Setting Up Controlbook Code and Python Environment for ME EN 431 / EC EN 483

Christian Hales, TA

Fall 2024

Table of Contents

Contents

1 Introduction **3**

2 Getting Started **3**
- 2.1 Prerequisites . . . 3
- 2.2 Setting Up VS Code . . . 3
- 2.3 Making a Private Clone of the Public Repo . . . 4
  - 2.3.1 Creating a Private Repo . . . 4
  - 2.3.2 Cloning the Repo and Connecting to the Upstream Remote 4
  - 2.3.3 Fetching and Merging from the Public Repo . . . 6

3 Setting Up Python Environment **7**
- 3.1 Why Virtual environments . . . 7
- 3.2 Setting `venv` . . . 7
- 3.3 Activating `venv` . . . 8
  - 3.3.1 Windows Instructions . . . 8
  - 3.3.2 Linux/macOS Instructions . . . 8
- 3.4 Installing Required Libraries . . . 8

4 Updating Personal Repo **9**
- 4.1 General Process for Personal Repo . . . 9
  - 4.1.1 Summary of Updating Personal Project Repo . . . 10
- 4.2 Grabbing Changes from Controlbook . . . 10
- 4.3 Pull Requesting Controlbook . . . 10

5 Test Matplotlib **10**

6 Conclusion **12**

A Other Virtual Environments **13**
A.1 1. `venv` (Built-in Python Virtual Environment) . . . 13
A.2 2. `conda` (Environment and Package Manager) . . . 13
A.3 3. `mamba` . . . 14
A.4 4. `Docker` (Containerization Tool) . . . 14
A.5 5. `virtualenv` . . . 14
A.6 6. `pipenv` . . . 15
A.7 7. `poetry` . . . 15
A.8 8. `pyenv` + `pyenv-virtualenv` . . . 15
A.9 9. `tox` . . . 16
A.10 10. `pew` . . . 16
A.11 11. `Hatch` . . . 16
A.12 12. `direnv` . . . 17

B How to Make Changes to Public Repository with Forks **18**
B.1 Fork the Public Repository . . . 18
B.2 Clone Your Forked Repository Locally . . . 18
B.3 Make Your Changes . . . 19
B.4 Create a Pull Request (PR) . . . 19
B.5 Wait for Review and Feedback . . . 20
B.6 Question: Why Use Forks and Pull Requests? . . . 20

C Jupyter Notebook to PDF **20**
C.1 Directly with `nbconvert` . . . 20
C.1.1 Installing TEX Distributions: TeXLive and MiKTeX . . . 22
C.2 To Python Script to PDF . . . 22
C.3 To HTML to PDF . . . 23

D Editing This Document with Github **23**

1 Introduction

This document explains how to set up your environment for the ME 431 / ECE 483 course.

These instructions are created by Christian Hales by combining his own instructions on virtual environments and Git upstream branches with Dr. Killpack's general instructions on how to use the controlbook code, as well as 2022 TA Carson Moon's instructions on general setup for creating a private copy of a public repository.

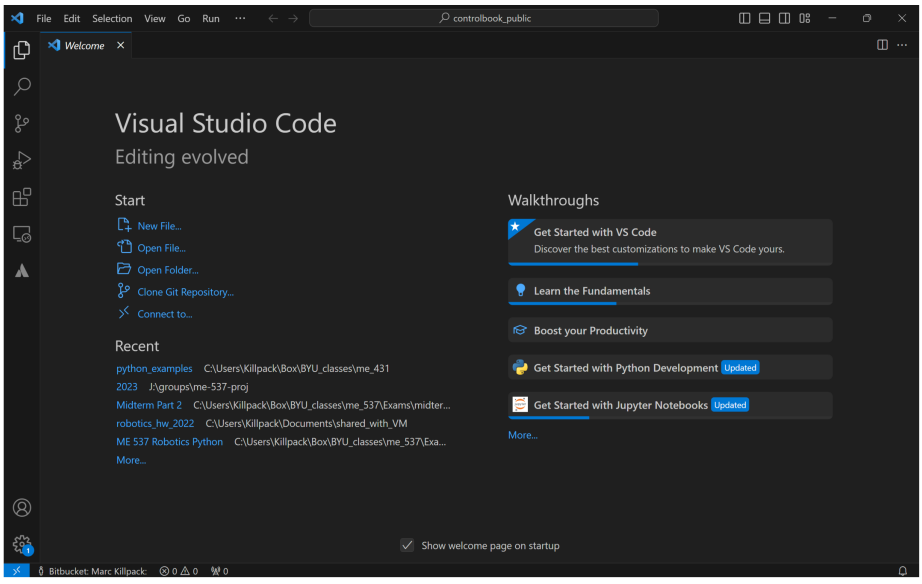
2 Getting Started

2.1 Prerequisites

Before setting up your environment locally, you need to install the following:

- **Python**: You can download and install it from python.org.
- **Git**: Download and install it from git-scm.com.
- **VS Code**: Download and install it from Microsoft's website.

