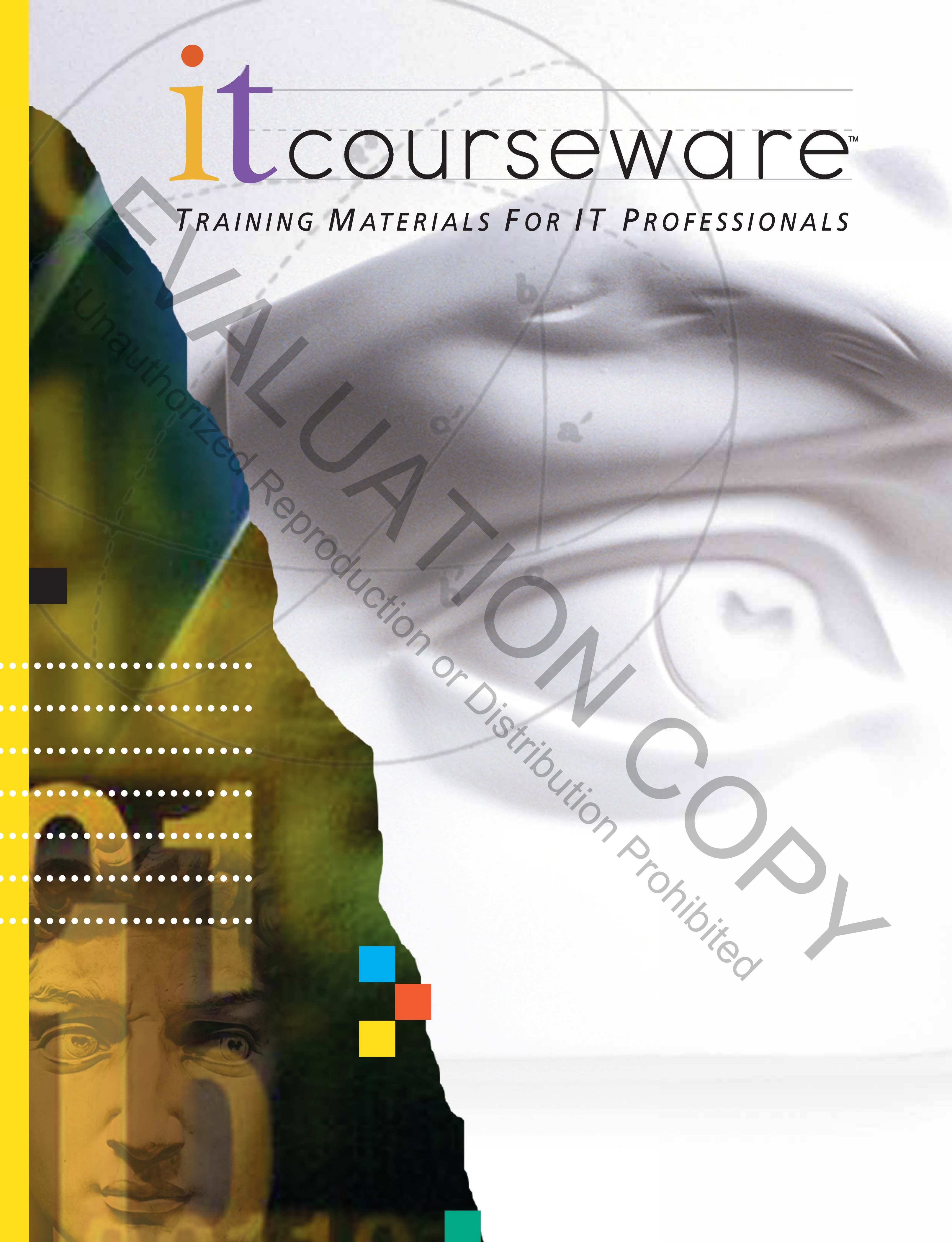


EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Copyright © 2020 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

Version: PYT252.2.0.0

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Stephen Withrow (Editor)

Stephen has over 30 years of experience in training, development, and consulting in a variety of technology areas including Python, Java, C, C++, XML, JavaScript, Tomcat, JBoss, Oracle, and DB2. His background includes design and implementation of business solutions on client/server, Web, and enterprise platforms. Stephen has a degree in Computer Science and Physics from Florida State University.

Roger Sakowski (Editor)

Roger has over 35 years of experience in technical training, programming, data management, network administration, and technical writing for companies such as NASA, Sun Microsystems, Bell Labs, GTE, GE, and Lucent among other Fortune 100 companies.

Class Files

Download the class files used in this manual at https://www.webucator.com/class-files/index.cfm?versionId=4883.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata.cfm.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Table of Contents

LESSON 1. JupyterLab....1
Exercise 1: Creating a Virtual Environment....2
Exercise 2: Getting Started with JupyterLab....6
Jupyter Notebook Modes....10
Exercise 3: More Experimenting with Jupyter Notebooks....12
Markdown....16
Exercise 4: Playing with Markdown....17
Magic Commands....20
Exercise 5: Playing with Magic Commands....21
Getting Help....29
LESSON 2. NumPy....31
Exercise 6: Demonstrating Efficiency of NumPy....32
NumPy Arrays....34
Exercise 7: Multiplying Array Elements....46
Multi-dimensional Arrays....49
Exercise 8: Retrieving Data from an Array....54
More on Arrays....57
Using Boolean Arrays to Get New Arrays....60
Random Number Generation....61
Exploring NumPy Further....63

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

LESSON 3. pandas....65
Getting Started with pandas....65
Introduction to Series....66
np.nan....69
Accessing Elements in a Series....72
Exercise 9: Retrieving Data from a Series....75
Series Alignment....77
Exercise 10: Using Boolean Series to Get New Series....81
Comparing One Series with Another....82
Element-wise Operations and the apply() Method....83
Series: A More Practical Example....85
Introduction to DataFrames....92
Creating a DataFrame using Existing Series as Rows....94
Creating a DataFrame using Existing Series as Columns....96
Creating a DataFrame from a CSV....97
Exploring a DataFrame....98
Exercise 11: Practice Exploring a DataFrame....103
Changing Values....106
Getting Rows....107
Combining Row and Column Selection....112
Boolean Selection....114
Pivoting DataFrames....117
Be careful using properties!....120
Exercise 12: Series and DataFrames....123
Plotting with matplotlib....125
Exercise 13: Plotting a DataFrame....136
Other Kinds of Plots....138

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

LESSON 1

JupyterLab

Topics Covered

☑ JupyterLab.

☑ Jupyter notebooks.

☑ Markdown.

> "Twins, by **Jupiter**!" was all he said for a minute, then turning to the women with an appealing look that was comically piteous, he added, "Take 'em quick, somebody! I'm going to laugh, and I shall drop 'em."
>
> *– Little Women, Louisa May Alcott*

Introduction

IPython stands for Interactive Python. It provides an enhanced command shell, the IPython console, and a browser-based programming environment called **JupyterLab**. This lesson will focus on JupyterLab, which we will use throughout the rest of the lessons.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 1: Creating a Virtual Environment

15 to 25 minutes

As you are likely to work on Python projects that are not geared towards data analysis, you should set up a virtual environment for your data analysis projects. For our purposes, this environment will contain the following libraries:

1. `jupyterlab`
2. `numpy`
3. `pandas`
4. `matplotlib`
5. `mysql-connector-python`
6. `ipywidgets`

❖ Create a Project Folder

To create a virtual environment, you will use Python's built-in `venv` module.

