Resources that I used:

- Blog post: Vladimir Iglovikov
- Lecture and companion notebook: cookiecutter data science lecture,cookiecutter data science refactoring notebook
- Pyscaffold installation
- Pyscaffold blog post by Florian Willhelm
- Pyscaffold documentation

1 Create a conda environment

To begin, ensure your version of Python is 3.11.4. On a Mac, I installed `pyenv` using

```
brew update
brew install pyenv
```

Python 3.11.4 can then be installed using

```
pyenv install 3.11.4
```

Note that you you might have to install additional packages. For example, I had to install Tkinter using

```
brew install tcl-tk
```

I then created a conda environment (in this case, I called it `ds_project`)

```
conda create -n ds_project python=3.11.4 numpy pandas scikit-learn
```

and activate the conda environment using

```
conda activate ds_project
```

From this point on, all code was executed within this conda environment.

2 Creation of data science project and pre-commit install

This content is largely based on the PyScaffold installation page.

The project structure was created using the data science extension of PyScaffold. First, I installed PyScaffold using (enter `y` to proceed).

```
conda install -c conda-forge pyscaffoldext-dsproject
```

and then created a data science project called `ds_project` using. Note that it is a good idea to first set the directory in the terminal to a project folds on your desktop. In my case, I ensured that the directory was the `Projects` folder on my desktop.

```
cd Desktop/Projects
putup --dsproject ds_project
```

I then installed the `pre-commit` package (enter `y` to proceed) and installed it for my project. Note that this requires setting the directory to the `ds_project` directory.

```
conda install pre-commit
cd ds_project
pre-commit install
```

Also update hooks specified in the pre-commit file

```
pre-commit autoupdate
```

3 Setting up R and autocompletion for code

In Step 3, the `rpy2` package is installed so that R code can be used in concert with Python in Jupyter notebooks (for more information, see rpy2 documentation). By installing `rpy2`, R code can be e

```
pip install rpy2
```

To enable language server protocol (i.e., dynamic code software that enhances workflow largely through code autocompletion), I installed `jupyterlab-lsp` to enable language server protocol and then installed packages that allow code completion within Jupyter for R and Python. Importantly, I spent several hours trying to figure this out, and it turns out that, for some reason, I only got code completion to work for R and Python by using `pip` to install the packages (I could only use `conda` to install languageserver, which enables code autocompletion for R code).

```
pip install 'jupyterlab>=3.0.0,<4.0.0a0' jupyterlab-lsp
pip install 'python-lsp-server[all]' #code completion for Python
conda install -c conda-forge r-languageserver #code completion for R
```

4 Specify pre-commits in the .pre-commit-config.yaml

For a pretty good explanation of the importance of including pre-commits, see this response by ChatGPT that provides four reasons for including pre-commits:

1. *Immediate Feedback*: Pre-commits are checks that run on your code before you even commit it to your version control system (e.g., Git). They provide immediate feedback to developers, allowing them to catch and address issues early in the development process.
2. *Local Environment*: Pre-commits run in the developer's local environment, which means developers can quickly identify issues on their own machines before sharing code with others. This can lead to faster debugging and issue resolution.
3. *Consistency*: Pre-commits enforce coding standards, formatting, and other guidelines consistently across the development team. This ensures that code is written and structured in a uniform way, making it easier to read and maintain.

4. *Preventing Bad Commits*: Pre-commits can prevent bad or problematic code from being committed to the shared repository, reducing the likelihood of breaking the build or introducing bugs into the codebase.

Also see this blog post by Serio Pérez. For this example, I used the pre-commits in the `.pre-commit-config.yaml` in my example project. I suspect that, as we learn more about pre-commits, we will update this file.

5 Create an environment.yml file

At this point, it is a good idea to create the `environment.yml` file to store all the dependencies in your conda environment. I have experimented with several methods and found the best one to be

```
conda env export --no-builds > environment.yml
```

This pretty much gets the job done when we later create a conda environment for GitHub actions later one.

6 Install package and start up Jupyter lab

This step largely pulls from Steps 1–4 in a blog post by Florian Wilhelm.

At this point, a data science project called `ds_project` has been created, packages have been installed to enable pre-commits and facilitate working in Jupyter notebooks, and we have created an `environment.yml` file. Note that a Python package is embedded within the project structure (i.e., in addition to package files such as `pyproject.toml` and `README.md`, there is a `src` folder). At this point I will show a workflow for writing source code and using it in a `.ipynb` notebook (i.e., Jupyter notebook).

