbox how to build MATLAB toolboxes on top of Python packages. It relies on the CDS Python

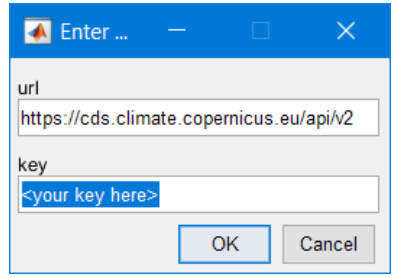
[55] Reload Out-of-Process Python Interpreter: https://www.mathworks.com/help/matlab/matlab_external/reload-python-interpreter.html

[56] Rob Purser (2022). Climate Data Store Toolbox for MATLAB https://github.com/mathworks/climatedatastore

[57] MATLAB File Exchange: https://www.mathworks.com/matlabcentral/fileexchange/

API [58] created by the European Centre for Medium-Range Weather Forecasts (ECMWF). The toolbox will automatically configure Python, download and install the CDSAPI package (you can manually do it using `pip install cdsapi`). You will need to create an account on https://cds.climate.copernicus.eu/ to retrieve data.

The first time you use it, it will prompt you for CSAPI credentials.



A well written toolbox like this one throwing an error coming from Python will forward this error:

```
  datasetName ="satellite-sea-ice-thickness";
  options.version = "1_0";
  options.variable = "all";
  options.satellite = "cryosat_2";
  options.cdr_type = ["cdr","icdr"];
  options.year = ["2011","2021"];
  options.month = "03";
  [downloadedFilePaths,citation] =
climateDataStoreDownload('satellite-sea-ice-thickness',options);
```

```
2022-01-20 19:33:13,558 INFO Welcome to the CDS
2022-01-20 19:33:13,577 INFO Sending request to https://cds.climate.
copernicus.eu/api/v2/resources/satellite-sea-ice-thickness
Error using api>_api
Python Error: Exception: Client has not agreed to the required terms
and conditions.. To access this resource, you first need to accept
the terms of 'Licence to use Copernicus Products' at
https://cds.climate.copernicus.eu/cdsapp/#!/terms/licence-to-use-
copernicus-products
Error in api>retrieve (line 348)

Error in climateDataStoreDownload (line 60)
retrieveFromCDS(name,options,zipfilePath);
```

[58] CDS Python API: https://github.com/ecmwf/cdsapi

This error for instance indicates that an exception has been raised on the Python side. In this case the culprit is located in the following MATLAB function:

```
function retrieveFromCDS(name,options,zipfilePath)
% Utility function to isolate python code so that we don't trigger the
% python check until after python install checks are done.

% Copyright 2021 The MathWorks, Inc.
    c = py.cdsapi.Client();

    % Don't show the progress information
    c.quiet = true;
    c.progress = false;
    c.retrieve(name,options,zipfilePath);
end
```

Once imported with Python, the NetCDF files are read with MATLAB using ncread[59] and storing information as timetable [60] with the function readSatelliteSeaIceThickness[61]:

```
 ice2011 = readSatelliteSeaIceThickness("satellite-sea-ice-
thickness\ice_thickness_nh_ease2-250_cdr-v1p0_201103.nc");
 ice2021 = readSatelliteSeaIceThickness("satellite-sea-ice-
thickness\ice_thickness_nh_ease2-250_icdr-v1p0_202103.nc");
 head(ice2021)
```

```
ans = 8×3 timetable
```

	time	lat	lon	thickness
1	01-Mar-2021	47.6290	144.0296	2.4566
2	01-Mar-2021	47.9655	144.0990	2.5800
3	01-Mar-2021	50.5072	148.0122	-0.0364
4	01-Mar-2021	50.8360	148.1187	1.0242
5	01-Mar-2021	50.3237	146.9969	0.0518
6	01-Mar-2021	51.1642	148.2269	0.2445
7	01-Mar-2021	50.9112	147.6573	0.8933
8	01-Mar-2021	50.6540	147.0948	0.1271

[59] Read data from variable in NetCDF data source https://www.mathworks.com/help/matlab/ref/ncread.html

[60] MATLAB timetable: https://www.mathworks.com/help/matlab/timetables.html

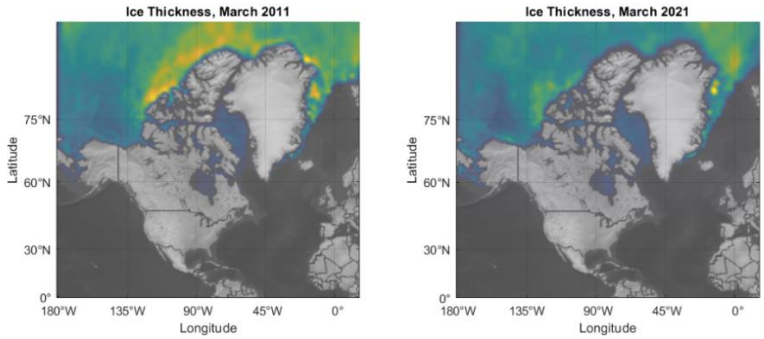
[61] https://github.com/mathworks/climatedatastore/blob/main/doc/readSatelliteSeaIceThickness.m

```
disp(citation)
```

```
Generated using Copernicus Climate Change Service information 2022
```

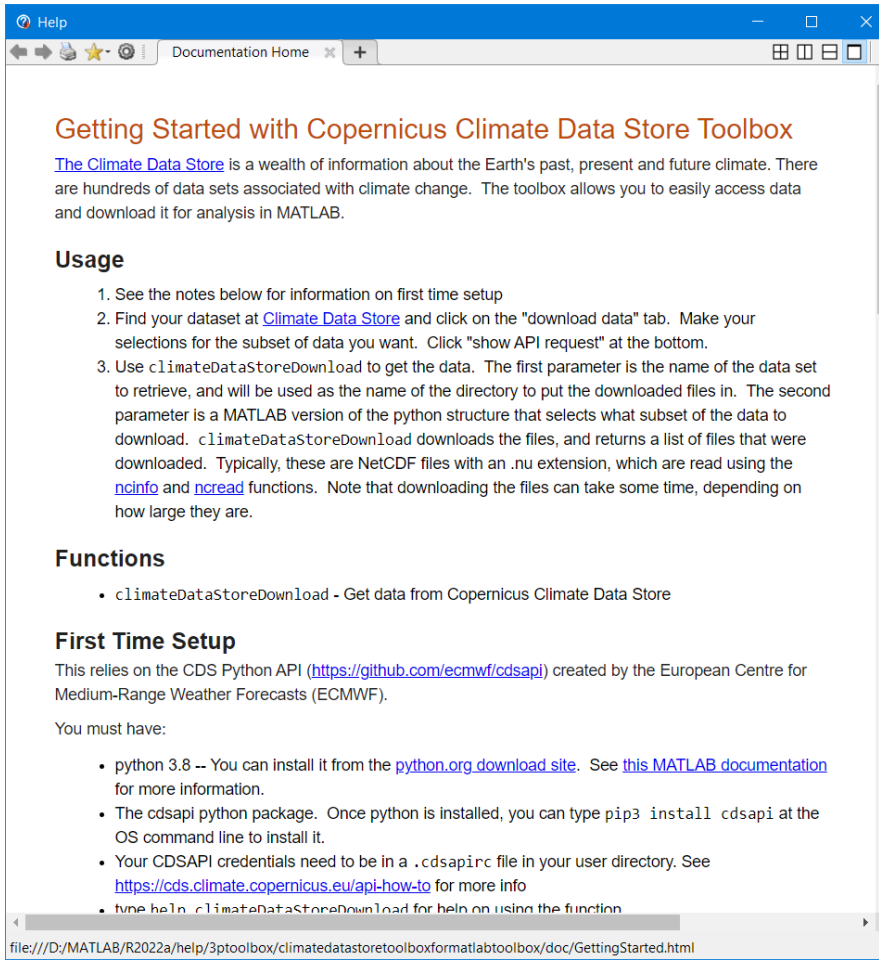
This toolbox leverages the beautiful geoplotting[62] capabilities of MATLAB:

```
subplot(1,2,1)
geodensityplot(ice2011.lat,ice2011.lon,ice2011.thickness,"FaceColor
","interp")
geolimits([23 85],[-181.4 16.4])
geobasemap("grayterrain")
title("Ice Thickness, March 2011")
subplot(1,2,2)
geodensityplot(ice2021.lat,ice2021.lon,ice2021.thickness,"FaceColor
","interp")
geolimits([23 85],[-181.4 16.4])
geobasemap("grayterrain")
title("Ice Thickness, March 2021")
f = gcf;
f.Position(3) = f.Position(3)*2;
```



In a well written toolbox like this one, you find a documentation that is packaged directly with it.

[62] Geographic density plot: https://www.mathworks.com/help/matlab/ref/geodensityplot.html



You can create your own toolbox and share it with others. These files can include MATLAB code, data, apps, examples, and documentation. When you create a toolbox, MATLAB generates a single installation file (.mltbx) that enables you or others to install your toolbox.

Read more on how to create and share toolboxes[63]

4.6. Debug Python code called by MATLAB

One of the first difficulty you will face when developing bilingual applications, is debugging across the language boundary. In the following examples we will demonstrate how to attach a MATLAB session to a VSCode or Visual Studio process

[63] Create and share toolboxes: https://www.mathworks.com/help/matlab/matlab_prog/create-and-share-custom-matlab-toolboxes.html

to debug the Python part of your app. In the next chapter, we will see how to do the opposite, debug the MATLAB part with the nice MATLAB Debugger.

4.6.1. Debug with Visual Studio Code

This section is showing in 8 steps how to debug Python code called from MATLAB with VSCode[64]:

1. Install VS Code and create a project.
See the official tutorial[65] for instructions on how to install Visual Studio Code, set up a Python project, select a Python interpreter, and create a `launch.json` file.

2. In a terminal, install the debugpy module using, for example,

```
$ python -m pip install debugpy
```

3. In VS Code, add the following debugging code to the top of your Python module.

```
import debugpy
debugpy.debug_this_thread()
```

4. Configure the `launch.json` file to select and attach to MATLAB using the code below.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Attach to MATLAB",
            "type": "python",
            "request": "attach",
            "processId": "${command:pickProcess}"
        }
    ]
}
```

5. Add breakpoints to your code.

[64] How can I debug Python code using MATLAB's Python Interface and Visual Studio Code: https://www.mathworks.com/matlabcentral/answers/1645680-how-can-i-debug-python-code-using-matlab-s-python-interface-and-visual-studio-code

[65] Configure and run the debugger: https://code.visualstudio.com/docs/python/python-tutorial#_configure-and-run-the-debugger

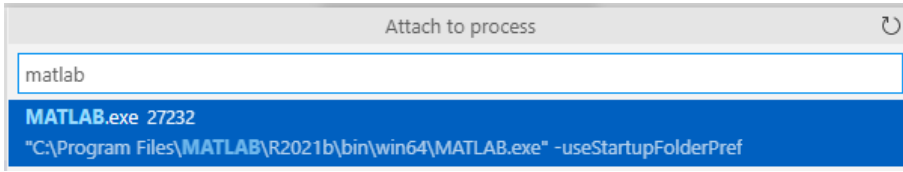
6. Set up your Python environment in MATLAB and get the `ProcessID` number. In this example, the `ExecutionMode` is set to `InProcess`.

```
>> pyenv

ans =

  PythonEnvironment with properties:

         Version: "3.9"
      Executable:
"C:\Users\username\AppData\Local\Programs\Python\Python39\python.exe"
         Library:
"C:\Users\username\AppData\Local\Programs\Python\Python39\python39.dl
l"
            Home:
"C:\Users\username\AppData\Local\Programs\Python\Python39"
          Status: Loaded
    ExecutionMode: InProcess
        ProcessID: "27664"
      ProcessName: "MATLAB"
```

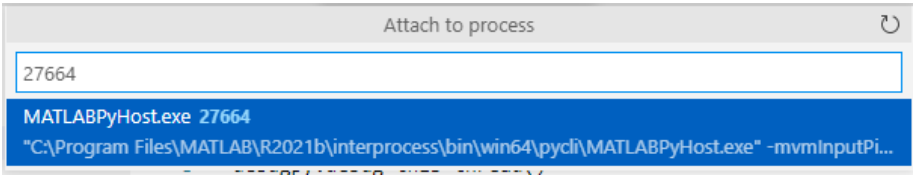
If you see `Status: NotLoaded`, execute any Python command to load the Python interpreter (for example `>> py.list`) then execute the `pyenv` command to get the `ProcessID` for the MATLAB process.

7. Attach the MATLAB process to VS Code.
In VS Code, select "Run and Debug" (Ctrl+Shift+D), then select the arrow to Start Debugging (F5). In this example, the green arrow has the label "Attach to MATLAB". Note that this corresponds to the value of the "name" parameter that you specified in the `launch.json` file. Type "matlab" in the search bar of the dropdown menu and select the "MATLAB.exe" process that matches the ProcessID from the output of the pyenv command. Note that if you are using OutOfProcess execution mode, you will need to search for a "MATLABPyHost.exe" process.

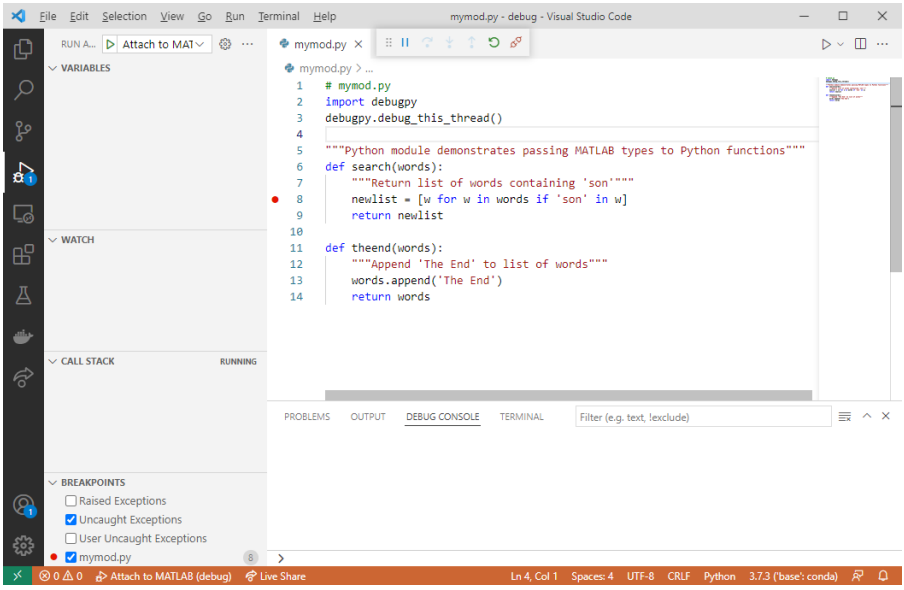
In-process:



Out-of-Process:



8. Invoke the Python function from MATLAB. Execution should stop at the breakpoint.

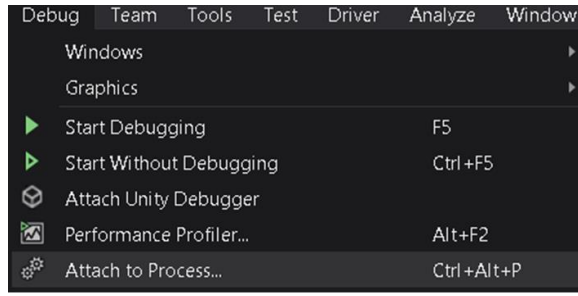


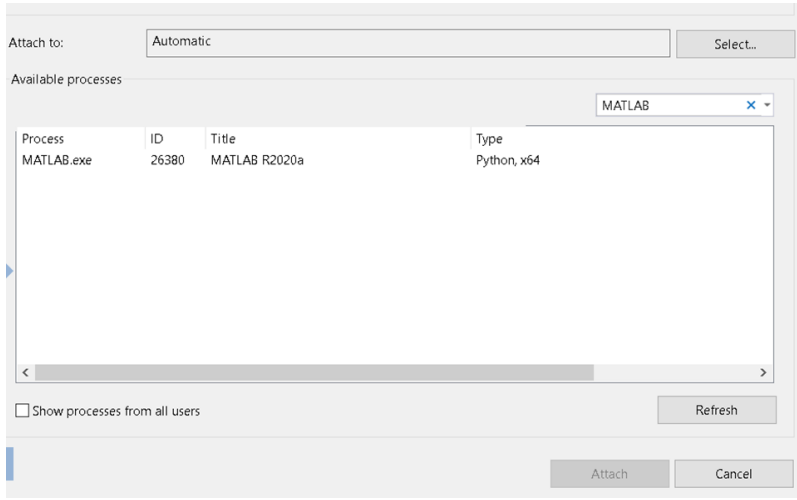
Run the following MATLAB code to step into the Python function `search`:

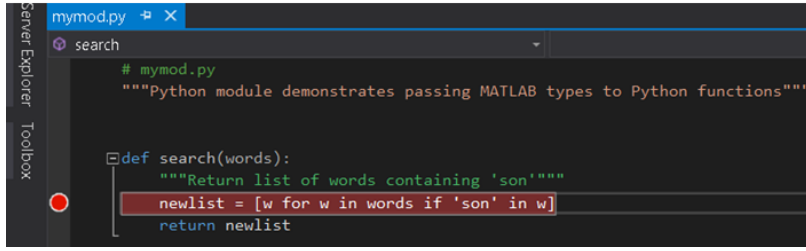
```
>> N = py.list({'Jones','Johnson','James'})
>> py.mymod.search(N)
```

4.6.2. DEBUG WITH VISUAL STUDIO

If you have access to Visual Studio and you are more familiar with it, you can do the same as before with Visual Studio[66]. Open Visual Studio and create a new Python project from existing code. Then, select Attach to Process from the Debug menu:







[66] Calling Python from Matlab - how to debug Python code? https://stackoverflow.com/questions/61708900/calling-python-from-matlab-how-to-debug-python-code

4.7. MAPPING DATA BETWEEN PYTHON AND MATLAB

In his book about *Python for MATLAB Development*[67], Albert Danial shares some clever functions to convert MATLAB variables into an equivalent Python-native variable with mat2py[68], and vice-versa with py2mat[69].

Converting data [70] returned by Python function inside of MATLAB may require understanding some of the differences in the native datatypes of the two languages:

- Scalars (integers, floating point numbers, …), text and Booleans
- Dictionaries and lists
- Arrays and dataframes

Some specialized MATLAB data types like *timetable* or *categorical* will require some extra love and need to be converted manually. Of course, we can still use these data types in our functions, but the functions need to return types that the Python interpreter can understand.

4.7.1. SCALARS

The table below shows the mappings for common scalar data types:

MATLAB	Python
double, single	float
complex single, complex double	complex
(u)int8, (u)int16, (u)int32, (u)int64	int
NaN	float(nan)
Inf	float(inf)
string, char	str
logical	bool

By default, numbers in MATLAB are double, whereas numbers without decimal point in Python are integers.

```
a = py.dataExchange.get_float()
```

```
a = 1
```

[67] Python for MATLAB Development, Albert Danial: https://link.springer.com/book/10.1007/978-1-4842-7223-7
[68] mat2py: https://github.com/Apress/python-for-matlab-development/blob/main/code/matlab_py/mat2py.m
[69] py2mat: https://github.com/Apress/python-for-matlab-development/blob/main/code/matlab_py/py2mat.m
[70] Converting data: https://www.mathworks.com/help/matlab/matlab_external/passing-data-to-python.html

```
  class(a)
```

```
ans = 'double'
```

```
  b = py.dataExchange.get_complex()
```

```
b = 2.0000 + 0.0000i
```

```
  class(b)
```

```
ans = 'double'
```

There are several kinds of integers in MATLAB, depending on the precision you require.

For instance uint8 can only store positive numbers between 0 and 255, whereas int8 covers the range $[-2^7,2^7-1]$. The most generic type to convert Python integers are int64, which you can do explicitly.