1. Open `Webucator/Python` in Windows Explorer or Finder.
2. Create a new folder called `data-analysis`.
3. Drag the following three folders into the new `data-analysis` folder:
   - `jupyter`
   - `numpy`
   - `pandas`
4. Open a command prompt at the `data-analysis` folder. Then, run the following command:

   ```
   PS …\Python\data-analysis> python -m venv .venv
   ```

   This will create and populate a new `.venv` directory.
5. Using Finder, Windows Explorer, or the command line (`ls .venv`), take a look at the directory contents. They should look something like this:
   Windows

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

.venv

Include

Lib

Scripts

pyvenv.cfg

Mac / Linux

.venv

bin

include

lib

pyvenv.cfg

The contents will differ by operating system. Included in this directory is a `Scripts` (Windows) or `bin` (Mac) folder that contains the `python` executable file and scripts for activating the virtual environment.

6. To work within your virtual environment, you must first activate it. The command for activating a virtual environment varies by operating system. At the terminal, run one of the following:

Windows

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

Mac / Linux

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

7. The prompt text varies by operating system, terminal type, and settings. However, when the virtual environment is activated, its name will always appear enclosed in parentheses before the prompt. Here's what it looks like in PowerShell:

```
(.venv) PS …\Python\data-analysis>
```

If you don't see the virtual environment name in parentheses before the prompt, you are not in the virtual environment.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

8. You can now invoke the Python interpreter and/or install additional packages (using `pip`) within the virtual environment. Install JupyterLab:

```
(.venv) PS …\Python\data-analysis> pip install jupyterlab
```

9. Check to see what version of JupyterLab you installed:

```
(.venv) PS …\Python\data-analysis> jupyter lab --version
2.1.5
```

Now, install the other libraries you will need in the environment:

1. NumPy:

```
(.venv) PS …\Python\data-analysis> pip install numpy
```

2. pandas:

```
(.venv) PS …\Python\data-analysis> pip install pandas
```

3. matplotlib:

```
(.venv) PS …\Python\data-analysis> pip install matplotlib
```

4. `mysql-connector-python` – We will use this to connect to a MySQL database in the `pandas` lesson:

```
(.venv) PS …\Python\data-analysis> pip install mysql-connector-python
```

5. `pip install ipywidgets` – We use this to create quizzes used within a Jupyter notebook:

```
(.venv) PS …\Python\data-analysis> pip install ipywidgets
```

Finally, you need to install the `ipywidgets` notebook and the JupyterLab extensions to get the widgets working:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
(.venv) PS …\Python\data-analysis> jupyter nbextension enable --py widgetsnbextension
(.venv) PS …\Python\data-analysis>
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

RecursionError

If you get a `RecursionError` while installing the jupyter labextension, re-running the command should resolve it.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

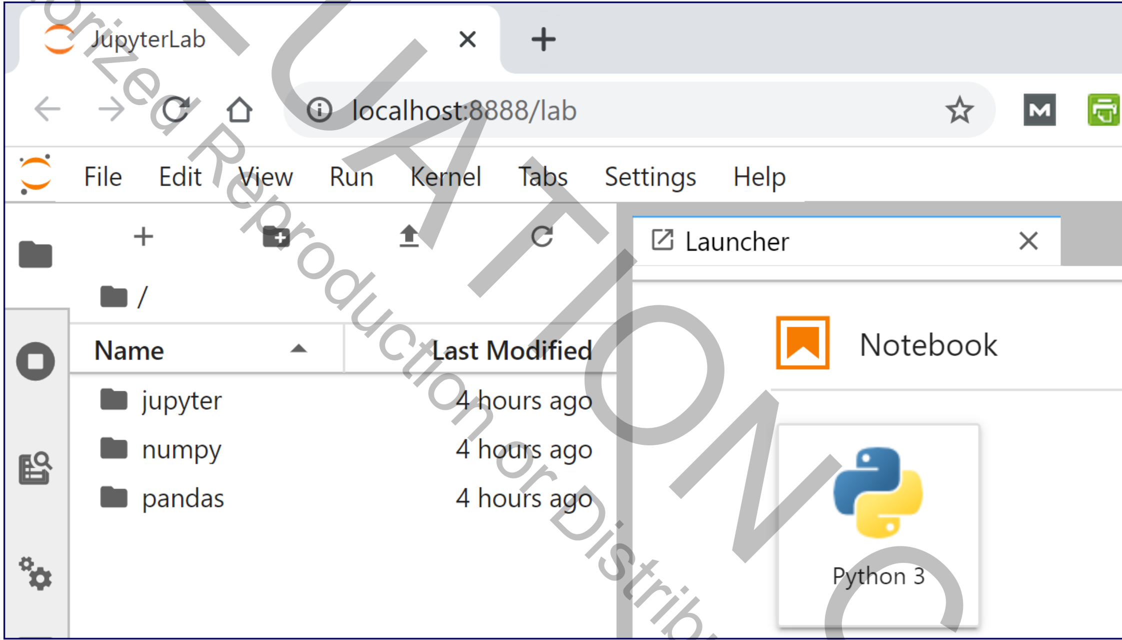
Exercise 2: Getting Started with JupyterLab

15 to 25 minutes

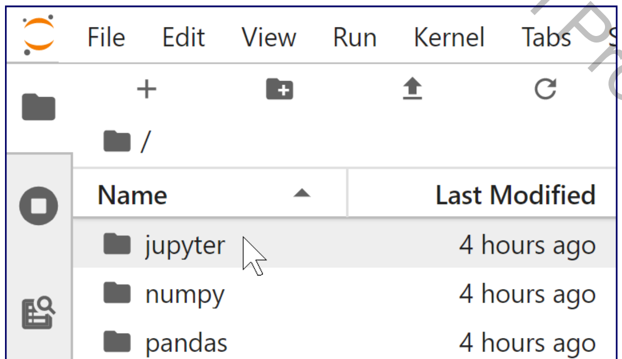
To get started with JupyterLab, run `jupyter lab` at the command line:

```
(.venv) PS …\Python\data-analysis> jupyter lab
```

A browser window or tab should automatically open showing JupyterLab. If a browser window does not open automatically, open a browser and navigate to `http://localhost:8888/lab`:

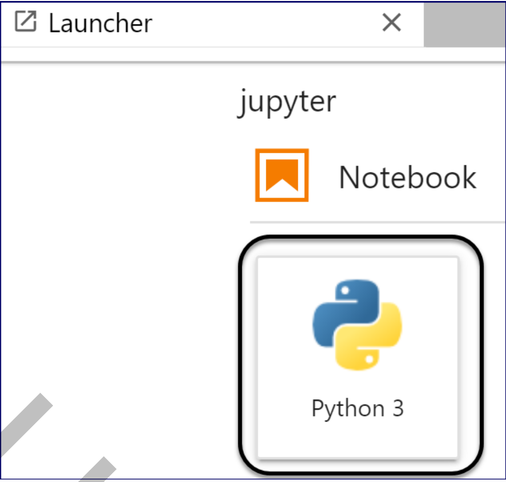


1. In the left sidebar, double-click on the `jupyter` folder to open it:

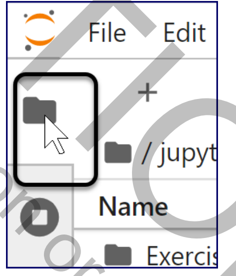


EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

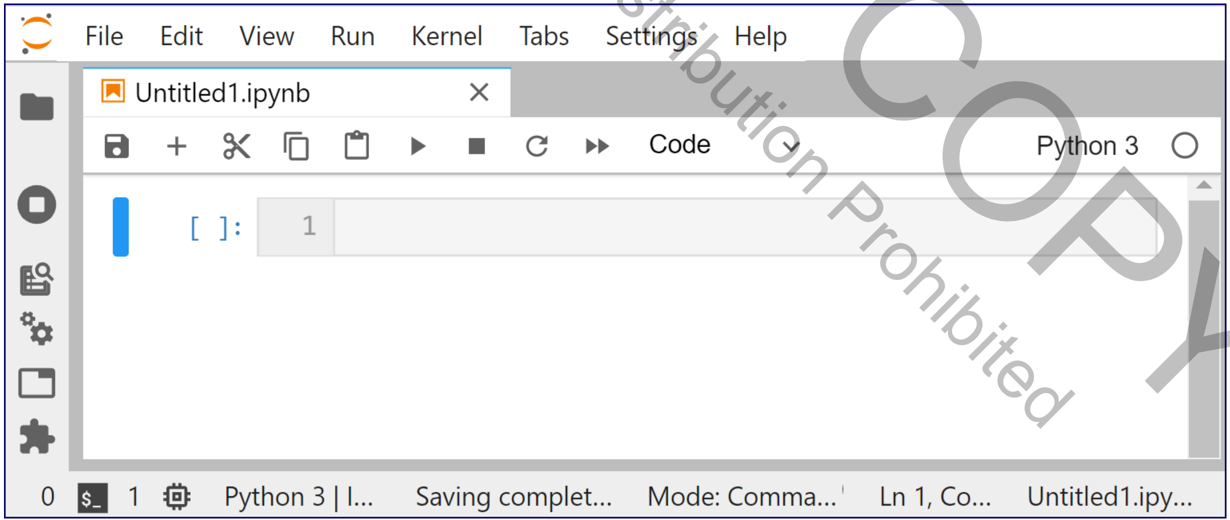
2. In the main area, under **Notebook**, click **Python 3** to start a new notebook:



3. A new browser tab should open with a new untitled notebook. Click the folder icon in the upper left to hide the left sidebar:

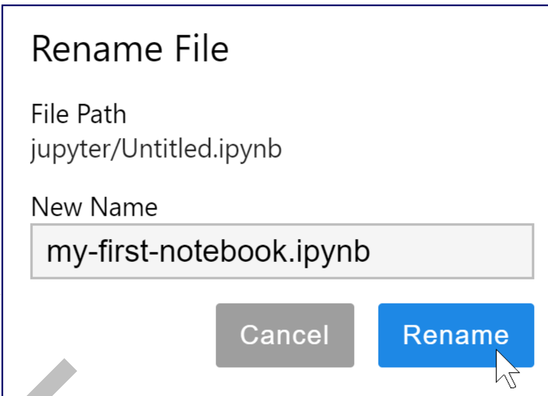


JupyterLab should now look like this:



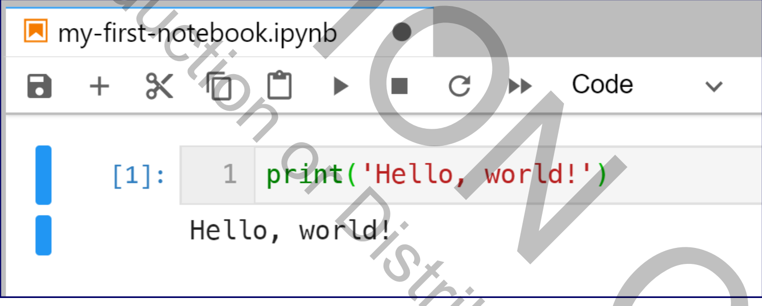
EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

4. From the **File** menu, select **Rename Notebook…** and name your new notebook "my-first-notebook.ipynb":

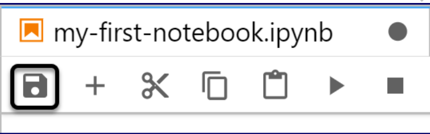


5. A notebook is made up of cells, in which you write your code. The notebook starts with a single cell. Click on the cell to enter text.

6. Enter `print('Hello, world!')` in the cell and run it by selecting **Run Selected Cells and Don't Advance** from the **Run** menu or use the shortcut key: **Ctrl+Enter** (Windows) or **Cmd+Enter** (Mac). Your notebook should now look like this:

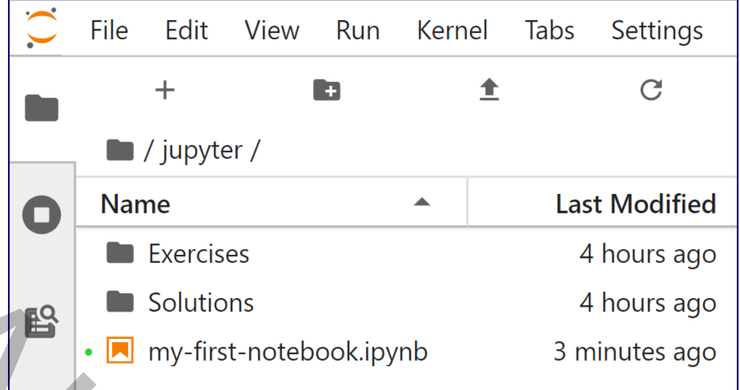


7. Save your file by clicking on the **Save** icon on the toolbar or by pressing **Ctrl+S** (Windows) or **Cmd+S** (Mac):

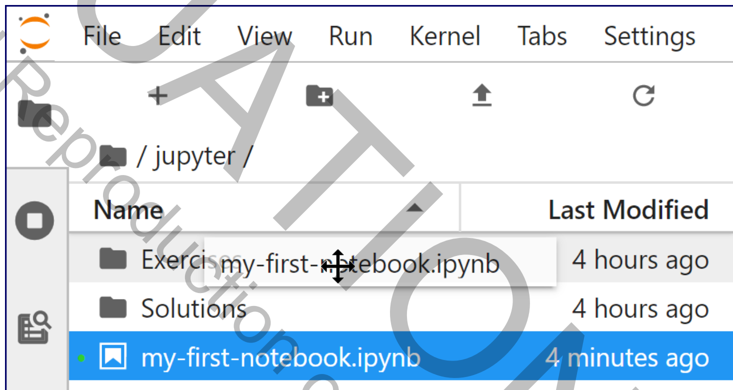


8. Re-open the left sidebar (**View > Show Left Sidebar**). You should see your new notebook listed:

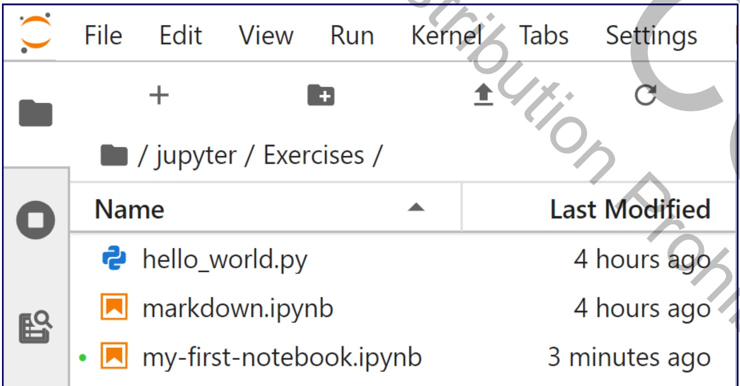
EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



9. In the sidebar, click and hold on `my-first-notebook.ipynb` and drag it into the `Exercise` folder:



10. Double-click on the `Exercises` folder to open it. It should look like this:



11. Leave `my-first-notebook.ipynb` open. We'll do some more work on it shortly.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Jupyter Notebook Modes

Jupyter notebooks have two modes:

1. **Edit mode** – for editing text in cells. **Keyboard shortcut**: **Enter**.
2. **Command mode** – for navigating around the notebook. **Keyboard shortcut**: **Esc**.