First, let's install the Python package (i.e., `ds_project`) using

```
python setup.py develop
```

Second, I will start Jupyter Lab by running `jupyter lab` in the terminal. Make sure the directory in your terminal is set to the folder of your data science project (i.e., `ds_project`). This should already be the case. After entering `jupyer lab` a web browser will open where you can access any file in your data science project. Let's open a new Jupyter notebook and title it `0-initials-workflow.ipynb`.

6.1 Using autoreload

At the top of the Jupyter notebook, we will include the follow magic commands (note my comments are for didactic purposes and should not actually be included in your code, as comments cannot be added to magic commands).

```
%load_ext autoreload  #load autoreload extension
%autoreload 2  #Configure autoreload to the highest level (reload all
               #imported modules and their dependencies)
%load_ext rpy2.ipython #load R capabilities
```

Now, go offline in your IDE and, in the `src/ds_project` folder, create a module called `add_numbers.py` with the following code:

```python
def add_numbers(x, y):
    return x + y
```

Now, create a new cell and import this function (press `Esc` to enter command mode, then `b` to create new code cell below your current cell).

```python
#python packages
from ds_project.add_numbers import add_numbers
```

Create a new cell below this one and now you can use the `add_numbers()` function (at this point, autocompletion should be kicking in. I find it sometimes has a delayed response).

```python
add_numbers(4, 5) #returns 9
```

Now, change the source code of `add_numbers()` to

```python
def add_numbers(x, y):
    return x + 10
```

and rerun the cell in the Jupyter notebook. It will now return `14`. The automatic update occurred because of the `autoreload` extension loaded by the magic commands.

6.2 Using R code

Now we will load and use R code. To do so, create a new cell and load the utils and base packages using

```python
import rpy2
from rpy2.robjects.packages import importr

# import R's "base" package
base = importr('base')

# import R's "utils" package
utils = importr('utils')

# select a mirror for R packages
utils.chooseCRANmirror(ind=1) # select the first mirror in the list
```

Create a new cell and install `ggplot2` (this will take some time).

```python
utils.install_packages("ggplot2")
```

Now, create a new cell and use the magic command `%%R` at the beginning. Use `library(ggplot2)` and you should not be able to use `ggplot2` and also see the autocompletion taking effect.

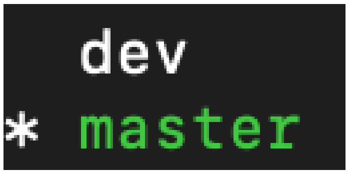
```
library(ggplot2)

data <- data.frame(x = c(1,4), y = c(3,4))

ggplot(data = data, mapping = aes(x=x, y=y)) +
  geom_line()
```

7 Connecting to a GitHub repository

Now we are ready to connect our project to Git repository and then connect it to GitHub. We first set up a Git repository and create a `dev` branch.

```
git init
git branch dev
```

If we list the current branches using `git branch`, we now see two branches of `master` and `dev` (see below).



```
  dev
* master
```

To ensure we are connected to the `dev` branch, use `git checkout dev`.