```
  c = py.dataExchange.get_integer()
```

```
c =
  Python int with properties:

    denominator: [1×1 py.int]
           imag: [1×1 py.int]
      numerator: [1×1 py.int]
           real: [1×1 py.int]

    3
```

```
  class(c)
```

```
ans = 'py.int'
```

```
  int64(c)
```

```
ans = int64
3
```

When getting a string from a Python function, the convertion isn't obvious. It can either be turned into a char (character array) or a string. You can distinguish them by the single quotation marks for chars, and double quotes for strings.

```
abc = py.dataExchange.get_string()
```

```
abc =
  Python str with no properties.
    abc
```

```
char(abc)
```

```
ans = 'abc'
```

```
class(char(abc))
```

```
ans = 'char'
```

```
string(abc)
```

```
ans = "abc"
```

```
class(string(abc))
```

```
ans = 'string'
```

Finally, the last basic datatype that contains a logical information is called a boolean in Python:

```
py.dataExchange.get_boolean()
```

```
ans = logical
   1
```

4.7.2. Dictionaries and Lists

This is how containers map to each other between the two languages:

MATLAB	Python
structure	dict
cell arrays	list, tuple

To illustrate the conversion of Python dictionaries and lists into MATLAB containers, we will reuse the example from chapter 2. JSON data are really close to dictionaries in Python, which makes the data processing very easy when accessing data from web services.

```
 url=
webread("https://samples.openweathermap.org").products.current_weath
er.samples{1};
 r = py.urllib.request.urlopen(url).read();
 json_data = py.json.loads(r);
 py.weather.parse_current_json(json_data)
```

```
ans =
  Python dict with no properties.

    {'temp': 280.32, 'pressure': 1012, 'humidity': 81, 'temp_min':
279.15, 'temp_max': 281.15, 'speed': 4.1, 'deg': 80, 'lon': -0.13,
'lat': 51.51, 'city': 'London', 'current_time': '2022-05-22
22:15:18.161296'}
```

Dictionaries can contain scalars, but also other datatypes like lists.

```
 url2 =
webread("https://samples.openweathermap.org").products.forecast_5day
s.samples{1};
 r2 = py.urllib.request.urlopen(url2).read();
 json_data2 = py.json.loads(r2);
 forecast = struct(py.weather.parse_forecast_json(json_data2))
```

```
forecast = struct with fields:
    current_time: [1×40 py.list]
            temp: [1×40 py.list]
             deg: [1×40 py.list]
           speed: [1×40 py.list]
        humidity: [1×40 py.list]
        pressure: [1×40 py.list]
```

```
 forecastTemp = forecast.temp;
 forecastTime = forecast.current_time;
```

Lists containing only numeric data can be converted into doubles since MATLAB R2022a:

```
double(forecastTemp)
```

```
ans = 1×40
  261.4500  261.4100  261.7600  261.4600  260.9810  262.3080
263.7600 ⋯
```

And any lists can be converted to string (even those containing a mix of text and numeric data).

```
forecastTimeString = string(forecastTime);
datetime(forecastTimeString)
```

```
ans = 1×40 datetime
30-Jan-2017 18:00:0030-Jan-2017 21:00:0031-Jan-2017 00:00:0031-Jan-
2017 03:00: ⋯
```

Before MATLAB R2022a, Python lists need to be converted into MATLAB cell arrays[71]. Cells can then be transformed to double, strings, with the cellfun[72] function. The previous code would look like this until R2021b:

```
forecastTempCell = cell(forecastTemp)
```

```
forecastTempCell = 1×40 cell
```

	1	2	3	4	5	6	7	⋯
1	261.4500	261.4100	261.7600	261.4600	260.9810	262.3080	263.7600	

```
cellfun(@double,forecastTempCell)
```

```
ans = 1×40
  261.4500  261.4100  261.7600  261.4600  260.9810
262.3080  263.7600 ⋯
```

```
forecastTimeCell = cell(forecastTime)
```

```
forecastTimeCell = 1×40 cell
```

	1	2	3	4	5	6	7	⋯
1	1×19 str	1×19 str	1×19 str	1×19 str	1×19 str	1×19 str	1×19 str	

[71] MATLAB Cell Arrays: https://www.mathworks.com/help/matlab/cell-arrays.html
[72] MATLAB cellfun Function: https://www.mathworks.com/help/matlab/ref/cellfun.html

cellfun(@string,forecastTimeCell)

```
ans = 1×40 string
"2017-01-30 18:0…  "2017-01-30 21:0…  "2017-01-31 00:0…
"2017-01-31 03:0…  "2 …
```

4.7.3. ARRAYS

By modifying the parse_forecast_json function in the weather module, we output Python arrays[73] instead of lists. There exists indeed a native array datatype in base Python.

```
forecast2 = struct(py.weather.parse_forecast_json2(json_data2))
```

```
forecast2 = struct with fields:
    current_time: [1×40 py.list]
            temp: [1×40 py.array.array]
             deg: [1×40 py.array.array]
           speed: [1×40 py.array.array]
        humidity: [1×40 py.array.array]
        pressure: [1×40 py.array.array]
```

The MATLAB double function will convert the Python array into a MATLAB array

```
double(forecast2.temp)
```

```
ans = 1×40
  261.4500  261.4100  261.7600  261.4600  260.9810  262.3080
263.7600 …
```

Those data conversion also apply to Numpy arrays:

```
npA = py.numpy.array([1,2,3;4,5,6;7,8,9])
```

```
npA =
  Python ndarray:

     1     2     3
     4     5     6
     7     8     9
```

[73] Python Array: https://docs.python.org/3/library/array.html

```
    Use details function to view the properties of the Python object.

    Use double function to convert to a MATLAB array.
```

```
  double(npA)
```

```
ans = 3×3
      1      2      3
      4      5      6
      7      8      9
```

4.7.4. DATAFRAMES

One common question on data transfer, is how to exchange data between MATLAB tables and Pandas Dataframes. Since 24a, Pandas Dataframes can be casted into MATLAB tables:

```
  d.a=[1,2,3];
  d.b=[4,5,6];
  df = py.pandas.DataFrame(d)
```

```
df =
  Python DataFrame with properties:

          T: [1×1 py.pandas.core.frame.DataFrame]
         at: [1×1 py.pandas.core.indexing._AtIndexer]
      attrs: [1×1 py.dict]
       axes: [1×2 py.list]
    columns: [1×1 py.pandas.core.indexes.base.Index]
     dtypes: [1×1 py.pandas.core.series.Series]
      empty: 0
      flags: [1×1 py.pandas.core.flags.Flags]
        iat: [1×1 py.pandas.core.indexing._iAtIndexer]
       iloc: [1×1 py.pandas.core.indexing._iLocIndexer]
      index: [1×1 py.pandas.core.indexes.range.RangeIndex]
        loc: [1×1 py.pandas.core.indexing._LocIndexer]
       ndim: [1×1 py.int]
      shape: [1×2 py.tuple]
       size: [1×1 py.numpy.int32]
      style: [1×1 py.pandas.io.formats.style.Styler]
     values: [1×1 py.numpy.ndarray]
```

```
      a    b
0  1.0  4.0
1  2.0  5.0
2  3.0  6.0
```

```
T = table(df)
```

```
T = 3×2 table
```

	a	b
1	1	4
2	2	5
3	3	6

Before 24a, the recommended solution was to rely on Parquet files[74]. Parquet is a columnar storage format that enables to store & transfer tabular data between languages. It is available to any project in the Hadoop big data ecosystem, regardless of the choice of data processing framework, data model or programming language (more on Parquet[75]).

This example demonstrates a back and forth between Pandas DataFrames and MATLAB Tables:

pq_CreateDataframe.py

```
import pandas as pd
import numpy as np

# create dataframe
df = pd.DataFrame({'column1': [-1, np.nan, 2.5],
'column2': ['foo', 'bar', 'tree'],
'column3': [True, False, True]})
print(df)
```

[74] Parquet file support in MATLAB: https://www.mathworks.com/help/matlab/parquet-files.html
[75] Apache Parquet project: https://parquet.apache.org/

```
# save dataframe to parquet file via pyarrow library
df.to_parquet('data.parquet', index=False)
```

Read in parquet file

```
% info = parquetinfo('data.parquet')
data = parquetread('data.parquet')
```

data = 3×3 table

	column1	column2	column3
1	-1	"foo"	1
2	NaN	"bar"	0
3	2.5000	"tree"	1

Examine datatype of a particular column

```
class(data.column2)
```

```
ans = 'string'
```

Change data in table

```
data.column2 = ["orange"; "apple"; "banana"];
```

Write the results back to parquet

```
parquetwrite('newdata.parquet', data)
```

Finally read the modified DataFrame back in Python:

pq_ReadTable.py

```
import pandas as pd

# read parquet file via pyarrow library
df = pd.read_parquet('newdata.parquet')
print(df)
```

5. CALL PYTHON AI LIBRARIES FROM MATLAB

In this Chapter we will look at different Python libraries for Artificial Intelligence, both Machine Learning & Deep Learning (like Scikit-learn and TensorFlow) and how to call them from MATLAB.

Those steps can be integrated in a typical AI workflow[76]:

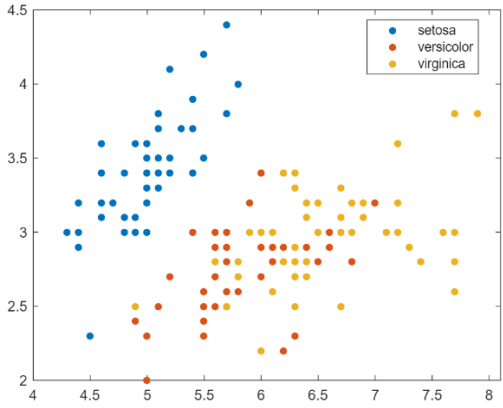
5.1. CALL SCIKIT-LEARN FROM MATLAB

The Iris flower dataset[77] is a multivariate data set introduced by the British statistician and biologist Ronald Fisher. This data set consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray. The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.

You can also find this dataset in MATLAB, as it is shipped with a list of Sample Data Sets[78]:

```
load fisheriris.mat
gscatter(meas(:,1),meas(:,2),species)
```

[76] AI Workflow: https://www.mathworks.com/discovery/artificial-intelligence.html
[77] Iris dataset: https://en.wikipedia.org/wiki/Iris_flower_data_set
[78] Sample Data Sets in MATLAB: https://www.mathworks.com/help/stats/sample-data-sets.html



Or retrieve the dataset from the Scikit-learn library[79] (inside of MATLAB still):

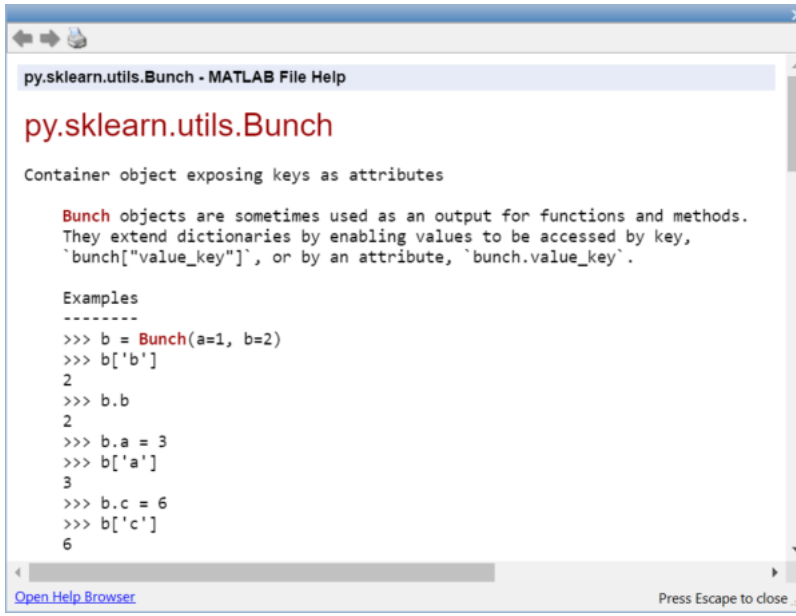
```
iris_dataset = py.sklearn.datasets.load_iris()
```

```
iris_dataset =
  Python Bunch with no properties.

    {'data': array([[5.1, 3.5, 1.4, 0.2],
           [4.9, 3. , 1.4, 0.2],
           [4.7, 3.2, 1.3, 0.2],
           [4.6, 3.1, 1.5, 0.2],
           [5. , 3.6, 1.4, 0.2],
           [5.4, 3.9, 1.7, 0.4],
...
               [6.2, 3.4, 5.4, 2.3],
           [5.9, 3. , 5.1, 1.8]]),

'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,
           0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1,
           1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1,
           1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2,
```

[79] Iris dataset in Scikit-learn: https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

```
           2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2,
           2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]),
'frame': None, 'target_names': array(['setosa', 'versicolor',
'virginica'], dtype='<U10'), 'DESCR': ...
```

Scikit-learn datasets are returned as a “Bunch object”. You can access the Python modules documentation directly from within MATLAB:



This dataset can be passed to MATLAB as a struct:

```
struct(iris_dataset)
```

```
ans = struct with fields:
             data: [1×1 py.numpy.ndarray]
           target: [1×1 py.numpy.ndarray]
            frame: [1×1 py.NoneType]
     target_names: [1×1 py.numpy.ndarray]
            DESCR: [1×2782 py.str]
    feature_names: [1×4 py.list]
         filename: [1×8 py.str]
      data_module: [1×21 py.str]
```

The data manipulated in this dataset are by default stored as Numpy arrays. Read more on how to Pass Matrices and Multidimensional Arrays to Python[80]

```
X_np = iris_dataset{'data'}
```

```
X_np =
  Python ndarray:
    5.1000    3.5000    1.4000    0.2000
    4.9000    3.0000    1.4000    0.2000
    4.7000    3.2000    1.3000    0.2000
       ...

    Use details function to view the properties of the Python object.

    Use double function to convert to a MATLAB array.
```

```
X_ml = double(X_np);
X = X_ml(:,1:2)
```

```
X = 150×2
    5.1000    3.5000
    4.9000    3.0000
    4.7000    3.2000
    4.6000    3.1000
    5.0000    3.6000
    5.4000    3.9000
    4.6000    3.4000
    5.0000    3.4000
    4.4000    2.9000
    4.9000    3.1000
      ⋮
```

```
y = iris_dataset{'target'}
```

```
y =
  Python ndarray:
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

[80] https://www.mathworks.com/help/matlab/matlab_external/passing-data-to-python.html#mw_b64f3777-2204-45e9-8ced-f3f363096a49 - Pass Matrices and Multidimensional Arrays to Python

```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2

    Use details function to view the properties of the Python object.
    Use int64 function to convert to a MATLAB array.
```

We won't translate the Python ndarray into a MATLAB datatype just yet, as we will use a cool feature of Python to translate the list of ordinal values into a list of categorical species. Those features can be leveraged in MATLAB with a few calls to pyrun[81]

```
pyrun('dict = {0: "setosa",1: "versicolor", 2: "virginica"}')
% pass y as input, and retrieve species as output
s = pyrun('species=[dict[i] for i in y]','species',y = y)
```

```
s =
  Python list with no properties.
    ['setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
'setosa', 'setosa', ...
```

Finally, you can retrieve the Python list as a MATLAB categorical variable:

```
s = string(s);
species = categorical(s)
```

```
species = 1×150 categorical array
     setosa      setosa      setosa      setosa      setosa
setosa      setosa      ...
```

Another approach for the preprocessing in Python can be performed with pyrunfile

```
[X,y,species] = pyrunfile('iris_data.py',{'Xl','y','species'})
```

[81] pyrun: https://www.mathworks.com/help/matlab/ref/pyrun.html

```
X =
  Python list with no properties.
    [[5.1, 3.5], [4.9, 3.0], [4.7, 3.2], [4.6, 3.1], [5.0, 3.6],
[5.4, 3.9], [4.6, 3.4],...

y =
  Python ndarray:
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1   1   1   1   1   1   1   1   1   1   1   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
2   2   2   2   2   2   2

    Use details function to view the properties of the Python object.
    Use int64 function to convert to a MATLAB array.

species =
  Python list with no properties.
    ['setosa', 'setosa', 'setosa', 'setosa', 'setosa', 'setosa',
'setosa', 'setosa', ...
```

This is what the python scripts looks like:

iris_data.py

```
from sklearn import datasets
iris = datasets.load_iris()

X = iris.data[:, :2]  # we only take the first two features (sepal)
Xl = X.tolist()
y = iris.target
dict = {0: "setosa",1: "versicolor", 2: "virginica"}
species = [dict[i] for i in y]
```

In this case, we are retrieving a list of lists, instead of a Numpy array. This will require some manual data marshalling:

```
Xc = cell(X)'
```

```
Xc = 150×1 cell array
    {1×2 py.list}
    {1×2 py.list}
    {1×2 py.list}
    {1×2 py.list}
    {1×2 py.list}
       ...
```

```
Xc1 = cell(Xc{1})
```

```
Xc1 = 1×2 cell array
    {[5.1000]}    {[3.5000]}
```

```
cell2mat(Xc1)
```

```
ans =
    5.1000    3.5000
```

The previous steps are included in the helper function `dataprep` (at the end of the live script):

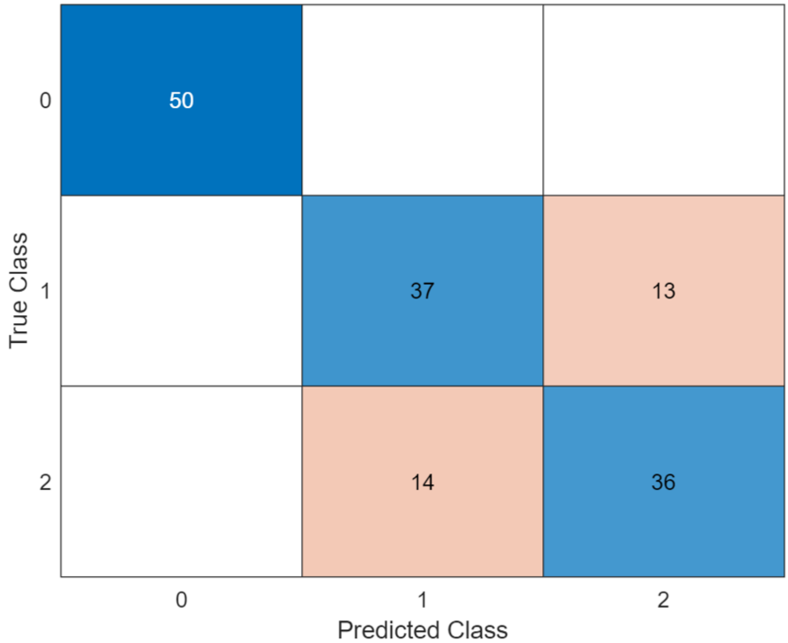
```
function Xp = dataprep(X)
  Xc = cell(X)';
  Xcc = cellfun(@cell,Xc,'UniformOutput',false);
  Xcm = cellfun(@cell2mat,Xcc,'UniformOutput',false);
  Xp = cell2mat(Xcm);
end
```

```
X_ml = dataprep(X);
y_ml = double(y);
s = string(species);
species = categorical(s);
```

Call the Scikit-Learn Logistic Regression and its fit and predict methods directly:

```
model = py.sklearn.linear_model.LogisticRegression();
model = model.fit(X,y); % pass by object reference
```

```
y2 = model.predict(X);
y2_ml = double(y2);
confusionchart(y_ml,y2_ml)
```



