Getting started with Python for MIKE+

Ryan Murray Rocco Palmitessa

2025-05-15

Table of contents

Introduction

DHI offers a range of free, open-source Python libraries that enable automated and reproducible MIKE+ workflows, as well as unlock the potential for robust and flexible analyses. This course is designed for experienced MIKE+ modelers who are new to Python, providing a practical foundation to begin applying concepts to real projects. You'll gain essential skills to read, run, and modify Python scripts relevant to MIKE+ modelling through focused, hand-tailored examples. The course will orient you to a new way of working, guiding you through the transition from a GUI to a script-based environment, helping you navigate common challenges, and giving you the confidence to continue exploring Python and seek out resources to further develop your skills.

Why Python with MIKE+?

Using Python alongside MIKE+ provides the following advantages:

- Efficient handling of various file types, including dfs0, res1d, and xns11
- Conversion of data between MIKE+ and third-party formats such as CSV and Excel
- Flexibility to modify MIKE+ databases, access tools, and run simulations
- Automation of modelling tasks using a straightforward scripting syntax
- Reproducible and documented workflows that enhance model quality assurance

Intended Audience

This course is ideal for MIKE+ modelers who:

- Are eager to explore Python's potential in MIKE+ modelling
- Want to enhance, automate, or document parts of their workflows with Python
- Seek more flexible and robust techniques for advanced modelling needs

Course Structure

The course focuses on practical applications of Python for common MIKE+ modelling tasks. Content generally consists of a combination of videos, live sessions, and hands-on exercises. We will cover Python libraries such as MIKE IO, MIKE IO 1D, and MIKE+Py.

Module 1 | Foundations Topics: Python and Python Packages, Visual Studio Code, GitHub, Jupyter Notebooks, LLMs for coding, Pandas, Matplotlib, Documentation

Module 2 | Time Series Topics: dfs0 files, plotting, statistics, selections, resampling, basic data validation

Module 3 | Network Results Topics: network result files (e.g. res1d, res, res11), selecting data, extracting results, geospatial formats (e.g. shapefiles)

Module 4 | Calibration Plots and Statistics Topics: basic statistics and plots relevant for model calibration

Module 5 | MIKE+Py Topics: databases and SQL, modifying MIKE+ databases, accessing GUI tools, running simulations

Module 6 | Putting Everything Together Topics: final project applying lessons of previous modules.

Course Objectives

After completing this course, you should be able to:

- Install Python and related packages for use with MIKE+
- Apply Python to create reproducible and automated workflows
- Explore documentation and run example notebooks and scripts
- Connect with the open-source Python community and MIKE+ modelers

Part I

Module 1 - Foundations

Welcome to Module One!

In this module, you'll gain the skills to confidently set up your coding environment, access course materials, and run Python code with ease. Our focus is on:

1. Mastering essential tools (e.g. GitHub, Visual Studio Code, uv).
2. Running and understanding Python scripts and Jupyter notebooks.
3. Learning basic Python syntax and concepts.
4. Exploring core libraries (e.g. NumPy, Pandas, Matplotlib) for data analysis.

Don't worry if it feels fast-paced — you'll practice these concepts in later modules. Let's get you set up and ready to dive in!

1 GitHub

GitHub is a website for storing, sharing, and collaborating on software development projects. It's an especially popular platform for open-source software. DHI uses GitHub for hosting its entire open-source Python ecosystem, including documentation and examples.

1.1 DHI's Python Ecosystem on GitHub

DHI's Python ecosystem is organized into modular Python packages based on functionality. This is a common pattern in Python that empowers users to flexibly combine functionalities to meet specific project needs. An overview of Python packages useful for MIKE+ modelling is provided in the table below.

Package	Description
MIKE IO	Read, write and manipulate dfs0, dfs1, dfs2, dfs3, dfsu and mesh files.
MIKE IO 1D	Read, manipulate, and analyze res1d, res, resx, out, and xns11 files.
MIKE+Py	MIKE+Py is a python interface for MIKE+.
MODELSKILL	Compare MIKE model results and observations.
TSOD	Anomaly Detection for time series data.

Feel free to browse additional open-source packages on DHI's GitHub profile.

