0

Getting Started with Time Series Analysis

When embarking on a journey to learn coding in **Python**, you will often find yourself following instructions to install packages and import libraries, followed by a flow of a code-along stream. Yet an often-neglected part of any data analysis or data science process is ensuring that the right development environment is in place. Therefore, it is critical to have the proper foundation from the beginning to avoid any future hassles, such as cluttered setups or package conflicts, and dependency crises. Having the right environment setup will serve you in the long run when you complete your project, ensuring you are ready to package your deliverables in a reproducible and production-ready manner.

Once you master the fundamentals covered in this chapter, you will realize how much time you will save as you proceed with future chapters in this book. This knowledge will also serve you well in your professional career. It is this foundation that differentiates a seasoned developer and professional-grade work from the pack. Like any project, whether it is a machine learning project, a data visualization project, or a data engineering project, it all starts with planning and ensuring that all the required pieces are in place before you even begin with the core development.

In this chapter, you will learn how to set up a **Python virtual environment**, and we will introduce you to two common approaches for doing so. The steps will cover commonly used environment and package management tools. This chapter is designed to be hands-on so that you avoid too much jargon and can dive into creating your virtual environments in an iterative and fun way.

As we progress throughout this book, there will be several new Python libraries that you will need to install, specific to **time series analysis**, **time series visualization**, **machine learning**, and **deep learning** on time series data. It is advised that you don't skip this chapter, regardless of the temptation to do so, as it will help you establish the proper foundation for any code development that follows. By the end of this chapter, you will have mastered the necessary skills to create and manage your Python virtual environments using either `conda` or `venv`.

The following recipes will be covered in this chapter:

- Setting up your development environment
- Installing Python libraries
- Installing JupyterLab and JupyterLab extensions

Technical requirements

In this chapter, you will be primarily using the command line. For macOS and Linux, this will be the default Terminal (`bash` or `zsh`), while on Windows, you will use the **Anaconda Prompt**, which comes as part of the **Anaconda** or **Miniconda** installation. Installing Anaconda or Miniconda will be discussed in the following *Getting ready* section.

You can use **Visual Studio Code** (**VS Code**) as your primary **integrated development environment** (**IDE**), which is available for free at `https://code.visualstudio.com`. It will also be useful for later tasks such as creating YAML configuration files. VS Code supports Linux, Windows, and macOS.

Other valid alternative options that will allow you to follow along include the following:

- **Sublime Text** at `https://www.sublimetext.com`
- **Spyder** at `https://www.spyder-ide.org`
- **PyCharm Community Edition** at `https://www.jetbrains.com/pycharm/download/`
- **Jupyter Notebook** at `https://jupyter.org`

If you are using a machine that does not allow you to install software, or if you are using an older machine with limited capacity or performance, then do not worry. There are several alternatives that enable you to follow the recipes in this book and engage in hands-on practice. Here are some options:

- **Google Colab**: A cloud-based platform to write and run Python code in notebooks that already have some of the most popular data science packages preinstalled, such as TensorFlow, PyTorch, pandas, statsmodels, and scikit-learn. Colab also offers AI-powered features for code completion and natural language to code generation through **Gemini Code Assist**. Additionally, Colab lets you install additional packages directly in the notebook using `pip install` and provides options for using a CPU, GPU, or TPU at no cost. You can explore Colab by visiting `https://colab.research.google.com/`.
- **Kaggle Notebooks**: Similar to Colab, Kaggle provides hosted notebooks with many popular data science packages already preinstalled. It also allows you to use `pip install` to install any additional packages that are needed. For more information, please refer to `https://www.kaggle.com/docs/notebooks`.
- **Replit**: Offers a free, browser-based IDE that supports more than 50 programming languages, including Python. Replit includes AI-powered features such as code generation from natural language prompts, autocompletion, and debugging assistance to help you develop faster. To start coding, just create an account at `https://replit.com/`.

- **Binder**: An online open source platform that allows you to transform a Git repository into a collection of interactive notebooks. You can explore Binder by visiting `https://mybinder.org`.
- **Deepnote**: Similar to Colab, an online platform for writing, running, and collaborating on Python notebooks. Deepnote offers AI capabilities that automatically query, analyze, and interpret your data, making it easier for teams to work together on data science projects. Deepnote also integrates with various data sources and tools, and offers a free plan, which you can check out here: `https://deepnote.com`.

The source code for this chapter is available at `https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/code/Ch1`.

Setting up your development environment

As we dive into the various recipes provided in this book, you will be creating different Python virtual environments to install all your dependencies without impacting other Python projects.

You can think of virtual environments as isolated buckets or folders, each with a Python interpreter and associated libraries. The following diagram illustrates the concept behind isolated, self-contained virtual environments, each with a different Python interpreter and different versions of packages and libraries installed:

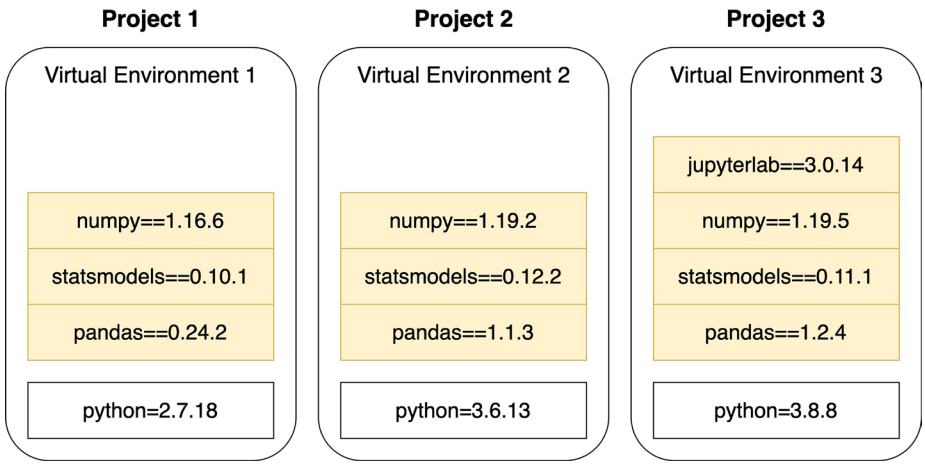

Figure 0.1: An example of three different Python virtual environments, one for each Python project

If you installed Anaconda, then these environments are typically stored and contained in separate folders inside the `envs` subfolder within the main Anaconda folder installation. As an example, on macOS, you can find the `envs` folder under `Users/<yourusername>/opt/anaconda3/envs/`. On Windows, it may look more like `C:\Users\<yourusername>\anaconda3\envs`. If you installed Miniconda, then the main folder will be `miniconda3` instead of `anaconda3`.

Each environment (folder) contains a **Python interpreter**, as specified during the creation of the environment, such as a Python 2.7.18 or Python 3.9 interpreter.

Generally speaking, upgrading your Python version or packages can lead to many undesired side effects if testing is not part of your strategy. A common practice is to replicate your current Python environment to perform the desired upgrades for testing purposes before deciding whether to move forward with the upgrades. This is the value that environment managers (`conda` or `venv`) and package managers (`conda` or `pip`) bring to your development and production deployment process.

Getting ready

In this section, it is assumed that you have the latest Python version installed by doing one of the following.

The recommended approach is to install through a Python distribution such as Anaconda (`https://www.anaconda.com/download`), which comes preloaded with all the essential packages and supports Windows, Linux, and macOS (including M1 support as of version 2022.05). Alternatively, you can install **Miniconda** (`https://docs.conda.io/en/latest/miniconda.html`) or **Miniforge** (`https://github.com/conda-forge/miniforge`).

Download an installer directly from the official Python site: `https://www.python.org/downloads/`.

If you are familiar with **Docker**, you can download the official Python image. You can visit Docker Hub to determine the desired image to pull, at `https://hub.docker.com/_/python`. Similarly, Anaconda and Miniconda can be used with Docker by following the official instructions here: `https://www.anaconda.com/docs/tools/working-with-conda/applications/docker`.

