FOLDER STRUCTURE & PACKAGES

ERT 474/574

Open-Source Hydro Data Analytics

Sep 10th 2025

University at Buffalo The State University of New York

Recap

- Assign the list `[1,2]` to a variable `a`

- If `a` is not equal to `b`, print "`a and b are different`";

  otherwise, print "`a and b are the same`".

- if `a > b` and `a > c`, print "`a is larger than b and c`";

  otherwise, if `a > b` or `a > c`, print "`a is larger than either b or c`";

  otherwise, print "`a does not exceed b or c`"

Practice

Write a Python program that:

- Write a function to calculate the average score.
  - Input: a list of scores
  - Output: average of all scores
- Write a function to print who passed and who failed:
  - Input: `students` (data type: `dict`)
  - Output example: "`Alice passed the test`"
- Uses proper indentation and comments.

```
# Define the dictionary
# Key: student names
# Value: student scores
students = {
    "Alice": 85,
    "Bob": 58,
    "Charlie": 72,
    "Diana": 49
}
```

Tips:

How can we get all keys in a dictionary?

```
students.keys() # behaves like a set
```

How can we turn it into a list?

```
list(students.keys())
```

How can we get all values as a list?

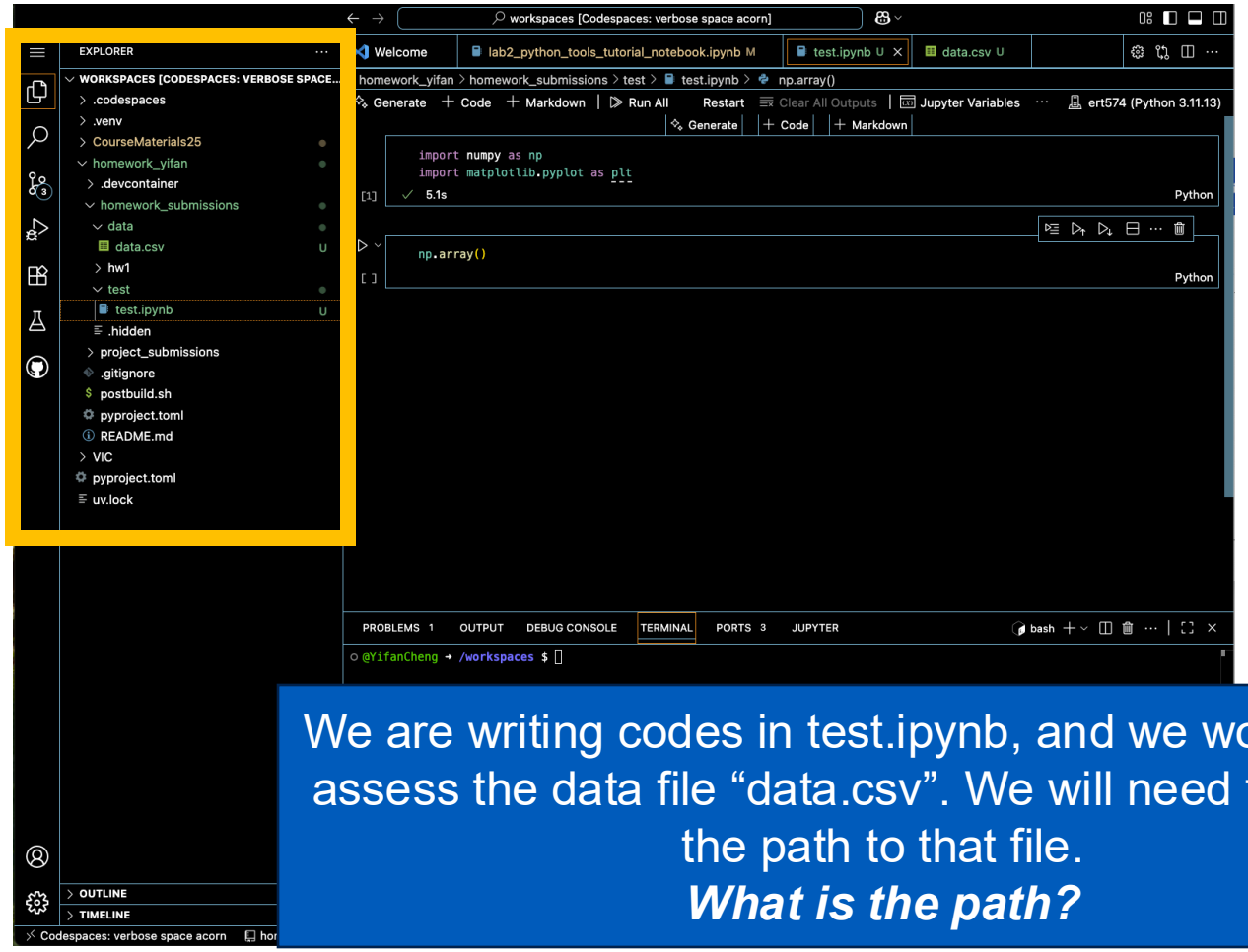
```
list(students.values())
```

Practice answer

```
# Step 2: Define a function to calculate the average score
def calculate_average(scores):
    total = sum(scores)
    count = len(scores)
    average = total / count
    return average
```