Call the Scikit-Learn model through a wrapper module:

```
model = py.iris_model.train(X,y);
y2 = py.iris_model.predict(model, X)
```

```
y2 = 
  Python ndarray:

    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    2    2    2
1    2    1    2    1    2    1    1    1    1    1    1    2    1    1    1    1    1
1    1    1    2    2    2    2    1    1    1    1    1    1    1    2    2    1    1
1    1    1    1    1    1    1    1    1    1    1    2    1    2    2    2    2    1
2    2    2    2    2    2    1    1    2    2    2    2    1    2    1    2    1    2
2    1    1    2    2    2    2    2    2    1    2    2    2    1    2    2    2    1
2    2    2    1    2    2    1

     Use details function to view the properties of the Python object.
```

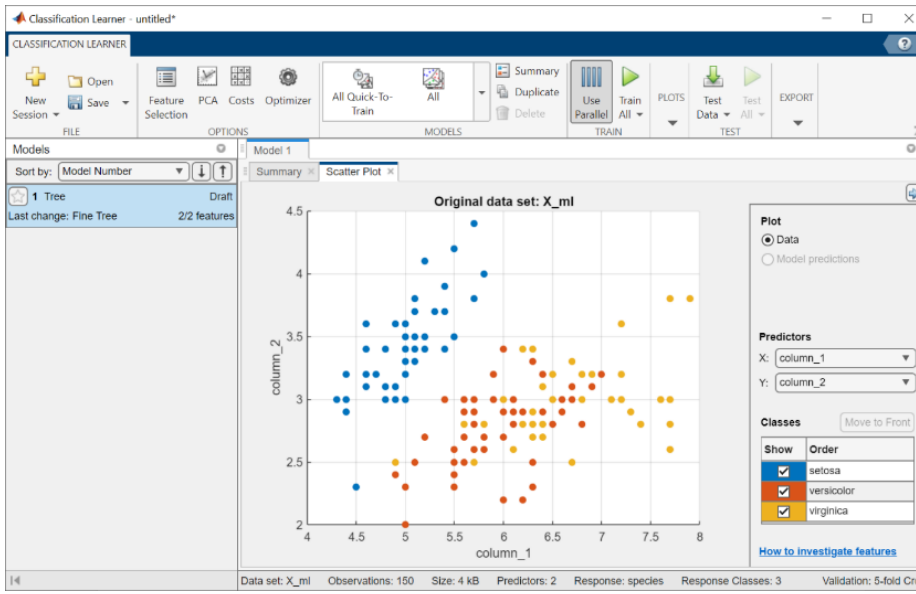
```
    Use int64 function to convert to a MATLAB array.
```

```
 sum(y_ml == y2)/length(y_ml) % precision of the model based on
training set
```

```
ans = 0.8200
```

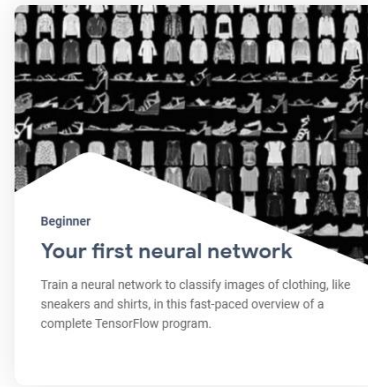
Alternatively, you can train all sorts of classification models in MATLAB. If you don't feel too comfortable with the various machine learning methods, you can simply try out the results from different types of models with an app:

```
 classificationLearner(X_ml,species)
```



5.2. CALL TENSORFLOW FROM MATLAB

Let's introduce the use of Tensorflow with the getting started tutorial[82]:



This guide uses the Fashion MNIST[83] dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels).

This example is curated by Zalando, under a MIT License.

First let's **load tensorflow** explicitely, and check the version of tensorflow installed:

```
tf = py.importlib.import_module('tensorflow');
pyrun('import tensorflow as tf; print(tf.__version__)')
```

```
2.8.0
```

Then let's **retrieve the dataset**

```
fashion_mnist = tf.keras.datasets.fashion_mnist;
train_test_tuple = fashion_mnist.load_data();
```

And store the images and labels for training and testing separately.

Indexing into Python tuples in MATLAB[84] is done with curly brackets: `pytuple{1}`

(Remember that indexing starts at 1 in MATLAB unlike Python starting at 0)

```
% ND array containing gray scale images (values from 0 to 255)
train_images = train_test_tuple{1}{1};
test_images = train_test_tuple{2}{1};
% values from 0 to 9: can be converted as uint8
train_labels = train_test_tuple{1}{2};
test_labels = train_test_tuple{2}{2};
```

[82] Basic classification – Classify images of clothing: https://www.tensorflow.org/tutorials/keras/classification
[83] Fashion MNIST: https://github.com/zalandoresearch/fashion-mnist
[84] Python tuples in MATLAB: www.mathworks.com/help/matlab/matlab_external/pythontuplevariables.html

Define the list of classes directly in MATLAB:

```
 class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress",
"Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

```
class_names = 1×10 string
"T-shirt/top"  "Trouser"    "Pullover"   "Dress"      "Coat"
"Sandal"      …
```

If we want to use the index of the training labels from the list above in MATLAB, we need to shift the range from [0:9] to [1:10]

```
 tl = uint8(train_labels)+1; % shifting range from [0:9] to [1:10]
 l = length(tl)
```

```
l = 60000
```

The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels:

```
 train_images_m = uint8(train_images);
 size(train_images_m)
```

```
ans = 1×3
       60000          28          28
```

To **resize a single image** from the dataset, use the reshape function:

```
 size(train_images_m(1,:,:))
```

```
ans = 1×3
     1    28    28
```

```
 size(reshape(train_images_m(1,:,:),[28,28]))
```

```
ans = 1×2
    28    28
```

You can add a live control to your live script to **explore your dataset**:

```
 i = 42 ;
 img = reshape(train_images_m(i,:,:),[28,28]);
 imshow(img)
 title(class_names(tl(i)))
```

You must **preprocess the data** before training the network.

If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

```
train_images = train_images / 255;
test_images = test_images / 255;
```

Finally, **build the model** with the function specified in the tf_helper file / module:

```
model = py.tf_helper.build_model();
```

You can look at the architecture of the model by retrieving the layers in a cell array:

```
cell(model.layers)
```

ans = 1×3 cell

	1	2	3
1	1×1 Flatten	1×1 Dense	1×1 Dense

```
py.tf_helper.compile_model(model);
py.tf_helper.train_model(model,train_images,train_labels)
```

```
Epoch 1/10

   1/1875 [..............................] - ETA: 12:59 - loss:
159.1949 - accuracy: 0.2500
  39/1875 [..............................] - ETA: 2s - loss: 52.8977
- accuracy: 0.5256
  76/1875 [>.............................] - ETA: 2s - loss: 34.8739
- accuracy: 0.6049
```

```
 113/1875 [>.............................] - ETA: 2s - loss: 28.4213
- accuracy: 0.6350
 157/1875 [=>............................] - ETA: 2s - loss: 22.9735
- accuracy: 0.6616
 194/1875 [==>...........................] - ETA: 2s - loss: 20.3405
- accuracy: 0.6740
 229/1875 [==>...........................] - ETA: 2s - loss: 18.3792
- accuracy: 0.6861
 265/1875 [===>..........................] - ETA: 2s - loss: 16.7848
- accuracy: 0.6943

...
```

Evaluate the model by comparing how the model performs on the test dataset:

```
 test_tuple =
py.tf_helper.evaluate_model(model,test_images,test_labels)
```

```
313/313 - 0s - loss: 0.5592 - accuracy: 0.8086 - 412ms/epoch -
1ms/step
test_tuple =
  Python tuple with values:

    (0.5592399835586548, 0.8086000084877014)

    Use string, double or cell function to convert to a MATLAB array.
```

```
 test_acc = test_tuple{2}
```

```
test_acc = 0.8086
```

Test the model on the first image from the test dataset:

```
 test_images_m = uint8(test_images);
 prob =
py.tf_helper.test_model(model,py.numpy.array(test_images_m(1,:,:)))
```

```
prob =
  Python ndarray:

    0.0000    0.0000    0.0000    0.0000    0.0000    0.0002    0.0000
0.0033    0.0000    0.9965
```

```
    Use details function to view the properties of the Python object.

    Use single function to convert to a MATLAB array.
```

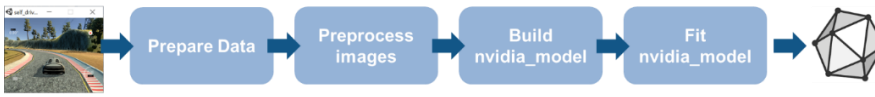
```
  [argvalue, argmax] = max(double(prob))
```

```
argvalue = 0.9965
argmax = 10
```

```
  imshow(reshape(test_images_m(1,:,:),[28,28])*255)
  title(class_names(argmax))
```

5.3. IMPORT TENSORFLOW MODEL INTO MATLAB

To illustrate the TensorFlow & ONNX import/export capabilities[85], we will take a workflow around an autonomous driving use case.

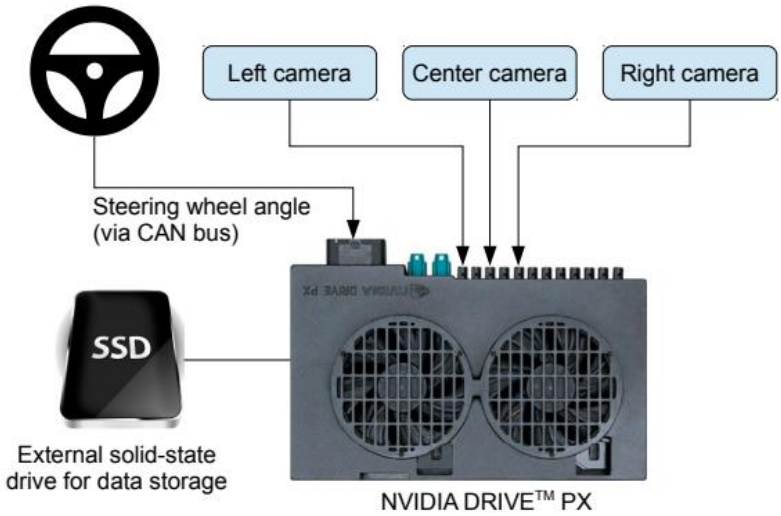


The data is generated by a simple open-source driving simulator[86] from Udacity. And the model comes from a real-life experiment from NVIDIA about End-to-end learning for self-driving cars[87].

[85] blogs.mathworks.com/deep-learning/2022/03/18/importing-models-from-tensorflow-pytorch-and-onnx/

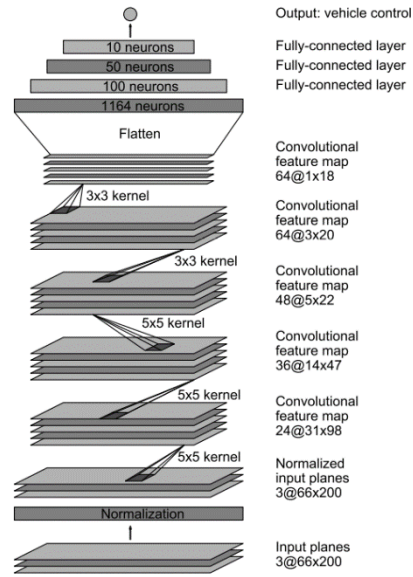
[86] Open-source driving simulator: https://github.com/udacity/self-driving-car-sim

[87] End-to-end learning for self-driving cars: https://arxiv.org/pdf/1604.07316.pdf



The inputs of the neural network are images from the camera and the output to predict in the steering angle (between -1 and 1). We will simplify the problem with only 5 classes(from left to right).

First, import csv file created by the driving simulator with images locations and steering ground truth:




```
 filename = "driving_log.csv";
 drivinglog = import_driving_log(
filename );
 drivinglog = drivinglog(2:end,:)
```

`drivinglog` = 7935×8 table

	VarName1	center	left	…
1	0	"center_2021_04_25_11_32_45_622.jpg"	"left_2021_04_25_11_32_45_622.jpg"	
2	1	"center_2021_04_25_11_32_45_689.jpg"	"left_2021_04_25_11_32_45_689.jpg"	

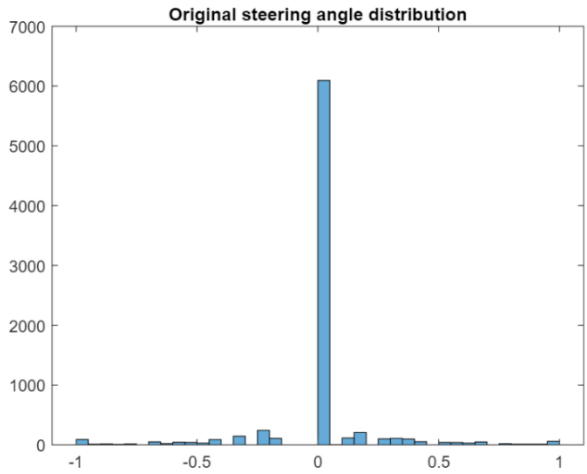
	VarName1	center	left	...
3	2	"center_2021_04_25_11_32_45_765.jpg"	"left_2021_04_25_11_32_45_765.jpg"	
4	3	"center_2021_04_25_11_32_45_834.jpg"	"left_2021_04_25_11_32_45_834.jpg"	
5	4	"center_2021_04_25_11_32_45_905.jpg"	"left_2021_04_25_11_32_45_905.jpg"	

⋮

Prepare the data

Analyze the range of values for the steering angle to find the optimal class values.

```
histogram(drivinglog.steering);
title("Original steering angle distribution");
```



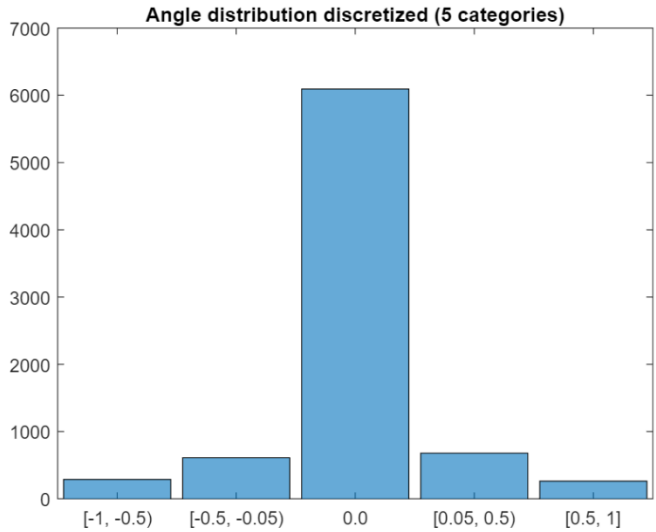
Use discretize to group the steering angles into discrete bins.

```
steeringLimits = [-1 -0.5 -0.05 0 0.05 0.5 1];
steeringClasses=discretize(drivinglog.steering,steeringLimits,'categorical');
classNames = categories(steeringClasses);
```

Merge the two bins that represent the angle close to 0 degrees.

```
steeringClasses=mergecats(steeringClasses,["[-0.05, 0)","[0, 0.05)"], "0.0");
```

```
histogramClasses = histogram(steeringClasses);
title("Angle distribution discretized (5 categories)");
```



Create image datastore and balance data (undersampling)

The previous histogram shows that the dataset is highly unbalanced. Use countEachLabel to check how many instances there are of each class.

```
imds=imageDatastore("sim_data/"+drivinglog.center,"Labels",
steeringClasses);
countEachLabel(imds)
```

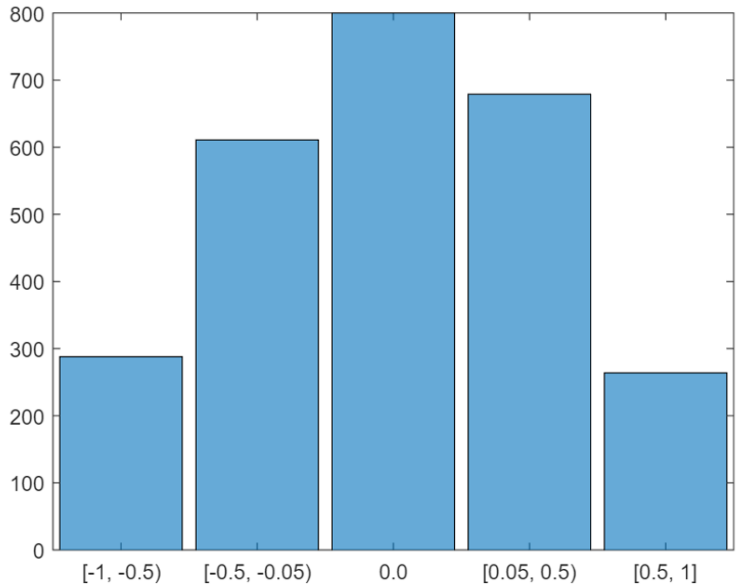
```
ans = 5×2 table
```

	Label	Count
1	[-1, -0.5)	288
2	[-0.5, -0.05)	611
3	0	6093
4	[0.05, 0.5)	679
5	[0.5, 1]	264

Define how many samples of the unbalanced class should be kept and randomly select these samples.

```
maxSamples = 800;
```

```
countLabel = countEachLabel(imds);
[~, unbalancedLabelIdx] = max(countLabel.Count);
unbalanced = imds.Labels == countLabel.Label(unbalancedLabelIdx);
idx = find(unbalanced);
randomIdx = randperm(numel(idx));
downsampled = idx(randomIdx(1:maxSamples));
retained = [find(~unbalanced) ; downsampled];
imds = subset(imds, retained');
histogram(imds.Labels)
```



Separate the dataset into training, validation and testing

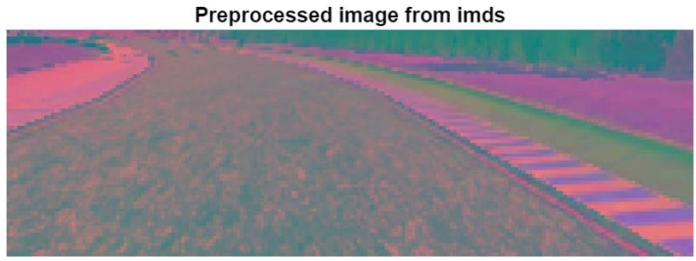
Extract 90% of the data for training and the remaining for testing and validation.

```
[imdsTrain, imdsValid,imdsTest] = splitEachLabel(imds, 0.9, 0.05,
0.05);
```

Preprocess the images by resizing it and converting it to the YCbCr color space.

```
trainData = transform(imdsTrain, @imagePreprocess, "IncludeInfo",
true);
testData = transform(imdsTest, @imagePreprocess, "IncludeInfo",
true);
```

```
 valData = transform(imdsValid, @imagePreprocess, "IncludeInfo",
true);
 imds_origI = imdsTrain.read;
 imds_newI = trainData.read{1};
 subplot(211), imshow(imds_origI), title("Original image from imds")
 subplot(212), imshow(imds_newI), title("Preprocessed image from
imds")
```



Modify the model:

Load the network from keras model and display with Deep Network Designer[88]

It is recommended to save and import the model in the SavedModel format instead of the HDF5 format[89] (you might get a warning).