Useful Shortcut Keys

- **Ctrl+S** (Windows) / **Cmd+S** (Mac) – Save the notebook.
- **Enter** – Enter edit mode.
- **Esc** – Enter command mode.
- **Ctrl+Enter** (Windows) / **Cmd+Enter** (Mac) – Run code in current cell.
- **Shift+Enter** – Run code in current cell and select next cell (inserting new one if current cell is the last cell in the notebook).
- **Alt+Enter** – Run code in current cell and insert new cell below.
- **A** (in command mode) – Insert cell above current cell.
- **B** (in command mode) – Insert cell below current cell.
- **D,D** (in command mode) – Delete cell.
- **Shift+L** (in command mode) – Show/Hide line numbers.
- **C** (in command mode) – Copy selected cells.
- **X** (in command mode) – Cut selected cells.
- **V** (in command mode) – Paste cells.

In command mode, use the **Up** and **Down** arrow keys to navigate between cells.

Use **Shift+Click** to select multiple consecutive cells. You can then copy and paste or drag the selected cells to a new location in the notebook.

Enclosing Code Snippets

In edit mode, enclose text with quotation marks, parentheses, and brackets by highlighting the text and then pressing the opening enclosing symbol. For example, if you have the following text:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
print(Hello, World!)
```

Wrap `Hello, World!` in quotes by highlighting it and pressing the quotation mark key.

You can do the same with any of the following characters:

1. " – Wrap in double quotes.
2. ' – Wrap in single quotes.
3. (– Wrap in parentheses.
4. [– Wrap in square brackets.
5. { – Wrap in curly braces.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 3: More Experimenting with Jupyter Notebooks

15 to 20 minutes

In this exercise, you will continue to experiment with your first Jupyter notebook. Open `my-first-notebook.ipynb` if it's not still open. Follow these instructions to familiarize yourself with navigating a notebook:

1. From the **Kernel** menu, select **Restart Kernel and Clear All Outputs…**. Click **Restart** in the dialog that pops up. This won't delete any of your code, but it will clear all the outputs and give you a fresh start.
2. **Enter** – Click inside the cell that contains `print('Hello, world!')` and press **Enter**. A new line is added:

```
[ ]:  1  print('Hello, world!')
      2  |
```

3. Press **Ctrl+Enter** (Windows) or **Cmd+Enter** (Mac). This will run the code in the cell:

```
[1]:  1  print('Hello, world!')
      2

      Hello, world!
```

4. Click inside the cell again and press **Shift+Enter**. This will run the code and *advance*, meaning it will move on to the next cell, creating a new one if one doesn't already exist.
5. In the new cell, enter `'Hello, world!'` and press **Ctrl+Enter** (Windows) or **Cmd+Enter** (Mac):

```
[2]:  1  print('Hello, world!')
      2

      Hello, world!

[3]:  1  'Hello, world!'

[3]:  'Hello, world!'
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Run the Cell

When we want you to run the contents of a cell without advancing, we will instruct you to "run the cell." You should do so by pressing **Ctrl+Enter** (Windows) or **Cmd+Enter** (Mac).

The **[3]** line below the cell shows the return value of the last operation in the cell. Notice that the printed value ("Hello, world!") in the first cell has no **[#]** label. That is because the `print()` function does not return anything; it just prints.

6. To illustrate, add another cell by pressing **B** and enter the following code:

```
result = print('Hello, world!')
print(result)
```

7. Press **Ctrl+Enter** to run the cell. The `print()` function will still run, printing "Hello, world!", but because `print()` returns no value, `result` will contain `None`:

```
Hello, world!
None
```

8. Let's take this a little further. Insert a new cell below the current one by pressing **B**.
9. Type the following function:

```
def my_print(text):
    print(text)
    return 'I printed ' + text
```

10. Press **Shift+Enter** to run the cell and insert a new cell below it.
11. In the new cell, type:

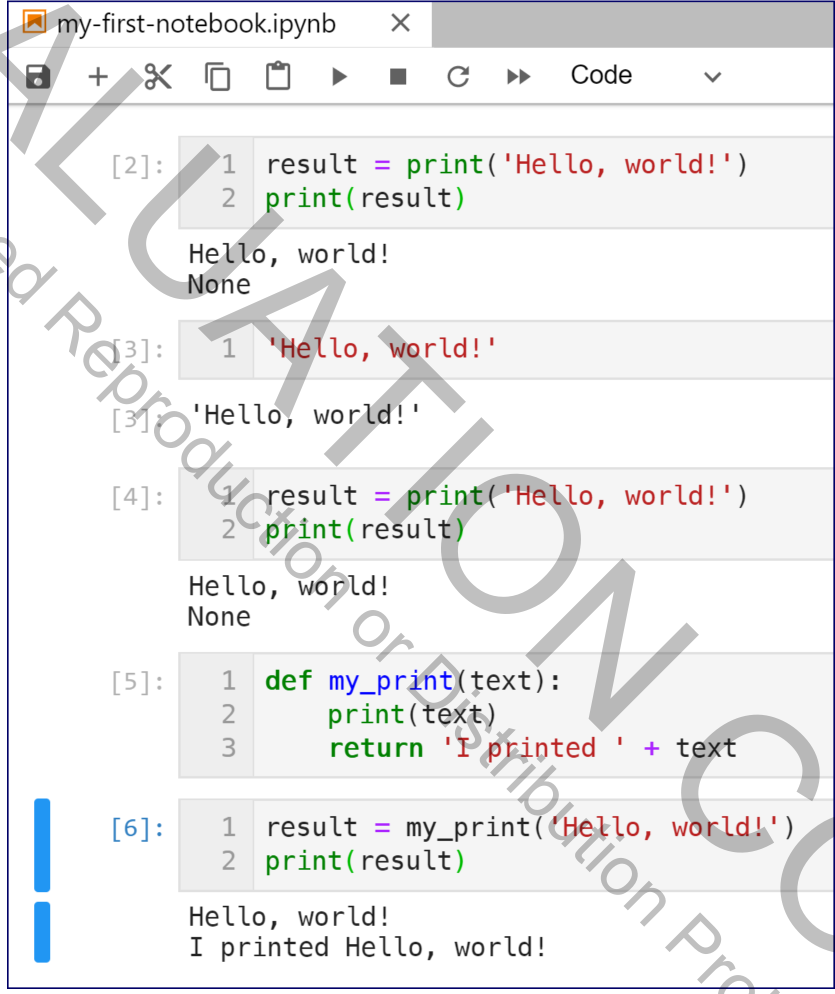
```
result = my_print('Hello, world!')
print(result)
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

12. Run the cell. The `my_print()` function prints "Hello, world!", but it also returns a message. `result` returns that message:

```
Hello, world!
I printed Hello, world!
```

13. Your notebook should look something like this now:



Don't worry if your input and output numbers are different.

14. Click in the cell that contains the call to `my_print()` (`[6]` in the screenshot above).

15. At the end of the cell, press **Enter** and type `result`. Then press **Ctrl+Enter** to run the cell. The result should look like this:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
[7]:  1 result = my_print('Hello, world!')
      2 print(result)
      3 result

      Hello, world!
      I printed Hello, world!
[7]:  'I printed Hello, world!'
```

16. Notice the difference between printing and returning. Calls to `print()` will write text below the cell, but it will not be labeled with a `[#]` label. Also note that every call to `print()` will be printed, but only the final return value will be output. To illustrate this, press **Shift+Enter** to add a new cell and enter:

```
1+1
2+2
```

Run the cell and notice that only the last result is output:

```
[8]:  1 1+1
      2 2+2

[8]:  4
```

To output multiple values, comma-delimit them on the last line of the cell:

```
[9]:  1 sum1 = 1+1
      2 sum2 = 2+2
      3 sum1, sum2

[9]:  (2, 4)
```

Really, this is just one result: a tuple containing multiple values.

17. Click in the first cell and press **Shift+Enter**. Notice that the code runs and the cursor advances to the next cell. Click in the first cell again and press **Alt+Enter**. Notice that the code runs and a new cell is added.

18. **edit mode** and **command mode**. When the cursor is inside of a cell, you are in *edit mode*. The cell will have a highlighted border. You can move to *command mode* by pressing the **Esc** key or by clicking to the left of the cell. You will see a thick vertical line to the left of the cell:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Press **Enter** to enter *edit mode* again.

- **Enter** – *command mode* to *edit mode*.
- **Esc** – *edit mode* to *command mode*.

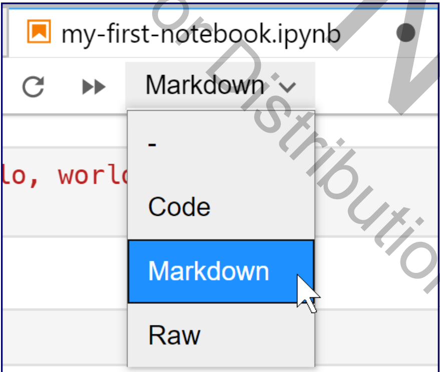
When you are in *command mode*, you can run, but not edit code.

❖ The Numbers in Brackets

The numbers in brackets indicate the order in which the code was run. You do not have to run cells in order. The number to the left of the cell is the "input number" and the number to the left of the output below the cell is the "output number."

Markdown

Jupyter notebook supports Markdown, a simple formatting language that gets converted to HTML. To convert a Jupyter notebook cell to Markdown, press **M** while in *command mode* or select **Markdown** from the drop-down menu on the toolbar:



You may find the Markdown language useful for taking notes directly in the Jupyter notebook files used in these lessons. The easiest way to learn Markdown is to play with it a bit.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 4: Playing with Markdown

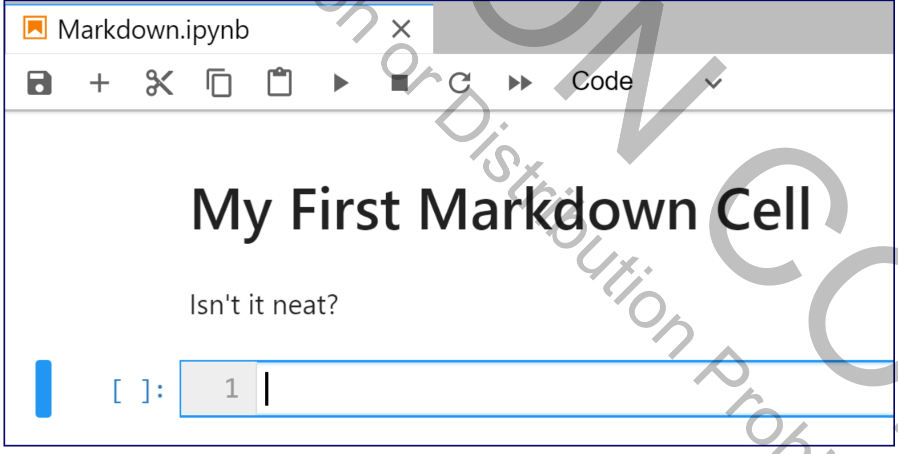
15 to 25 minutes

1. Open the `markdown.ipynb` notebook in `jupyter/Exercises`.
2. Click the square brackets to the left of the cell.
3. Press **M** to make the cell a Markdown cell.
4. Press **Enter** to enter edit mode.
5. Type the following:

```
# My First Markdown Cell
Isn't it neat?
```

   - A single hashmark followed by a space at the beginning of a line creates a top-level heading.
   - The second line contains no markdown, so it just creates a plain-text paragraph.

6. Press **Shift+Enter** to run the cell and create a new one.
7. Save. Your notebook should look like this:



8. Double-click on the Markdown cell to enter edit mode again and change the code as follows:

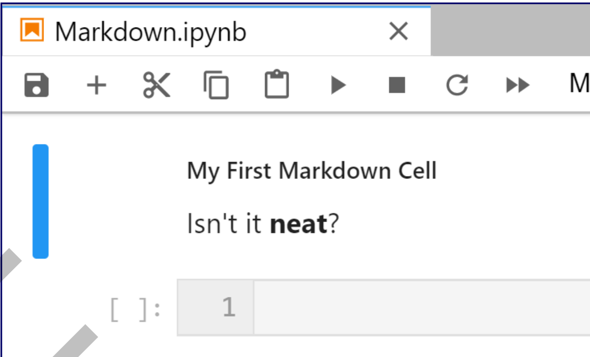
```
###### My First Markdown Cell
Isn't it **neat**?
```

   - There are six heading levels. Headings get smaller with each hashmark added.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

- Surrounding text with two asterisks on each side makes it bold.