At the time of writing, the latest Python version that's available is Python 3.14.0.

Latest Python version supported in Anaconda

The latest version of Anaconda, based on the `2025.06` metapackage, ships with Python 3.13.5 as the default base interpreter. You can create a Python virtual environment with another version, such as Python 3.11.7, using `conda create`, provided that version is available in the repositories. You will learn how to do this later in this recipe.

The simplest and most efficient way to get you up and running quickly and smoothly is to go with a Python distribution such as Anaconda or Miniconda. For beginners, I recommend starting with Anaconda, as it includes many popular libraries and tools.

If you are a macOS or Linux user, once you have Anaconda installed, you are pretty much all set for using your default Terminal. To verify the installation, open your Terminal and type the following:

```
$ conda info
```

The following screenshot shows the standard output when running `conda info`, which outlines information regarding the installed `conda` environment. You should be interested in the listed versions for both `conda` and Python:

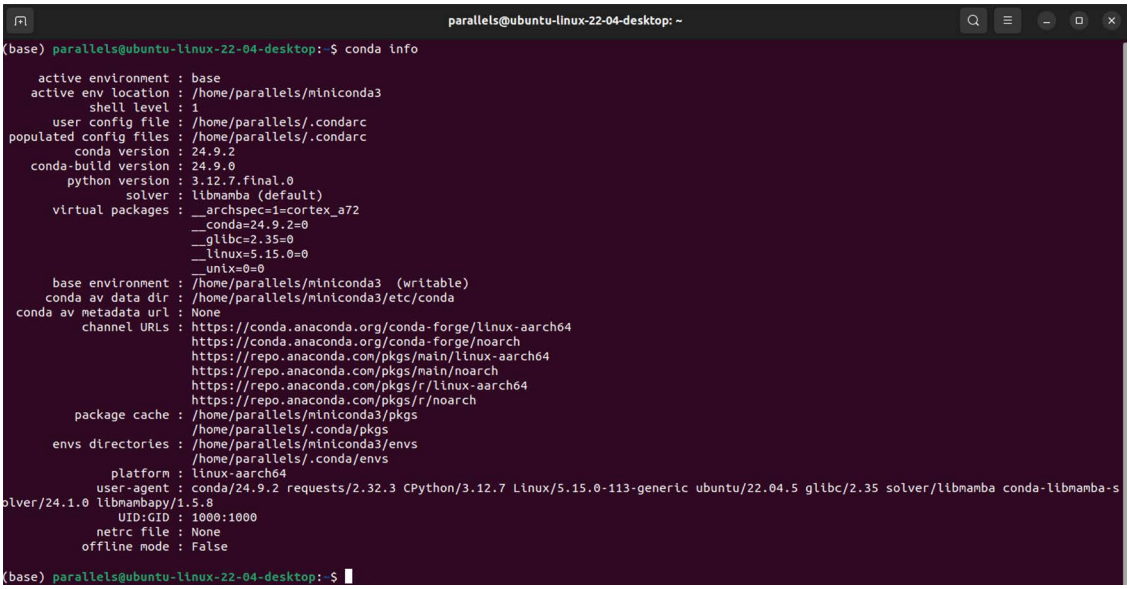

Figure 0.2: Verifying Conda's installation on Linux (Ubuntu) using Terminal

If you installed Anaconda on Windows, you need to use the Anaconda prompt. To launch it, you can type `Anaconda` in the Windows search bar and select one of the Anaconda prompts listed (**Anaconda Prompt** or **Anaconda PowerShell Prompt**). Once Anaconda Prompt has been launched, you can run the `conda info` command:

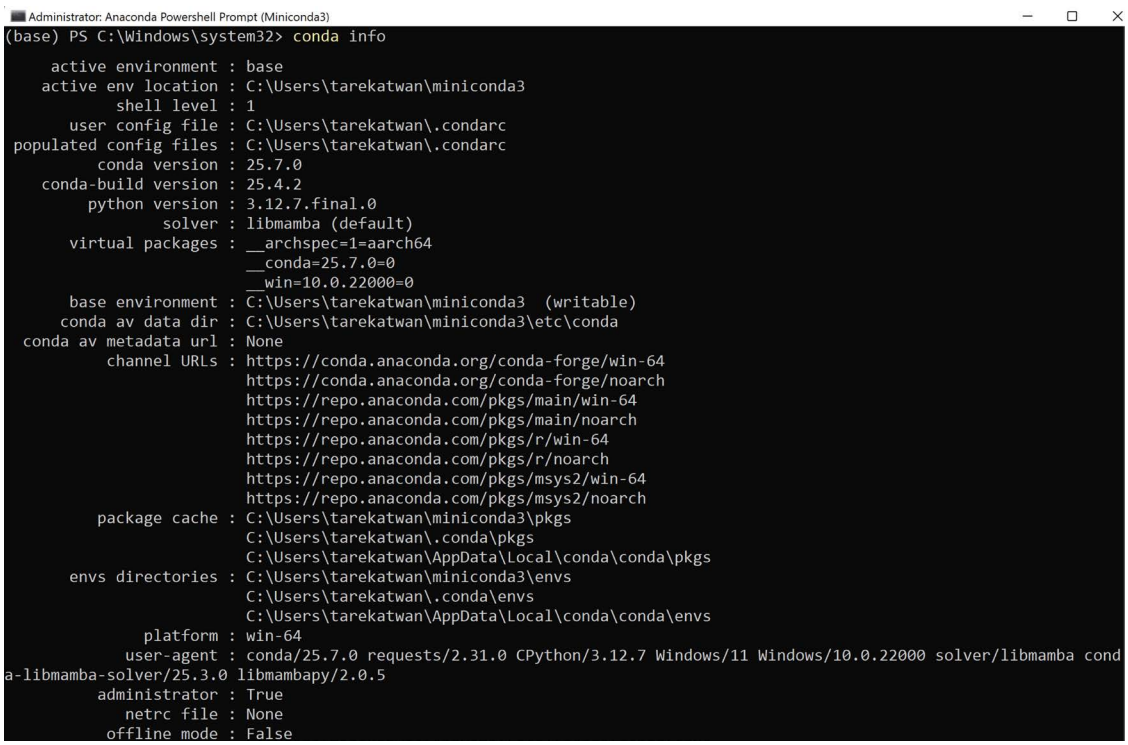

Figure 0.3: Verifying Conda's installation on Windows using Anaconda Prompt

How to do it...

In this recipe, I will cover two popular environment management tools. If you have Anaconda, Miniconda, or Miniforge installed, then `conda` should be your preferred choice since it provides both **package dependency management** and **environment management** for Python (and supports many other languages). On the other hand, the other option is using `venv`, which is a built-in Python module that provides environment management and requires no additional installation.

Both `conda` and `venv` allow you to create multiple virtual environments for your Python projects that may require different Python interpreters (for example, 3.4, 3.8, or 3.9) or different Python packages. In addition, you can create a sandbox virtual environment to experiment with new packages to understand how they work without affecting your base Python installation.

Creating a separate virtual environment for each project is a best practice taken by many developers and data science practitioners. Following this recommendation will serve you well in the long run, helping you avoid common issues when installing packages, such as package dependency conflicts.

Using Conda

Start by opening your Terminal (Anaconda Prompt for Windows):

1. First, let's ensure that you have the latest `conda` version. This can be done by using the following command:

   ```
   $ conda update conda -y
   ```

   The preceding code will update the `conda` package manager itself to the latest version. This is helpful if you are using an existing installation; it ensures you have the latest version. If everything is up to date, it will display the following:

   ```
   # All requested packages already installed.
   ```

2. If you already have Anaconda installed, you can update it to the latest version using the following command:

   ```
   $ conda update anaconda
   ```

   If everything is up to date, it will display the following:

   ```
   # All requested packages already installed.
   ```

3. Next, create a new virtual environment named `py310` with a specific Python version—in this case, Python 3.10:

   ```
   $ conda create -n py310 python=3.10
   ```

   Here, `-n` is a shortcut for `--name`.