```
 layers = importKerasLayers("tf_model.h5")
```

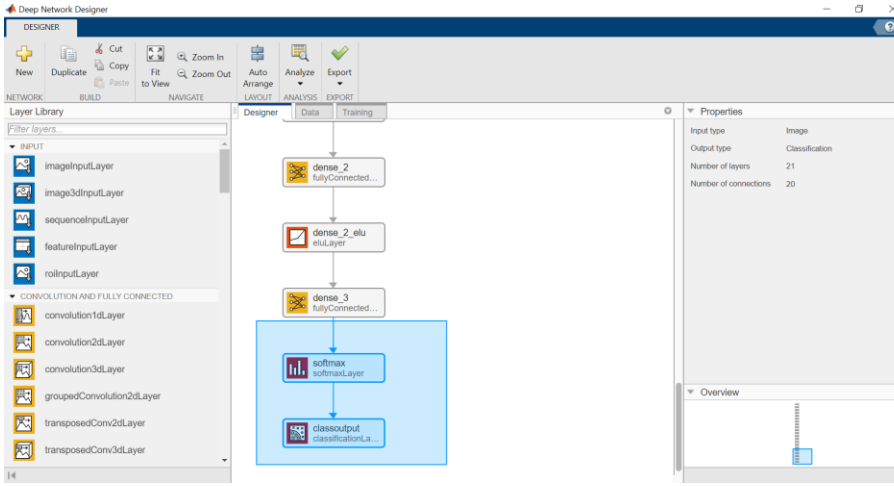
```
Warning: File 'tf_model.h5' was saved in Keras version '2.4.0'.
Import of Keras versions newer than '2.2.4' is not supported. The
```

[88] https://www.mathworks.com/help/deeplearning/gs/get-started-with-deep-network-designer.html
[89] www.tensorflow.org/tutorials/keras/save_and_load#save_the_entire_model

```
imported model may not exactly match the model saved in the Keras
file.
layers =
  20×1 Layer array with layers:

     1   'conv2d_5_input'            Image Input         66×200×3
images
     2   'conv2d_5'                  Convolution         24 5×5
convolutions with stride [2  2] and padding [0  0  0  0]
     3   'conv2d_5_elu'              ELU                 ELU with
Alpha 1
     4   'conv2d_6'                  Convolution         36 5×5
convolutions with stride [2  2] and padding [0  0  0  0]
     5   'conv2d_6_elu'              ELU                 ELU with
Alpha 1
     6   'conv2d_7'                  Convolution         48 5×5
convolutions with stride [2  2] and padding [0  0  0  0]
     7   'conv2d_7_elu'              ELU                 ELU with
Alpha 1
     8   'conv2d_8'                  Convolution         64 3×3
convolutions with stride [1  1] and padding [0  0  0  0]
     9   'conv2d_8_elu'              ELU                 ELU with
Alpha 1
    10   'conv2d_9'                  Convolution         64 3×3
convolutions with stride [1  1] and padding [0  0  0  0]
    11   'conv2d_9_elu'              ELU                 ELU with
Alpha 1
    12   'flatten'                   Keras Flatten       Flatten
activations into 1-D assuming C-style (row-major) order
    13   'dense'                     Fully Connected     100 fully
connected layer
    14   'dense_elu'                 ELU                 ELU with
Alpha 1
    15   'dense_1'                   Fully Connected     50 fully
connected layer
    16   'dense_1_elu'               ELU                 ELU with
Alpha 1
    17   'dense_2'                   Fully Connected     10 fully
connected layer
    18   'dense_2_elu'               ELU                 ELU with
Alpha 1
    19   'dense_3'                   Fully Connected     1 fully
connected layer
    20   'RegressionLayer_dense_3'   Regression Output   mean-
squared-error
```

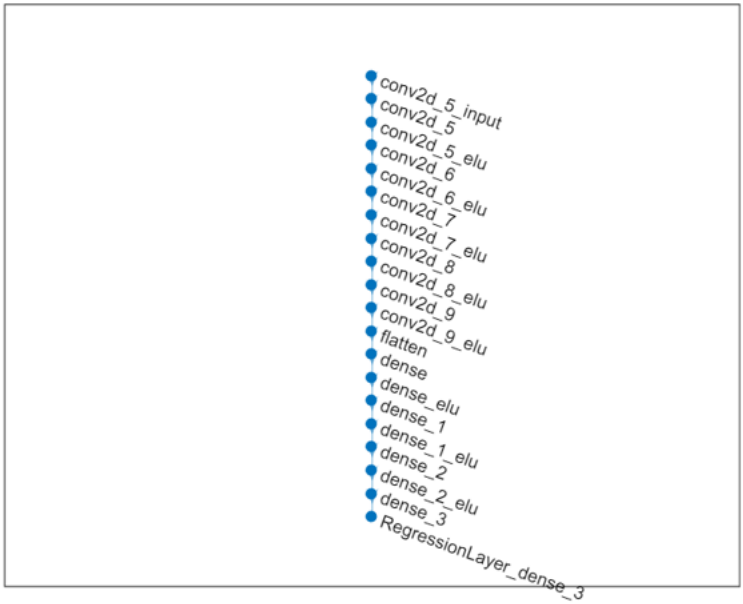
```
deepNetworkDesigner(layers)
```

Remove the last layer used for regression and add the layers for a classification with 5 classes (then export net as **layers_1**)



(programmatic alternative) Remove layers used for regression and add the layers for a classification with 5 classes

```
netGraph = layerGraph(layers);
clf; plot(netGraph)
```



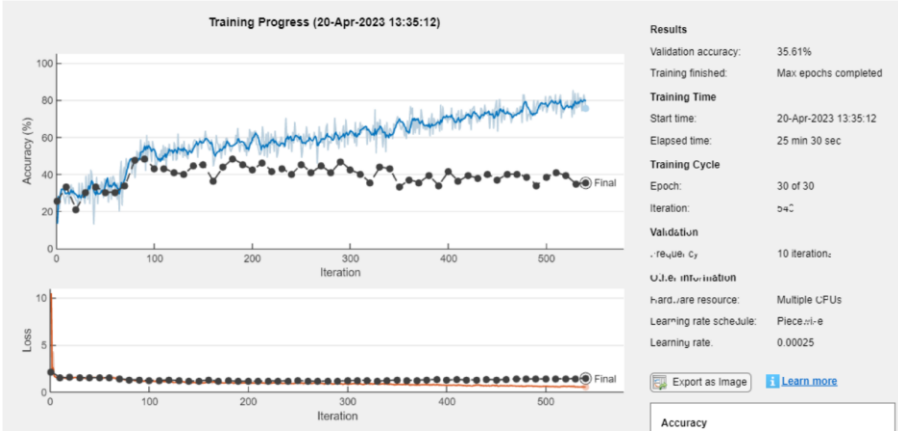

```
 classificationLayers = [fullyConnectedLayer(5,"Name","dense_3"),...
                         softmaxLayer("Name","softmax"),...
                         classificationLayer("Name","classoutput")];
 netGraph = removeLayers(netGraph, {'dense_3',
'RegressionLayer_dense_3'});
 netGraph = addLayers(netGraph,classificationLayers);
 layers_1 = netGraph.Layers
```

```
layers_1 =
  21×1 Layer array with layers:

     1   'conv2d_5_input'   Image Input             66×200×3 images
     2   'conv2d_5'         Convolution             24 5×5
convolutions with stride [2  2] and padding [0  0  0  0]
     3   'conv2d_5_elu'     ELU                     ELU with Alpha 1
     4   'conv2d_6'         Convolution             36 5×5
convolutions with stride [2  2] and padding [0  0  0  0]
     5   'conv2d_6_elu'     ELU                     ELU with Alpha 1
     6   'conv2d_7'         Convolution             48 5×5
convolutions with stride [2  2] and padding [0  0  0  0]
     7   'conv2d_7_elu'     ELU                     ELU with Alpha 1
     8   'conv2d_8'         Convolution             64 3×3
convolutions with stride [1  1] and padding [0  0  0  0]
     9   'conv2d_8_elu'     ELU                     ELU with Alpha 1
```

```
    10   'conv2d_9'         Convolution             64 3×3
convolutions with stride [1  1] and padding [0  0  0  0]
    11   'conv2d_9_elu'     ELU                     ELU with Alpha 1
    12   'flatten'          Keras Flatten           Flatten
activations into 1-D assuming C-style (row-major) order
    13   'dense'            Fully Connected         100 fully
connected layer
    14   'dense_elu'        ELU                     ELU with Alpha 1
    15   'dense_1'          Fully Connected         50 fully
connected layer
    16   'dense_1_elu'      ELU                     ELU with Alpha 1
    17   'dense_2'          Fully Connected         10 fully
connected layer
    18   'dense_2_elu'      ELU                     ELU with Alpha 1
    19   'dense_3'          Fully Connected         5 fully connected
layer
    20   'softmax'          Softmax                 softmax
    21   'classoutput'      Classification Output   crossentropyex
```

Train the model: (I am using my CPU here, but I recommend to speed it up on a GPU)

```
initialLearnRate = 0.001;
maxEpochs = 30;
miniBatchSize = 100;

options = trainingOptions("adam", ...
    "MaxEpochs",maxEpochs, ...
    "InitialLearnRate",initialLearnRate, ...
    "Plots","training-progress",  ...
    "ValidationData",valData, ...
    "ValidationFrequency",10, ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropPeriod",10, ...
    "LearnRateDropFactor",0.5, ...
    "ExecutionEnvironment","parallel",...
    "Shuffle","every-epoch");

net = trainNetwork(trainData, layers_1, options);
```



Save the model: Save the new trained network in a MAT format.

```
model_name = "net-class-30-1e-4-drop10-0_5";
% classification-epochs-learning_rate-drop_period-drop_factor
save("model_name+".mat","net")
```

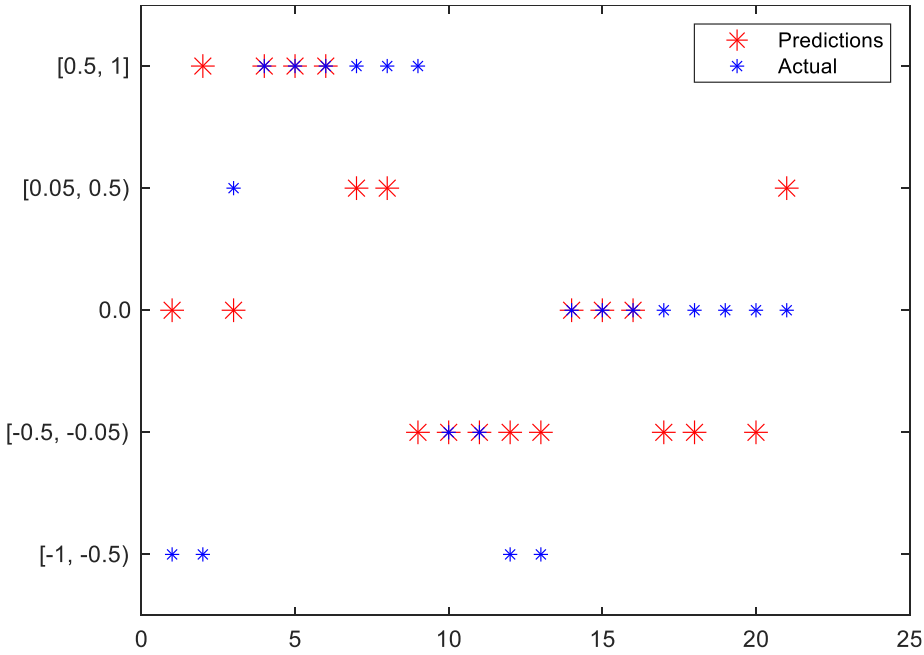
Export it to a ONNX Network format.

```
exportONNXNetwork(net,model_name+".onnx")
```

Test the model:

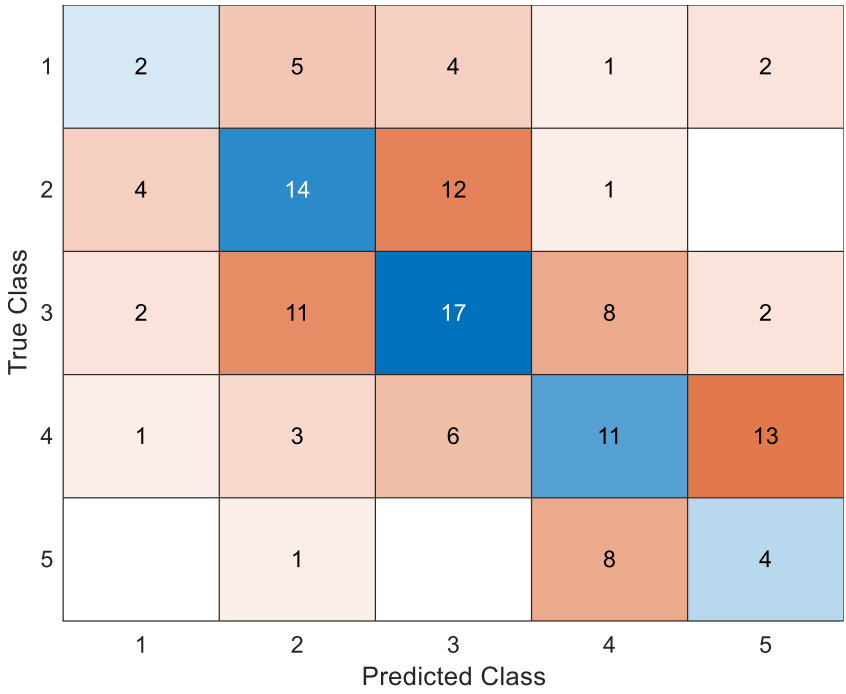
Plot predicted and ground truth values for steering angle using the testing dataset.

```
model_name = "net-class-30-1e-4-drop10-0_7"; % classification-
epochs-learning_rate-drop_period-drop_factor
load("models/"+model_name+".mat","net")
predSteering = classify(net, testData);

figure
startTest = 1;
endTest = 20;
plot(predSteering(startTest:endTest), 'r*', "MarkerSize",10)
hold on
plot(imdsTest.Labels(startTest:endTest), 'b*')
legend("Predictions", "Actual")
hold off
```



Display the confusion matrix.

```
confMat = confusionmat(imdsTest.Labels, predSteering);
confusionchart(confMat)
```



Display the testing image and the predicted label along with the ground truth.

```
 numberImages = length(imdsTest.Labels);
 i = 42;
 img = readimage(imdsTest, i);
 imshow(img),title(char(imdsTest.Labels(i)) + "/"
+char(predSteering(i)));
```

To test your model directly with the driving simulator, you can switch the simulator to “autonomous mode” and run the script drive.py.

6. CALL MATLAB FROM PYTHON

If you are a Python user and you are wondering *why you should consider MATLAB*, this chapter is probably a better entry point into this book. One of my favorite colleagues, Lucas Garcia – Deep Learning Product Manager – tried to answer this question at a Python Meetup[90] in Madrid:

- **Facilitate** AI development by using a simplified MATLAB workflow
- Need **functionality** available in MATLAB (e.g. Simulink)
- Leverage the work from the MATLAB **community**

But first let's start with a few basics on how to call MATLAB from Python.

6.1. GETTING STARTED WITH THE MATLAB ENGINE API FOR PYTHON

First, make sure you have the MATLAB Engine for Python installed (as described in section 3.8).

In your python script or jupyter notebook, the first statement you will need to enter in order to load the MATLAB Engine package is:

```
>>> import matlab.engine
```

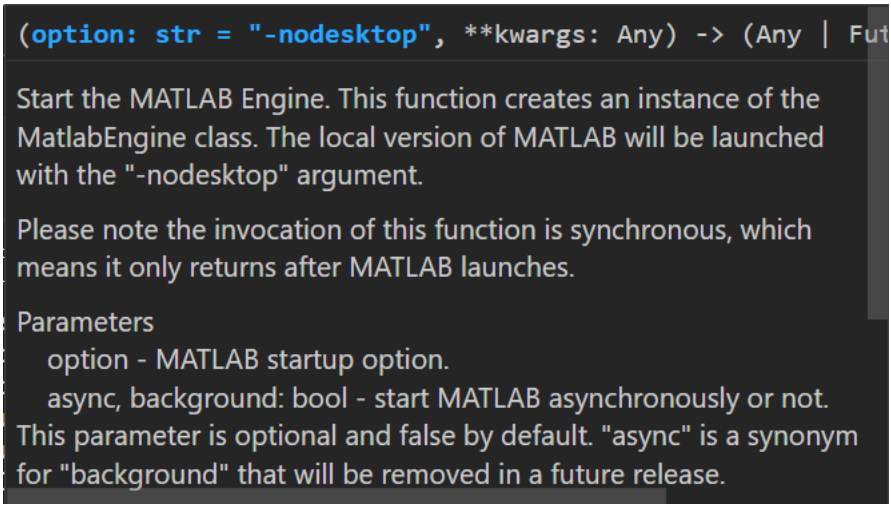
You have then two options to interact with MATLAB from Python:

1. Start a new session (in batch or interactive)

By default, the engine will be started in batch with the "-nodesktop" mode:

```
>>> m = matlab.engine.start_matlab()
```

[90] Lucas Garcia – Python Madrid Meetup:
https://mathinking.github.io/blog/en/a-trick-you-dont-know-about-python-matlab/



(this is the contextual help provided by VSCode when you enter the function)

If you wish to have the MATLAB Desktop apparent to visualize which values are stored in the workspace or to debug interactively with the console, you can specify the “desktop” argument:

```
>>> m = matlab.engine.start_matlab("-desktop")
```

2. Connect to an existing session

First you need to start MATLAB manually. For convenience, it’s easier to have the MATLAB Desktop and your Python development environment (Jupyter or VSCode) open side by side. To share the MATLAB Engine session, simply type inside of MATLAB:

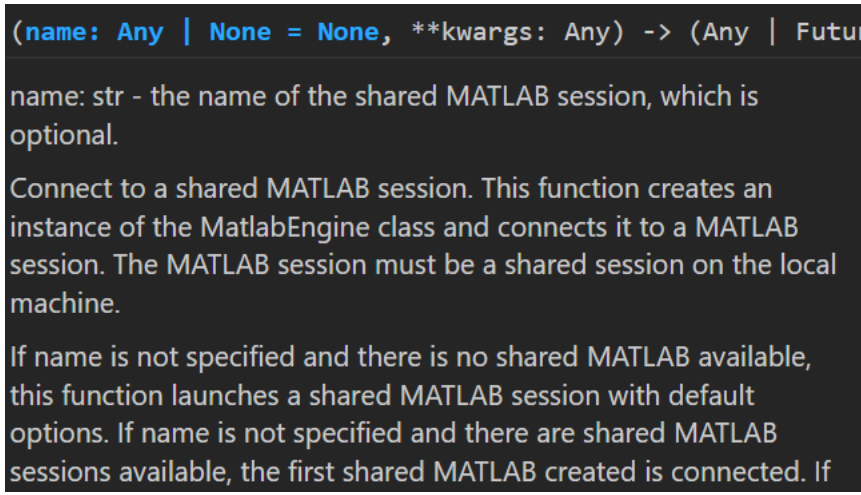
```
>> matlab.engine.shareEngine
```

You can also request the name of the MATLAB Engine session, in case Python doesn’t find it automatically:

```
>> matlab.engine.engineName
ans =
    'MATLAB_11388'
```

Then on the Python side, enter the following command:

```
>>> m = matlab.engine.connect_matlab()
```



(this is the contextual help provided by VSCode when you enter the function)

If Python does not find automatically the running session, you can enter the engine name requested previously in MATLAB (`'MATLAB_11388'`).

6.2. Facilitate AI development by using MATLAB Apps

6.2.1. Data Cleaner App

The first step in an AI pipeline is often to clean the data. This process requires some level of interactivity for the data analyst to understand which variables she is manipulating. Once the input format of the data is fixed, this process can be automated to scale it to the whole dataset.

Let's take an example with the weather data from chapter 2. In this first example, we will start MATLAB in `-nodesktop` mode (which is the default mode for the engine). In the next two sections, we will use the `-desktop` mode to show how to use the MATLAB desktop to interact with the data, but also connect to an already running MATLAB session.

Set up the environment

```
!git clone https://github.com/hgorr/weather-matlab-python
```

```
import matlab.engine
m = matlab.engine.start_matlab()
m.cd('weather-matlab-python') # returns the previous dir location
```

```
 # m.cd('..')
```

```
'C:\\Users\\ydebray\\Downloads\\python-book-github'
```

```
 m.pwd()
```

```
'C:\\Users\\ydebray\\Downloads\\python-book-github\\weather-matlab-
python'
```