1.2 Why visit a Python package's GitHub page?

You'll use GitHub for:

- Accessing documentation and examples.
- Creating 'issues' and/or 'discussions' when you need help.
- Checking out changes with new package versions.
- Browsing source code and/or contributing code you think is generally useful.

1.3 Typical structure of a Python package on GitHub

Please watch the video below for a guided tour of how DHI organizes their Python packages on GitHub.

https://www.youtube.com/embed/skF41BJT7m4?si=R1YSH6WLC7PdA4LV

2 Python Management

Python continuously releases new versions. Similarly, individual Python packages (hosted on PyPI) also continuously release new versions. Python scripts usually have dependencies on specific Python versions and packages, which highlights the need to carefully managing these. This is similar to different versions of MIKE+: you would not expect a MIKE+ 2025 model to run with MIKE+ 2023.

2.1 Tools

There are several tools for managing Python and packages together. Two common options are:

1. uv
2. Miniforge

This course uses *uv*. Please install *uv* according to their official installation instructions. Use the "standalone installer" for Windows.

Confirm you properly installed *uv* by opening a terminal and running:

```
uv --version
```

> Learn basics of terminals
>
> Installing and using *uv* requires using a terminal. Being familiar with terminals is generally useful for Python. This course assumes basic knowledge. If you've never used a terminal before, then please refer to an introductory resource such as: Windows PowerShell.

2.2 Installing Python with uv

You can install Python with *uv* from the command line:

```
uv python install
```

By default, this installs the latest version of Python (3.13.2 at the time of writing).

Confirm it installed correctly by running:

```
uv run python --version
```

2.3 Virtual Environments

> **Note**
>
> Virtual environments are an advanced Python topic, however, they are fundamental to using *uv*. Therefore, they will not be covered in depth, but explained just enough to be useful.

Virtual environments are useful for isolating dependencies between projects. For example, let's say you work on two projects: Project A and Project B. If Project A requires a different version of Python than Project B, then you can handle that by creating virtual environments for each project. This avoids a common issue encountered when not using virtual environments. Conceptually, a virtual environment is a single Python version and set of Python packages.

Create a new folder, and make a virtual environment:

```
uv venv
```

> **Tip**
>
> Use the terminal `cd` command to change its current directory. Alternatively, install Windows Terminal to easily launch a terminal from a folder within File Explorer via the right-click context menu.

Notice a folder called `.venv` was created. Explore that folder to see what it contains. Can you find the file `Python.exe`and the folder `site-packages`?

It's good practice to create a single virtual environment in the root directory of each project. Therefore, the remainder of this course assumes you always run *uv* from within a folder containing a virtual environment.

Refer to uv's documentation for additional details.

2.4 Python package management

uv provides two different approaches for Python package management. This course uses their *pip interface*. Common workflows are shown in the following sections. Refer to uv's documentation for more details.

2.4.1 Install packages

Install Python packages with uv as follows:

```
uv pip install <package-name>
```

For example, install *mikeio* as follows:

```
uv pip install mikeio
```

Look at the `site-packages` folder again. Notice that it now includes mikeio and many other packages. When a package is installed, all of its dependencies are also installed automatically.

2.4.2 List installed packages

List all installed Python packages and their versions with:

```
uv pip list
```

2.4.3 Upgrade packages

Upgrade an older package version to the latest version as follows:

```
uv pip install --upgrade mikeio
```

2.4.4 Install specific package versions

Occasionally there's a need to install an older version of a package, which can be done as follows:

```
uv pip install mikeio==1.7.1
```

2.5 Example video

https://www.youtube.com/embed/aOzYOZqSV60?si=0YKKOnakhujo8dPC

3 Integrated Development Environments (IDEs)

An Integrated Development Environment (IDE) is a software that bundles together tools convenient for software development. This course uses Visual Studio Code as an IDE, which is a popular free and open-source software provided by Microsoft.

3.1 Why use an IDE?

There are several benefits to using an IDE compared to using a text editor like Notepad:

- Designed for easy code writing, with several shortcuts
- Syntax highlighting for more readable code
- Automatic code completion
- Integrated terminal
- Integrated LLM chat and code completion
- Highly customizable with extensions

3.2 Install Visual Studio Code

Install Visual Studio Code (VSCode) according to their official instructions.