   During the setup, `conda` may identify additional packages that need to be downloaded and installed. You will be prompted on whether you want to proceed or not. Type `y` and press *Enter* to proceed.

4. You can skip the confirmation message in the previous step by adding the `-y` option. Use this if you are confident in what you are doing and do not require the confirmation message, allowing `conda` to proceed immediately without prompting for a response. Update the command by adding the `-y` or `--yes` option, as shown in the following code:

```
$ conda create -n py310 python=3.10 -y
```

5. Once the setup is complete, you are ready to **activate** the new environment. Activating a Python environment updates the `$PATH` environment variable to point to the specified Python interpreter from the virtual environment (folder). You can confirm this by using the `echo` command:

```
$ echo $PATH
>> /Users/tarekatwan/opt/anaconda3/bin:/Users/tarekatwan/opt/anaconda3/
condabin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

   The preceding command works on Linux and macOS. If you are using the Windows Anaconda Prompt, use `echo %path%`, and on Anaconda PowerShell Prompt, use `echo $env:path`.

   Here, we can see that our `$PATH` variable is still pointing to the base `conda` environment and not our newly created virtual environment.

6. Now, activate your new `py310` environment and check the `$PATH` environment variable again. You will notice that it is now pointing to the `envs` folder—specifically, the `py310/bin` subfolder:

```
$ conda activate py310
$ echo $PATH
> /Users/tarekatwan/opt/anaconda3/envs/py310/bin:/Users/tarekatwan/opt/
anaconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

   Your actual path output will differ based on your operating system, username, and installation location.

7. Another way to confirm that our new virtual environment is the active environment is by running the following command:

```
$ conda info --envs
```

   The command will list all the `conda` environments that have been created. Notice that `py310` is listed with an `*` character, indicating that it is the active environment. Here is an example of the output:

```
# conda environments:
#

base                     /opt/anaconda3
py310                 *  /opt/anaconda3/envs/py310
```

8. Once you activate a specific environment, any package you install will only be available in that isolated environment. For example, let's install the pandas library and specify the version to install in the `py310` environment. At the time of writing, pandas 2.3.3 is the latest version:

```
$ conda install pandas=2.3.3
```

Notice that `conda` will prompt you for confirmation, displaying which additional packages will be downloaded and installed. Here, `conda` is checking for all the dependencies that pandas 2.2.3 requires and installs them. You can skip this confirmation step by adding the `-y` or `--yes` option at the end of the command.

Package version syntax differences

When specifying package versions, `conda` and `pip` use different syntax.

With `conda`, use a single equals sign (`conda install pandas=2.2.3`).

With `pip`, use a double equals sign (`pip install pandas==2.2.3`).

The message will also indicate the environment location where the installation will occur.

If you encounter `PackagesNotFoundError`, you may need to add the `conda-forge` channel to install the latest version of pandas (e.g., 2.3.3). You can accomplish this using the following command:

```
$ conda config --add channels conda-forge
```

`conda-forge` provides builds for different platforms and architectures and will automatically select the appropriate build for your system.

As an example, if you want to specify a `conda-forge` build for macOS ARM, then you can specify the build as shown in the following:

```
$ conda config --add channels conda-forge/osx-arm64
```

9. After pressing `y` and hitting *Enter*, `conda` will begin downloading and installing the specified packages.
10. When you are finished working in the `py310` environment, you can deactivate it and return to the base Python environment with the following command:

```
$ conda deactivate
```

11. If you no longer need the `py310` environment and wish to delete it, use the `env remove` command. The command will completely delete the environment and all the installed libraries. In other words, it will delete (remove) the entire folder for that environment:

```
$ conda env remove -n py310
```

Using venv

Once Python 3x is installed, you get access to the built-in `venv` module, which allows you to create virtual environments (similar to `conda`). Notice that when using `venv`, you will need to specify a path to where you want the virtual environment (folder) to be created. If no path is provided, the environment will be created in the current directory from which you ran the command. In the following example, we will create the virtual environment in the `Desktop` directory.

Follow these steps to create a new environment, install a package, and then delete the environment using `venv`:

1. First, decide where you want to place the new virtual environment and specify the path. In this example, I have navigated to the `Desktop` directory and run the following commands:

   ```
   $ cd Desktop
   $ python -m venv py310
   ```

   The preceding code will create a new `py310` folder in the `Desktop` directory. The `py310` folder contains several subdirectories, the Python interpreter, standard libraries, and other supporting files. The folder structure is similar to how `conda` creates its environment folders in the `envs` directory.

2. Next, activate the `py310` environment and examine the `$PATH` environment variable to verify that it is active. The following commands work for Linux and macOS (`bash` or `zsh`) and assume you are running the command from the `Desktop` directory:

   ```
   $ source py310/bin/activate
   $ echo $ PATH
   > /Users/tarekatwan/Desktop/py310/bin:/Users/tarekatwan/opt/anaconda3/
   bin:/Users/tarekatwan/opt/anaconda3/condabin:/usr/local/bin:/usr/bin:/
   bin:/usr/sbin:/sbin
   ```

   Here, we can see that the `py310` environment has been activated.

   On Windows, using Anaconda PowerShell Prompt, there is no `bin` subfolder, so you will need to activate the environment with the following command, assuming you are in the `Desktop` directory:

   ```
   $ .py310\Scripts\activate
   ```

   There are two activation files in the `Scripts` folder: `activate.bat` and `Activate.ps1`. The latter is intended for the Anaconda PowerShell Prompt, while `activate.bat` is for the Anaconda Windows Command Prompt. Generally, in PowerShell, if you omit the file extension, the correct script will be executed. However, while PowerShell can infer file extensions, it is generally safer and more explicit to specific the file extension—for example, specifying `Activate.ps1`, as shown:

   ```
   .\py310\Scripts\Activate.ps1
   ```

3. Now, check which Python version has been installed by using the following command:

```
$ python --version
> Python 3.10.10
```

4. Once you are done developing using the `py310` environment, you can deactivate it to return to the base Python environment using the `deactivate` command:

```
$ deactivate
```

5. If you no longer need the `py310` environment and wish to remove it, simply delete the entire `py310` folder, and that's it.

How it works...

Once a virtual environment is activated, you can validate the location of the active Python interpreter to confirm that you are using the correct one. Earlier, you saw how the `$PATH` environment variable changes once a virtual environment is activated. You can achieve similar results using the `which` command in Linux and macOS, `Get-Command` in Windows PowerShell, or the `where` command in Windows Command Prompt.

The following is an example you can run on either macOS or Linux:

```
$ which python
> /Users/tarekatwan/opt/anaconda3/envs/py310/bin/python
```

The following is an example on Windows (Windows Command Prompt or PowerShell):

```
$ where.exe python
```

Alternatively, you can use `Get-Command` in PowerShell, as shown:

```
$ Get-Command python
```

These commands will output the path to the active Python interpreter. The output of the preceding statements will show a different path, depending on whether the active environment was created with `conda` or `venv`. When activating a `conda` virtual environment, it will be inside the `envs` folder, as shown in the following on macOS:

```
/Users/tarekatwan/opt/anaconda3/envs/py310/bin/python
```

When activating a `venv` virtual environment, the path will match the location you provided during creation, as shown here on macOS:

```
/Users/tarekatwan/Desktop/py310/bin/python
```

Any additional packages or libraries that you install after you have activated a virtual environment will be isolated from other environments and reside in that environment's folder structure.