```
 # Make sure that your Python interpreter follows along
import os
os.getcwd()
 os.chdir('weather-matlab-python')
 # os.chdir('..')
```

Retrieve Weather Data

```
 import weather
appid ='b1b15e88fa797225412429c1c50c122a1'
json_data =
weather.get_forecast('Muenchen','DE',appid,api='samples')
data = weather.parse_forecast_json(json_data)
data.keys()
```

```
dict_keys(['current_time', 'temp', 'deg', 'speed', 'humidity',
'pressure'])
```

```
 print(len(data['temp']))
 data['temp'][0:5]
```

```
36
[286.67, 285.66, 277.05, 272.78, 273.341]
```

```
 t = matlab.double(data['temp'])
 t
```

```
matlab.double([[286.67,285.66,277.05,272.78,273.341,275.568,276.478,2
76.67,278.253,276.455,275.639,275.459,275.035,274.965,274.562,275.648
,277.927,278.367,273.797,271.239,269.553,268.198,267.295,272.956,277.
422,277.984,272.459,269.473,268.793,268.106,267.655,273.75,279.302,27
9.343,274.443,272.424]])
```

Format into a Timetable

```
 # Transform into a timetable for data cleaning
 m.workspace['data'] = data
```

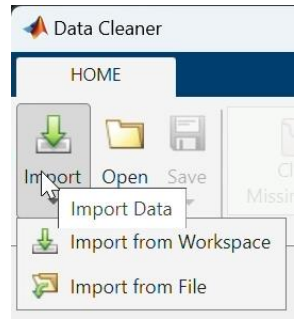
```
 m.eval("TT =
timetable(datetime(string(data.current_time))',cell2mat(data.temp)',
'VariableNames',{'Temp'})",nargout=0)
 m.who()
```

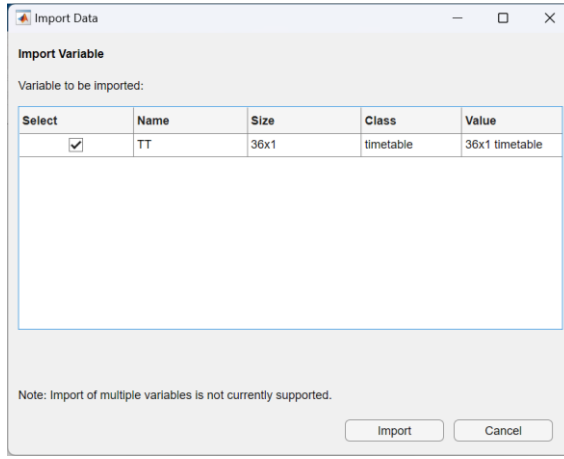
```
['TT', 'data']
```

Interact manually with the app

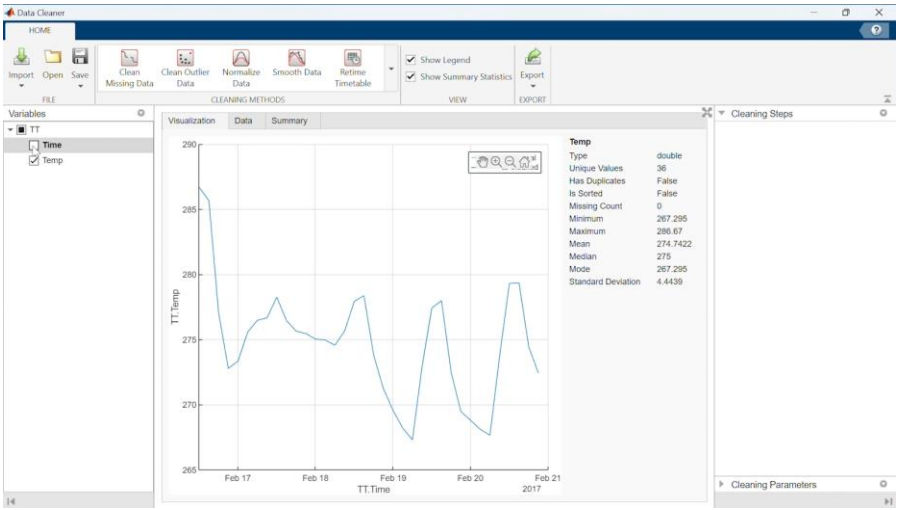
```
 m.dataCleaner(nargout=0)
```

The app will appear, with a blank canvas, giving you the option to import data. Select the timetable from your workspace.





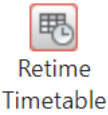
This will open your data into the app main view:

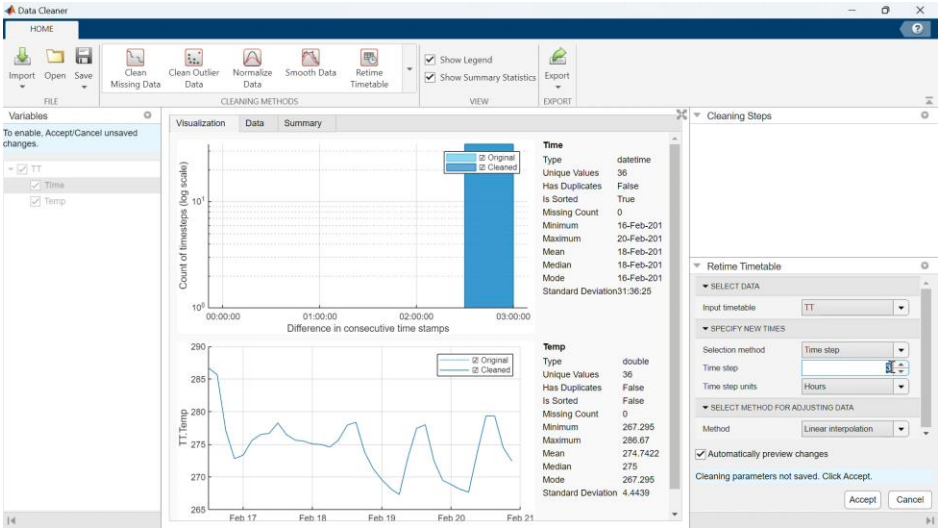


In the left panel, you can select the variables that you want to visualize and manipulate.

We will select the Time variable, and use the **Retime Timetable** cleaning method:

This will display options in the right panel, where we will specify the new sampling, Time step: 1 (hour). Once you are happy with the results, click accept (bottom right).



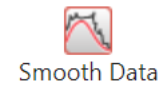


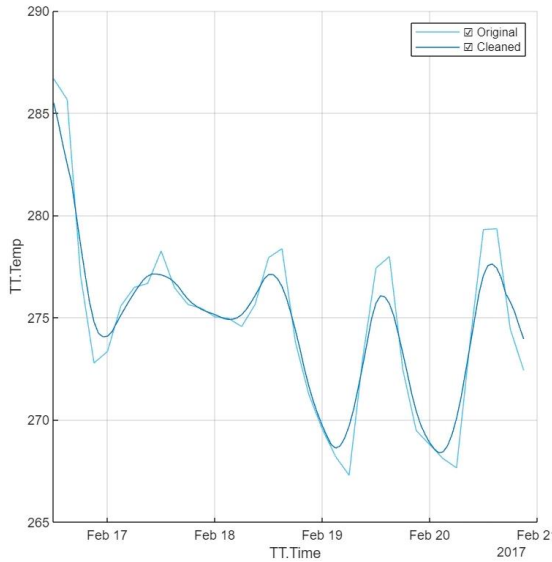
You can see the results of this transformation on your data by changing the tabs in the center panel:

Visualization | Data | Summary

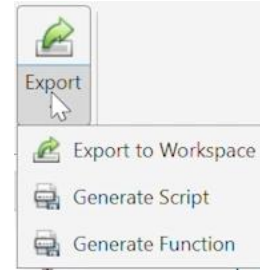
	Time	1 Temp
	Min 16-Feb-2017 12:00:00 Max 20-Feb-2017 21:00:00 Mean 18-Feb-2017 16:30:00 Std Dev 30:44:36 Missing 0 Class datetime	Min 267.295 Max 286.67 Mean 274.6515 Std Dev 4.113 Missing 0 Class double
1	16-Feb-2017 12:00:00	286.6700
2	16-Feb-2017 13:00:00	286.3333
3	16-Feb-2017 14:00:00	285.9967
4	16-Feb-2017 15:00:00	285.6600
5	16-Feb-2017 16:00:00	282.7900
6	16-Feb-2017 17:00:00	279.9200
7	16-Feb-2017 18:00:00	277.0500
8	16-Feb-2017 19:00:00	275.6267
9	16-Feb-2017 20:00:00	274.2033
10	16-Feb-2017 21:00:00	272.7800

The second transformation we will operate on this is **Smooth Data**. You can select the smoothing method (we will stick with the default moving average) and play around with the smoothing factor.





Export the cleaning steps



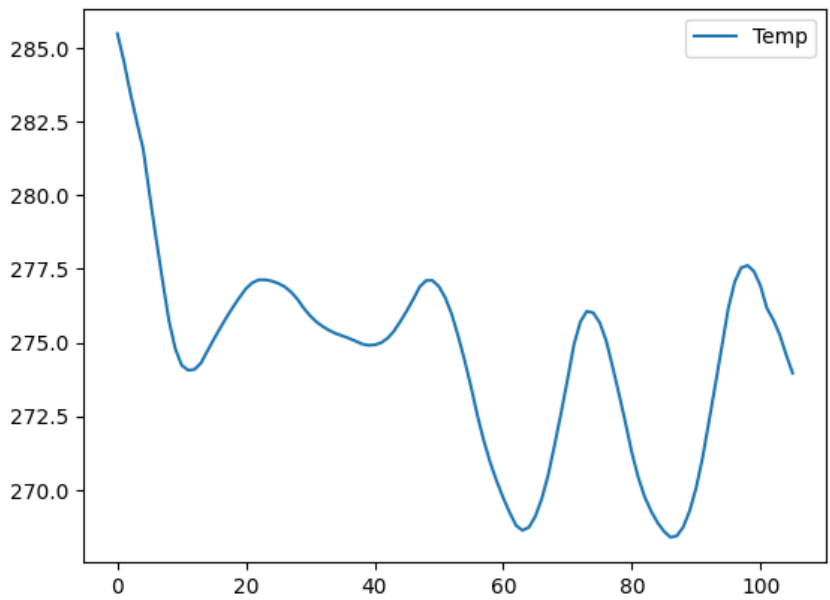
Once you are happy with the way your data looks, you can save your manual operations as a function that you will apply to any new weather data that comes in, to automate the preprocessing.

preprocess.m

```
function TT = preprocess(TT)
        % Retime timetable
        TT = retime(TT,"regular","linear","TimeStep",hours(1));
        % Smooth input data
        TT = smoothdata(TT,"movmean","SmoothingFactor",0.25);
end
```

You can call this function from Python, and test that it works.

```
TT = m.workspace['TT']
TT2 = m.preprocess(TT)
m.parquetwrite("data.parquet",TT2,nargout=0)
import pandas as pd
pd.read_parquet('data.parquet').plot(y='Temp')
```



6.2.2. Regression and Classification Learner Apps

We will take the Boston housing example that is part of the Scikit-Learn sample datasets to call MATLAB from Python.

Open a Jupyter notebook and connect to a running MATLAB session from Python:

```
 import matlab.engine
m = matlab.engine.connect_matlab()
```

Retrieve the dataset:

```
 import sklearn.datasets
dataset = sklearn.datasets.load_boston()
dataset.keys()
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename',
'data_module'])
```

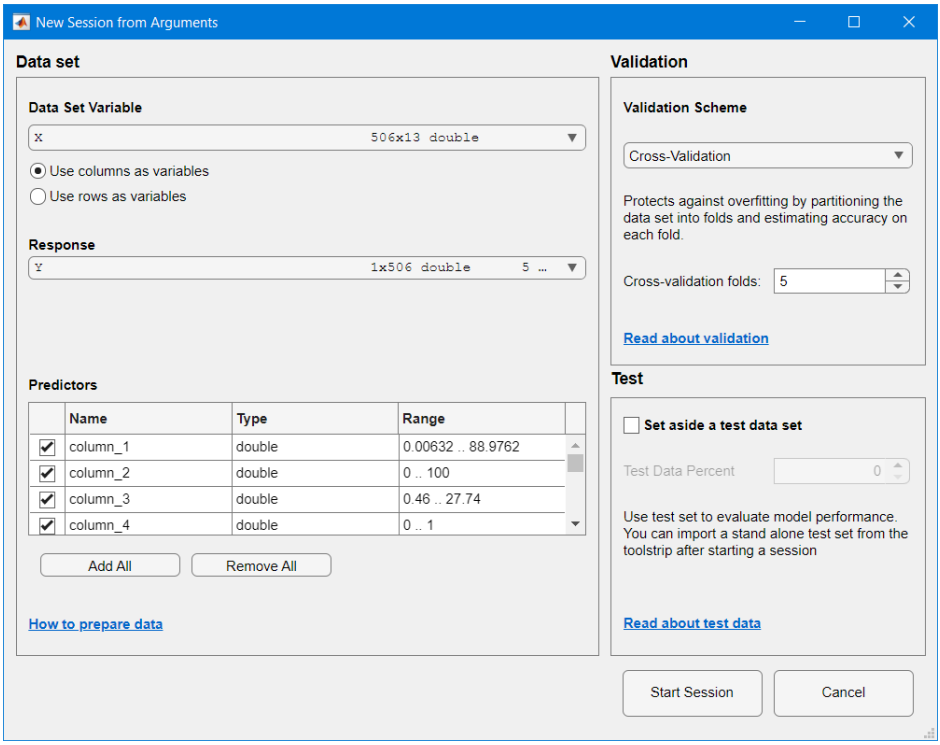
```
 data = dataset['data']
target = dataset['target']
```

Depending on the version of MATLAB you are you using, you might require converting the data and target arrays to MATLAB double:

- Before 22a, Numpy arrays were not accepted, so you need to translate them to lists.
- In 22a, Numpy arrays can be passed into MATLAB object constructor (double, int32, …).
- From 22b, Numpy arrays can be passed directly into MATLAB functions.

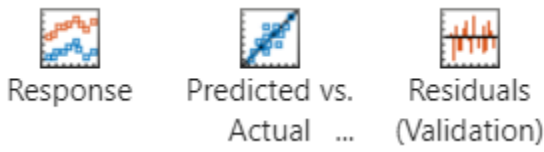
```
 # Before 22a
X_m = matlab.double(data.tolist())
Y_m = matlab.double(target.tolist())
# In 22a
X_m = matlab.double(data)
Y_m = matlab.double(target)
# From 22b
X_m = data
Y_m = target
 # Call the regression learner app with data coming from Python
 m.regressionLearner(X_m,Y_m,nargout=0)
```

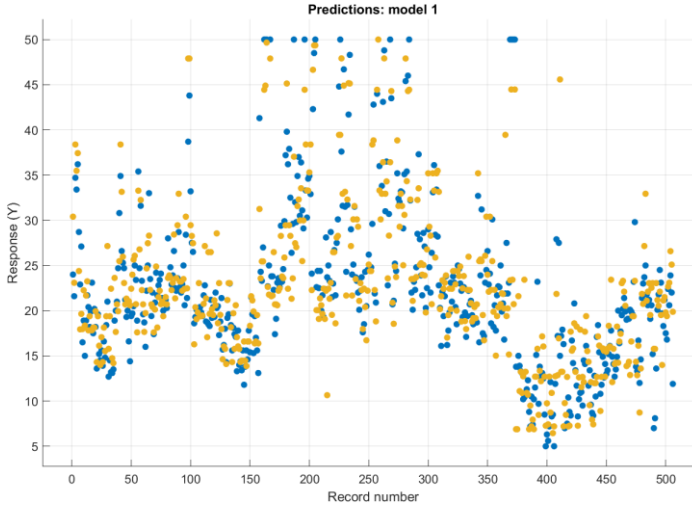
The session is automatically created in the Regression Learner, with the passed data:

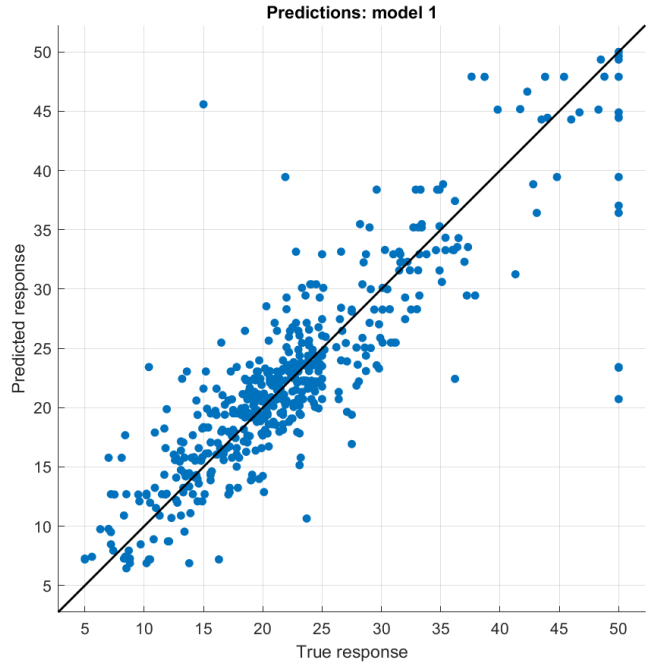


You have several models and categories to choose from:

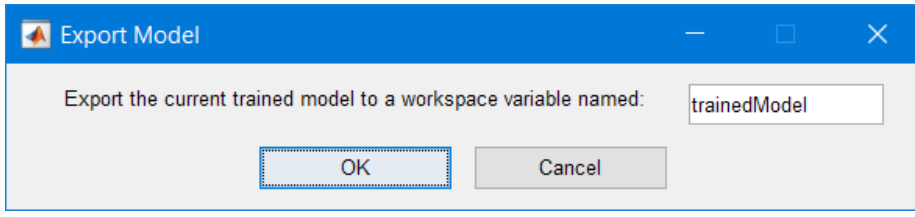
You can visualize certain indicators during the training:







Once you are happy with one of the models you've trained, you can generate a function or export it:



Informations are shared in the MATLAB Command Window:

```
Variables have been created in the base workspace.

Structure 'trainedModel' exported from Regression Learner.

To make predictions on a new predictor column matrix, X:

  yfit = trainedModel.predictFcn(X)

For more information, see How to predict using an exported model.
```

Finally, you can retrieve the model to assign the prediction function to a variable in Python:

```
  model = m.workspace['trainedModel']
  m.fieldnames(model)
```

```
['predictFcn', 'RegressionTree', 'About','HowToPredict']
```

```
  predFcn = model.get('predictFcn')
```

This way, you can test the model directly from within Python:

```
  X_test = data[0]
y_test = target[0]
X_test,y_test
```

```
(array([6.320e-03, 1.800e+01, 2.310e+00, 0.000e+00, 5.380e-01,
6.575e+00,
        6.520e+01, 4.090e+00, 1.000e+00, 2.960e+02, 1.530e+01,
3.969e+02,
        4.980e+00]),
 24.0)
```

```
  m.feval(predFcn,X_test)
```

```
23.46666666666667
```

You can iterate and test another model to see if the predictions are closer to the test target.

6.2.3. IMAGE LABELER APP

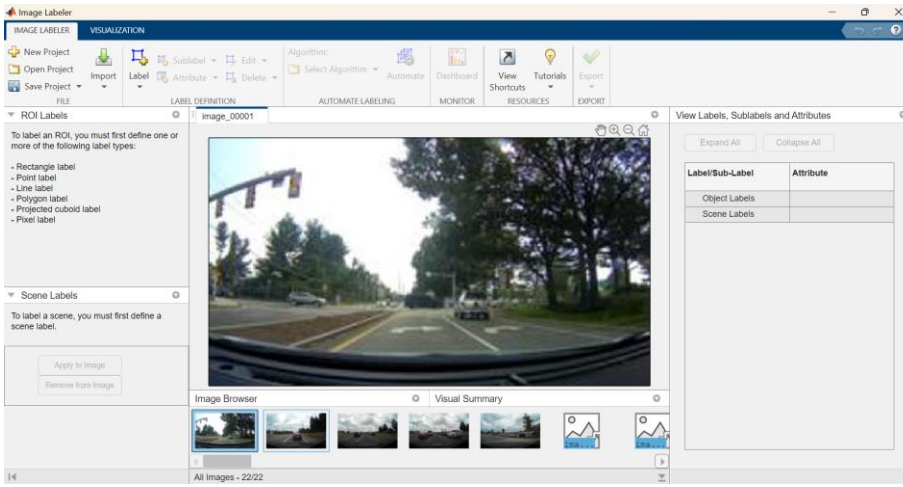
Data preparation is key in developing Machine Learning and Deep Learning applications. No matter how much effort you put into your ML model, it will likely perform poorly if you didn't spend the right time in preparing your data to be consumed by your model.

In this example, we start with a set of images to label for a Deep Learning application.