> **Caution**
>
> You may stumble upon a software called Visual Studio, which is **different** than Visual Studio Code.

3.3 Getting Started

VS Code provides excellent documentation. Please refer to their getting started guide for a basic introduction.

3.4 Visual Studio Code Extensions

This course uses the Python extension for VS Code. Extensions can be installed from within VS Code. Refer to VS Code's documentation for guidance.

3.5 Opening Projects

VS Code can be used in different ways. This course uses a common workflow of opening VS Code from the root directory of a project folder. Alternatively, open a project folder via "Open Folder" from within VS Code.

3.6 Selecting Python Interpreters

VS Code should automatically detect virtual environments located in the root project directory.

Otherwise, there's an option of manually selecting which Python Interpreter VS Code uses. Access it via the Command Palette (CTRL + SHIFT + P) and typing "Python: Select Interpreter".

VS Code uses the selected interpreter for running scripts, as well as for other features like auto completion.

3.7 Example - Setting up a fresh project

https://www.youtube.com/embed/ROvfBhSLquw?si=MfLDx9HFVadnWZne

4 Python Scripts

A Python script is a file with the extension *.py* that contains Python code that's executable via Python's interpreter.

4.1 Running Python Scripts

Python is most powerful when scripts are reused. Therefore, it's important to understand both how to run scripts others have sent you, as well as how to explain how others can use scripts you wrote.

4.1.1 Running in Terminal

You can run a script from the terminal by running:

```
uv run python example_script.py
```

4.1.2 Running in VS Code

You can run scripts from VS Code's user interface. Under the hood, it executes the script in the terminal, so this is only a matter of preference. Refer to VS Code's documentation on how to run Python code.

> Tip
>
> Running scripts in debug mode is more convenient via VS Code's user interface. This lets you walk through code line by line as it executes, which is helpful when investigating unexpected outcomes.

4.2 Script Dependencies

As previously mentioned, Python code includes dependencies on a set of Python packages (e.g. mikeio). If a script is run with a virtual environment that is missing these dependencies, there'll be an error along the lines of: *ModuleNotFoundError: No module named 'mikeio'*. The package listed in the error message (e.g. mikeio) needs to be installed before running the script.

> Tip
>
> *uv* provides a way of defining dependencies within the script itself, such that they are automatically detected and installed when running the script with *uv*. Refer to uv's documentation on script inline metadata for details.

4.3 Example - running scripts

https://www.youtube.com/embed/TL2ZS1TeZeY?si=wZ3vy2gxBk9YdtJE

5 Jupyter Notebooks

A Jupyter Notebook is a file with the extension *.ipynb* that combines code, its output, and markdown into an interactive notebook-like experience.

5.1 Comparison with Python Scripts

A key difference is that notebooks are *interactive*, whereas scripts simply run from start to end. Generally, notebooks are more useful for exploratory or visual workflows (e.g. making plots, or analyzing data). It's also a great tool for learning Python.

5.2 Terminology

The following are fundamental concepts of Jupyter Notebooks:

Cell A Jupyter Notebook is a collection of cells.
Code Cell A cell containing Python code, whose output shows below after execution.
Cell Output The output after executing a cell, which could be many things (e.g. a number, plot, or table)
Markdown Cell A cell containing markdown for nicely formatted text.
Kernel Responsible for executing cells. Same as Python virtual environment for the purposes of this course.

5.3 Running a Jupyter Notebook

The Python extension for VS Code allows opening jupyter notebook files (.ipynb).

Upon opening a notebook, all cells are displayed along with any saved output of those cells.

Running a notebook first requires selecting the kernel (i.e. the Python virtual environment). If the virtual environment has not installed the package `ipykernel`, then VS Code will ask to do that. Alternatively, manually install it via:

```
uv pip install ipykernel
```

Next, "Run All" to run all cells from top to bottom. It's also possible to run (or re-run) cells individually in any order.

> **Tip**
>
> It's good practice to organize notebooks such that they run from top to bottom.

5.4 Creating a Jupyter Notebook

Create a Jupyter Notebook from within VS Code by opening the Command Palette (CTRL + SHIFT + P) and typing "Create: New Jupyter Notebook".