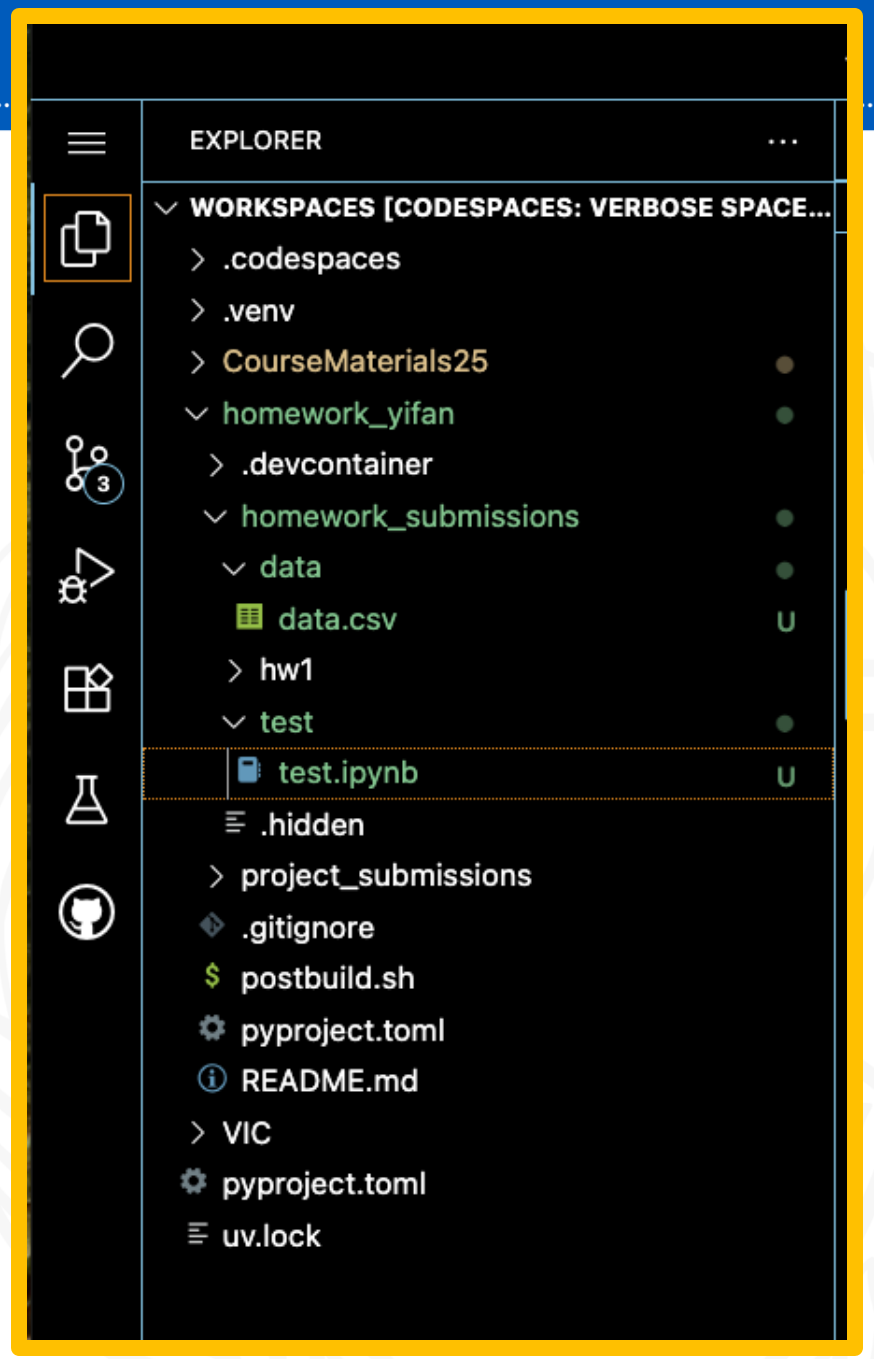
```
# Step 3: Use control flow to print who passed and who failed print("Results:")
for name, score in students.items():
    if score >= 60:
        print(f"{name} passed with a score of {score}.")
    else:
        print(f"{name} failed with a score of {score}.")


# Step 4: Print the average score
average_score = calculate_average(list(students.values()))
print(f"Average score: {average_score}")
```