2.2 Setting Up VS Code

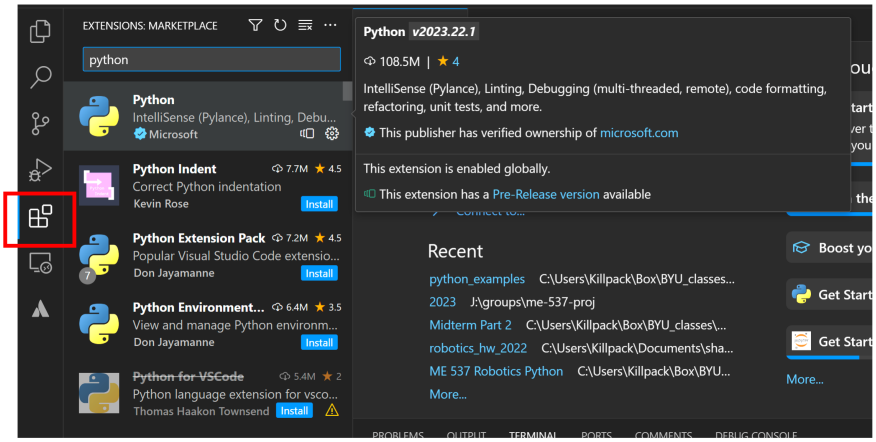


figureVisual Studios Home page

Once you have installed VS Code, you need to install helpful extensions. Follow these steps:

Table of Contents

1. Click on the Extensions button in the sidebar.
2. Search for "Python" and install the Python extension by Microsoft.



Installing Python Extension

2.3 Making a Private Clone of the Public Repo

To avoid sharing code and keeping with BYU Honor Code, you will want to create a private clone of the public repository.

Follow these steps:

2.3.1 Creating a Private Repo

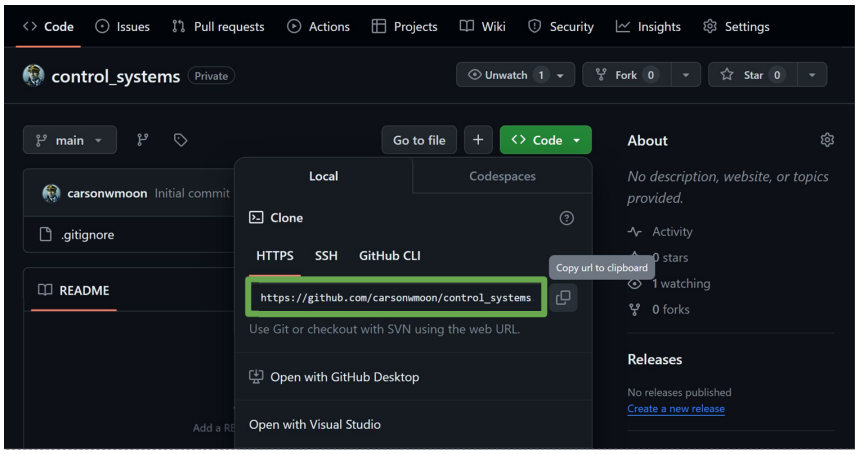
1. Go to GitHub's new repository page.
2. Name your new repository (e.g., `control_systems`).
3. Ensure it is set to "Private" and create the repository.

2.3.2 Cloning the Repo and Connecting to the Upstream Remote

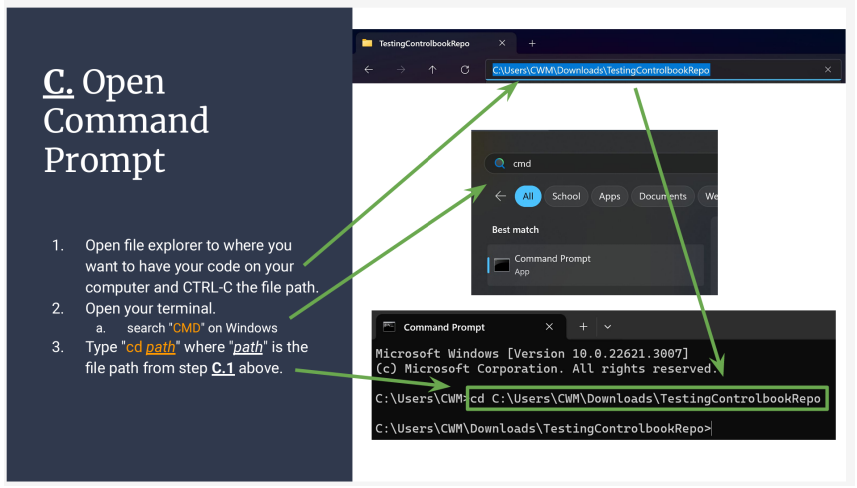
In your terminal:

1. Clone your new private repository using the HTTPS URL provided.

Table of Contents



Grabbing the private repo's link



Using command line – Windows Powershell is used, but any shell can be used.[1][2]

2. Then `cd` into the repository.
3. Connect to the public Controlbook repo by running:

   ```
   git remote add upstream https://github.com/byu-controlbook/controlbook_public.git
   ```

4. Type `git remote -v` to check your connection to both repositories.

[1] A shell refers to Git/Linux Bash, Powershell, Mamba, etc. Note that Git must be installed for Powershell to run Git commands.

[2] Also note that in VSCode, any of these shells can be opened by pressing Ctrl-‘, where ”‘” is a tickmark found to the left of the number-1 button.

Table of Contents

Previous two steps using Carson's slides

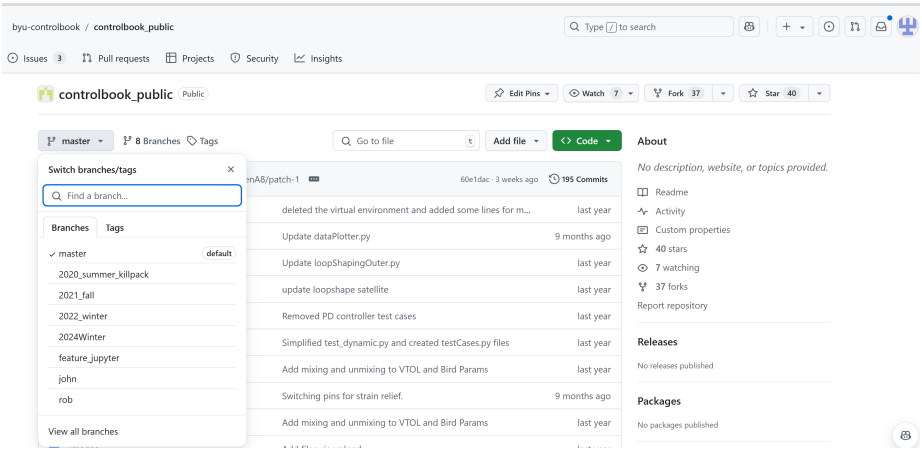
2.3.3 Fetching and Merging from the Public Repo

To get the first copy of the public repository, and to fetch changes from the public repo in the future (do not copy verbatim):

```
git fetch upstream <current Branch>
git merge --allow-unrelated-histories upstream/<current Branch>
git push
```

In the above `Git` commands, `<current Branch>` is this semester's branch of code that will be used.[3]

[3] Look to the TAs for which branch of the code to use. In Fall2024, the `2024Winter` branch was used. Often in industry, the main branch that runs all of the code is the `master` or `main` branch. Development would then be done on separate branches which are then merged into the `master` or `main` branch.



Examples of the different branches

3 Setting Up Python Environment

Below are two different methods for setting up your Python environment for this course:

- **Option A:** Install necessary Python libraries directly on your local machine.
- **Option B:** Set up a virtual environment to manage dependencies.

DO NOT CHOOSE Option A.

3.1 Why Virtual environments

A simple web search or AI search will provide dozens of reasons why virtual environments are important. Here are two reasons that are relevant to the Controls class:

- **Easier Collaboration and Debugging**: When working in a team (or amongst multiple students using the same class code), using virtual environments helps ensure that everyone has the same environment setup.
- **Dependency Isolation**: Each Python project may rely on different versions of the same libraries. Installing these globally on your system can lead to conflicts between projects.

3.2 Setting `venv`

More than a dozen different industry-standard options for virtual environments are available.[4] In this class, virtual environments are installed directly in VS-

[4] See Appendix A for more information.

Code by using VSCode's built-in features, which implement Option 1 in Appendix A (the *venv* method), as explained in this link. Please use that link to set up the virtual environment in VSCode.

3.3 Activating `venv`

After setting up a virtual environment as described in the VS Code documentation, you will need to activate it in your terminal. The steps for creating and activating the environment are slightly different depending on whether you're using a Windows machine or a Linux/macOS machine. **You will need to do this each time you open a terminal in the project.** [5]

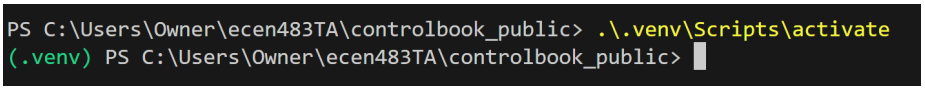
3.3.1 Windows Instructions

On Windows, after creating the virtual environment, you activate it by running the following command in the terminal:

```
.\.venv\Scripts\activate
```

3.3.2 Linux/macOS Instructions

```
source .venv/bin/activate
```



Activating venv on CLI (Command Line Interface[6])

Once activated, you can move on to the next step.