Save the notebook in a project folder to help VS Code automatically find the project's virtual environment. Then, start adding and running cells.

5.5 Useful Keyboard Shortcuts

There's a few useful keyboard shortcuts when working with notebooks:

1. Shift + Enter: Run the current cell and move to the next.
2. Ctrl + Enter: Run the current cell.
3. A: Insert a new cell above.
4. B: Insert a new cell below.

5.6 Additional resources

For additional information, refer to VS Code's documentation on jupyter notebooks.

5.7 Example - Using Jupyter Notebooks

https://www.youtube.com/embed/hBQ7lS-6crY?si=87M3mjkNJykX2w7s

6 Python Basics

This section provides a crash course on basic Python concepts used throughout the course. It is purposefully brief, with additional resources provided at the end.

> **Tip**
>
> Follow along by running these cells in a Jupyter Notebook. Alternatively, run them in the Python Interpreter which can be started with `uv run python`.

6.1 Using libraries

Most functionality useful for MIKE+ modelling exists in Python packages (e.g. mikeio). Therefore, it's important to understand how to access functionality in a Python package.

> **Note**
>
> The terms package, library, and module are used interchangeably throughout this course.

6.1.1 Import libraries

Import libraries using the `import` statement:

```
import math
```

Or import specific functionality from a library:

```
from math import sqrt
```

6.1.2 Objects

All imports are *objects* containing some functionality. Objects have *members* accessible via the dot notation:

```
math.pi
```

```
3.141592653589793
```

Dot accessors can be chained together, since all members are also objects.

```
math.pi.is_integer()
```

```
False
```

There are a few common types of objects to be aware of:

1. Modules: *reusable code you can import into your program.*
2. Classes: *templates for creating objects with specific properties and behaviors.*
3. Functions / Methods: *blocks of code that return a result.*
4. Data: *any stored information (e.g. numbers, text).*

See the type of an object with:

```
type(math)
```

```
module
```

See the members of an object with:

```
dir(math)
```

Get help for an object:

```
help(math)
```

Good libraries have documentation. For example, see the documentation for math.

6.1.3 Using Functions / Methods

> **Note**
>
> This course will use the terms 'function' and 'method' interchangeably.

Use a function by *invoking* it with round brackets:

```
sqrt(25)
```

```
5.0
```

Between the brackets is the function *arguments*. There's different ways of specifying arguments. For example, there could be a list of arguments:

```
math.pow(2, 3)
```

```
8.0
```

6.1.4 Using Classes

Some library functionality is provided via a *class* that needs to be *instantiated* before using it.

Below, the `Random` class is instantiated and assigned to the identifier `my_random` for reference later on.

```
from random import Random
my_random = Random()
```

An instantiation of a class is called an *instance*, and is also an object whose functionality is accessible with the dot notation:

```
my_random.random() # returns a random number
```

```
0.2685844172375744
```

6.2 Variables

Store data/objects in named variables by using the assignment operator `=`.

```
result = 1 + 1
result
```

```
2
```

> **Note**
>
> A valid name must be used. In general, this means it must start with a letter or underscore.

Variable names can be referenced anywhere after their definition.

```
result = result * 2
result
```

```
4
```

6.3 Collections

A common need is to have a collection of related data. Perhaps the most common type of collection is a `list`, which is briefly introduced below.

Create a list with square brackets. Optionally include comma separated elements, otherwise an empty list is created.

```
my_numbers = [1, 2, 3]
my_numbers
```

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

Append elements to an existing list.

```
my_numbers.append(4)
```

Access a specific element by *indexing* the list with the zero-based index. Zero refers to the first element, one the second, and so on.

```
my_numbers[0]
```

```
1
```

Access a subset of a list by *slicing* it. The example below accesses elements with index 0 up to, but excluding, 2.

```
my_numbers[0:2]
```

```
[1, 2]
```

6.4 Control Logic

Control logic allows the flow of a program to be controlled via boolean conditions.

6.4.1 Conditional statements

Use `if` statements to execute code only if the specified condition is true.

```
if 100 > 10:
    print("100 is greater than 10")
```

```
100 is greater than 10
```

> **Note**
>
> The code that the `if` statement applies to is called a *block*, which must be indented.