If we compare the folder structures of both `venv` and `conda`, you can see similarities, as shown in the following screenshot:

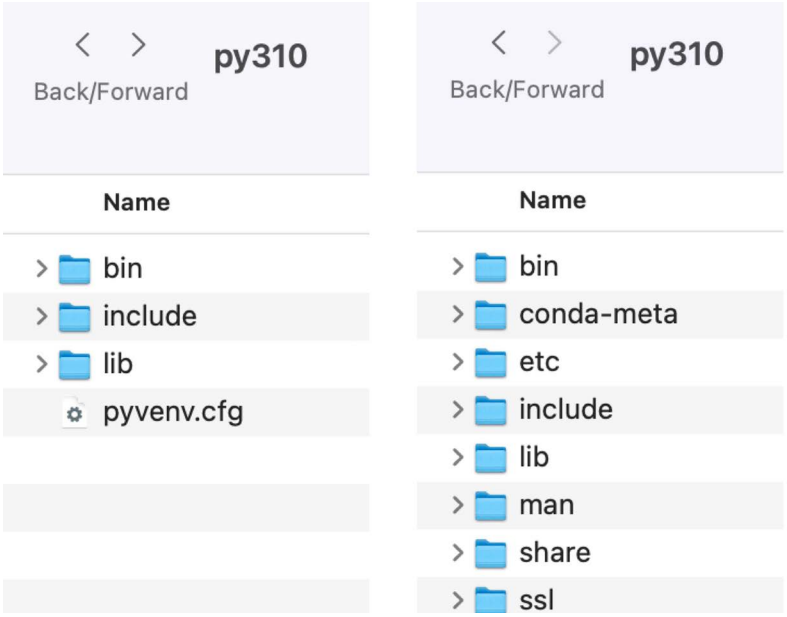

Figure 0.4: Comparing folder structures using conda and venv

Recall that when using `conda`, all environments will, by default, be created in the `/envs/` location inside the `anaconda3/` or `miniconda3/` directory. When using `venv`, you need to provide a path to specify where to create the environment directory; otherwise, it will default to the current directory where you ran the command.

You can create a `conda` environment in any directory of your choice by using the `-p` or `–prefix` option followed by your desired location path. Note, when using custom paths, you will need to manually provide the path whenever you want to activate that environment, for example, `conda activate /path/to/yourcustom/env`.

When using `venv`, you are typically limited to creating environments with the same Python version that you use to run the command. For example, if you run the command with Python 3.10, the virtual environment will use Python 3.10. This is in contrast to `conda`, which allows you to specify a different Python version, regardless of the base Python version installed. For example, if the current Python version for the base environment is 3.12.7, you can still create a 3.13.x environment using the following:

```
conda create -n py313 python=3.13.0 -y
```

This code will create a new `py313` environment with Python 3.13.0.

You can check which Python versions are available for installation with `conda` using the following `search` command:

```
conda search python
```

This will list all available Python versions and their associated builds. The following is an example output:

```
python                    3.13.0 h013362c_0_cp313t  conda-forge
python                    3.13.0 h206b6c5_100_cp313  conda-forge
python                    3.13.0 h4862095_100_cp313  pkgs/main
python                    3.13.0 h75c3a9f_100_cp313  conda-forge
python                    3.13.0 hf5be2b4_0_cp313t  conda-forge
```

An advantage of `conda` is that it provides two features: a package and dependency manager and a *virtual environment manager*. This means you can use the same `conda` environment to create additional environments using `conda create`, and install packages using `conda install <package name>`, which you will use in the next recipe, *Installing Python libraries*.

Keep in mind that when using `venv`, it only serves as a virtual environment manager, so you will still need to rely on `pip` as a package manager to install packages, such as `pip install <package name>`.

When using `conda` to install packages, it checks for any conflicts and will prompt you with recommendations, such as the need to upgrade, downgrade, or install additional package dependencies.

Finally, an added benefit of using `conda` is that you can create environments for other languages and not just Python. This includes Julia, R, Lua, Scala, Java, and more.

There's more...

In the preceding examples, you were able to create Python virtual environments from scratch using either `conda` or `venv`. The virtual environments you created may not contain the desired packages yet, so you will need to specifically install such packages for your project. You will explore how you can install packages in the upcoming recipe, *Installing Python libraries*.

There are other ways to create your virtual environment in Conda, which we will discuss here.

Creating a virtual environment using a YAML file

You can create a virtual environment from a **YAML** file. This option gives greater control in defining many aspects of the environment, including all the packages that should be installed in one step.

You can create a YAML file in VS Code. Here is an example of a YAML file (`environment.yml`) that creates a `conda` environment labeled `tscookbook` using Python 3.10:

```
# A YAML for creating a conda environment
# file: environment.yml
# example creating an environment named tscookbook

name: tscookbook
channels:
  - conda-forge
  - defaults
```

```
dependencies:
  - python=3.10
  - pip
  # Data Analysis
  - statsmodels
  - scipy
  - pandas
  - numpy
  - tqdm

  # Plotting
  - matplotlib
  - seaborn

  # Machine Learning
  - scikit-learn
  - jupyterlab
```

To create your virtual environment using the `environment.yml` file, you can use `conda env create` with the `-f` or `--file` option, like so:

```
$ conda env create -f environment.yml
```

Once this process is completed, you can activate the environment:

```
$ conda activate tscookbook
```

You can also bootstrap your YAML file from an existing environment. This is very useful if you want to share your environment configurations with others or create a backup for later use. The following three commands will achieve the same results of exporting the `py310` conda environment to a YAML file named `environment.yml`, using slightly different syntax options:

```
$ conda env export -n py310 > environment.yml
$ conda env export -n py310 -f environment.yml
$ conda env export --name py310 --file environment.yml
```

This will generate the `environment.yml` file for you in the current directory.

Cloning a virtual environment from another environment

Cloning is helpful if you want to experiment with new packages or upgrade existing ones without risking changes that may break existing code in a current project. Using the `--clone` option, you can create a copy of your environment in one step. This achieves the same result as exporting an environment to a YAML file and then creating a new environment based on that file. The following example will clone the `py310` Conda environment to a new environment named `py310_clone`:

```
$ conda create --name py310_clone --clone py310
```

Generally, cloning is a great option for quick duplication of environments on the same machine. If you want portability across machines or platforms, then exporting/importing through a YAML file is a better choice.

See also

It is worth mentioning that Anaconda comes with another tool called `anaconda-project` for packaging your `conda` project artifacts and creating a YAML file to ensure reproducibility. This is ideal for creating, sharing, and ensuring reproducibility for your data science projects. Think of this as an alternative approach to manually developing your YAML.

For a list of arguments, type the following in your Terminal:

```
$ anaconda-project --help
```

If you did not install the full Anaconda Python distribution and, for example, opted for Miniconda, then you will need to install `anaconda-project` manually:

```
$ conda install anaconda-project
```

For more information on Anaconda Project, you can reference the official GitHub repository (`https://github.com/Anaconda-Platform/anaconda-project`) and the official documentation (`https://anaconda-project.readthedocs.io/en/latest/index.html`).

While `conda` and `venv` are the focus of this recipe, it is worth mentioning other popular alternatives, such as **Mamba**, a drop-in replacement for Conda that offers a fast dependency resolution. Mamba uses identical syntax to Conda, which makes the transition much easier. You can learn more about Mamba here: `https://mamba.readthedocs.io/en/latest/`. Another noteworthy alternative is **Poetry**, a modern Python dependency and packaging tool that you can read more about here: `https://python-poetry.org`.

Installing Python libraries

In the previous recipe, you were introduced to the YAML environment configuration file, which allows you to create a Python virtual environment with all the necessary packages in one step using a single command:

```
$ conda env create -f environment.yml
```

Throughout this book, you will be installing various Python libraries specific to time series analysis, such as **statsmodels** (for statistical models), **PyOD** (for outlier detection), or **sktime** (for scikit-learn compatible time series tools). Understanding these package installation methods will ensure that you can properly set up environments for each recipe.

There are various methods for installing Python libraries, which you will explore in this recipe.

Getting ready

In this recipe, you will create and use different files, including `requirements.txt`, `environment_history.yml`, and other files. These files are available to download from the GitHub repository for this book: `https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/code/Ch1`.