```
 import os
cwd = os.getcwd()
vehicleImagesPath = os.path.join(cwd, "vehicleImages")
vehicleImagesPath
```

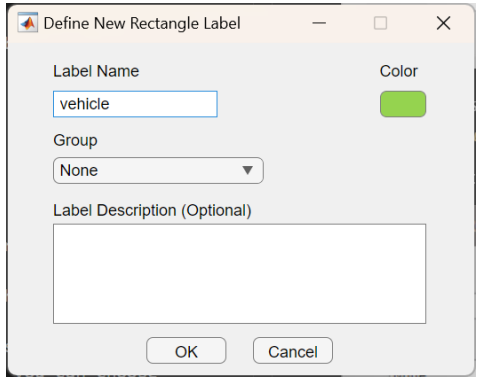
We then start the MATLAB Engine API for Python and open the *Image Labeler App*, passing the location of the images as an input:

```
 import matlab.engine
m = matlab.engine.start_matlab()
 m.imageLabeler(vehicleImagesPath, nargout=0)
```

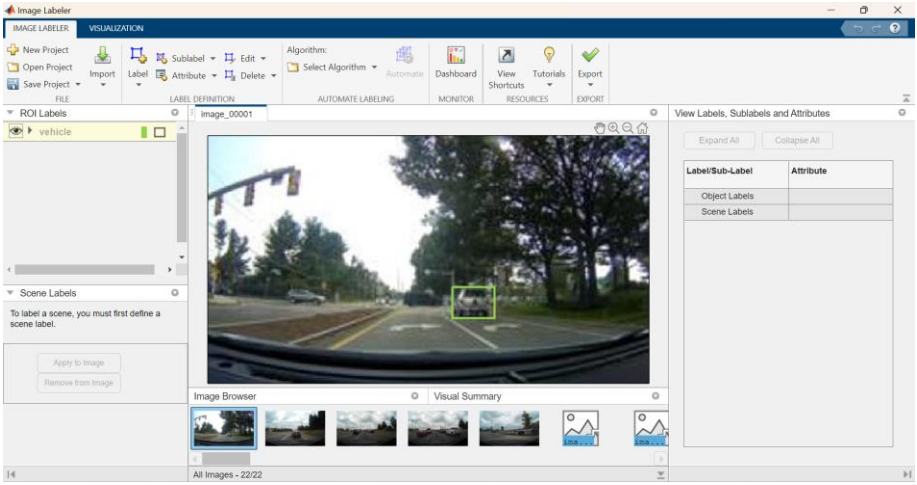


Note that because the App returns no output arguments back to Python, you need to specify `nargout=0` *(number of output arguments equals 0).*

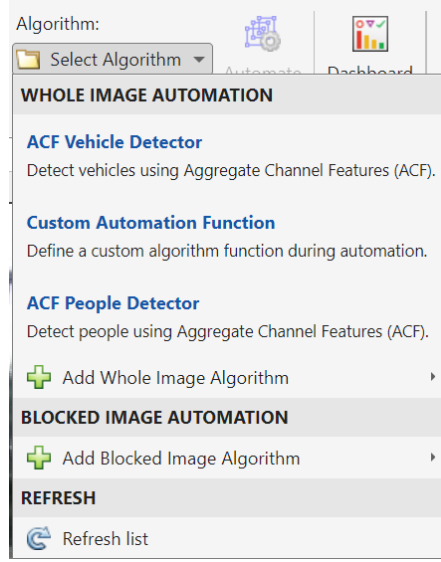
Now you can interactively create a new ROI Label:



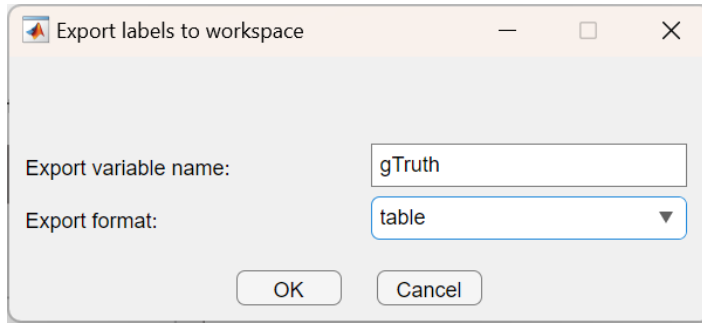
and start to manually label your vehicles:



This process is rather tedious, especially considering that the number of required labeled images for the problem might be significant for Deep Learning workflows. Thus, the following shows how to facilitate labeling by automating (or semi-automating) the labeling process. After selecting the images you would like to automatically label, you can choose among various algorithms (*ACF vehicle detector*, *ACF people detector*, or import your custom detector). In this particular case, after choosing *ACF vehicle detector*, the selected images are automatically labeled. Earlier I mentioned that the process is semi-automated, as it might not detect all vehicles, or you might want to correct some bounding boxes before exporting your results.



Finally, export your labeling process as a MATLAB table to continue your work back in Python:



Back in Python, gather the variables you are interested in:

```
 imageFilename = m.eval("gTruth.imageFilename")
labels = m.eval("gTruth.vehicle")
```

and put them into a convenient form to continue your work:

```
 import pandas as pd
import numpy as np
# Bring data to convenient form as DataFrame
labels = [np.array(x) for x in labels]
df = pd.DataFrame({"imageFileName":imageFilename, "vehicle":labels})
```

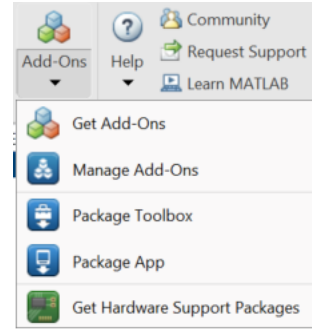
Labeled data is now conveniently shaped into a DataFrame with information regarding file location and bounding boxes for each vehicle, and can be easily accessed:

```
df.iloc[[13]]
```

```
                                          imageFileName
13  c:\Users\ydebray\Downloads\matlab-with-python-...  \

                                                vehicle
13  [[32.0, 118.0, 40.0, 25.0], [76.0, 110.0, 40.0...
```

```
m.exit()
```

6.3. LEVERAGE THE WORK FROM THE MATLAB COMMUNITY



MATLAB has a vibrant and established community. It is actually quite complementary to the Python community, that is younger and growing fast in particular in areas related to Machine/Deep Learning. MATLAB File Exchange[91] enables you to share your own files or browse and download files contributed by other users. It does not require you to use source control with Git, but can be connected with your code repositories on GitHub if you want to. This enables to access community contributions directly from MATLAB via the add-on manager.

In this example, we will use a MATLAB community toolbox to fit a sine function[92] over weather data.

Set-up the environment

```
!git clone https://github.com/hgorr/weather-matlab-python

# download the zip file and unzip it
url_zip = 'https://www.mathworks.com/matlabcentral/mlc-
downloads/downloads/a1ca242b-82c2-4a89-b280-38d2243276da/4d399976-
```

[91] https://www.mathworks.com/matlabcentral/fileexchange/
[92] https://www.mathworks.com/matlabcentral/fileexchange/66793-sine-fitting

```
e76f-418a-a227-c97d2f7a85f7/packages/zip'

import requests, zipfile, io, os
r = requests.get(url_zip)
z = zipfile.ZipFile(io.BytesIO(r.content))
os.makedirs('sineFit', exist_ok=True)
z.extractall('sineFit')
```

Retrieve weather data

```
 import os
os.chdir('weather-matlab-python')
```

We will use the sample dataset from the forecast in Munich.

```
 import weather
appid ='b1b15e88fa797225412429c1c50c122a1'
json_data =
weather.get_forecast('Muenchen','DE',appid,api='samples')
data = weather.parse_forecast_json(json_data)
```

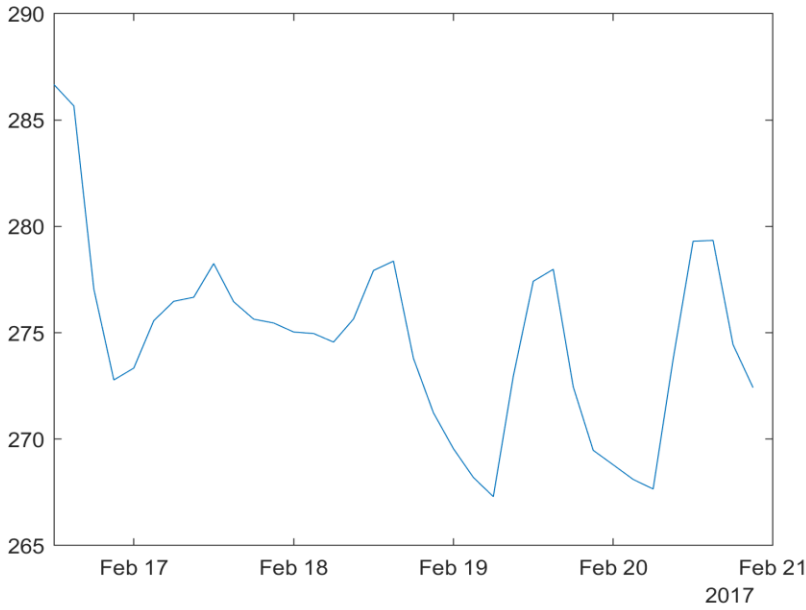
The MATLAB engine is going to start in the current directory of the Python interpreter.

```
 import matlab.engine
m = matlab.engine.connect_matlab()
 m.pwd()
```

```
'C:\\Users\\ydebray\\...\\code\\weather-matlab-python'
```

Let's convert the data into MATLAB datatypes (datetime and double precision floats).

```
 dt = m.datetime(data['current_time'])
temp = matlab.double(data['temp'])
m.plot(dt,temp)
m.print(m.tempdir()+"myPlot.png",'-dpng','-r300',nargout=0)
 from IPython.display import Image
Image(m.tempdir()+"myPlot.png")
```



You can create a function to simplify the process of saving MATLAB plots in your notebook.

```
 from IPython.display import Image
def mplot(x,y):
    m.plot(x,y)
    m.print(m.tempdir()+"myPlot.png",'-dpng','-r300',nargout=0)
    return Image(m.tempdir()+"myPlot.png")
# mplot(dt,temp)
```

Sine Fitting

Retrieve description from the webpage

```
 import requests
from bs4 import BeautifulSoup
from IPython.display import HTML
url='https://www.mathworks.com/matlabcentral/fileexchange/66793-
sine-fitting'
r = requests.get(url)
soup = BeautifulSoup(r.text, 'html.parser')
description = soup.find('div', id='description')#.get_text()
HTML(str(description))
```

sineFit is a function to detect the parameters of a noisy sine curve, even less than one period long. It requires only x and y values and no additional parameters as input. No toolbox required

Syntax:

[SineParams]=sineFit(x,y,optional)

optional: plot graphics if ommited. Do not plot if 0

Input:

x and y values, y = offs + amp * sin(2pi * f * x + phi) + noise

x and y may be row or column vectors

x may be not equidistant

Output:

SineParams(1):offset (offs)

SineParams(2): amplitude (amp)

SineParams(3): frequency (f)

SineParams(4): phaseshift (phi)

SineParams(5): MSE , if negative then SineParams are from FFT

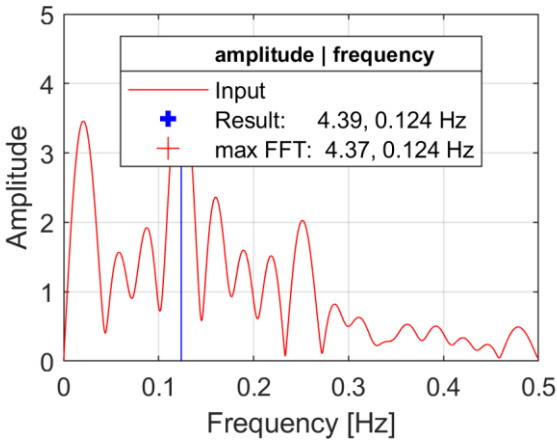
```
 m.cd('..')
m.cd('sineFit/')
```

```
'C:\\Users\\ydebray\\Downloads\\python-book-github\\code'
```

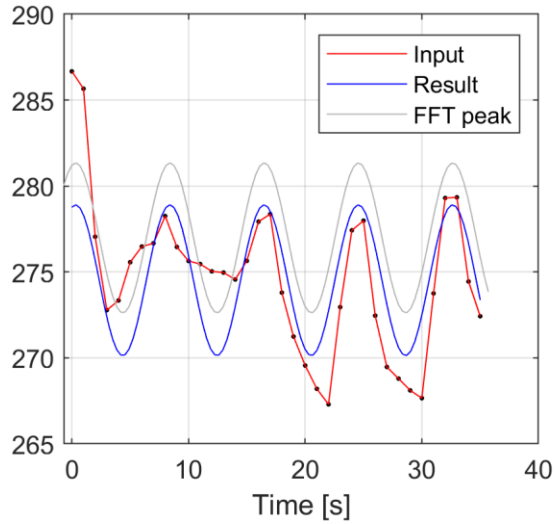
```
 t = matlab.double(range(36))
SineParams = m.sineFit(t,temp)
SineParams
```

```
matlab.double([[274.51842044220115,4.390270295890297,0.12378139169121
775,1.3187000578012196,9.61367024285807]])
```

```
 m.print(m.tempdir()+"fft.png",'-dpng','-r300',nargout=0)
m.close(m.gcf()) # close the current figure
Image(m.tempdir()+"fft.png")
```




```
 m.print(m.tempdir()+"sine.png",'-dpng','-r300',nargout=0)
m.close(m.gcf())
Image(m.tempdir()+"sine.png")
```



Sine Fit 2: retrieve the values of the sine model

Create a function in MATLAB that returns the values of the sine model for a given set of parameters and a given set of x values.

```
 m.edit('sineFit2.m',nargout=0)
```

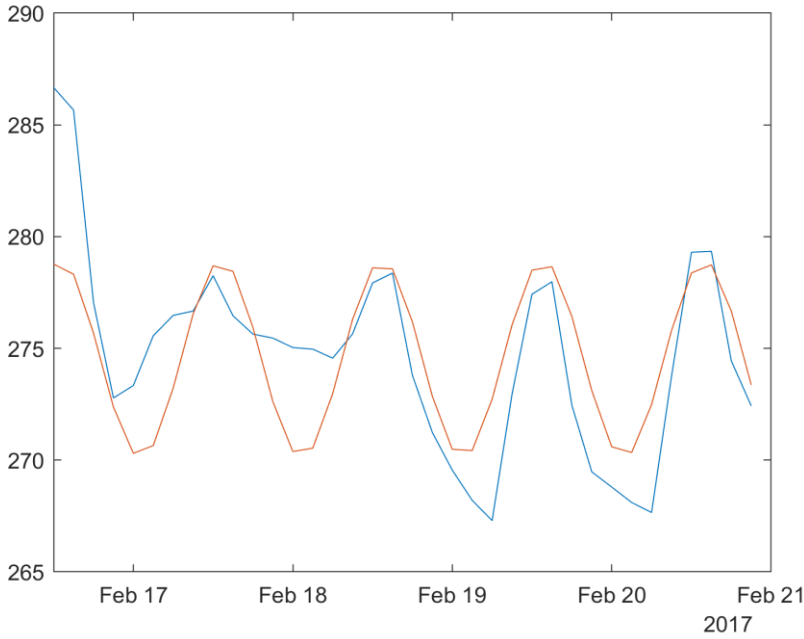
```
function Sine=sineFit2(y)
    s = size(y);
```

```
    if s(1)>s(2)
        y = y';
    end
    n=length(y);
    x = linspace(1,n,n);
    SineP = sineFit(x,y,0); % does not generate plots
    Sine = SineP(1)+SineP(2)*sin(2*pi*SineP(3)*x+SineP(4));
end
```

```
 mat_temp = matlab.double(temp)
SineVal = m.sineFit2(mat_temp)[0]
SineVal
```

```
matlab.double([278.76990973043,278.3159918142903,275.67845197258595,2
72.3738730375523,270.3023846684254,270.65509065371265,273.22918539644
996,276.5445669343253,278.6948941220223,278.44373092737726,275.935495
9419373,272.6124218319101,270.38527296408427,270.5346577616464,272.97
46801119398,276.3023295473256,278.60421089340406,278.55674462500303,2
76.187223800081,272.85812057806464,270.4836660592461,270.429169272222
6,272.72596581629216,276.05339992247,278.4982002304407,278.6546089515
687,276.432691224009,273.11004757040575,270.59719484552426,270.339020
91123986,272.48397552754307,275.7987118855729,278.37725981779397,278.
7369567822915,276.67097737588534,273.3672577389005])
```

```
 m.plot(dt,temp,dt,SineVal)
m.print(m.tempdir()+"sine2.png",'-dpng','-r300',nargout=0)
m.close(m.gcf())
Image(m.tempdir()+"sine2.png")
```




```
m.quit()
```

Sine Fit 2: retrieve only the parameters

This second implementation does a minimal work from the MATLAB side, of fitting the parameters *(a,b,c,d)* of the Sine model:

$$a + b * \sin(2 * \pi * c + d)$$

```
# need to grab the first element of the matlab double array, to
return a list
SineP = SineParams[0]
SineP
```

```
matlab.double([274.51842044220115,4.390270295890297,0.123781391691217
75,1.3187000578012196,9.61367024285807])
```

```
from math import sin,pi
# SineP(1)+SineP(2)*sin(2*pi*SineP(3)*tt+SineP(4)));
def Sine(t):
    return SineP[0]+SineP[1]*sin(2*pi*SineP[2]*t+SineP[3])
```

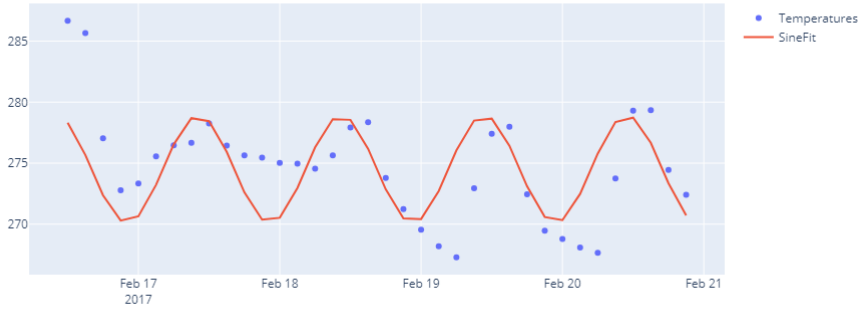
```
 # generate a list from 1 to 40
t = list(range(1,41))
Sine(t[0])
```

```
278.3160060160971
```

```
 # need to use the concept of list comprehension in Python to
generate a list of Sine values
SineFit = [ Sine(x) for x in t ]
SineFit
```