3.4 Installing Required Libraries

Install libraries in the virtual environment by running:

```
pip3 install -r requirements.txt
```

If, for some reason, there is no `requirements.txt` in the project[7], one can get started in the project by looking up how to install `numpy`, `control` and `matplotlib` libraries on your specific shell[8]. Other libraries can be installed as the need arises.

For TAs: If a `requirements.txt` or any other big change is needed, it be created by a Pull Request (`PR`). See Appendix B for more information.

[5] A new virtual environemnt can be made by calling `python -m venv .venv` (or `python3` for MacOS/Linux). Note that `.venv` is a common user-assigned name, but any name for the virtual environment could be chosen. If one deisred, multiple environments could be made.

[6] The venv should already be configured with VSCode. This step is only necessary for installation of libraries. While venv CLI can indeed run python files, this pdf will teach you how to run files in venv through VSCode's GUI.

[7] As of October 2024, the only branch of the project to have this was the **2024Winter** branch.

[8] A shell refers to `Git/Linux Bash`, `Powershell`, `Mamba`, etc.

4 Updating Personal Repo

4.1 General Process for Personal Repo

After making changes to your project files, follow the process to add, commit, and push your changes to your Git repository.

NOTE: If you are looking for a simple list of bash commands to follow, go to the commands found in Summary below.

1. **Stage the changes**: The first step is to tell Git which files have changed and need to be tracked. You can stage all files or specific files using the following commands:

```
# To stage all changed files (includes newly created files):
git add .

# Alternatively, to stage a specific file:
git add <filename>
```

2. **Commit the changes**: Once the files are staged, you need to commit the changes. Committing records the changes to the repository and allows you to add a message explaining the changes you made:

```
# To commit all staged files:
git commit -m "Descriptive message about the changes"

# Alternatively, to commit and stage all files that have been staged before
# (This does NOT include new, untracked files)
# (This CAN replace "git add" if there are no newly created files):
git commit -am "Descriptive message about the changes"
```

   A good commit message describes what changes were made and why they were necessary, helping track the project's history

3. **Push the changes**: The final step is to push your committed changes to your remote repository (e.g., on GitHub). This updates the remote repository with the latest code from your local machine:

```
# Option 1: Push to specific branch
git push origin <branchname>
# Option 2: Push all changes generally
git push
```

   Replace `branchname` with the name of the branch you are working on (e.g., 'main', 'development', etc.). This ensures your changes are saved and accessible from other locations.

Table of Contents

4.1.1 Summary of Updating Personal Project Repo

In summary, a good general process is:

```
git add .
git commit -am "Write a unique message here"
git push
```

4.2 Grabbing Changes from Controlbook

Occasionally, changes will be made to the class code, which will require grabbing the changes from the public repo. Follow the steps in Fetching and Merging from the Public Repo to do so.[9]

4.3 Pull Requesting Controlbook

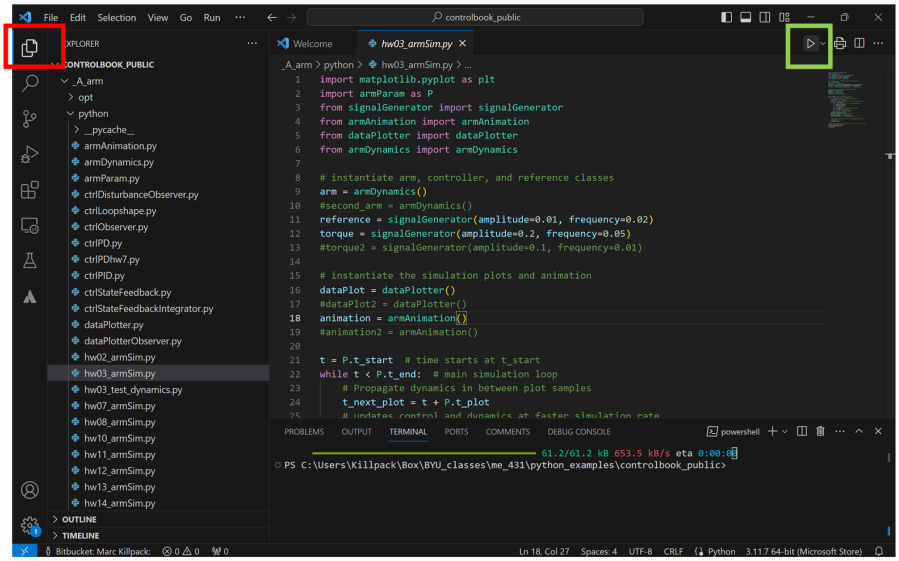
For TAs: You will not push changes to controlbook, but if changes do need to be made, a separate fork can be made, updating the specific problem on your fork and submitting a request to the project with the specific changes. See Appendix B for more details on the subject.