In this recipe, you will learn how to generate a `requirements.txt` file and install libraries in bulk.

How to do it...

The easiest way to install a collection of libraries at once is by using a `requirements.txt` file.

In a nutshell, the `requirements.txt` file lists the Python libraries and their associated versions that you want to install. You can create your `requirements.txt` file manually or export it from an existing Python environment.

The file does not need to be named `requirements.txt`; it is more of a naming convention and a very common one embraced by the Python community. Some tools, such as PyCharm, will autodetect the `requirements.txt` file if it is placed in the root project directory.

Using any text editor of choice, create a `requirements.txt` file with the following:

```
pandas==2.2.3
matplotlib==3.9.2
statsmodels==0.14.4
```

Using Conda

With Conda, you have different options for installing our packages in bulk. You can either create a new environment and install all the packages listed in a `requirements.txt` file at once (using the `conda create` command), or install the Python packages to an existing environment (using the `conda install` command):

- **Option 1**: Create a new `conda` environment and install the libraries in one step. For example, you can create a new environment for each chapter and use a `requirements.txt` file. The following example will create a new environment named `ch1` and install all the packages listed in the `requirements.txt` file:

  ```
  $ conda create -n ch1 --file requirements.txt -y
  ```

- **Option 2**: Install the necessary libraries into an existing `conda` environment. In this example, you have an existing environment named `timeseries`. You will need to activate the environment, then install the libraries from the `requirements.txt` file:

  ```
  $ conda create -n timeseries -y
  $ conda activate timeseries
  $ conda install --file requirements.txt -y
  ```

Using venv and pip

Since `venv` is just an environment manager, you will need to use `pip` as your package manager. Start by using `venv` to create a new environment, then use `pip` to install the packages. The following examples create the environment folder on the desktop, but you can choose any location you prefer. Keep in mind the command examples are being executed from the directory where the `requirements.txt` file is located:

- **On MacOS/Linux**: Create and activate the `venv` environment, then install the packages:

```
$ python -m venv ~/Desktop/timeseries
$ source ~/Desktop/timeseries/bin/activate
$ pip install -r requirements.txt
```

- **On Windows**: Create and activate the `venv` environment, then install the packages:

```
$ python -m venv C:\Users\<your username>\Desktop\timeseries
$ C:\Users\<your username>\Desktop\timeseries\Scripts\activate
$ pip install -r requirements.txt
```

Notice that in the preceding Windows code, the `activate` file extension was not specified (either `.bat` or `.ps1`). This is valid and will work on either Windows Prompt or PowerShell.

While we've been using `requirements.txt` for both `pip` and Conda installations, there are differences in how each package manager handles this file format. `pip` expects entries in the form of `package_name==version` (e.g., `numpy==1.21.2`), while Conda uses `package_name=version=build` (e.g., `numpy=1.21.2=py39hdbf815f_0`). Conda can accept the `pip` format directly, but `pip` cannot process Conda's format.

In the next section, you'll learn how to create these files automatically using a bootstrapping technique, ensuring compatibility with your package manager of choice.

How it works...

In the preceding code, the `requirements.txt` file was created manually to install the necessary libraries. There are two approaches to generating a `requirements.txt` file: the manual approach, as we did earlier, or using a **bootstrapping** method.

The file format is straightforward (given that Conda can accept the `pip` format). If you have a small list of libraries, you can create a `requirements.txt` file using any text editor, such as VS Code, and simply list the packages you want to install. If you do not specify the package version, the latest available version will be considered for installation. See the following example for a new `simple.txt` file:

```
pandas
matplotlib
```

Let's test this using `venv` and `pip`. I am running the following command on macOS:

```
$ python -m venv ~/Desktop/ch1
$ source ~/Desktop/ch1/bin/activate
$ pip install -r simple.txt
$ pip list

Package             Version
------------------- -----------
contourpy           1.3.0
cycler              0.12.1
fonttools           4.54.1
importlib_resources 6.4.5
kiwisolver          1.4.7
matplotlib          3.9.2
numpy               2.0.2
packaging           24.2
pandas              2.2.3
pillow              11.0.0
pip                 23.0.1
pyparsing           3.2.0
python-dateutil     2.9.0.post0
pytz                2024.2
setuptools          58.1.0
six                 1.16.0
tzdata              2024.2
zipp                3.21.0

$ deactivate
```

Since we did not specify the package version, the latest versions were installed for both `pandas` and `matplotlib`. But what are those additional packages? These are dependencies required by `pandas` and `matplotlib` that `pip` identified and automatically installed for us.

Now, let's use the same `simple.txt` file, but using `conda` this time:

```
$ conda create -n ch1 --file simple.txt python=3.12 -y
```

Once the installation is completed, you can activate the environment and list the packages that were installed:

```
$ conda activate ch1
$ conda list
```

You may notice that the list is pretty large. There are more packages installed compared to the `pip` approach. To get a count of the libraries installed, use the following command:

```
$ conda list | wc -l
> 68
```

The count may vary depending on your system and the versions being installed.

The larger number of packages installed by Conda compared to `pip` is due to Conda managing more than just Python packages—it also installs **non-Python dependencies** (such as C libraries) that may be required by the packages you're installing. For example, some libraries, such as `libblas`, are non-Python dependencies that Conda may include to ensure smooth functionality.

There are a few things to keep in mind here:

- Conda installs packages from the Anaconda repository as well as from the Anaconda Cloud
- `pip` installs packages from the **Python Package Index** (**PyPI**) repository
- Conda performs a thorough analysis of all the packages before installation and handles version conflicts more effectively than `pip`

Bootstrapping a file

An alternative option is to generate the `requirements.txt` file from an existing environment. This is very useful when you want to recreate environments for future use or share your list of packages and dependencies with others to ensure reproducibility and consistency. For example, if you have worked on a project and installed specific libraries, you can share your code along with the required libraries to help other users set up the same environment. This is where generating a `requirements.txt` file comes in handy. Similarly, you can export a YAML environment configuration file, as demonstrated earlier in the chapter.

Let's see how this can be done in both `pip` and `conda`. Keep in mind that both methods will export the list of currently installed packages along with their versions.

venv and pip freeze

The `pip freeze` command allows you to export all `pip`-installed libraries in your environment. First, activate the `ch1` environment you created earlier with `venv`, then export the list of packages to a `requirements.txt` file. The following example is on macOS using Terminal:

```
$ source ch1/bin/activate
$ pip freeze > requirements.txt
$ cat requirements.txt
>>
contourpy==1.3.0
cycler==0.12.1
fonttools==4.54.1
importlib_resources==6.4.5
kiwisolver==1.4.7
```

```
matplotlib==3.9.2
numpy==2.0.2
packaging==24.2
pandas==2.2.3
...
```

Once done, you can run the `deactivate` command.

Conda

To export a list of packages in a Conda environment, activate the environment (in this example, the `ch1` Conda environment) and export the list of packages:

```
$ conda activate ch1
$ conda list -e > conda_requirements.txt
$ cat conda_requirements.txt
>>
 # This file may be used to create an environment using:
# $ conda create --name <env> --file <this file>
# platform: osx-arm64
# created-by: conda 24.9.2
brotli=1.1.0=hd74edd7_2
brotli-bin=1.1.0=hd74edd7_2
bzip2=1.0.8=h99b78c6_7
ca-certificates=2024.8.30=hf0a4a13_0
certifi=2024.8.30=pyhd8ed1ab_0
contourpy=1.3.1=py313h0ebd0e5_0
cycler=0.12.1=pyhd8ed1ab_0
fonttools=4.54.1=py313heb2b014_1
freetype=2.12.1=hadb7bae_2
kiwisolver=1.4.7=py313hf9c7212_0
...
```

Once done, you can run `conda deactivate`.

Notice the different format used. `pip` uses a simple format (`package_name==version`), while Conda uses a more detailed format (`package_name=version=build`). As mentioned earlier, Conda can interpret the `pip` file format, but `pip` cannot read the Conda format. Conda also exports non-Python packages, resulting in a longer list of dependencies in the exported file.