Use `else` statements after an `if` statement to execute code only if the condition is untrue.

```
if 100 < 10:
    print("100 is less than 10")
else:
    print("of course, 100 is not less than 10")
```

```
of course, 100 is not less than 10
```

6.4.2 Loops

A `while` loop continuously executes a block of code *while* the specified condition is true.

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

```
0
1
2
```

A `for` loop executes a block of code per element in a specified collection.

```
for fruit in ["Apple", "Banana", "Orange"]:
    print(fruit)
```

```
Apple
Banana
Orange
```

6.5 Additional resources

Learning Python should be a continuous endeavor through practice. Luckily there's an abundance of high quality resources online. Here's a few examples:

- Official Python Documentation
- Learn X in Y minutes
- FreeCodeCamp: Scientific Computing with Python

> **Tip**
>
> Python is used by a wide variety of domains (e.g. web development). Try to use resources specific for engineering/science applications for a more efficient learning path.

6.6 Example - Using Python's Interpreter

https://www.youtube.com/embed/U2gxRmstAC0?si=u07FDPI3u1gtp6b5

7 LLMs for Coding

Large Language Models (LLMs) can significantly enhance coding efficiency. They're also a great tool for explaining code, which is helpful for learning Python.

7.1 Ways of using LLMs

LLMs for coding is an area under rapid development. Here are a few ways of using LLMs for coding, roughly in the order in which they became available for use:

1. Chat interfaces via web (e.g. ChatGPT, Mistral AI)
2. Chat interfaces via an IDE (e.g. GitHub Copilot Chat)
3. Inline chat and autocompletion in IDE (e.g. GitHub Copilot)
4. Agentic coding with specialized IDEs (e.g. Windsurf)

Using LLMs is completely optional for the course. However, since GitHub Copilot is free and integrated with VS Code, our suggestion is to try it out as a learning assistant.

7.2 Ideas of how to use LLMs in coding

A few ideas of how to use LLMs in coding:

- Write scripts from scratch based on a description of what's needed
- Explain a given script line by line to enhance understanding
- Understand cryptic error messages, and get potential solutions
- Review the quality of your code to see if it could be improved

7.3 Example - Andrej Karpathy using LLMs for coding

https://www.youtube.com/embed/EWvNQjAaOHw?si=VUpLr2-rEUsOWZEI&start=4471

8 Scientific Python

Python is a general purpose programming language that's used by a broad range of domains. MIKE+ modelling workflows most closely align with the *scientific python* community.

8.1 Package ecosystem for scientific Python

There are several useful packages for engineering and science. This course will use the following packages:

- NumPy
- Matplotlib
- pandas

Check out packages sponsored by NumFOCUS for an overview of useful libraries.

> **Tip**
>
> DHI builds their Python ecosystem on top of these packages, to enable better integration between them and allow scientists/engineers the flexibility that's often required.

8.2 NumPy

NumPy is a package that essentially enables faster numerical computing on large arrays than would otherwise be possible via Python collections. It is foundational to many other packages.

NumPy is imported as `np` by convention:

```
import numpy as np
```

> **Note**
>
> Import as 'np' simply imports numpy and creates an alias for it as 'np'.

Create a NumPy array from a Python collection:

```
my_array = np.array([1, 2, 3])
my_array
```

```
array([1, 2, 3])
```

Use *vectorized* operations on arrays. For example, multiply all elements of the previous array by 2:

```
my_array * 2
```

```
array([2, 4, 6])
```

Index and slice arrays the same way as Python collections:

```
my_array[0]
```

```
np.int64(1)
```

Perform aggregation functions on an array (e.g. sum, mean, max):

```
my_array.sum()
```

```
np.int64(6)
```

Refer to NumPy's official documentation for additional information.

8.3 Pandas

Pandas builds upon NumPy with a special focus on tabular data (like spreadsheets, or csv files).