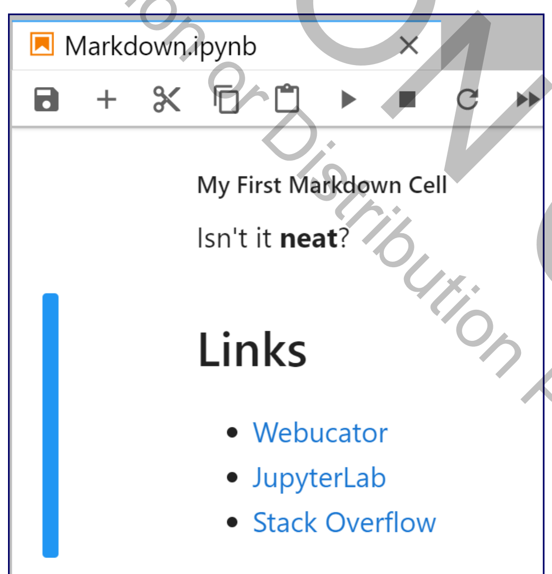
9. Press **Shift+Enter** and save. Your notebook should look like this:



10. Make the second cell a Markdown cell and enter the following to create a bulleted list of links:

```
## Links
* [Webucator](https://www.webucator.com)
* [JupyterLab](https://www.jupyterlab.com)
* [Stack Overflow](https://www.stackoverflow.com)
```

11. Run the cell. The notebook should now look like this:



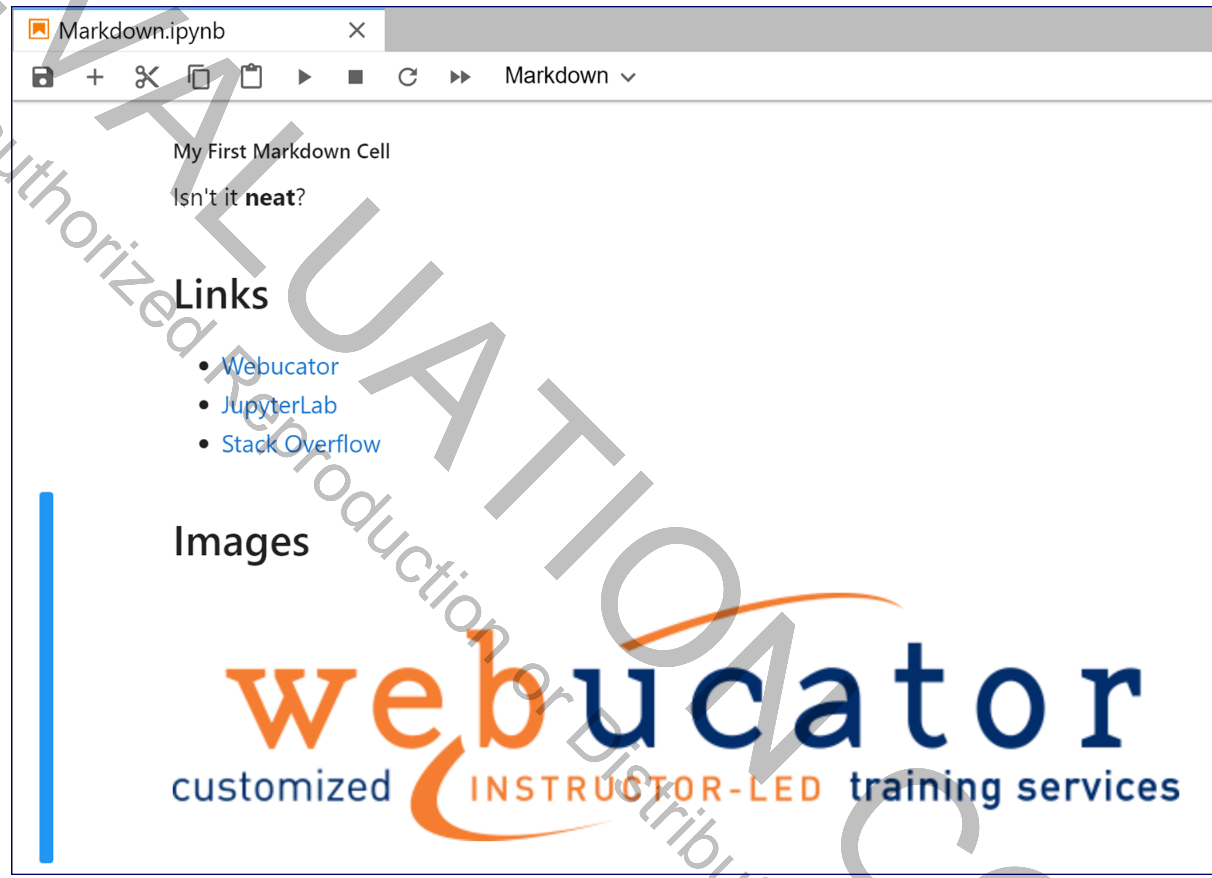
12. Press **B** to add a new cell, **M** to make it a Markdown cell, and **Enter** to enter edit mode.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

13. Enter the following to add an image:

```
## Images
![Webucator Logo](https://images.webucator.com/logos/webucator.png)
```

14. Run the cell. The notebook should now look like this:



15. From the **Help** menu, select **Markdown Reference** to see other Markdown formatting options. Play around a bit.

Solution

Open `jupyter/Solutions/markdown.ipynb` and browse through the notebook to learn additional tricks. Double-click on any cell to enter edit mode and see how the cell was created.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Magic Commands

IPython includes a list of more than 100 "magic" commands that can be used in Jupyter notebooks. These are not Python commands, but extra commands that help make development easier.

Magic commands are prefixed with a percent sign (`%`).

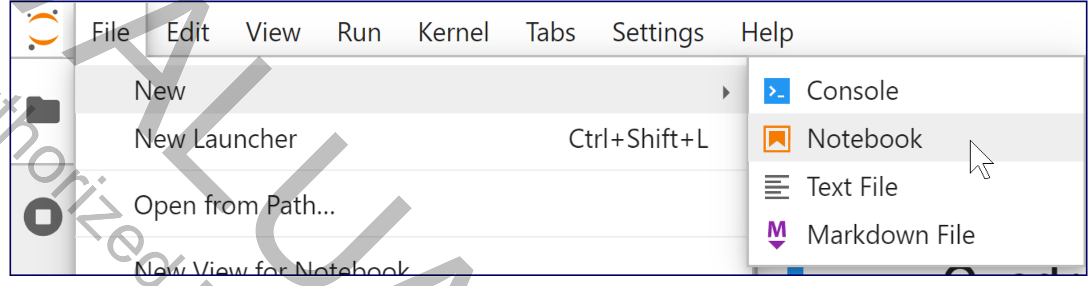
EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 5: Playing with Magic Commands

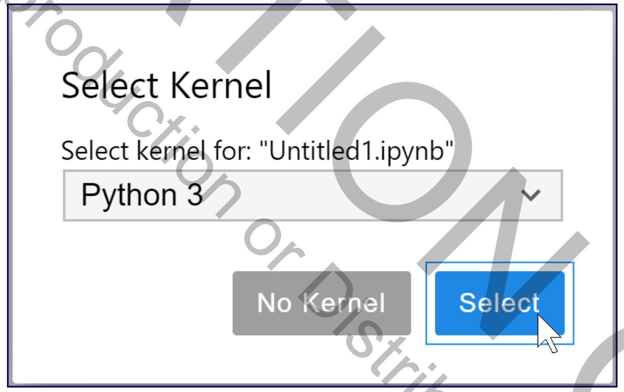
20 to 30 minutes

The easiest way to get familiar with magic commands is to play with them. You'll need a playground:

1. **File > New > Notebook**



2. Select "Python 3" for the terminal:



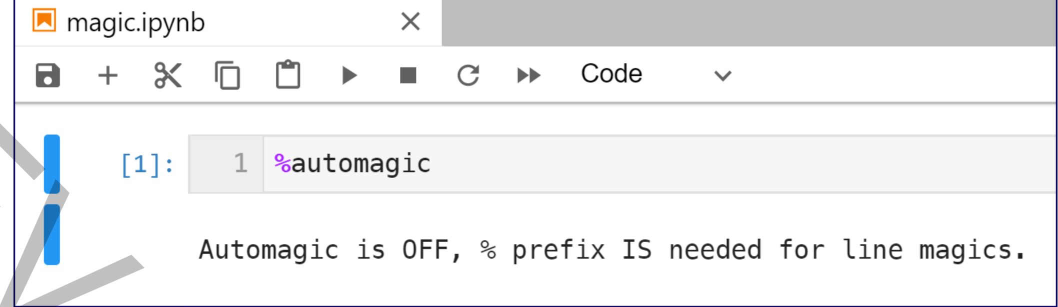
3. Rename the notebook "magic.ipynb".