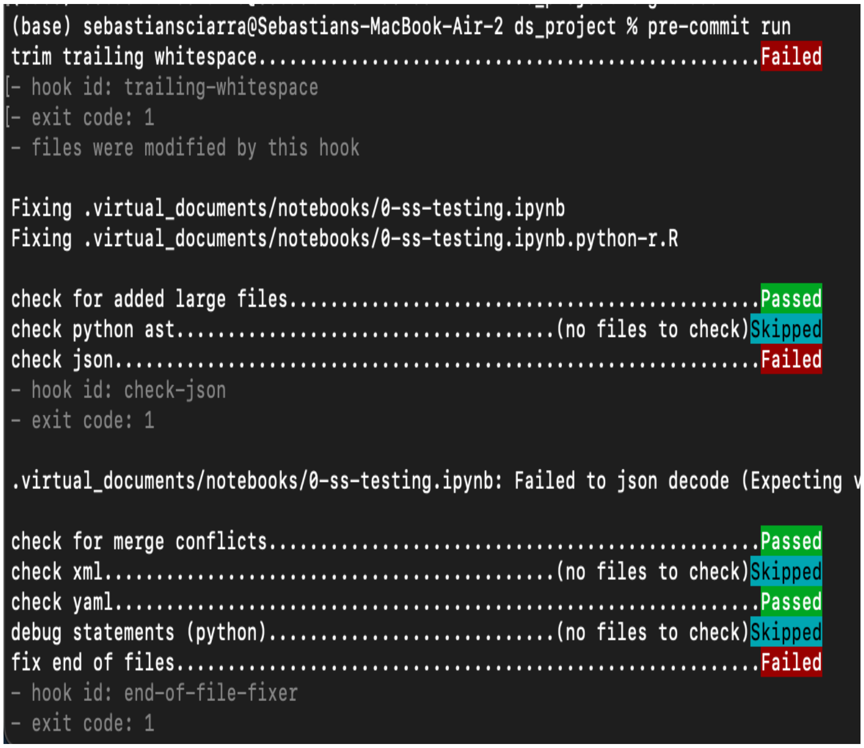
Now, create a new repository on Github and add it as a remote repository to your local Git repository.

```
git remote add origin https://github.com/sebsciarra/ds_project.git
```

Then, we add, commit, and push everything to this GitHub repository (note that we do so by commiting to the dev branch). Note that I run `pre-commit run` after adding every file for staging.

```
git add .
pre-commit
```

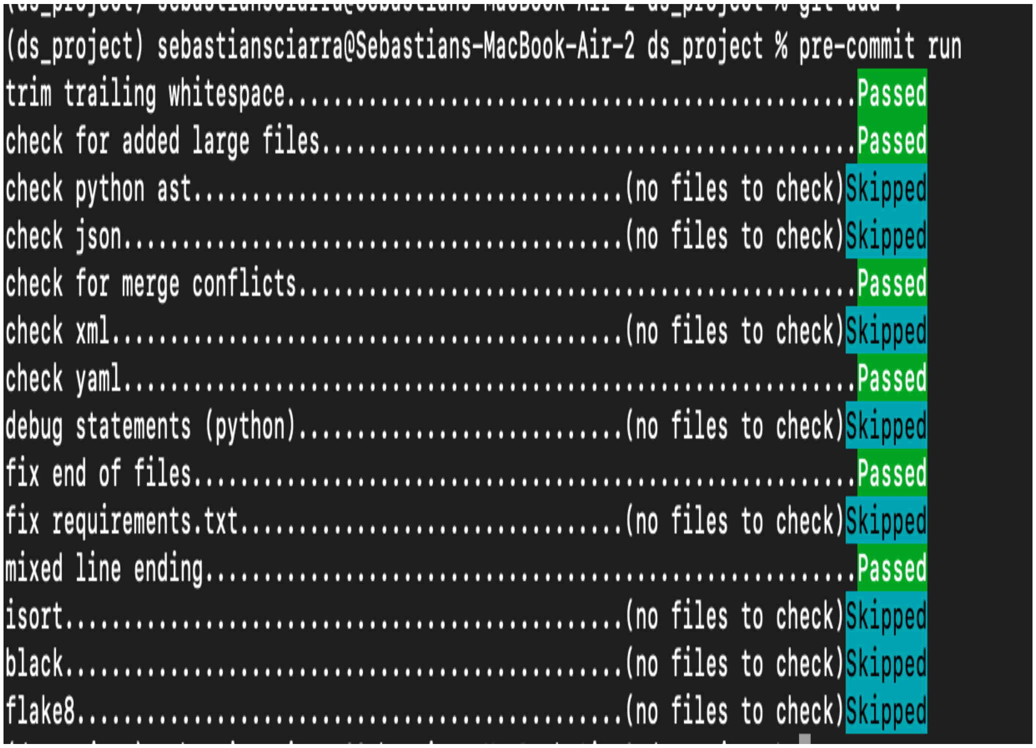
Initially, you will likely get some fails in the output (see below).



```
(base) sebastiansciarra@Sebastians-MacBook-Air-2 ds_project % pre-commit run
trim trailing whitespace.............................................Failed
[- hook id: trailing-whitespace
[- exit code: 1
- files were modified by this hook

Fixing .virtual_documents/notebooks/0-ss-testing.ipynb
Fixing .virtual_documents/notebooks/0-ss-testing.ipynb.python-r.R

check for added large files..........................................Passed
check python ast.................................(no files to check)Skipped
check json...........................................................Failed
- hook id: check-json
- exit code: 1

.virtual_documents/notebooks/0-ss-testing.ipynb: Failed to json decode (Expecting v

check for merge conflicts............................................Passed
check xml........................................(no files to check)Skipped
check yaml...........................................................Passed
debug statements (python)........................(no files to check)Skipped
fix end of files.....................................................Failed
- hook id: end-of-file-fixer
- exit code: 1
```