There's more...

When you exported the list of packages installed with `conda`, the `conda_requirements.txt` file contained a large list of packages. If you want to export only the packages that you explicitly installed (without the additional packages that `conda` added), then you can use the `conda env export` command with the `--from-history` flag:

```
$ conda activate ch1
$ conda env export --from-history > environment.yml
$ cat environment.yml
>>
name: ch1
channels:
  - conda-forge/osx-arm64
  - defaults
dependencies:
  - matplotlib==3.9.2
  - pandas==2.2.3
  - statsmodels==0.14.4
prefix: /opt/anaconda3/envs/ch1
```

Note that you do not need to activate that environment first, as we have been doing so far. Instead, you can add the `-n` or `--name` option to specify the name of the environment. Otherwise, it will default to the currently active environment. Here is what the modified command would look like (all of the following are equivalent):

```
conda env export -n ch1 --from-history > environment.yml
conda env export --name ch1 --from-history > environment.yml
conda env export -n ch1 --from-history -f environment.yml
conda env export --name ch1 --from-history --file environment.yml
```

See also

To find a list of all the available packages from Anaconda, you can visit `https://anaconda.org/anaconda/repo`.

To search for packages in the PyPI repository, you can visit `https://pypi.org/`.

Installing JupyterLab and JupyterLab extensions

Throughout this book, you can follow along using your favorite Python IDE (for example, PyCharm or Spyder) or text editor (for example, VS Code). Alternatively, you can use an interactive, web-based approach with **Jupyter Notebook** or **JupyterLab**, which are particularly preferred choices for learning, experimenting, and following along with the recipes in this book. Notebooks allow you to view code execution results directly within the interface, thus enhancing the learning experience.

While any platform can be used, JupyterLab will be the primary tool used throughout this book, and the provided code files on the GitHub repository are formatted for JupyterLab (the `.ipynb` extension). JupyterLab also allows for easy installation of helpful extensions to support additional features.

Interestingly, the name "Jupyter" is derived from the three programming languages: Julia, Python, and R. Other options for notebook-style environments are suggested in the *Technical requirements* section, including **Google Colab** and **Kaggle Notebooks**. If you are new to Jupyter Notebook or JupyterLab, you can find more information here: `https://jupyter.org/`.

In this recipe, you will install Jupyter Notebook, JupyterLab, and additional JupyterLab extensions. You will also learn how to install individual packages, in addition to the bulk approach covered in earlier recipes. This can be useful when you only have a handful of specific libraries you want to install.

Using Conda in future examples

Moving forward, when a new environment is created, the code uses the `conda` commands. Previous recipes covered two approaches to creating virtual environments (`venv` vs. `conda`) and installing packages (`pip` vs. `conda`), so you can proceed with whichever setup you prefer.

Getting ready

We will create a new environment and install the main packages needed for this chapter, primarily `pandas`:

```
$ conda create -n timeseries python=3.12 pandas -y
```

This code creates a new Python 3.12 environment named `timeseries`. The last portion of the statement lists the individual packages that you will be installing. If the list of packages is large, you should use a `requirements.txt` file instead. If there are a handful of packages, then they can be listed individually, separated by spaces, as follows:

```
$ conda create -n timeseries python=3.12 pandas matplotlib statsmodels -y
```

Once the environment has been created and the packages have been installed, go ahead and activate it:

```
$ conda activate timeseries
```

How to do it...

Now that we have created our environment and activated it, let's install Jupyter:

1. With the environment activated, use `conda install` to install additional packages that were not included in `conda create`:

   ```
   $ conda install -c conda-forge jupyterlab -y
   ```

2. To launch your JupyterLab instance, type the following command:

   ```
   $ jupyter lab
   ```

The command starts a local web server and automatically launches the JupyterLab interface in your default browser, pointing to `localhost:8888/lab`. The following is an example output that JuptyerLab provides on how you can access the server:

```
To access the server, open this file in a browser:
        file:///Users/tarekatwan/Library/Jupyter/runtime/jpserver-92368-
open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/
lab?token=bbd7fb73e59304fb36eb86b31a9a402ba9a93781971adeab
        http://127.0.0.1:8888/
lab?token=bbd7fb73e59304fb36eb86b31a9a402ba9a93781971adeab
```

3. To terminate the web server, press *Ctrl + C* twice on your terminal or click **Shut Down** from the **File** menu in the JupyterLab interface, as shown in the following screenshot:

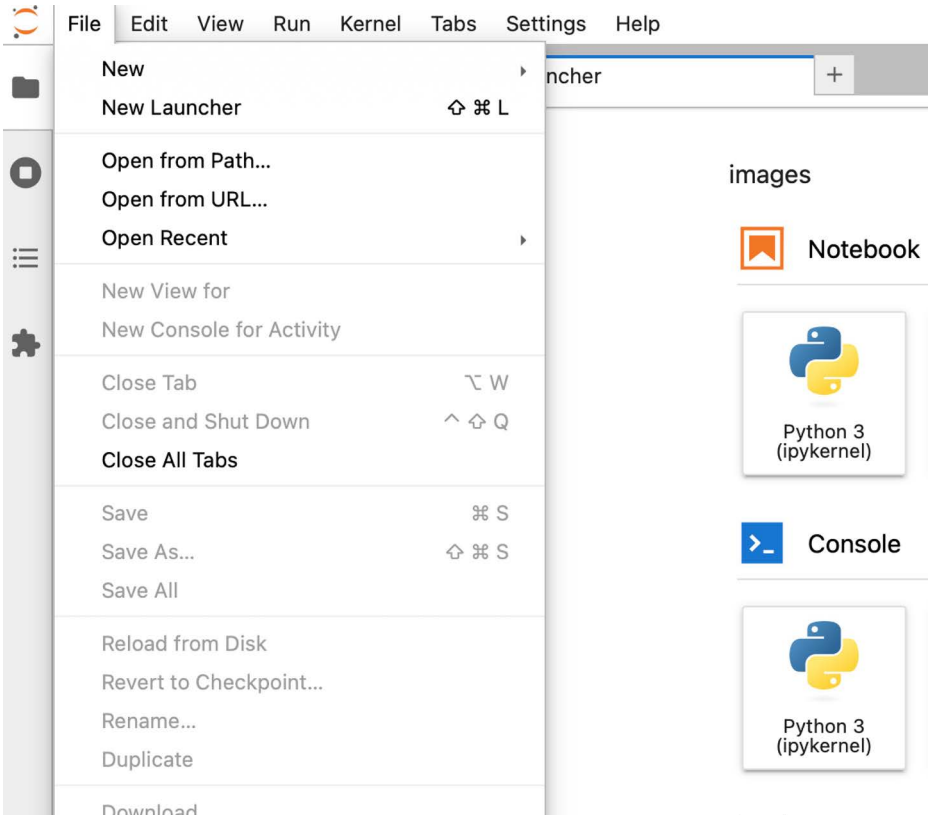

Figure 0.5: Shutting down the JupyterLab server

4. You can then safely close your browser.
5. Notice that in the preceding example, when JupyterLab was initiated, it launched in your default browser. If you wish to use a different browser, you can use the following command:

```
$ jupyter lab --browser=chrome
```

In this example, we specify Chrome instead of Safari, which is the default on macOS. You can replace `chrome` with your preferred browser, such as Firefox, Opera, and so on.