❖ Automagic

The `%automagic` command is used to toggle whether magic functions are callable without having to type the initial `%`.

1. In the first cell of the `magic.ipynb` notebook, enter `%automagic` and run the cell. This will toggle the ON/OFF value of **automagic**:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



2. If the output says "Automagic is ON" then you **do not need** the `%` prefix for magic commands. If it says "Automagic is OFF" then you **do need** the `%` prefix for magic commands.

Whether `automagic` is on or off, using the % prefix is always allowed. In notebooks you intend to reuse or share, it's a good idea to include the % prefix so the scripts are not dependent upon the state of automagic.

❖ Autosave

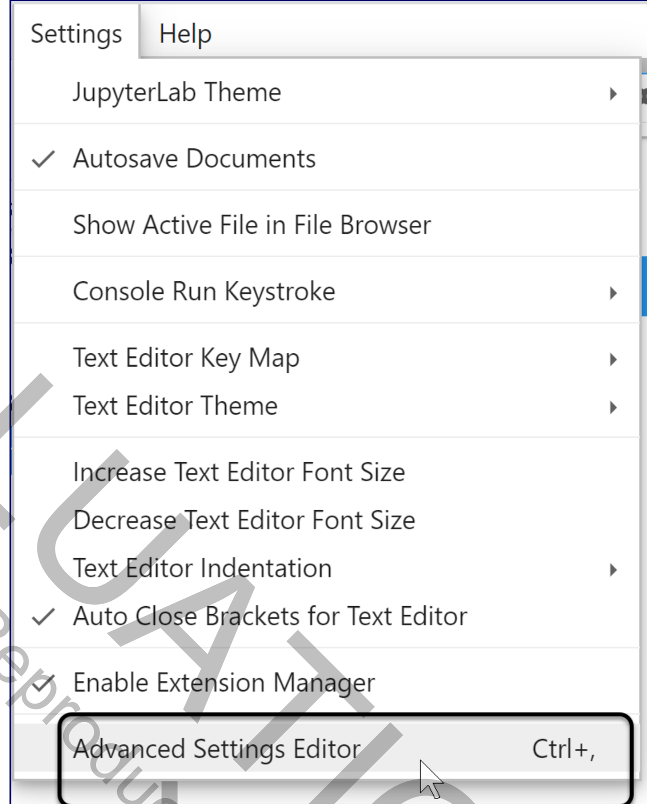
The `%autosave` command is used to set the autosave interval in the notebook (in seconds):

- `%autosave 60` – Sets the autosave interval to 60 seconds
- `%autosave 0` – Disables autosave.

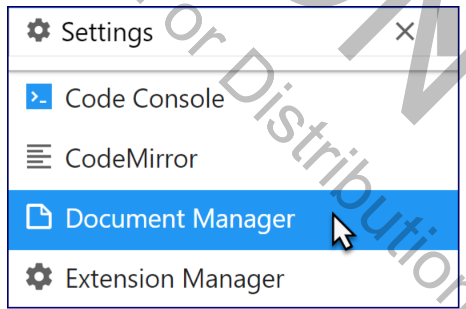
The default value for `autosave` is 120. To change that for all notebooks:

1. Select **Advanced Settings Editor** from the **Settings** menu:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

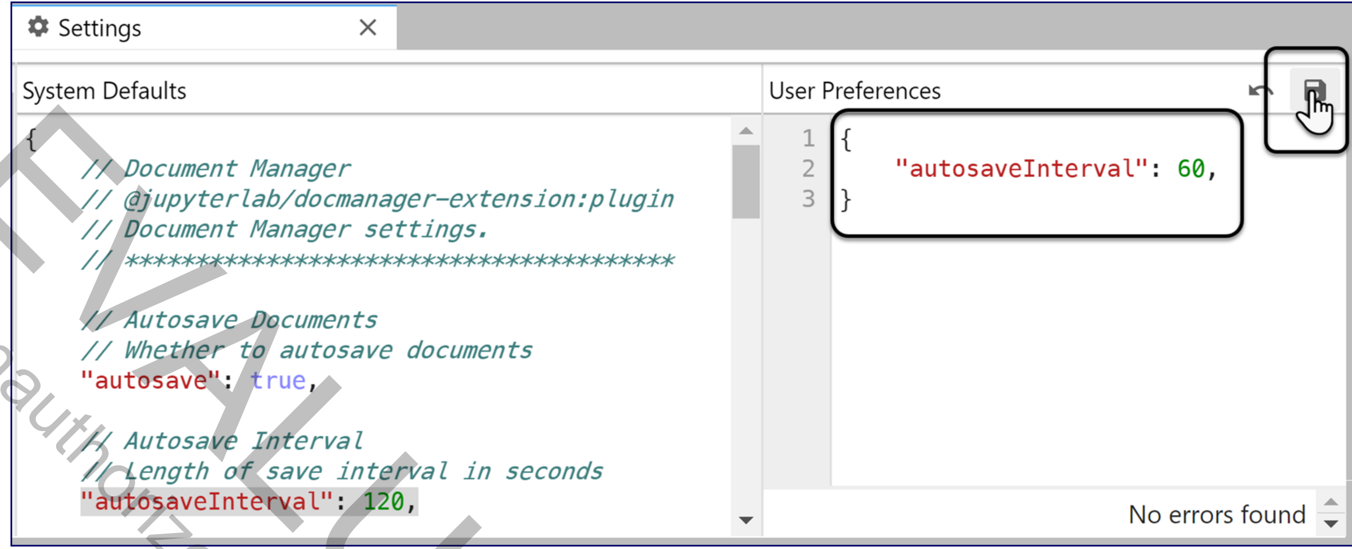


2. Select **Document Manager**:



3. Under **User Preferences**, add an `autosaveInterval` setting and click the **Save** icon:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

❖ Directory Commands

- `%cd <dir>` – Changes the current directory to `<dir>`.
- `%pwd` – Returns the current working directory.

❖ Bookmarking

The `%bookmark` command is used to create bookmarks to specific directories to make it easy to navigate within JupyterLab.

- `%bookmark <name>` – Sets a `<name>` bookmark to the current directory.
- `%bookmark <name> <dir>` – Sets a `<name>` bookmark to `<dir>`.
- `%bookmark -l` – Lists all bookmarks.
- `%bookmark -d <name>`- Removes the `<name>` bookmark.
- `%bookmark -r` – Removes all bookmarks.