To accept these changes, simply readd the files and run the pre-commits again. There should be no more fails.

```
git add .
pre-commit
```



```
(ds_project) sebastiansciarra@Sebastians-MacBook-Air-2 ds_project % git add .
(ds_project) sebastiansciarra@Sebastians-MacBook-Air-2 ds_project % pre-commit run
trim trailing whitespace.................................................Passed
check for added large files..............................................Passed
check python ast.................................(no files to check)Skipped
check json.......................................(no files to check)Skipped
check for merge conflicts................................................Passed
check xml........................................(no files to check)Skipped
check yaml...............................................................Passed
debug statements (python)........................(no files to check)Skipped
fix end of files.........................................................Passed
fix requirements.txt.............................(no files to check)Skipped
mixed line ending........................................................Passed
isort............................................(no files to check)Skipped
black............................................(no files to check)Skipped
flake8...........................................(no files to check)Skipped
```

Now we commit the files and push them to the GitHub directory.

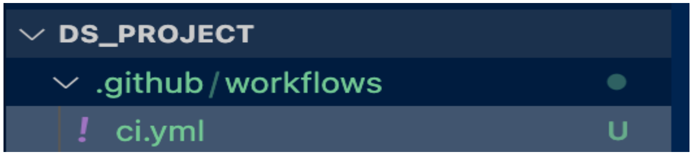
```
git commit -m "Initial commit"
git push --set-upstream origin dev
```

8 Setting up GitHub actions workflow

Now I will show how to enable GitHub actions. To do so, create a new directory in your project `.github/workflow` and create a new file for continuous integration called `ci.yaml`. In this file, copy the code from this file.

```
mkdir .github
mkdir .github/workflows
touch .github/workflows/ci.yml
```

Note that because the `.github` folder cannot be readily accessed from your Desktop, you need to access it from within your IDE to copy the code.



```
∨ DS_PROJECT
  ∨ .github / workflows                ●
    ! ci.yml                            U
```

Now we, as before, add all these changes to the GitHub repo. If pre-commit run shows any fails, readd the changes using `git add .`

```
git add .
pre-commit run
git commit -m "Add GitHub actions workflow"
git push
```

Now, go to your GitHub repository page and you should see that GitHub actions is running all the checks specified in the `ci.yml` file. A large portion of this processing is devoted to setting up a miniconda environment and loading all the dependencies specified in the `environment.yml` file.

9 Collaborating on GitHub

Now we have a GitHub repository that contains our project and its history. At this point, we are ready to collaborate. To collaborate, we will first copy all the contents from the `master` repository into a new branch where we can do our work. Although we can copy over the `master` branch using Git in the terminal, I prefer a more manual approach to avoid overwriting of the `master` branch. First, move up one level into the `Project` directory using `cd ...` Then, delete the `ds_project` folder using `sudo rm -r ds_project`. Now, making sure you are in the `Project` directory, copy the repository at `https://github.com/sebsciarra/ds_project.git` using

```
git clone -b master --force https://github.com/sebsciarra/ds_project.git
cd ds_project
```

Now, create a new branch and switch to it using

```
git checkout -b ex_branch
```

If the branch has any unmerged changes, then it will not be deleted (so make sure to resolve these unmerged changes).

Having copied the contents of the `master` branch to `ex_branch` we will now delete the copy of the `master` branch using

```
git branch -d master
```

Let's now add this branch to the GitHub repository so that other collaborators can see it. From what I understand, the `--set-upstream` flag is important because it sets up tracking between the remote and local copies of `ex_branch` (i.e., ensures `git pull` and `git push` use the `ex_branch` as their reference).

```
git push --set-upstream origin ex_branch
```

To delete the new branch, you can use

```
git push origin -d ex_branch
```

Note that this does not delete it offline.