If the preceding command does not launch Chrome automatically, you will need to register the browser with JupyterLab using `webbrowser.register()`. To do so, start by generating the JupyterLab configuration file with the following command:

```
$ jupyter lab --generate-config
```

This will create a `jupyter_lab_config.py` file in the `.jupyter` folder. The file path will vary by system, as in these examples:

- **Windows**: `C:\Users\<yourusername>\.jupyter\jupyter_lab_config.py`
- **Linux**: `/home/<username>/.jupyter/jupyter_lab_config.py`
- **macOS**: `/Users/<username>/.jupyter/jupyter_lab_config.py`

Open the `jupyter_lab_config.py` file and add the following at the top (Windows example):

```
import webbrowser
webbrowser.register(
    'chrome', None,
    webbrowser.GenericBrowser(
        'C:\\Program Files (x86)\\Google\\Chrome\\Application\\chrome.exe'
    )
)
```

Here, `webbrowser.register()` registers a new browser that JupyterLab can use to launch the interface. The first argument is for the name to reference the browser, which, in this case, is `chrome`. The second argument is for the controller, and with `None`, we specify the default behavior of the browser, and the final argument provides a path to the browser (Chrome) executable.

The `GenericBrowser` class is used to provide a custom path. If you're using a different system or have installed your browser in a different location, you'll need to find the correct path for your browser's executable file.

Save and close the file. You can rerun `jupyter lab --browser=chrome`, and this should launch the Chrome browser. If you don't want to register additional browsers and prefer to overwrite the default, then you can achieve this simply by updating the c.ServerApp.browser configuration. Here is an example on a Windows machine. Adding this line inside `jupyter_lab_config.py` would overwrite the default:

```
c.ServerApp.browser = 'C:/Program Files (x86)/Google/Chrome/Application/
chrome.exe %s'
```

Note that the %s placeholder is required and represents where the URL will be inserted. This means you can just run jupyter lab without the need to specify a browser.

6. If you prefer JupyterLab not to launch the browser automatically, you can disable the behavior with the following command:

```
$ jupyter lab --no-browser
```

The web server will start, and you can open your preferred browser manually and just point it to `http://localhost:8888`.

If you are prompted for a token, you can copy and paste the URL with the token as displayed in Terminal, which looks like this:

```
To access the server, open this file in a browser:
        file:///Users/tarekatwan/Library/Jupyter/runtime/jpserver-92368-
open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/
lab?token=bbd7fb73e59304fb36eb86b31a9a402ba9a93781971adeab
        http://127.0.0.1:8888/
lab?token=bbd7fb73e59304fb36eb86b31a9a402ba9a93781971adeab
```

7. Lastly, if the default port `8888` is already in use or you prefer to change the port, you can specify it with the `--port` option. Here is an example where the port is set to `8890`:

```
$ jupyter lab --browser=chrome --port 8890
```

This will launch Chrome at `localhost:8890/lab`.

8. If you have multiple Conda virtual environments and you want to use JupyterLab from the main `base` environment as the primary access point to all other available JupyterLab kernels (representing the Conda environments that you have created), then you will need to perform an additional step. Let's investigate the current behavior.

First, deactivate your current `timeseries` environment to return to base:

```
$ conda deactivate
$ jupyter lab
```

If JupyterLab is not installed in the base environment, then you may get an error saying `"jupyter-lab" not found`. In this case, you will need to install JupyterLab in the `base` environment.

Notice that when JupyterLab launches, you might only see one kernel listed under the `Notebooks/Console` sections: the base Python kernel. The expectation is to see multiple, including the `base` and `timeseries` environment you created earlier.

The following screenshot shows JupyterLab with only one kernel:

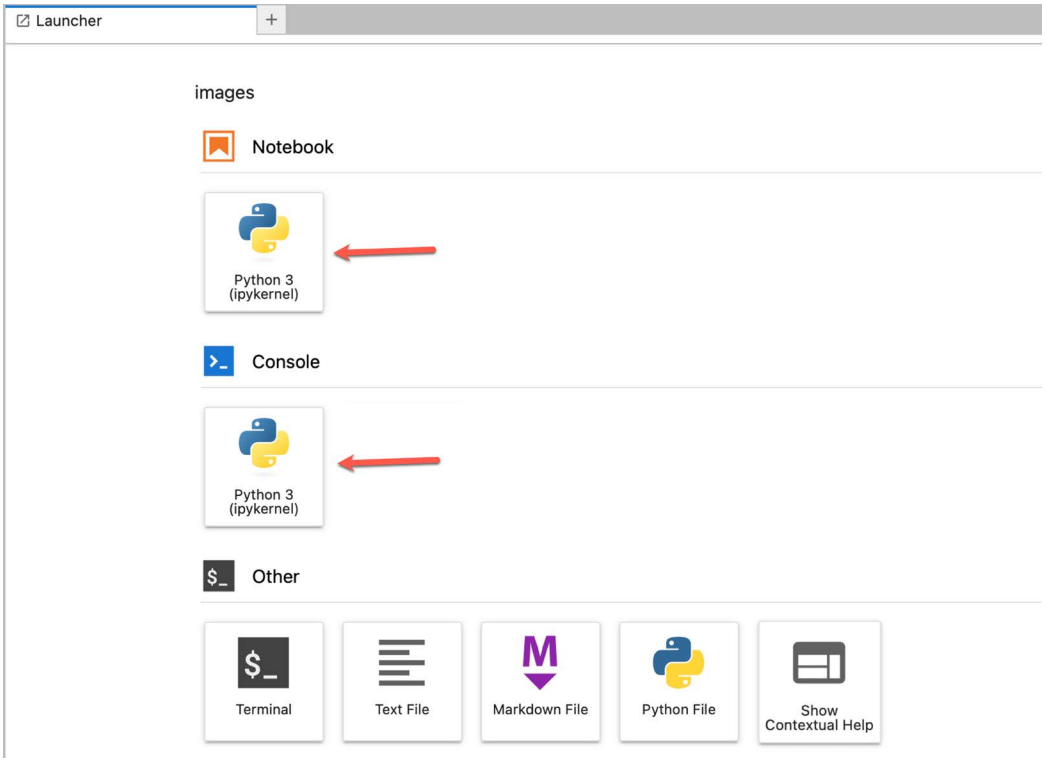

Figure 0.6: JupyterLab interface showing only one kernel, which belongs to the base environment

You can confirm the available environments with the following command :

```
$ conda info --envs
>>
# conda environments:
#
base                          *  /opt/anaconda3
timeseries                    /opt/anaconda3/envs/timeseries
```

Here, there are two environments listed, and the `base` environment is active, indicated by the asterisk, `*`.

9. To make additional Conda environments (such as `timeseries`) automatically available as kernels in JupyterLab, install the `nb_conda_kernels` package in the `base` environment. This enables JupyterLab to automatically detect and display all your environments without requiring any manual setup. Install `nb_conda_kernels` as follows:

```
conda install nb_conda_kernels
```

Next, just install `ipykernel` in the `timeseries` environment to make it accessible as a kernel:

```
$ conda install --name timeseries ipykernel
```

Now, you can launch JupyterLab again:

```
$ jupyter lab
```

The JupyterLab interface will now display the new `timeseries` kernel alongside the `base` kernel:

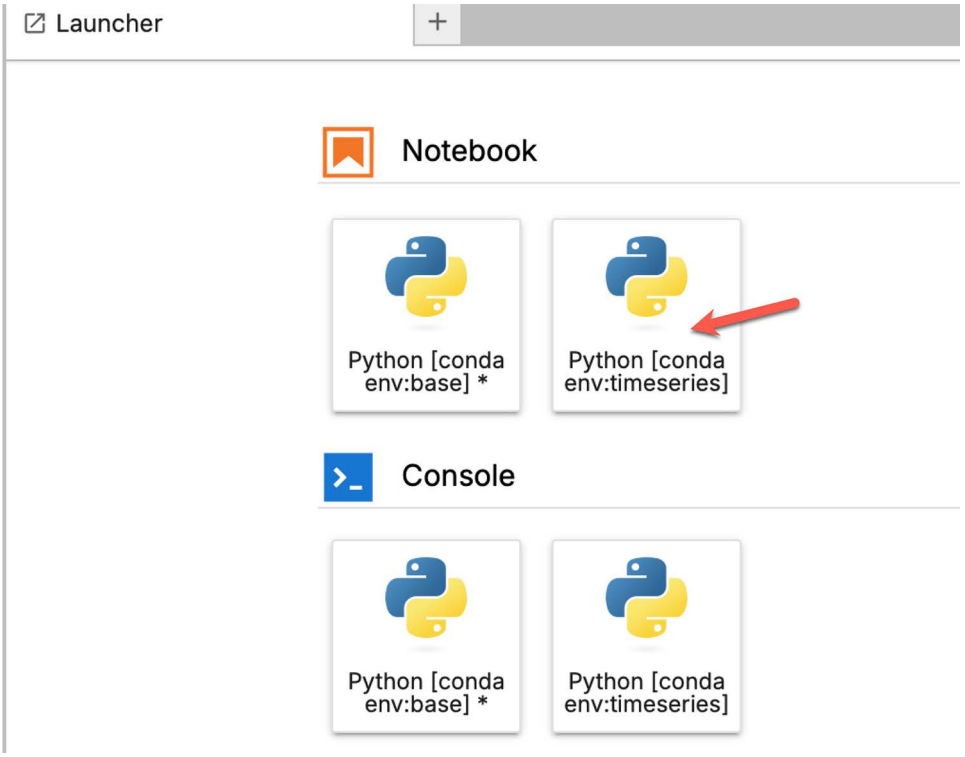

Figure 0.7: JupyterLab interface showing two kernels, base and timeseries

10. To check the list of kernels available for JupyterLab, use the following command:

```
$ python -m nb_conda_kernels list
```

The following is an example output on macOS that lists available kernels created and their location:

```
[ListKernelSpecs] nb_conda_kernels | enabled, 2 kernels found.
Available kernels:
  conda-env-timeseries-py       /opt/anaconda3/envs/timeseries/share/
jupyter/kernels/python3
  conda-base-py                 /opt/anaconda3/share/jupyter/kernels/
python3
```

These act as pointers that connect JupyterLab to the appropriate environment for executing Python code.

How it works...

When you created the new `timeseries` environment and installed your desired packages using `conda install`, it created a new subfolder inside the `/anaconda3/envs` directory. This isolates the environment and its installed packages from other environments, including the `base` environment.

The `nb_conda_kernels` package simplifies the process of managing multiple Conda environments by allowing you to access all available kernels from a single Conda environment, such as `base`. This eliminates the need for manual setup when adding new environments.

For example, to create a new `dev` environment, you can run the following:

```
$ conda create -n dev python=3.12 ipykernel -y
```

After creating the new environment, you can refresh the browser to see the new kernel added automatically. If that does not work, then shut down the JupyterLab server and start it again. You should see something similar to the following:

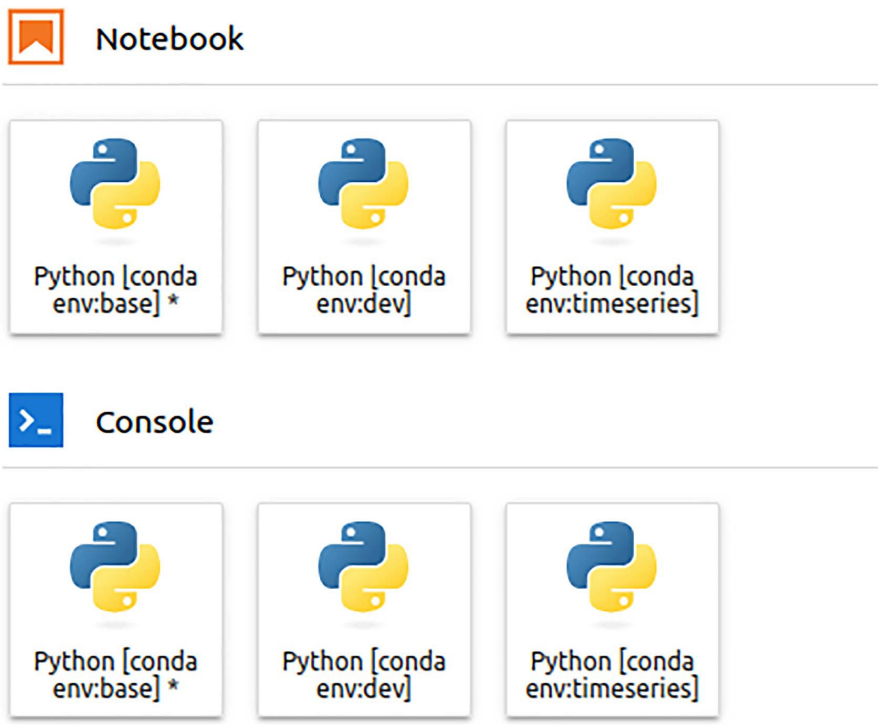

Figure 0.8: JupyterLab interface showing three kernels, base, dev, and timeseries

There's more...

JupyterLab allows you to install several useful extensions. Some of these extensions are developed and maintained by Jupyter, while others are contributed by the community.

You can manage JupyterLab extensions in two ways: through the **command line** using `jupyter labextension install <someExtension>` or through the JupyterLab interface using **Extension Manager**, which provides a graphical interface for browsing. The following screenshot shows what the Jupyter Extension Manager UI looks like:

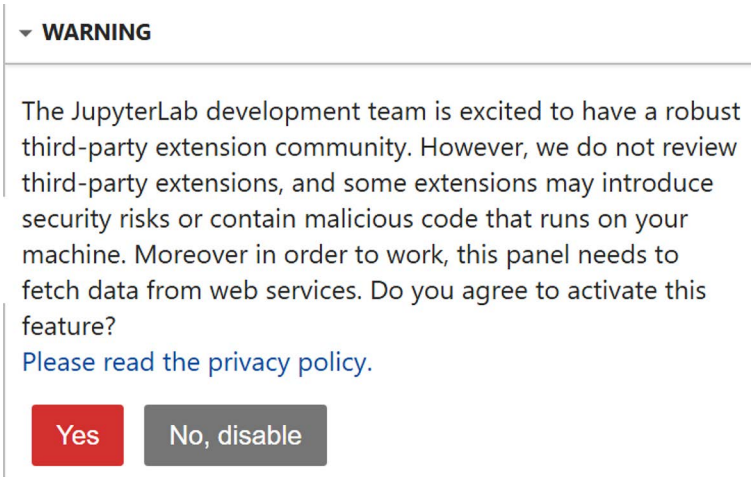

Figure 0.9: JupyterLab third-party extension warning

Once you click **Yes**, the Extension Manager will display a list of available extensions. To install an extension, just click the **Install** button. Installed extensions can be viewed under the **INSTALLED** section. Alternatively, you can list installed extensions via the terminal:

```
$ jupyter labextension list
```

Note: Some packages may require **Node.js** and **npm** to be installed. If these dependencies are missing, you will see a warning similar to the following:

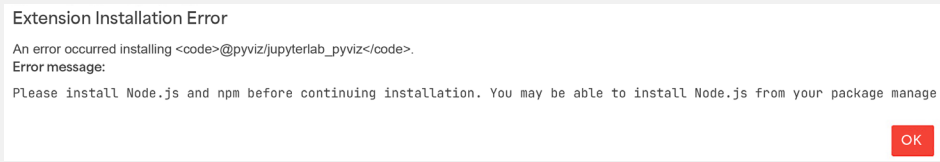

Figure 0.10: Extension Installation Error when Node.js and npm are required

You can download and install Node.js directly from `https://nodejs.org/en/`.

Alternatively, you can use `conda` to install Node.js with the following command:

```
$ conda install -c conda-forge nodejs
```

See also

To learn more about JupyterLab extensions, refer to the official documentation here: `https://jupyterlab.readthedocs.io/en/stable/user/extensions.html`.

If you want to learn more about how JupyterLab extensions are created with tutorials and demos, please refer to the official GitHub repository here: `https://github.com/jupyterlab/extension-examples`.

For more details about the `nb_conda_kernels` project, refer to the official GitHub repository here: `https://github.com/anaconda/nb_conda_kernels`.