```
[278.3160060160971,
 275.67847973668944,
 272.3739196764843,
 ...
 275.539336033496,
 278.2419051283501]
```

```
 import plotly.graph_objects as go
fig = go.Figure()
fig.add_trace(go.Scatter(x=data['current_time'], y=data['temp'],
                    mode='markers',
                    name='Temperatures'))
fig.add_trace(go.Scatter(x=data['current_time'], y=SineFit,
                    mode='lines',
                    name='SineFit'))

fig.show()
```



A recorded demo of this chapter can be found here:

- Part 1: https://github.com/yanndebray/matlab-with-python-book/assets/128002745/dc9b069e-bf73-4294-aa1d-79813df9c62d
- Part 2: https://github.com/yanndebray/matlab-with-python-book/assets/128002745/6b777298-76ea-405e-b3ee-57fa5a18e211

7. SIMULINK WITH PYTHON

Simulink is a software based on MATLAB that enables the modeling, simulation and generation of code for physical and control systems. It is heavily used in the development of embedded systems, in applications such as automotive vehicle dynamics or flight control.

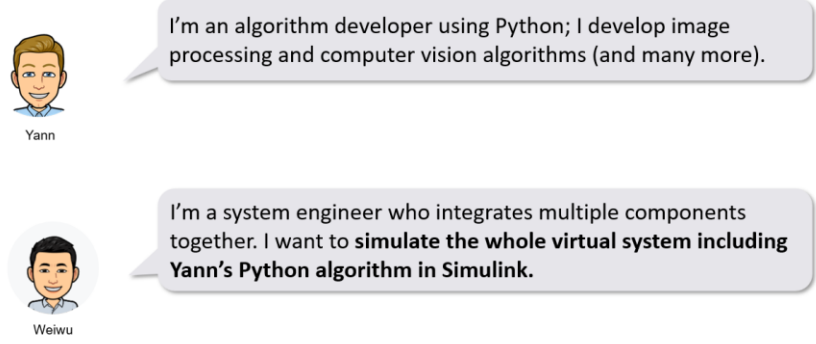
At the MATLAB Expo 2023[93], my colleague Weiwu and I looked at 4 typical scenarios we observe where Simulink is used with Python. In those scenarios, Weiwu is representing the Simulink usage, and I'm representing the Python usage (bearing in mind that in some cases they are the same users).

The 4 scenarios explored in this chapter:

- Bring Python code into Simulink as a library for co-execution
- Integrate TensorFlow and PyTorch models for both simulation and code generation
- Simulate a Simulink model directly from Python
- Export a Simulink model as a Python package for deployment

7.1. BRING PYTHON CODE INTO SIMULINK AS A LIBRARY FOR CO-EXECUTION

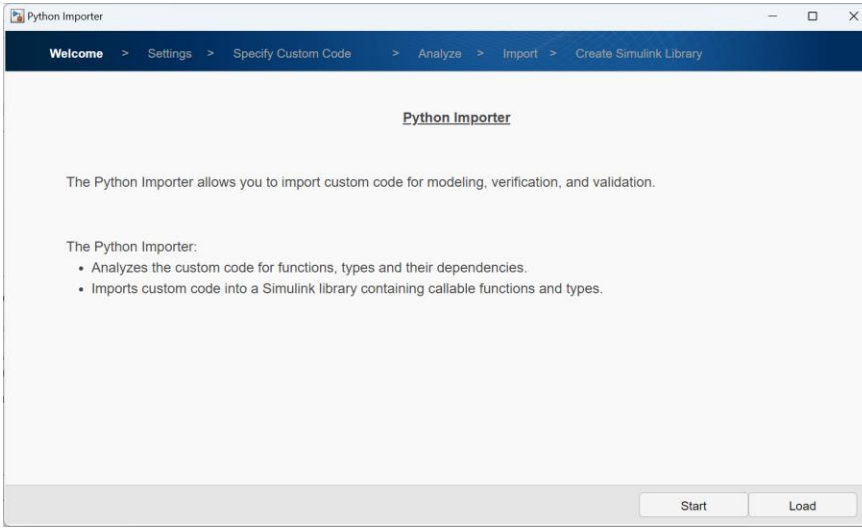
In the first scenario, an algorithm has been developed for computer vision using Python, and it needs to be integrated into a larger system for simulation.



[93] Video of Simulink with Python:
https://www.mathworks.com/videos/using-simulink-with-python-1683218506123.html

7.1.1. PYTHON IMPORTER FOR SIMULINK

In Simulink 23a, we shipped a new feature called the Python Importer. It can help you to bring your Python function in the Simulink as a library block easily. It provides a graphical wizard for step-by-step guidance. In this demo, we will use a Python Human Detector algorithm calling OpenCV from Python[94].



The importer will walk you through the following steps:

Welcome > Settings > Specify Custom Code > Analyze > Import > Create Simulink Library

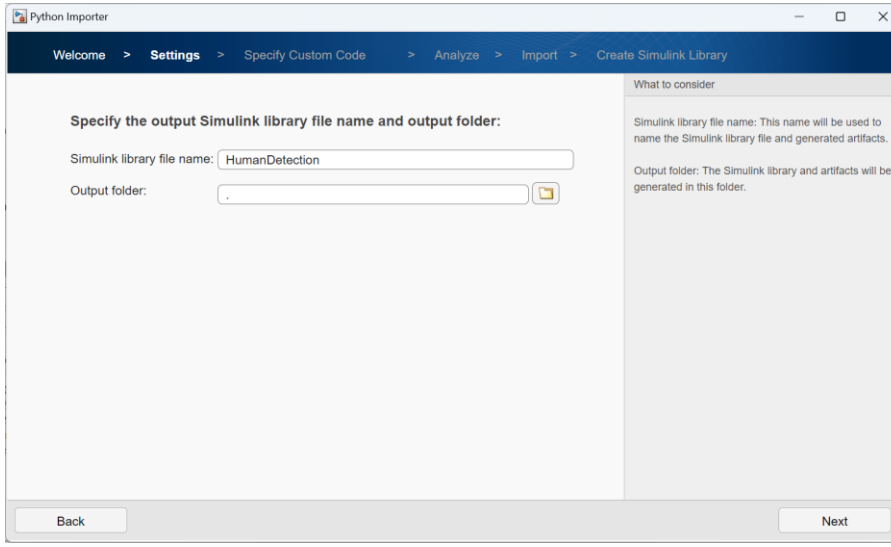
The Python Importer allows you to import custom code for modeling, verification, and validation.

The Python Importer:

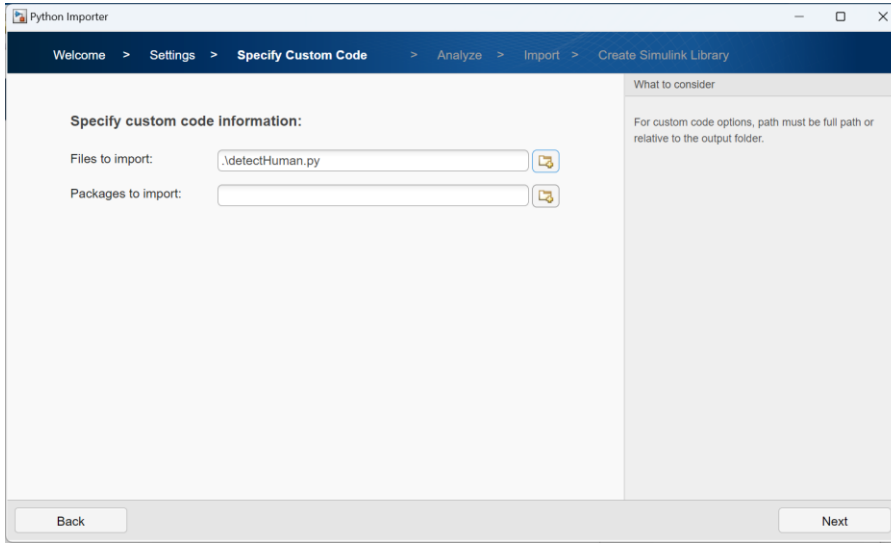
- Analyzes the custom code for functions, types and their dependencies.
- Imports custom code into a Simulink library containing callable functions and types.

[94] Demo calling OpenCV from Simulink:
https://github.com/mathworks/Integrate_Python_code_with_Simulink

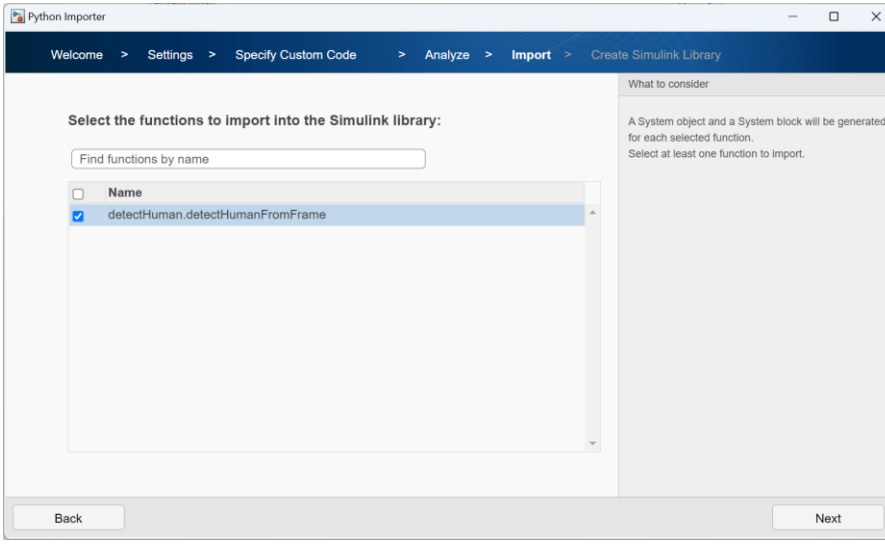
Settings

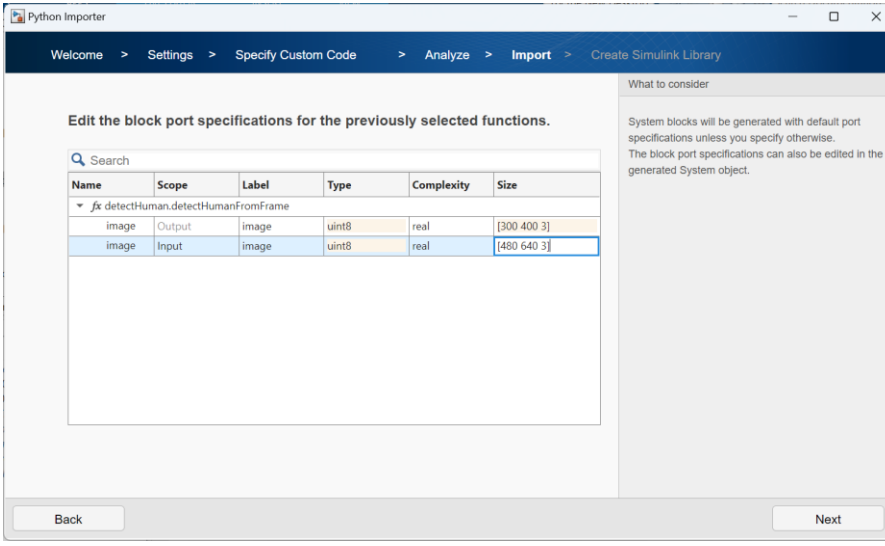


Specify Custom Code

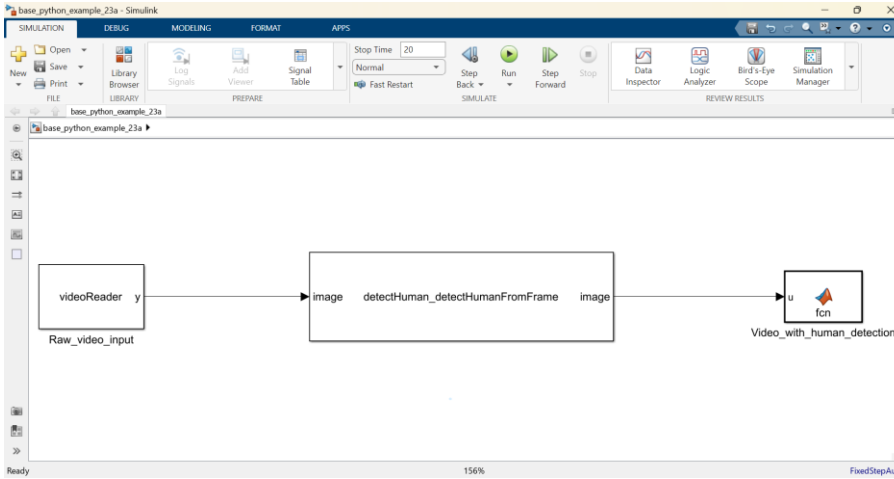


Analyze





The library is created and can be drag and dropped into a new diagram.

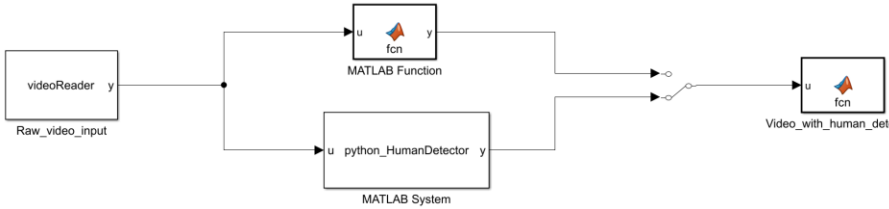


7.1.2. MATLAB FUNCTION AND SYSTEM OBJECTS

Prior to 23a, you had two different ways of calling Python from within a Simulink block:

- Using a MATLAB function
- Using a MATLAB system object

In both cases you are wrapping the Python code with the `py.` prefix.



The first option is with a MATLAB Function [95] . The only trick is the `coder.extrinsic` [96] function that specifies to explicitly call MATLAB and not generate code for the Simulink simulation.

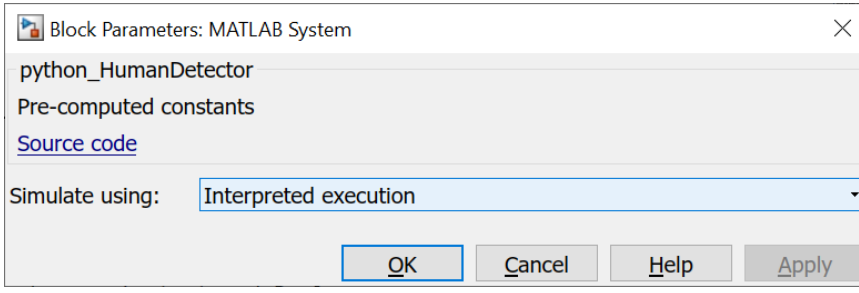
[95] Include MATLAB code in models: https://www.mathworks.com/help/simulink/slref/matlabfunction.html

[96] Declare a function as extrinsic: https://www.mathworks.com/help/simulink/slref/coder.extrinsic.html

If you double click on the block, you can see this code:

```
function y = fcn(u)
    coder.extrinsic('py.detectHuman.getHogObject')
    coder.extrinsic('py.detectHuman.detectHumanFromFrame')

    persistent hog;
    if isempty(hog)
        hog = py.detectHuman.getHogObject();
    end

    out = py.detectHuman.detectHumanFromFrame(u, hog);

    y = uint8(out);
end
```

The second option is with a MATLAB System[97] block. You need to set it up to *"Simulate using: Interpreted execution"*. In both of those options, you cannot generate the code for Python.



The source code for this MATLAB System follows the paradigm of Object-Oriented Programming[98]:

```
classdef python_HumanDetector < matlab.System
    % Pre-computed constants
    properties(Access = private, Nontunable)
        hog
    end

    methods(Access = protected)
```

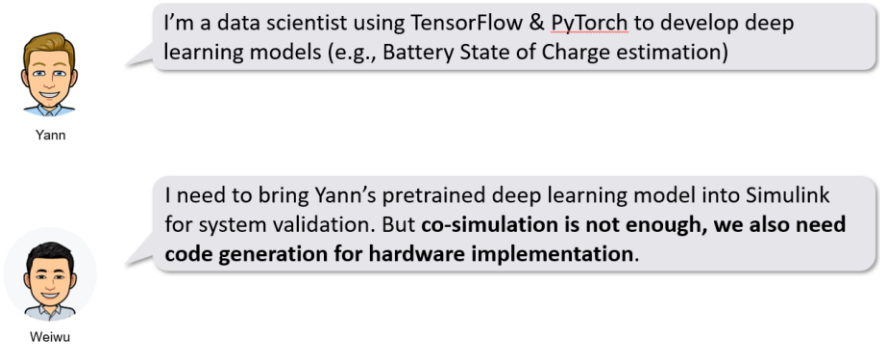
[97] Include System object in models: https://www.mathworks.com/help/simulink/slref/matlabsystem.html
[98] OOP in MATLAB: https://www.mathworks.com/products/matlab/object-oriented-programming.html

```
        function setupImpl(obj)
            % Perform one-time calculations, such as computing
constants
            obj.hog = py.detectHuman.getHogObject();
        end

        function y = stepImpl(obj,u)
            % Calculate y as a function of input u and discrete states.
            out = py.detectHuman.detectHumanFromFrame(u, obj.hog);
            y = uint8(out);
        end

        function out = getOutputSizeImpl(obj)
            out = [300 400 3];
        end

        function y1 = getOutputDataTypeImpl(obj)
            y1 = 'uint8';
        end

        function y1 = isOutputComplexImpl(~)
            y1 = false;
        end

        function out = isOutputFixedSizeImpl(obj)
            out = true;
        end

    end
end
```

The limitation with this approach is that you cannot generate code for embedded systems.

7.2. INTEGRATE TENSORFLOW AND PYTORCH MODELS FOR BOTH SIMULATION AND CODE GENERATION

In the second scenario, an AI deep learning model has been developed with TensorFlow or PyTorch, and it needs not only to be simulated with the whole system, but the code for AI model needs to be generated in order to be integrated into an embedded system.



One example of this workflow was presented by MBDA at the MATLAB UK Expo 2023[99].

Before we dive in, there are two options to integrate a python AI model into Simulink if code generation is a must have. The first one is importing the deep learning model in MATLAB directly with the Deep Learning Toolbox. And the second one is to simulate and generate code using TensorFlow Lite. The plus and delta for those two approaches are the following:

Those are library-free C and C++ code, some optimized code for Intel and ARM processor, and CUDA code for GPUs. The delta is, in this case, that the input process can sometimes be painful. And there's a bit of a need of custom code and manual validation testing.

- Import your deep learning model in MATLAB directly

  + Multi-platform code generation: library-free C/C++ code, optimized code for Intel and ARM processors, and CUDA code for NVIDIA® GPU
  Δ import process can be painful, need for custom code, and validation testing

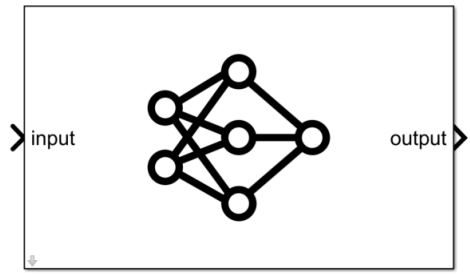
- Integrate deep learning models from TensorFlow Lite (TFLite)

  + requiring only a simple Python code to compile the model
  Δ requires the TensorFlow Lite interpreter and libraries built on the target hardware, which is currently limited to Windows and Linux targets

[99] Integrating AI into Simulink for Simulation and Deployment: https://www.mathworks.com/videos/integrating-ai-into-simulink-for-simulation-and-deployment-1699640610671.html

7.2.1. IMPORT YOUR DEEP LEARNING MODEL IN MATLAB DIRECTLY

You can convert models from TensorFlow, PyTorch, and ONNX into MATLAB. Once the model is converted in MATLAB, you can use the deep neural networks blocks to bring it in Simulink for both simulation and code generation.



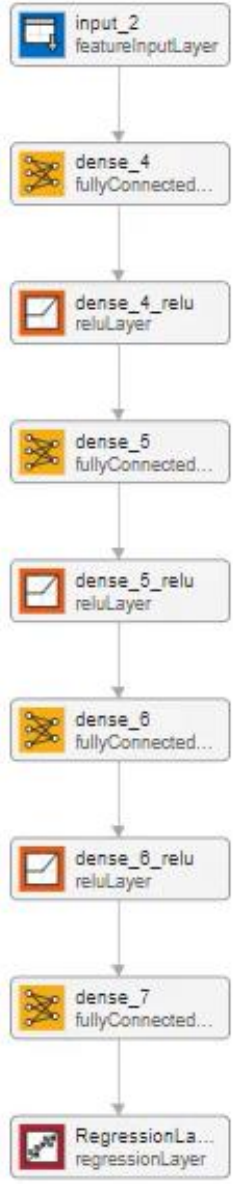
Let us use a TensorFlow model to predict the state of charge of a battery[100], and generate C-code to be integrated in the battery management system.