Pandas is imported as 'pd' by convention:

```
import pandas as pd
```

Create a DataFrame, which is like a 2D labeled array (rows + columns):

```
import pandas as pd
data = [['Alice', 25], ['Bob', 30]]
df = pd.DataFrame(data, columns=['name', 'age'])
df
```

	name	age
0	Alice	25
1	Bob	30

Select a single column by name:

```
df['age']
```

```
0    25
1    30
Name: age, dtype: int64
```

Perform aggregation operations just like as with NumPy:

```
df['age'].mean()
```

```
np.float64(27.5)
```

Import data from a csv file into a pandas DataFrame:

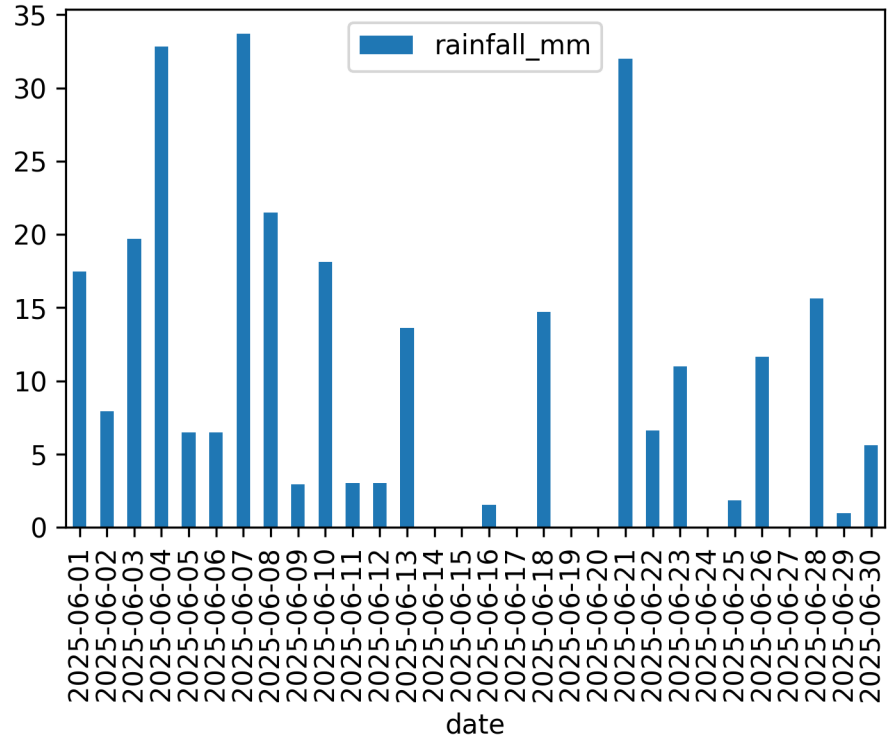
```
rainfall = pd.read_csv('data/fake_daily_rainfall.csv', index_col='date')
rainfall.head()
```

date	rainfall_mm
2025-06-01	17.450712
2025-06-02	7.926035
2025-06-03	19.715328
2025-06-04	32.845448
2025-06-05	6.487699

> **Note**
>
> Use the `head` method of a DataFrame to view the first five rows of very long DataFrames.

Create plots from a DataFrame:

```
rainfall.plot(kind='bar')
```



Export a DataFrame to csv, excel, or other formats:

```
rainfall.to_csv("temp.csv")
rainfall.to_excel("temp.xlsx")
```

Refer to pandas's official documentation for additional information.

8.4 Matplotlib

Matplotlib is a library for creating plots and is commonly used by other libraries.