File folder hierarchy



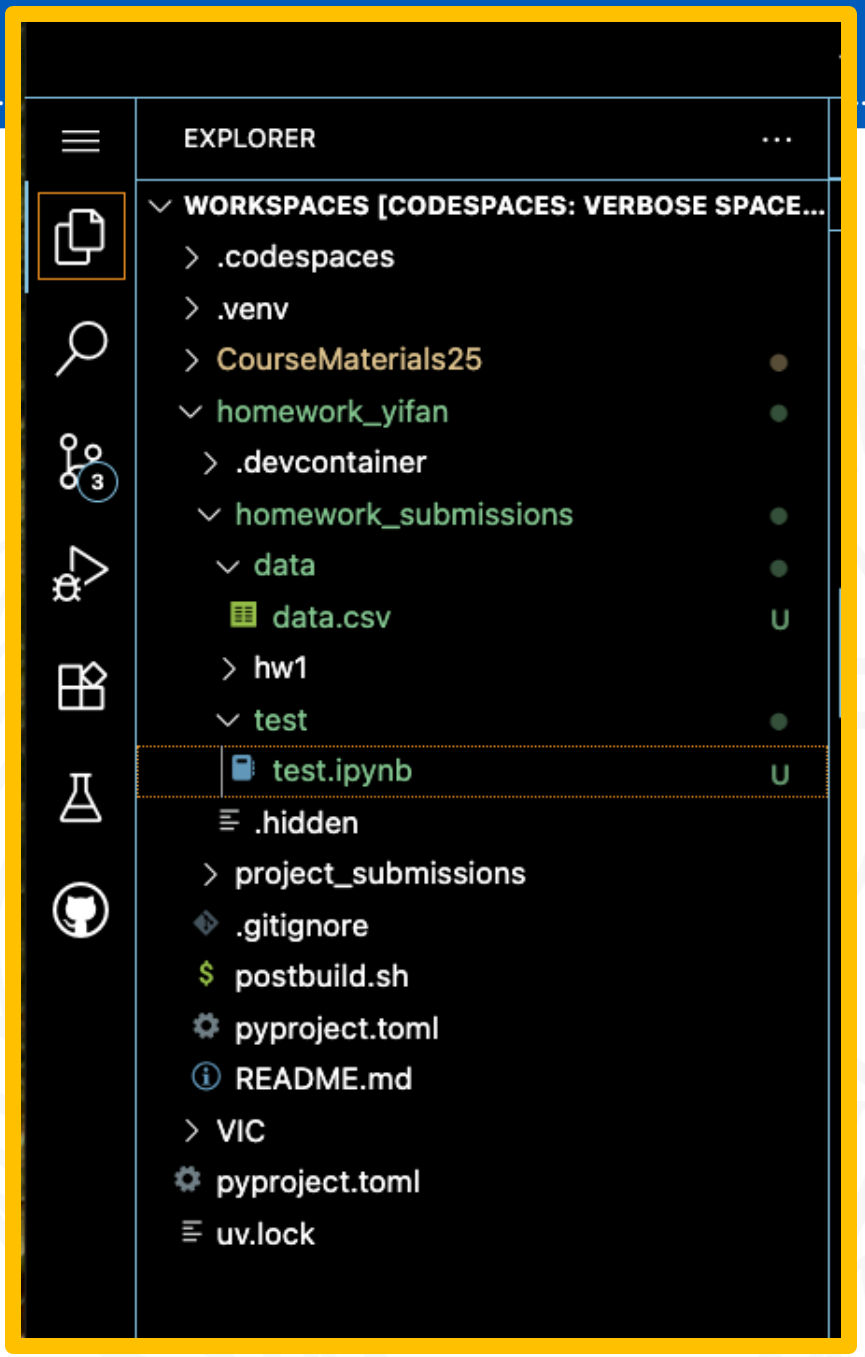
We are writing codes in test.ipynb, and we would like to assess the data file “data.csv”. We will need to provide the path to that file.

What is the path?



File folder hierarchy

- Absolute Path
  - Describes the **full path** from the root of the file system to the target file or folder.
  - Always starts from the **root directory** (e.g., **/** on Unix/Linux/macOS or a drive letter like C:\ on Windows).
- (Linux/MacOS) /workspaces/homework_yifan/homework_submissions/data/data.csv



We are writing codes in test.ipynb, and we would like to assess the data file “data.csv”. We will need to provide the path to that file.
What is the path?