```
>> battery_SOC_net = importTensorFlowNetwork('BatterySOC_net','OutputLayerType','regression')
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation. Assembling network...
Import finished.

battery_SOC_net =

  DAGNetwork with properties:

         Layers: [9×1 nnet.cnn.layer.Layer]
    Connections: [8×2 table]
     InputNames: {'input_2'}
    OutputNames: {'RegressionLayer_dense_7'}
```

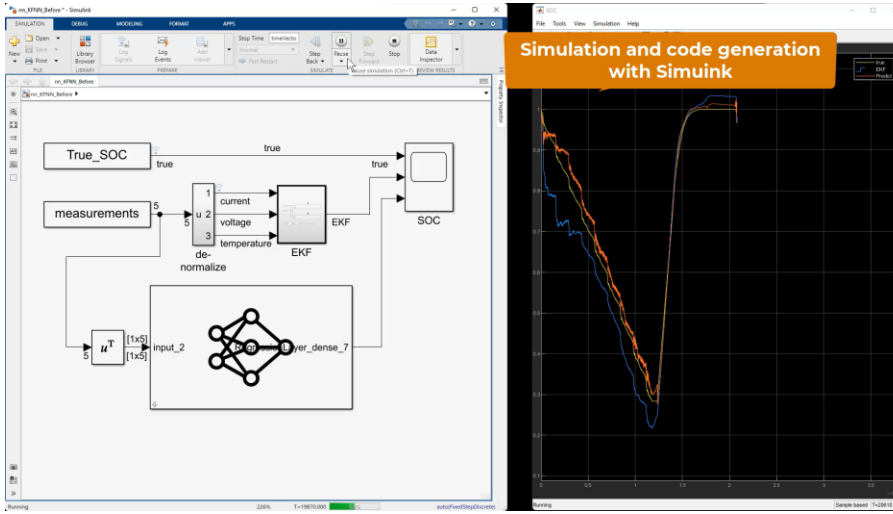
Convert the pretrained tensorflow model into MATLAB

You can visualize the model with the deep network designer, in order to understand what the imported network looks like.

[100] Deep learning-based battery state-of-charge estimation: https://www.mathworks.com/videos/integrate-tensorflow-model-into-simulink-for-simulation-and-code-generation-1649142078929.html



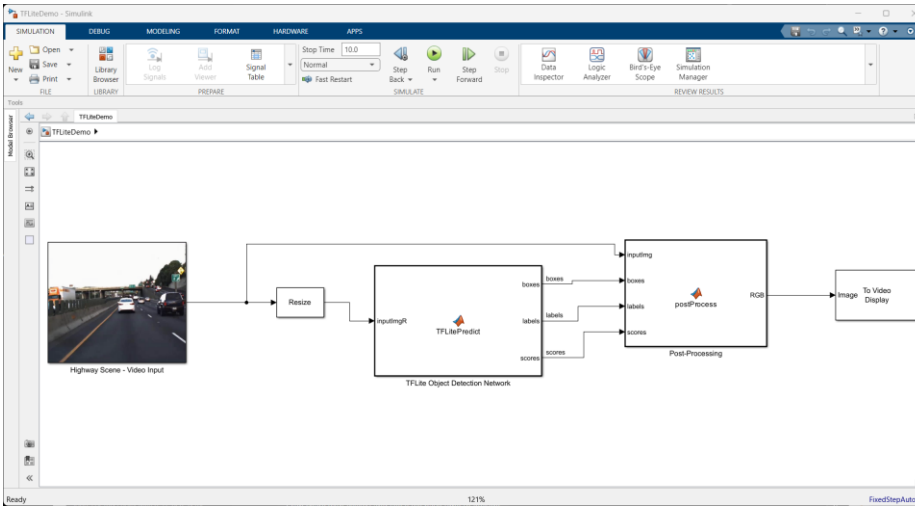
Then save the imported model as a .mat file and load it in a Simulink block.



Another use case for the integration of Python AI models into Simulink is the replacement of a control system or a physical system by a reduced model of this system.

7.2.2. INTEGRATE DEEP LEARNING MODELS FROM TENSORFLOW LITE (TFLITE)

In the scenario of a TensorFlow user willing to integrate AI into Simulink, one alternative approach would be to iterate through the use of TensorFlow Lite. We will illustrate this with an example of a TFLite Object Detector integrated through a MATLAB function. When generating the equivalent of this model, it will wrap the code with the TFLite runtime.




```
172  // Function for MATLAB Function: '<Root>/TFLite Object Detection Network'
173  static void TFLiteDemo_TFLiteModel_predict(coder_TFLiteModel_TFLiteDemo_T
174    *b_this, const uint8_T varargin_1[307200], real32_T varargout_1[400], real32_T
175    varargout_2[100], real32_T varargout_3[100], real32_T *varargout_4)
176  {
177    real_T count;
178    std::mem_fn(&invokeinterpreter::setVerbose)(b_this->Network, b_this->Verbose);
179    std::mem_fn(&invokeinterpreter::setProfiling)(b_this->Network,
180      b_this->EnableProfiling);
181    std::mem_fn(&invokeinterpreter::setNumThreads)(b_this->Network,
182      b_this->NumThreads);
183    std::mem_fn(&invokeinterpreter::setInputMean)(b_this->Network, b_this->Mean);
184    std::mem_fn(&invokeinterpreter::setInputStdDeviation)(b_this->Network,
185      b_this->StandardDeviation);
```

7.3. SIMULATE A SIMULINK MODEL DIRECTLY FROM PYTHON

In the third scenario, we start to look at the reverse problem, of calling Simulink from Python. I encounter this scenario a lot when it comes to test and automation of simulations by DevOps engineers, often as part of a CI/CD process[101].

[101] Continuous Integration and Delivery: https://about.gitlab.com/topics/ci-cd/

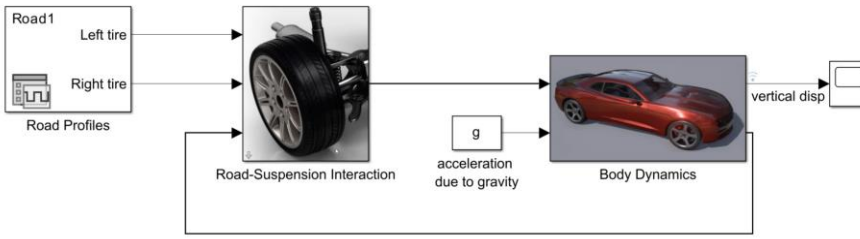
Weiwu

I use Simulink to model a dynamic system, for example, a vehicle suspension system.

Yann

I want to use a Python based automation framework to run Simulink simulations. I need to **invoke Weiwu's Simulink model from Python** for automated testing.

In this scenario and the next one, we will be using a road suspension model.



The first step is to use MATLAB functions to wrap the simulation. This function can make use of the simulation input object[102] in order to update the model parameters (in this case the body mass). Next, you need to use the sim[103] command in order to run the simulation in batch. You can then post-process the simulation results in MATLAB and/or Python.

MATLAB functions:

```
function res = sim_the_model(args)
% Utility function to simulate a Simulink model with the specified
parameters.
%
% Inputs:
%    StopTime: simulation stop time, default is nan
%    TunableParameters:
%       A struct where the fields are the tunanle referenced
```

[102] Simulation input object: https://www.mathworks.com/help/simulink/slref/simulink.simulationinput.html

[103] Simulate Simulink model: https://www.mathworks.com/help/simulink/slref/sim.html

```
%       workspace variables with the values to use for the
%       simulation.
%
%    Values of nan or empty for the above inputs indicate that sim
should
%    run with the default values set in the model.
%
% Outputs:
%    res: A structure with the time and data values of the logged
signals.

arguments
    args.StopTime (1,1) double = nan
    args.TunableParameters = []
end

    %% Create the SimulationInput object
    si = Simulink.SimulationInput('suspension_3dof');
    %% Load the StopTime into the SimulationInput object
    if ~isnan(args.StopTime)
        si = si.setModelParameter('StopTime', num2str(args.StopTime));
    end

    %% Load the specified tunable parameters into the simulation input
object
    if isstruct(args.TunableParameters)
        tpNames = fieldnames(args.TunableParameters);
        for itp = 1:numel(tpNames)
            tpn = tpNames{itp};
            tpv = args.TunableParameters.(tpn);
            si = si.setVariable(tpn, tpv);
        end
    end

    %% call sim
    so = sim(si);

    %% Extract the simulation results
    % Package the time and data values of the logged signals into a
structure
```

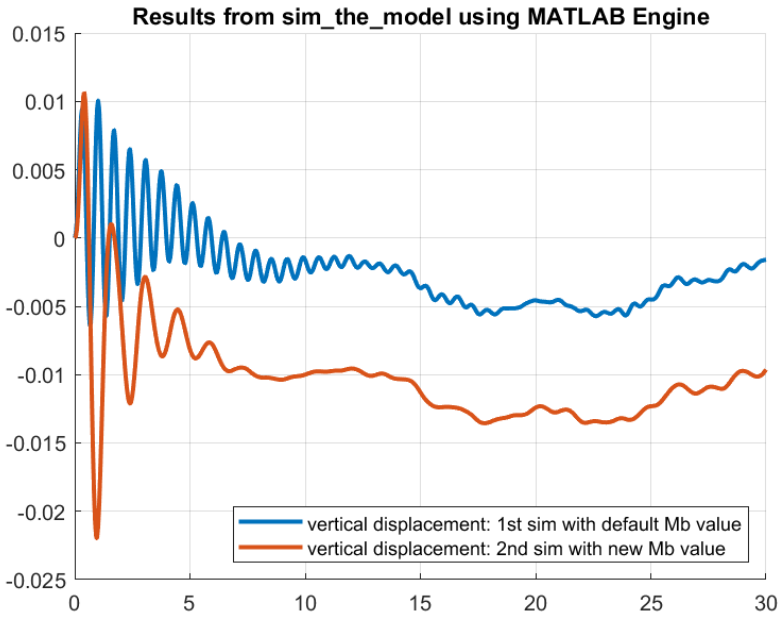
```
    res = extractResults(so,nan);

end % sim_the_model_using_matlab_runtime

function res = extractResults(so, prevSimTime)
    % Package the time and data values of the logged signals into a
structure
                                                ts                    =
simulink.compiler.internal.extractTimeseriesFromDataset(so.logsout);
    for its=1:numel(ts)
        if isfinite(prevSimTime)
            idx = find(ts{its}.Time > prevSimTime);
            res.(ts{its}.Name).Time = ts{its}.Time(idx);
            res.(ts{its}.Name).Data = ts{its}.Data(idx);
        else
            res.(ts{its}.Name).Time = ts{its}.Time;
            res.(ts{its}.Name).Data = ts{its}.Data;
        end
    end
end


function figHndl = plot_results(res, plotTitle)
%PLOT_RESULTS Plot results from call_sim_the_model

figHndl = figure; hold on; cols = colororder;


plot(res{1}.vertical_disp.Time,  res{1}.vertical_disp.Data,  'Color',
cols(1,:), ...
     'DisplayName', 'vertical displacement: 1st sim with default Mb
value');
plot(res{2}.vertical_disp.Time,  res{2}.vertical_disp.Data,  'Color',
cols(2,:), ...
    'DisplayName', 'vertical displacement: 2nd sim with new Mb value');


hold off; grid;

title(plotTitle,'Interpreter','none');
```

```
set(get(gca,'Children'),'LineWidth',2);
legend('Location','southeast');

end
```

Python code:

```
import matlab.engine

mle = matlab.engine.start_matlab()  # start the matlab engine

# Allocate res list to hold the results from 2 calls to sim_the_model
res = [0]*2
# 1st sim: with default parameter values: Mb = 1200 Kg
res[0] = mle.sim_the_model('StopTime', 30)

# 2nd sim: with new values for tunable parameters
tunableParams = {
    # use a new parameter for body mass: Mb = 5000 Kg
    'Mb': 5000.0
}
res[1]  =  mle.sim_the_model('StopTime',  30,  'TunableParameters',
tunableParams)

# callback into MATLAB to plot the results
mle.plot_results(res,  "Results  from  sim_the_model  using  MATLAB
Engine")

input("Press enter to close the MATLAB figure and exit ...")
mle.quit()  # stop the matlab engine
```

This code will produce the following MATLAB plot:



7.4. EXPORT A SIMULINK MODEL AS A PYTHON PACKAGE FOR DEPLOYMENT

In the fourth scenario, the goal is to deploy a Simulink model into a Python-based production system.

Weiwu

I use Simulink to model a dynamic system, for example, a vehicle suspension system.

Yann

We need to deploy Weiwu's Simulink model in a Python-based production environment. I want to **get a Python package which encapsulates a Simulink simulation** which can be used for deployment.

The steps are going to be the same here. The only difference is that you need to configure your model for deployment. That's because not all Simulink models are

deployable. So you need to add the following line to transform the simulation input object before calling sim:

```
si = simulink.compiler.configureForDeployment(si);
```

Then the next step is to compile the MATLAB function as a Python package:

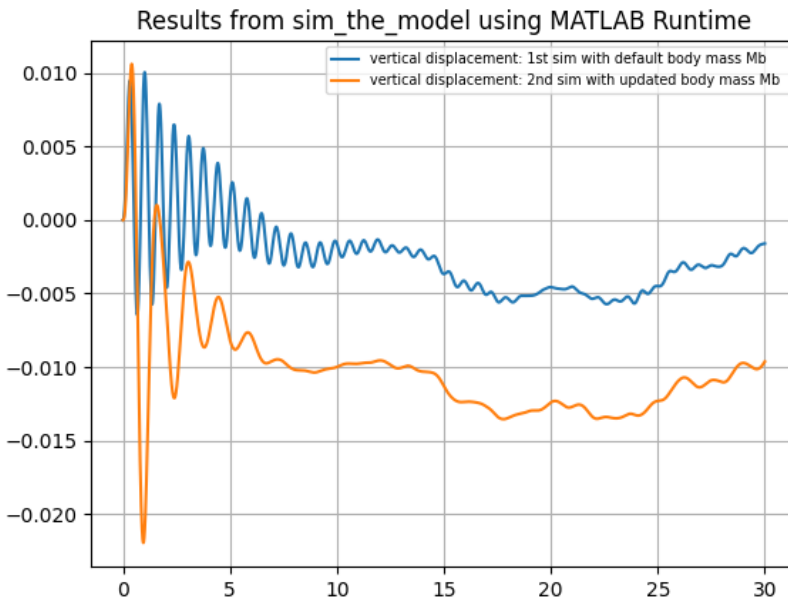
```
appFile = which('sim_the_model2');
outDir = fullfile(origDir,'sim_the_model2_python_package_installer');
compiler.build.pythonPackage(appFile, ...
        'PackageName','sim_the_model2', ...
        'OutputDir',outDir);
```

Finally, you need to install the Python package and call it from as such:

```
import sim_the_model2
import matplotlib.pyplot as plt
# initialize sim_the_model package
mlr = sim_the_model2.initialize()

# Allocate res list to hold the results from 2 calls to sim_the_model
res = [0]*2
# 1st sim: with default parameter values: Mb = 1200 Kg
res[0] = mlr.sim_the_model2('StopTime', 30)

# 2nd sim: with new values for tunable parameters
tunableParams = {
    'Mb': 5000.0   # use a new parameter for body mass Kg
}
res[1]  =  mlr.sim_the_model2('StopTime',  30,  'TunableParameters',
tunableParams)


# Plot the results
cols = plt.rcParams['axes.prop_cycle'].by_key()['color']
fig, ax = plt.subplots(1, 1, sharex=True)
ax.plot(res[0]['vertical_disp']['Time'],
res[0]['vertical_disp']['Data'], color=cols[0],
        label="vertical displacement: 1st sim with default body mass
Mb")
```

```
ax.plot(res[1]['vertical_disp']['Time'],
res[1]['vertical_disp']['Data'],
          color=cols[1], label="vertical displacement: 2nd sim with
updated body mass Mb ")


ax.grid()
lg = ax.legend(fontsize='x-small')
lg.set_draggable(True)
ax.set_title("Results from sim_the_model using MATLAB Runtime")
plt.show()

mlr.terminate()  # stop the MATLAB Runtime
```

This code will produce the following Python plot:



8. RESOURCES

8.1. GETTING PYTHON PACKAGES ON MATLAB ONLINE

MATLAB Online is a great pre-configured environment to demo MATLAB with Python. And I'm not just saying this now that I've joined the MATLAB Online product team (mid 2023). I have been using it since 2022 to run workshops with Heather, either for large public events like MATLAB EXPO, or for customer dedicated hands-on workshops. This avoids wasting the first half hour of any conversation in setting up the MATLAB & Python environments (see chapter 3), as Python 3 is pre-installed there.

But there's a catch: what always prevented me from using this nice online environment productively for bilingual workflows was the lack of Python packages, and a way to customize the environment. I obviously tried to find ways around this limitation, by uploading the packages sources as zipped files, and unzip it in MATLAB Online. However, for foundational packages like Numpy, it was already amounting for over 7,000 files to write to disk (only 50 Mb uncompressed). So, it takes a while…

Then recently, after discussing with my colleagues in the MATLAB Online team, they suggested a very simple approach to retrieve pip from a single script:

```
>> websave("get-pip.py","https://bootstrap.pypa.io/get-pip.py");
>> !python get-pip.py
>> !python -m pip --version
pip 24.2 from /home/matlab/.local/lib/python3.10/site-packages/pip (python 3.10)
```

You can now simply install a package like numpy as such:

```
>> !python -m pip install numpy
```

Bear in mind that the MATLAB Online environment is ephemeral, and that you will have to repeat this process each time you start a new session.

8.2. GENERATE THIS BOOK WITH QUARTO AND PANDOC

This book is written with different editing tools: Word, Live scripts, , Jupyter Notebook and Markdown. In order to maintain consistency and automate the generation of the different formats, different conversions are applied:

- live scripts > word
- live scripts > markdown
- live scripts > jupyter notebook
- jupyter notebook > markdown
- jupyter notebook > word

I am leveraging an open-source software called Quarto[104], developed by Posit (formerly known as the company RStudio). Quarto is built on Pandoc[105] a universal document converter that allows you to write in Markdown, and generate a book in multiple formats, including Word, PDF, HTML, Markdown, and Jupyter Notebooks.

Quarto is a great tool to write technical books, as it complements Pandoc by supporting code in multiple languages, including MATLAB and Python, with interactive code snippets, that can be executed in VS Code or Jupyter.

Here is one example of a command to generate word documents from the Jupyter notebooks:

```
quarto pandoc notebook.ipynb -s -o README.md
```

[104] https://quarto.org/
[105] https://pandoc.org/