5 Test Matplotlib

Now that the environment is set up, you should check to make sure your computer creates good plots. Navigate to an example file by clicking the button in red shown in the image below, and then selecting a file such as:

`A_arm->python->hw03_armSim.py`

[9] In group projects, best policy is to grab all changes to repositories and group items. This includes performing `git pull` at the start of every day.



Files in Project

After selecting this file, you can run the code by pushing the button in green shown above.

If the visualization blinks and doesn't render properly (most likely a problem on Windows computers), you can change what graphics library is being used to render the plots.

Do this by opening the relevant ____Animation.py file (which is case-study dependent) as shown below. Then uncomment any `matplotlib.use` line. For example:

```
matplotlib.use('tkagg')
```

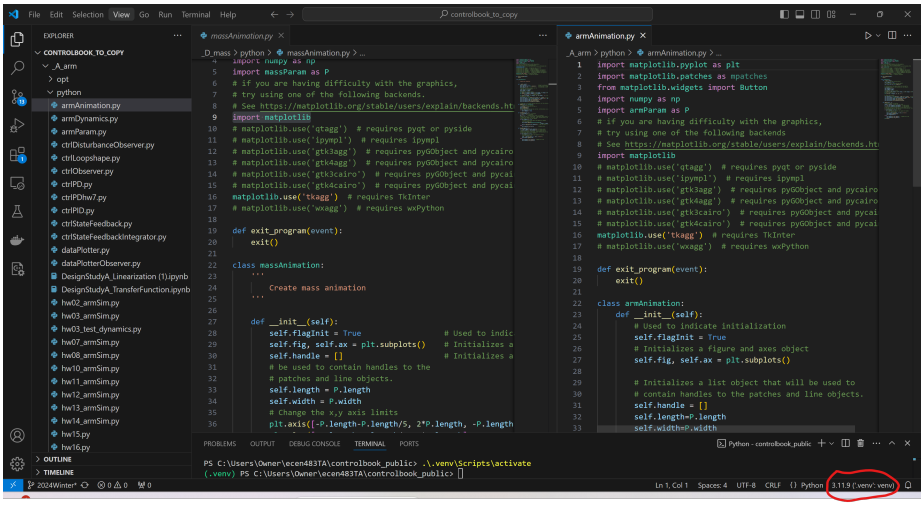


Table of Contents

Matplotlib files in Cases A and D. [10]

It should make a big difference. After uncommenting that line, navigate back to the main file to run it. You will have to do this on every case study, unfortunately, but only the first time.

If you still run into errors, it may be helpful to directly add the following to the top of your simulation file:

```
import matplotlib
matplotlib.use('tkagg')
```

6 Conclusion

This document covered the three main methods for setting up your environment for ME EN 431 / EC EN 483: installing libraries and Python IDE on your local machine, setting up a virtual environment, and configuring Matplotlib. Additionally, instructions for making a private clone of the public repository were provided.

[10] Note also the Virtual Environment kernel circled in the bottom right-hand corner. This shows that the environment is properly set up.

Table of Contents

A Other Virtual Environments

Virtual environments in Python are essential for managing dependencies and ensuring that different projects can use different versions of the same libraries without conflicts. Below is a list of common methods for creating and managing virtual environments, each suited to different use cases:

A.1 1. `venv` (Built-in Python Virtual Environment)

The `venv` module is a built-in tool in Python (from version 3.3 and up) that allows for the creation of lightweight virtual environments. It is a simple and effective way to isolate dependencies for each project.

- **Usage**:

```
python -m venv <env_name>
source <env_name>/bin/activate  # On Linux/macOS
.\<env_name>\Scripts\activate  # On Windows
```

- **Pros**: Built-in, lightweight, and easy to use.
- **Cons**: Basic functionality, lacks some advanced features like dependency management found in other tools.

A.2 2. `conda` (Environment and Package Manager)

`conda` is a powerful environment and package manager that is popular in the data science community. It manages both Python and non-Python dependencies and can create isolated environments for different projects.

- **Usage**:

```
conda create --name <env_name> python=3.x
conda activate <env_name>
```

- **Pros**: Excellent for managing complex dependencies, especially useful in data science projects. Works across programming languages.
- **Cons**: Environments tend to be larger in size, and it requires `conda` (or Anaconda/Miniconda) installation.

Table of Contents

A.3 3. mamba

mamba (Fast Conda Alternative) is a reimplementation of conda in C++ that focuses on speed. It significantly reduces the time needed for environment creation and dependency resolution while maintaining compatibility with conda.