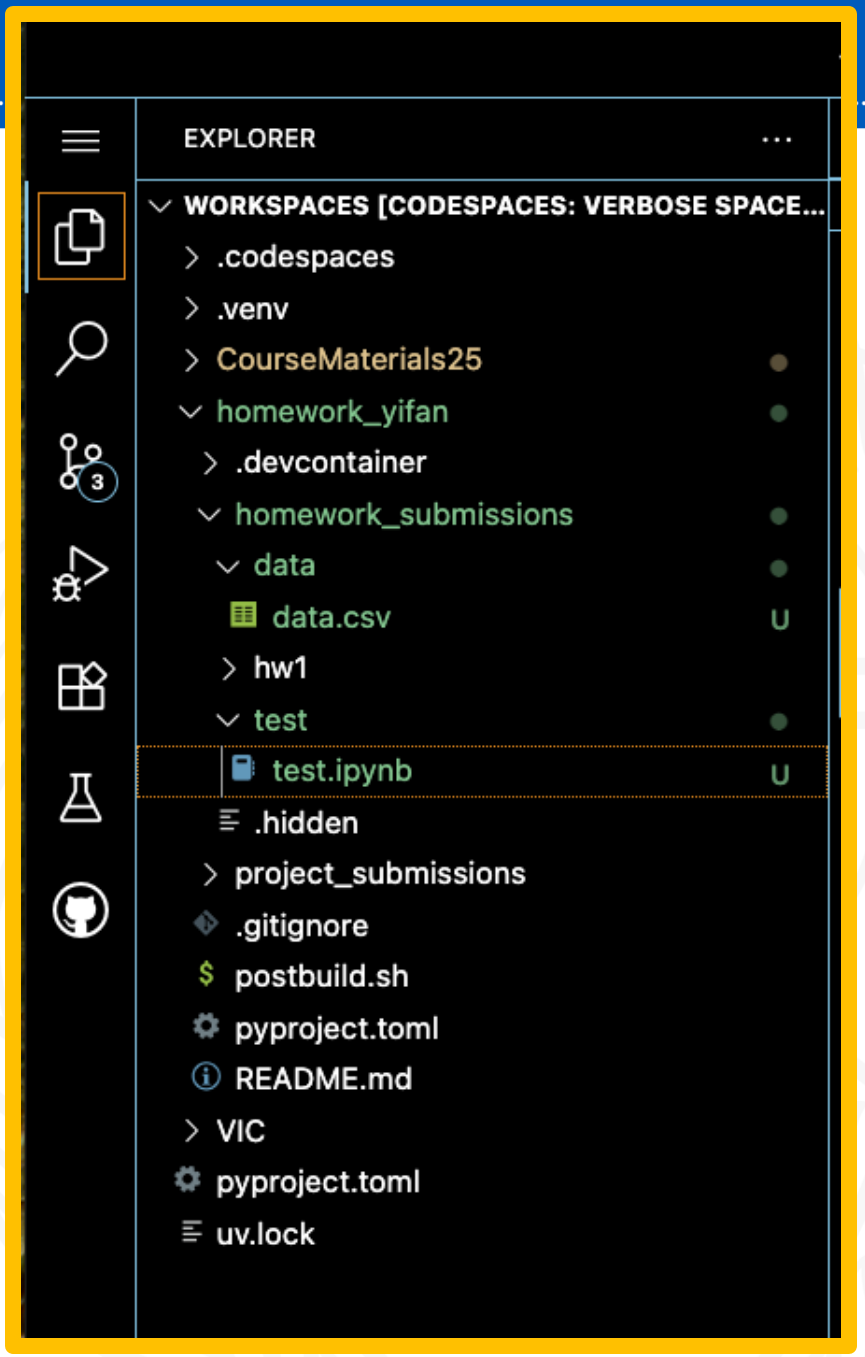
File folder hierarchy

- Relative Path
  - Describes the path **relative to the current working directory** (where your script or terminal is currently located).
  - Does **not** start with / or a drive letter.

- (Linux/MacOS) relative to **test.ipynb**

  ../data/data.csv

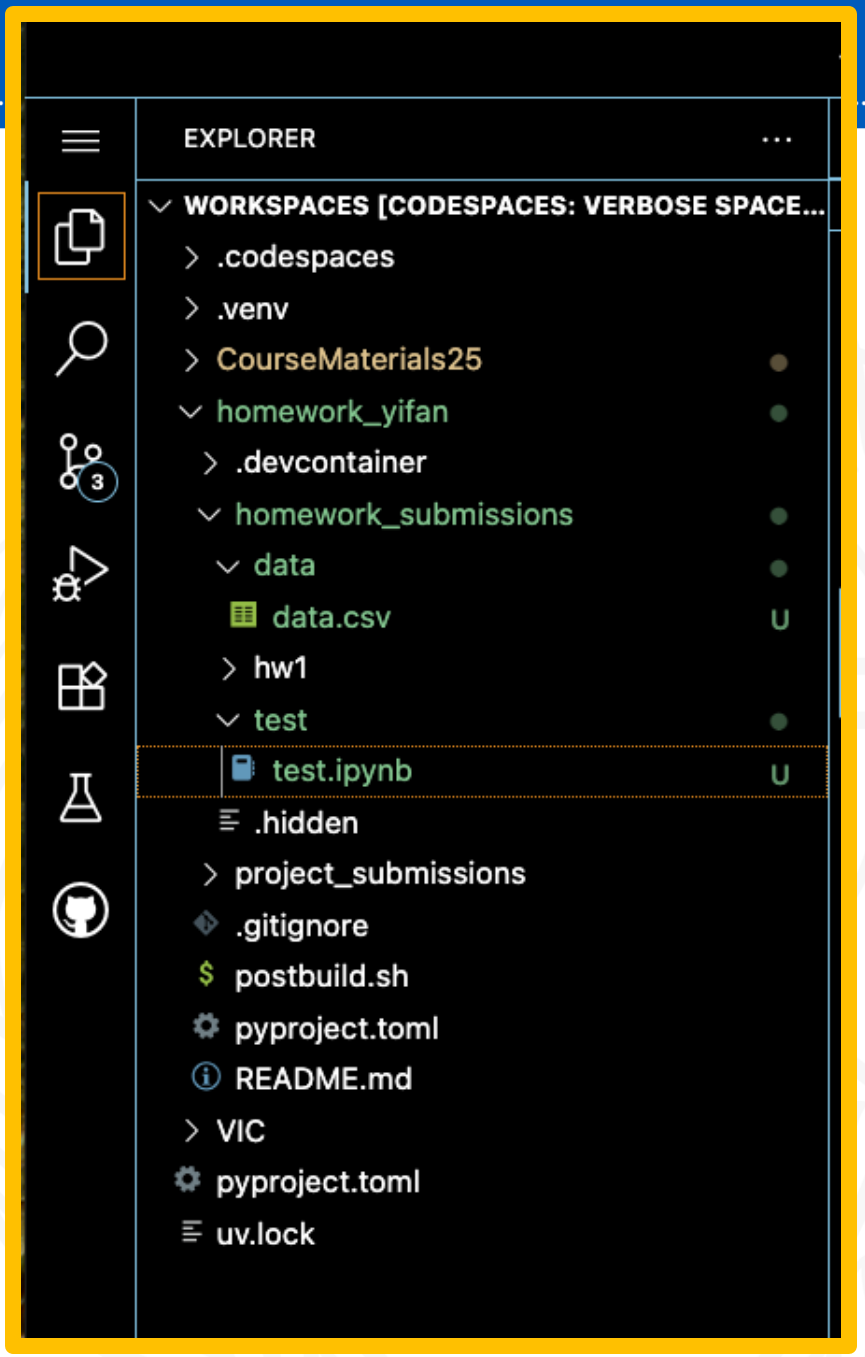
We are writing codes in test.ipynb, and we would like to assess the data file “data.csv”. We will need to provide the path to that file.
What is the path?