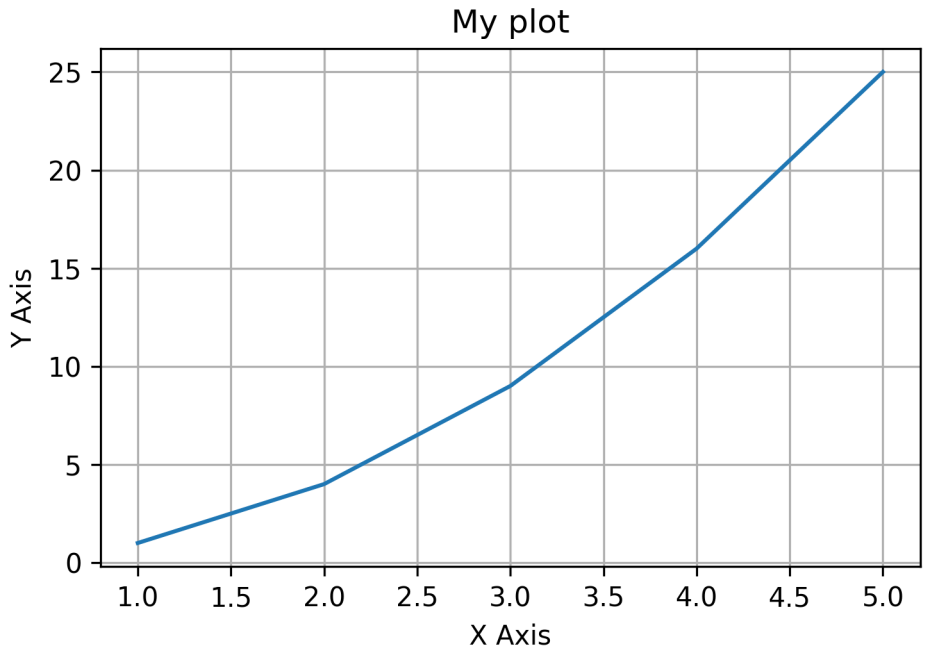
Matplotlib is imported as 'plt' by convention:

```
import matplotlib.pyplot as plt
```

Create a simple line plot:

```
# Create some data
x = np.array([1, 2, 3, 4, 5])
y = x ** 2

# Make the plot
plt.plot(x, y)              # Plots x vs y
plt.title("My plot")        # Gives a title to the plot
plt.xlabel("X Axis")        # Labels the x-axis
plt.ylabel("Y Axis")        # Labels the y-axis
plt.grid()                  # Turns on grid lines
```



Refer to Matplotlib's official documentation for additional information.

Also, feel free to check out their example gallery for a sense of what's possible.

8.5 Example - Importing and Plotting a Time Series CSV File

https://www.youtube.com/embed/biRU1_LfLL8?si=Ik-65D8AsbkLyNzZ

Homework

Exercise 1

1. Create a GitHub account.
2. Find and explore the *mikeio1d* repository. Can you find its documentation?
3. What's the current version of *mikeio1d*?
4. Search around GitHub and *star* some repositories you think are cool.

Exercise 2

1. Make a new folder somewhere on your PC.
2. Open the folder in Visual Studio Code.
3. Create a virtual environment in that folder using *uv* from VS Code's terminal.
4. Install *mikeio1d* in the virtual environment using *uv*.
5. List all the packages in the virtual environment. Do you recognize any?
6. Select the Python Interpreter in VS Code to be the virtual environment you created.

Exercise 3

1. From VS Code, create a new .py file under the project folder created in exercise two.
2. Copy the following code into the script:

```
import mikeio1d

print("I'm a script that uses mikeio1d version " + mikeio1d.__version__)
```

3. Run the script from VS Code's terminal using *uv*.
4. Run the script from VS Code's user interface (i.e. via the 'Run' menu).
5. Do you get the same output for steps 3 and 4?

Exercise 4

1. Install `ipykernel` into the same virtual environment of the previous exercises.
2. Create a new Jupyter Notebook from within VS Code.
3. Make sure the kernel matches your virtual environment, otherwise update it.
4. Paste the code from exercise three into a code cell.
5. Run the cell created in the previous step. Does the output match that of exercise three?

Exercise 5

1. Install the package `cowsay` into your virtual environment.
2. Create a new script, and import the function `cow` from `cowsay`.
3. Make a list containing the names of three countries you want to visit.
4. Loop over the list, and invoke the function `cow` by passing the current element of the list.
5. Run the script. What do you see?
6. Try to get the same output in a jupyter notebook by using two code cells.

Exercise 6

1. Download this time series csv file into your project folder.
2. Install `pandas` and `matplotlib` into your virtual environment.
3. Create a new Jupyter Notebook and import pandas
4. Load the downloaded csv file into a *DataFrame* using pandas.
5. Calculate the minimum, mean, and maximum values.
6. Plot the DataFrame. Do the values calculated from the previous step make sense?

Part II

Module 2 - Time Series

Part III

Module 3 - Network Results

Part IV

Module 4 - Calibration Plots and Statistics

Part V

Module 5 - MIKE+Py

Part VI

Module 6 - Putting Everything Together