- **Usage**:

```
mamba create --name <env_name> python=3.x
mamba activate <env_name>
```

- **Pros**: Much faster than conda, especially with large environments and complex dependencies.
- **Cons**: Requires mamba installation, but otherwise compatible with conda.

A.4 4. Docker (Containerization Tool)

While not strictly a virtual environment, Docker allows you to create isolated containers that can include the entire software stack (OS, libraries, dependencies). This makes it more powerful than just isolating Python dependencies.

- **Usage**:

```
docker run -it python:3.x bash
```

- **Pros**: Full isolation of the application, including system-level dependencies. Excellent for reproducibility and deployment.
- **Cons**: Requires knowledge of containerization, more overhead than typical virtual environments.

A.5 5. virtualenv

virtualenv is a widely-used tool for creating isolated Python environments. It predates venv, but offers more features and better support for older Python versions.

- **Usage**:

```
pip install virtualenv
virtualenv <env_name>
```

- **Pros**: Supports both Python 2.x and 3.x, provides more features than venv.
- **Cons**: Requires installation, overlaps with venv.

Table of Contents

A.6 6. pipenv

`pipenv` integrates environment creation with dependency management by using a `Pipfile` and `Pipfile.lock` for better reproducibility.

- **Usage**:

```
pip install pipenv
pipenv install
pipenv shell
```

- **Pros**: Combines virtual environments with dependency management, great for project reproducibility.
- **Cons**: Slower than `venv` or `virtualenv`, can be more complex.

A.7 7. poetry

`poetry` is a modern tool for dependency management and packaging in Python. It uses the `pyproject.toml` file for managing dependencies and packaging.

- **Usage**:

```
pip install poetry
poetry install
poetry shell
```

- **Pros**: Combines environment management, dependency resolution, and packaging in one tool.
- **Cons**: Heavier than `venv`, more complex for small projects.

A.8 8. pyenv + pyenv-virtualenv

`pyenv` allows for the management of multiple Python versions, while `pyenv-virtualenv` integrates virtual environments into this workflow.

- **Usage**:

```
pyenv install 3.x.x
pyenv virtualenv 3.x.x <env_name>
```

- **Pros**: Easily switch between multiple Python versions, integrates well with virtual environments.
- **Cons**: Limited to Linux/macOS, more setup required.

Table of Contents

A.9 9. tox

tox is a tool designed for automating testing across multiple Python environments. It is widely used in continuous integration (CI) pipelines.

- **Usage**:

```
pip install tox
tox
```

- **Pros**: Excellent for automating tests across multiple environments.
- **Cons**: Primarily focused on testing rather than general virtual environment management.

A.10 10. pew

pew is a tool that wraps around virtualenv, making it easier to manage multiple environments, and provides features like auto-activation and tab completion.

- **Usage**:

```
pip install pew
pew new <env_name>
pew workon <env_name>
```

- **Pros**: Adds useful functionality on top of virtualenv.
- **Cons**: Less widely used compared to venv or virtualenv.

A.11 11. Hatch

Hatch is a project management tool that also handles virtual environments and package management.

- **Usage**:

```
pip install hatch
hatch new <project_name>
hatch shell
```

- **Pros**: Integrated project management, virtual environments, and packaging.
- **Cons**: Less widely adopted than poetry or pipenv.

Table of Contents

A.12 12. direnv

`direnv` automatically loads and unloads environment variables when you enter or leave a directory, integrating well with virtual environments.

- **Usage**:

```
direnv allow
```

- **Pros**: Automatic environment activation when entering directories.
- **Cons**: Requires additional setup and configuration.

Table of Contents

B How to Make Changes to Public Repository with Forks

When working with public repositories, such as the ‘controlbook‘ repository, you won't have direct permission to push changes to the main repository. To contribute, the standard approach is to create a fork of the repository, make your changes in your fork, and then submit a pull request (PR) to the original repository. Here's how you can go about it.

B.1 Fork the Public Repository

Forking a repository creates a personal copy of the public repository in your own GitHub account. Follow these steps:

1. Go to the GitHub page of the public repository (e.g., controlbook public).
2. Click the "Fork" button at the top-right of the page to create a copy of the repository under your GitHub account.

Image Placeholder: Forking a Public Repo

figureForking the Public Repository on GitHub

B.2 Clone Your Forked Repository Locally

Once you've forked the repository, you'll need to clone your fork to your local machine to work on it. Here's how to do it:

1. In your forked repository (on GitHub), copy the HTTPS URL by clicking the green "Code" button.
2. In your terminal or Git Bash, clone the repository to your local machine by running:

```
git clone <your-fork-url>
```

   Replace ‘¡your-fork-url¿‘ with the URL you copied from your GitHub fork.