File folder hierarchy

- Relative Path
  - Describes the path **relative to the current working directory** (where your script or terminal is currently located).
  - Does **not** start with / or a drive letter.

```
./example.csv          # current directory
../example.csv         # parent directory
project/example.csv    # subdirectory
```



Practice

```
/home/student/HWERT574/
├── data/
│   ├── rainfall.csv
│   └── temperature.csv
├── scripts/
│   ├── analyze/
│   │   └── rainfall_analysis.py
│   └── summary.py
└── README.md
```

1. Identify Paths

What is the **absolute path** to temperature.csv?

What is the **relative path** from rainfall_analysis.py to:

rainfall.csv

temperature.csv

summary.py

2. Python Practice

In rainfall_analysis.py, write code to open both data files using **relative paths.**

- Syntax: `open(path)`

Python packages

- A **Python package** is a way of organizing related Python modules (files) into a directory hierarchy.
- It allows for **code reuse**, **modularity**, and **namespace management**.

Let's first talk about external packages!

Numpy

- Why Numpy?
  - A **high-performance** library for numerical computing.
  - Much faster and more memory-efficient than Python lists.
  - The foundation for many scientific libraries (e.g., Pandas, SciPy, Scikit-learn).

Load the numpy package

```
import numpy as np
```

Arrays vs Lists

```
import numpy as np
a = [1, 2, 3]
b = np.array([1, 2, 3])
```

Numpy

- Why Numpy?
  - A **high-performance** library for numerical computing.
  - Much faster and more memory-efficient than Python lists.
  - The foundation for many scientific libraries (e.g., Pandas, SciPy, Scikit-learn).

Load the numpy package

```
import numpy as np
```

Array Creation

```
# Creates a 2x3 array filled with zer
os
np.zeros((2, 3))

# Creates a 1D array of length 5 fill
ed with ones
np.ones(5)

# Creates an array with values from 0
 up to (but not including) 10, in ste
ps of 2 → [0, 2, 4, 6, 8]
np.arange(0, 10, 2)
```

Numpy

Lists don't support element-wise operations like arrays do.

Load the numpy package

```
import numpy as np
```

- Why Numpy?
  - A **high-performance** library for numerical computing.
  - Much faster and more memory-efficient than Python lists.
  - The foundation for many scientific libraries (e.g., Pandas, SciPy, Scikit-learn).

Array Operations

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b) # Output: [5 7 9]
print(a * 2) # Output: [2 4 6]
np.dot(a, b)
```

With Python Lists:

```
a = [1, 2, 3]
b = [4, 5, 6]
print(a + b) # Output: [1, 2, 3, 4, 5, 6] → concatenation, not addition
print(a * 2) # Output: [1, 2, 3, 1, 2, 3] → repetition, not multiplication
```

Numpy

- Why Numpy?
  - A **high-performance** library for numerical computing.
  - Much faster and more memory-efficient than Python lists.
  - The foundation for many scientific libraries (e.g., Pandas, SciPy, Scikit-learn).

Load the numpy package

```
import numpy as np
```

Shape and Reshape

```
a = np.array([[1, 2], [3, 4]])
a.shape
a.reshape((4,))
```

Matplotlib

- Matplotlib is the mostly widely used Python library for creating static, animated, and interactive visualizations. It's highly customizable and integrates well with NumPy and Pandas for data visualization.

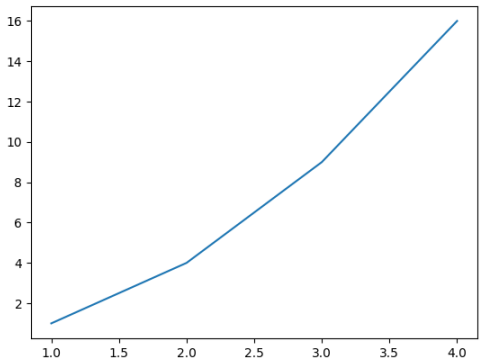
```
import matplotlib as mpl
```

The base **mpl** import is used for high level settings like setting a default figure or font size

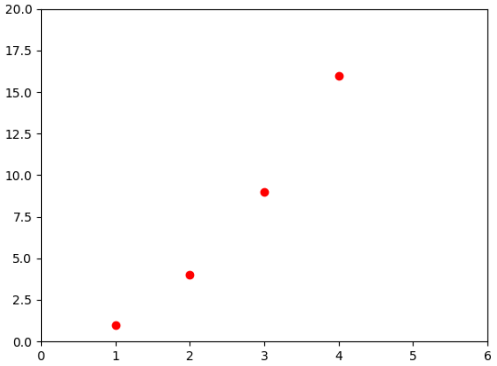
```
import matplotlib.pyplot as plt
```

The import of **plt** provides the main interface to the actual plotting functions

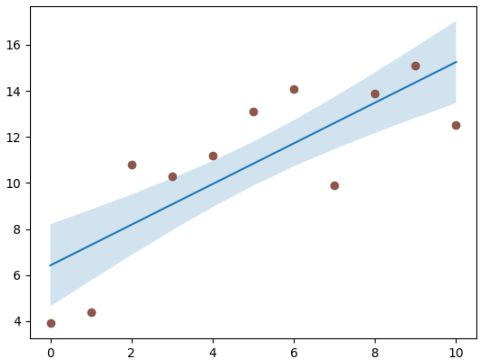
```
plt.plot
```



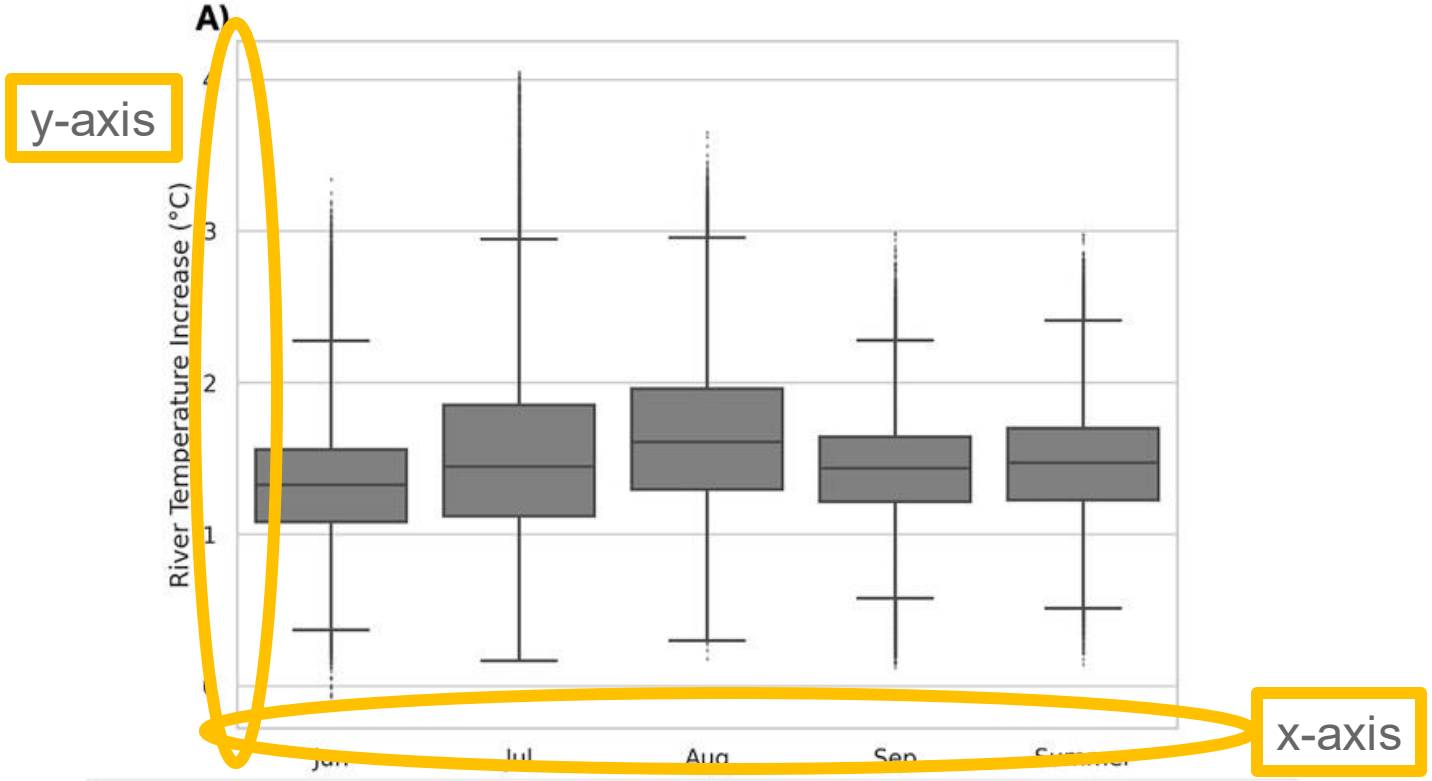