Image Placeholder: Cloning the Forked Repo

figureCloning Your Forked Repository Locally

Table of Contents

B.3 Make Your Changes

Now that you've cloned your fork locally, you can make the necessary changes. You can edit files, add new features, fix bugs, or improve documentation. Remember to follow good Git practices:

- Make changes on a new branch to keep the main branch clean.
- Test your changes locally before committing them.

After making your changes, use the following Git commands to commit and push them:

```
# Stage your changes
git add .

# Commit your changes with a descriptive message
git commit -m "Descriptive message about the changes"

# Push the changes to your GitHub fork
git push origin <branch_name>
```

Image Placeholder: Making and Committing Changes

figureMaking and Committing Changes in Your Fork

B.4 Create a Pull Request (PR)

After pushing your changes to your forked repository on GitHub, you can now submit a pull request to the original public repository. Here's how:

1. Navigate to your forked repository on GitHub.
2. Click on the "Pull Requests" tab.
3. Click the "New Pull Request" button.
4. Ensure that the base repository is the original public repository (e.g., 'randybeard/controlbook_public') and the base branch is the main or target branch (e.g., 'main', '2024Winter', etc.).
5. Fill out the pull request form, providing a detailed description of the changes you made and why they are important.
6. Submit the pull request.

Image Placeholder: Submitting a Pull Request

figureSubmitting a Pull Request from Your Fork to the Original Repository

Table of Contents

B.5 Wait for Review and Feedback

Once your pull request has been submitted, the maintainers of the original repository will review your changes. They may request changes or ask for further clarification. Keep an eye on the pull request conversation for feedback and be prepared to make adjustments.

B.6 Question: Why Use Forks and Pull Requests?

Using forks and pull requests is a standard practice in open-source development for several reasons:

- **Permission Control**: Forking allows you to work on a project even if you don't have direct permission to modify the main repository.
- **Isolated Changes**: Working in a fork ensures that any changes you make won't affect the original repository until they have been reviewed and merged.
- **Collaboration**: Pull requests create a structured way to propose changes and have them reviewed by project maintainers.

This workflow is essential for contributing to large-scale collaborative projects, maintaining code quality, and ensuring that updates are thoroughly reviewed before being merged into the main codebase.

C Jupyter Notebook to PDF

This section will be more applicable for TAs, who may need to print new versions of solution code in PDF form to Learning Suite.

There are several ways to convert a Jupyter Notebook to PDF format, depending on the tools available and the desired formatting. Here, we discuss three common methods: using `nbconvert` directly, converting from Python script to PDF, and generating an HTML file before printing it as a PDF.

C.1 Directly with `nbconvert`

The simplest way to convert a Jupyter Notebook (`.ipynb`) to a PDF is by using the `nbconvert` tool with the `-to pdf` option. This method uses LaTeX to compile the notebook, so ensure LaTeX is installed (e.g., TeXLive or MiKTeX).

NOTE: This can also be useful for students who are looking to print their coding work for easy grading. The TAs will thank you.

In a virtual environment, the setup can simply be installed using:

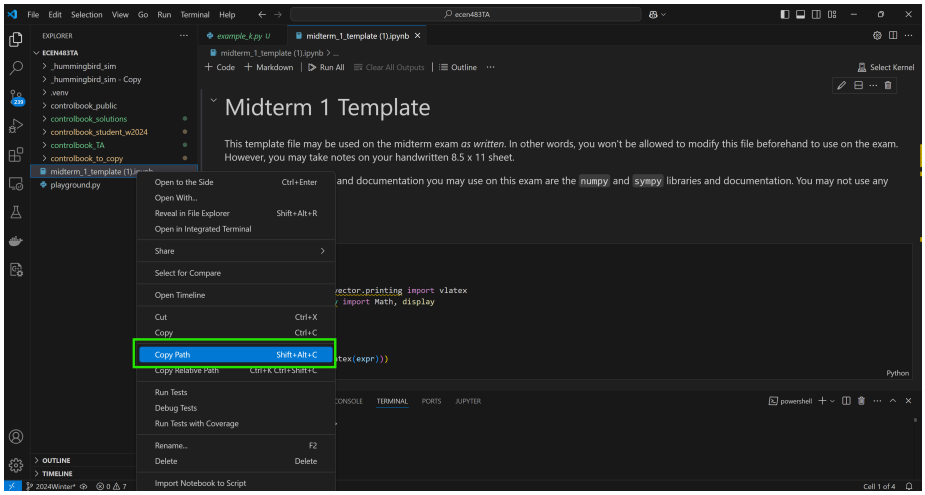
```
pip install nbconvert
```

Table of Contents

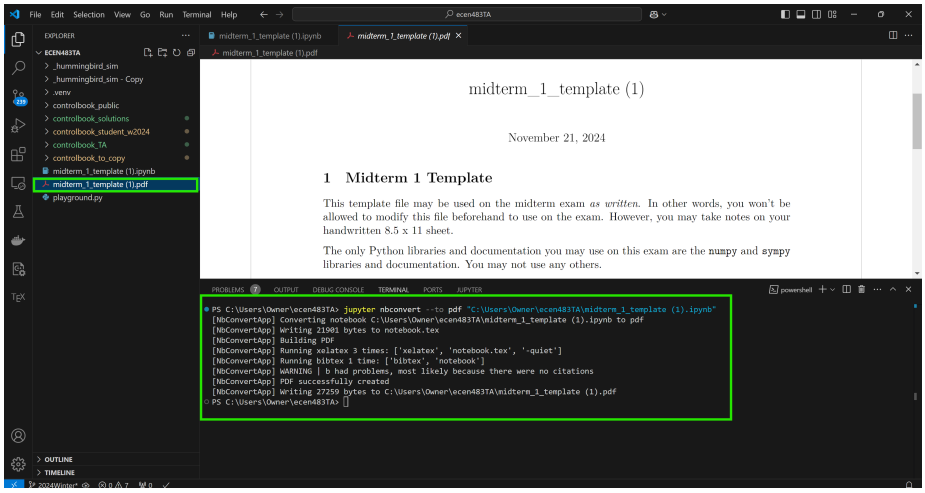
To print, follow the instructions:

- Grab the path of the jupyter notebook.
- apply the following command:

```
jupyter nbconvert --to pdf <current path>
```



How to grab Path with Midterm Template as an example. [11]



Result of converting to notebook with Midterm Template as an example.

This command will create a PDF file with the same name as the notebook file. If LaTeX is not installed or you encounter issues, consider applying the installations below. If those do not allow it to run, consider using one of the alternative methods below the installation instructions below it.

[11] This is different from Relative Path, which grabs the path relative to the project.

Table of Contents

C.1.1 Installing TEX Distributions: TeXLive and MiKTeX

To convert a Jupyter Notebook directly to PDF using `nbconvert`, a TEX distribution such as TeXLive or MiKTeX is required. Below are steps to install each of these distributions on various operating systems.

Installing TeXLive TeXLive is a comprehensive TEX distribution available for Linux, macOS, and Windows.

- **Linux**: TeXLive is usually available in the package manager. For instance, on Debian-based systems, run:

  ```
  sudo apt-get install texlive-full
  ```

  On other Linux distributions, use the equivalent package manager command.

- **macOS**: TeXLive can be installed via MacTeX, which includes TeXLive and other useful tools. Download it from https://www.tug.org/mactex/ and follow the installation instructions.

- **Windows**: Download the TeXLive installer from https://www.tug.org/texlive/. Run the installer and follow the on-screen instructions.

Installing MiKTeX MiKTeX is a lightweight TEX distribution, primarily used on Windows but also available for macOS and Linux.

- **Windows**: Download the MiKTeX installer from https://miktex.org/download. Run the installer and select the default settings for a typical installation.

- **macOS and Linux**: MiKTeX also provides installers for macOS and Linux, available on the same website. Download the appropriate installer and follow the installation instructions provided.

After installing either TeXLive or MiKTeX, ensure that the TEX distribution's binary directory is added to your system's PATH, so it is accessible from the command line. Once installed, `nbconvert` can use LATEX to compile notebooks directly into PDF format.

C.2 To Python Script to PDF

Another way to obtain a PDF from a Jupyter Notebook is to first convert the notebook to a Python script. This approach allows for further customization before generating the final PDF.

1. Convert the notebook to a Python script using the command line:

Table of Contents

```
jupyter nbconvert --to script notebook.ipynb
```

This command will produce a `.py` file containing the notebook code and markdown comments.

2. Alternatively, in Visual Studio Code (VSCode), a Jupyter Notebook can be exported as a Python file directly via the graphical interface. Open the notebook in VSCode and select the option to `Export As Python Script`.

3. Open the `.py` file in a text editor and make any adjustments if necessary. Then, convert it to PDF using a tool like `Pweave` (for formatting Python scripts) or by running the script and saving outputs directly.

C.3 To HTML to PDF

If LaTeX is not installed, an easy alternative is to convert the notebook to HTML format and print it as a PDF.

1. Convert the notebook to HTML:

```
jupyter nbconvert --to html notebook.ipynb
```

This command will generate an HTML file, which preserves the notebook's formatting and interactive elements.

2. Open the resulting HTML file in a web browser. From the browser's print menu, select `Print to PDF` to save the notebook as a PDF file.

Each of these methods has its benefits depending on the complexity of the notebook and the available software. For example, the HTML-to-PDF route may work best for basic formatting needs, while the `nbconvert -to pdf` method provides greater fidelity when LaTeX is installed.

D Editing This Document with Github

See this link.

Table of Contents