```
plt.scatter
```




```
plt.fill_between
```



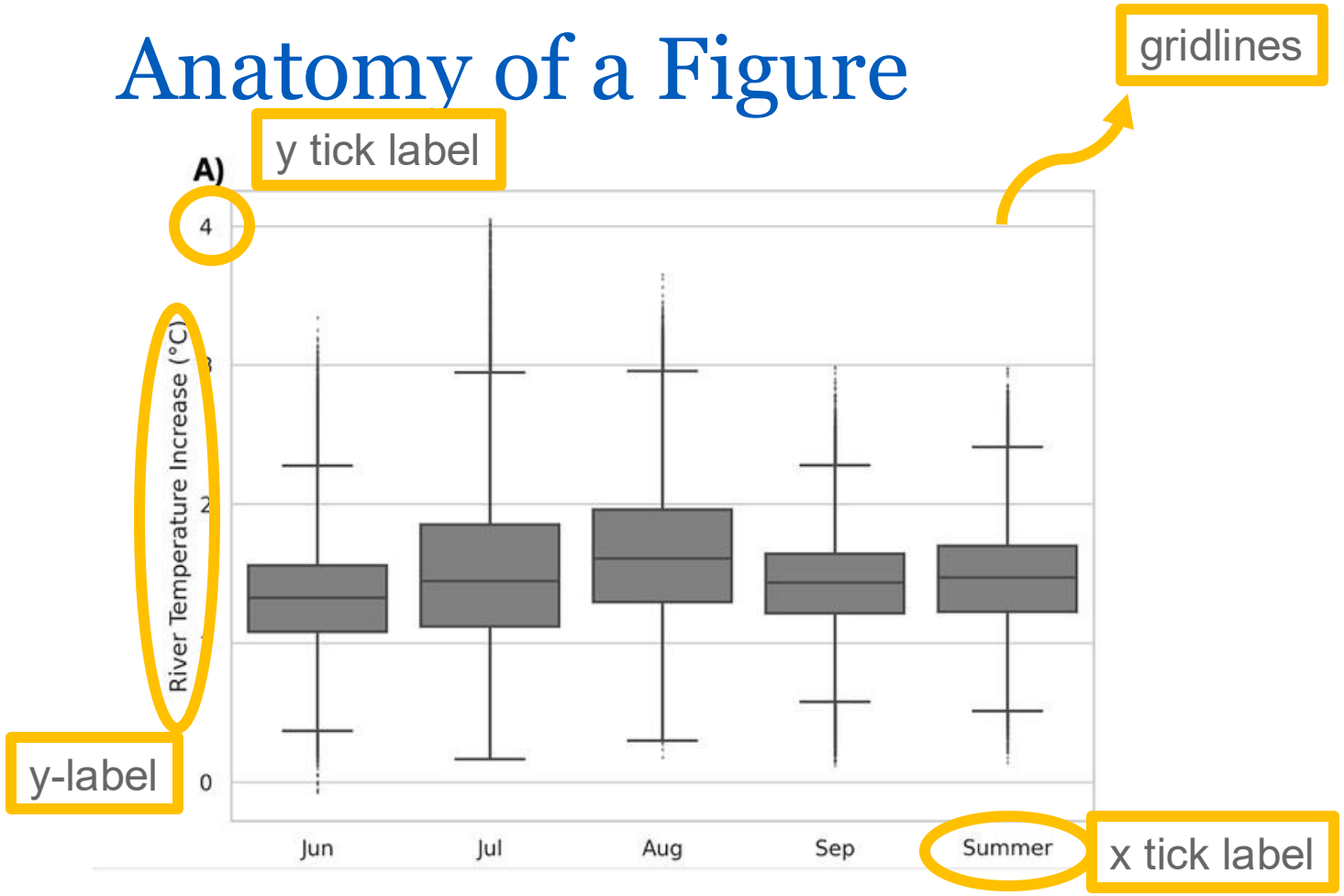
Anatomy of a Figure



Detailed anatomy can be found here:
https://realpython.com/python-matplotlib-guide/

Figure source:
https://journals.ametsoc.org/view/journals/hydr/26/5/JHM-D-24-0121.1.xml

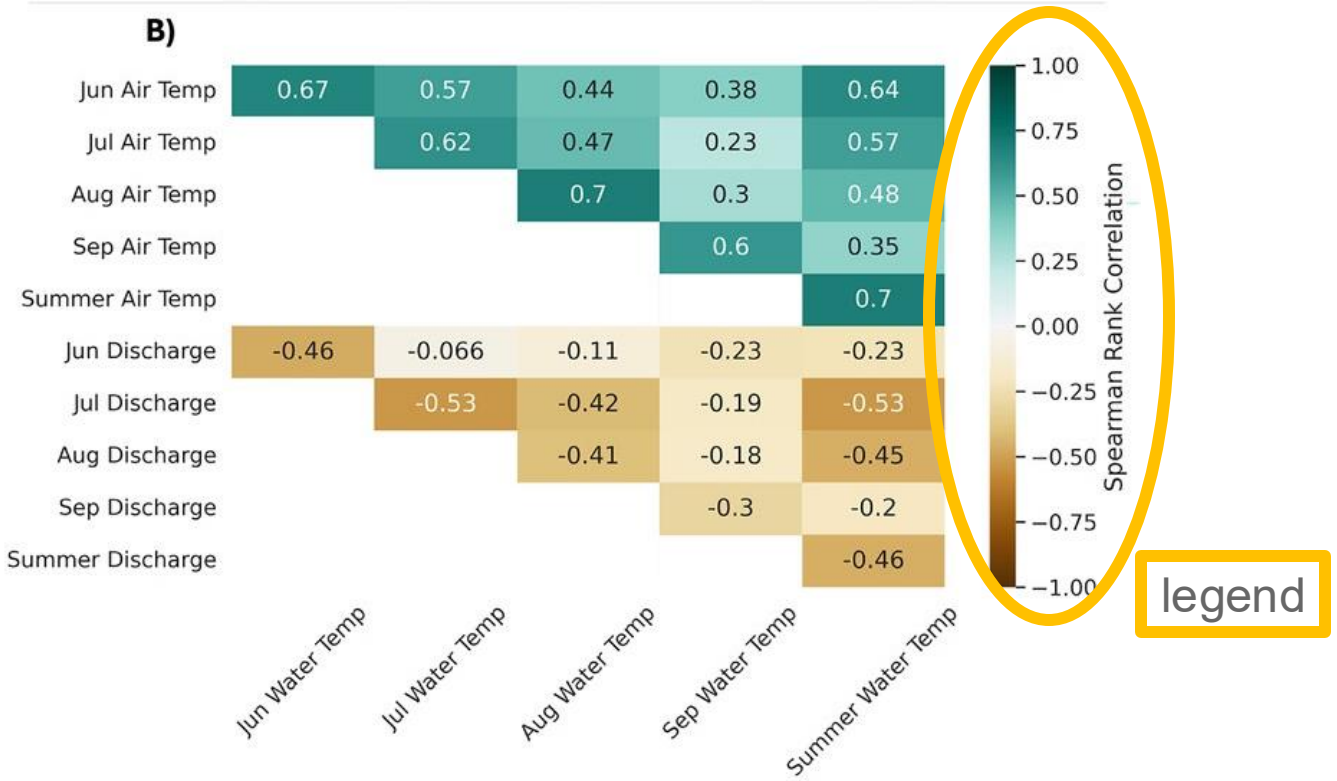
Anatomy of a Figure



Detailed anatomy can be found here:
https://realpython.com/python-matplotlib-guide/

Figure source:
https://journals.ametsoc.org/view/journals/hydr/26/5/JHM-D-24-0121.1.xml

Anatomy of a Figure



Detailed anatomy can be found here:
https://realpython.com/python-matplotlib-guide/

Figure source:
https://journals.ametsoc.org/view/journals/hydr/26/5/JHM-D-24-0121.1.xml

Self-define Package

- Step 1: Create the Package Directory
- Step 2: Add Functionality to Modules
- Step 3: Use the Package in a Script

Self-define Package

- **Step 1: Create the Package Directory**
- Step 2: Add Functionality to Modules
- Step 3: Use the Package in a Script

Create a folder named `mytools` with the following structure:

```
mytools/
├── __init__.py
├── math_utils.py
└── string_utils.py
```

Self-define Package

- Step 1: Create the Package Directory
- **Step 2: Add Functionality to Modules**
- Step 3: Use the Package in a Script

math_utils.py

```
def add(a, b):
    return a + b
def multiply(a, b):
    return a * b
```

string_utils.py

```
def capitalize_words(text):
    return ' '.join(word.capitalize() for word in text.split())
def count_vowels(text):
    return sum(1 for char in text.lower() if char in 'aeiou')
```

__init__.py

```
from .math_utils import add, multiply
from .string_utils import capitalize_words, count_vowels
__all__ = ['add', 'multiply', 'capitalize_words', 'count_vowels']
```

Self-define Package

- Step 1: Create the Package Directory
- Step 2: Add Functionality to Modules
- **Step 3: Use the Package in a Script**

Create a separate file called main.py in the same parent directory:

```
from mytools import add, capitalize_words

print(add(3, 5)) # Output: 8
print(capitalize_words("hello world")) # Output: Hello World
```

What if we leave __init__.py empty?

No Shortcut Imports

```
from mytools import add # ❌ won't work unless
add is exposed in __init__.py
```

However, we can always access functions through their module:

```
from mytools.math_utils import add # ✅ works
```

Questions?