Create a bookmark for the current directory:

1. Use `%pwd` to verify that you are currently in `Python/data-analysis/jupyter/Exercises`:

```
[2]:  1  %pwd

[2]: 'C:\\Webucator\\Python\\data-analysis\\jupyter\\Exercises'
```

2. Use `%cd` to navigate up to the `data-analysis` directory:

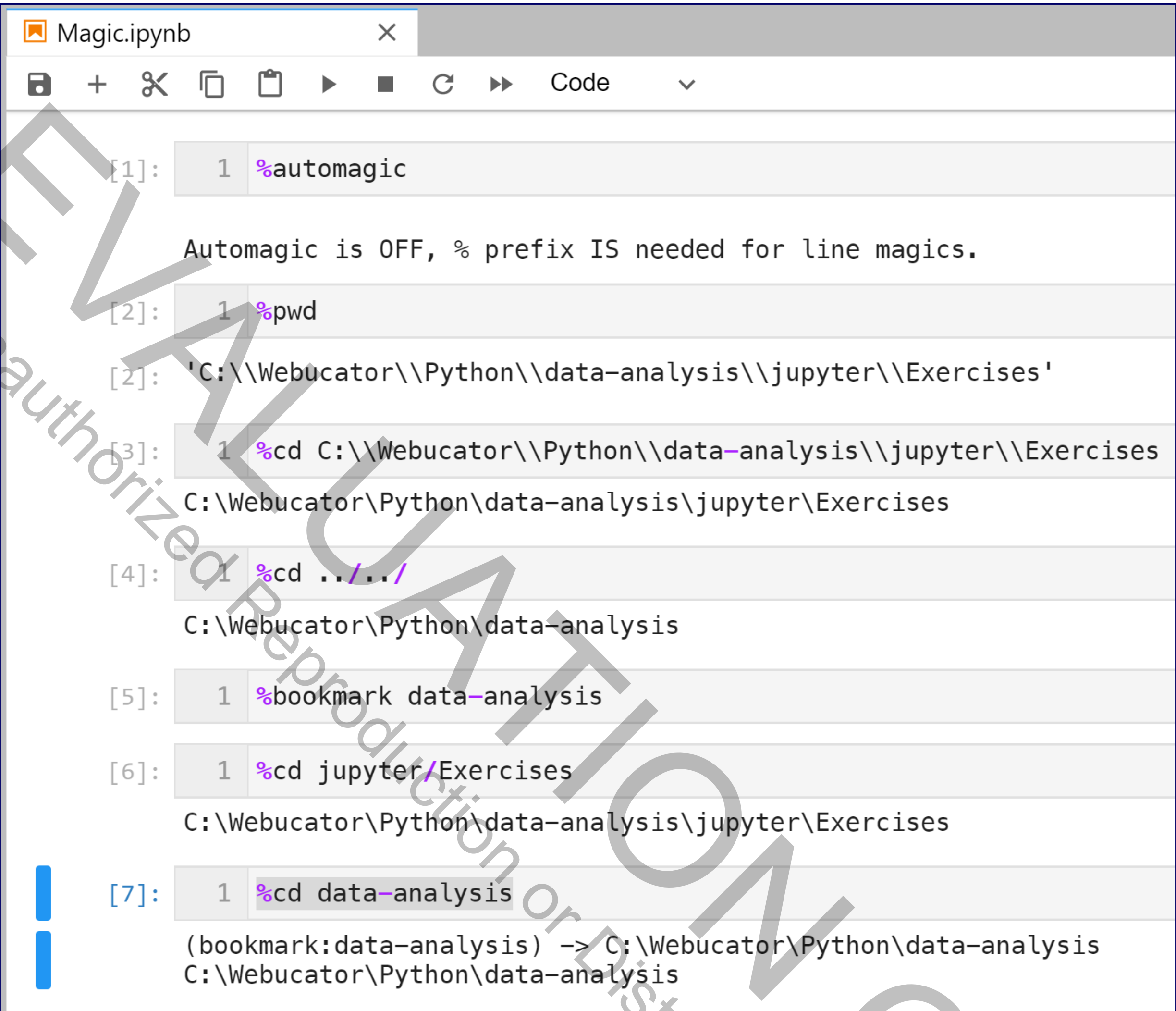
```
%cd ../../
```

3. Once you are in the `data-analysis` directory, enter `%bookmark data-analysis` and press **Shift+Enter**.
4. In the new cell that appears, enter `%cd jupyter/Exercises` and run the cell to return to the `jupyter/Exercises` directory.
5. Now, test your bookmark:

```
%cd data-analysis
```

Your notebook should looks something like this:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

❖ Command History

The `%history` command is used to get a history of your input commands. Type `%history` in a cell and press **Ctrl+Enter** to see the results.

❖ Environment Variables

The `%env` command is used to list all environment variables and their values.

You can also get specific values using `%env var`. For example, the following code would get the value of `PATH`.

```
%env PATH
```

You can assign the result to a variable. Enter and run the following code to list all the values in the PATH:

```
import platform

# Get the separator used by the operating system.
sep = ';' if platform.system() == 'Windows' else ':'

env_vars = %env
path = env_vars['PATH'].split(sep)
for i, item in enumerate(path,1):
    print(f'{i}. {item}')
```

You can also assign environment values using %env var = val or %env var val. They will exist until the kernel is stopped.

❖ Loading and Running Code from Files

- %load <source> – Loads code from <source> into cell.
- %run <source> – Runs code from <source>.

Try it out:

1. Add a new cell to your notebook.
2. Run the following to change into the Exercises directory:

   ```
   %cd data-analysis
   %cd jupyter/Exercises
   ```

3. In a new cell, enter and run %load hello_world.py. It should load the contents of hello_world.py and comment out the %load command:

   ```
   [10]:  1  # %load hello_world.py
          2  print('Hello, world!')
   ```

   You can then run the cell again to run the loaded code.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

4. In a new cell, enter `%run hello_world.py` and press **Ctrl+Enter**. This will run the file, but it won't load the file's contents into the cell:

```
[11]:   1  %run hello_world.py

        Hello, world!
```

❖ Shell Execution

- `!` – Runs shell command and captures output as a string. In a new cell, enter and run `!ping google.com`:

```
[12]:   1  !ping google.com

        Pinging google.com [172.217.4.46] with 32 bytes of data:
        Reply from 172.217.4.46: bytes=32 time=53ms TTL=117
        Reply from 172.217.4.46: bytes=32 time=54ms TTL=117
        Reply from 172.217.4.46: bytes=32 time=54ms TTL=117
        Reply from 172.217.4.46: bytes=32 time=54ms TTL=117

        Ping statistics for 172.217.4.46:
            Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
        Approximate round trip times in milli-seconds:
            Minimum = 53ms, Maximum = 54ms, Average = 53ms
```

- `%sx` (`!!` is shortcut and `%system` is an alias) – Runs shell command and captures output as a formatted list split on new line (`\n`) characters. In a new cell, enter and run `!!ping google.com`:

```
[13]:   1  !!ping google.com

[13]:  ['',
        'Pinging google.com [172.217.4.46] with 32 bytes of data:',
        'Reply from 172.217.4.46: bytes=32 time=54ms TTL=117',
        'Reply from 172.217.4.46: bytes=32 time=54ms TTL=117',
        'Reply from 172.217.4.46: bytes=32 time=60ms TTL=117',
        'Reply from 172.217.4.46: bytes=32 time=54ms TTL=117',
        '',
        'Ping statistics for 172.217.4.46:',
        '    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),',
        'Approximate round trip times in milli-seconds:',
        '    Minimum = 54ms, Maximum = 60ms, Average = 55ms']
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

❖ More Magic Commands

We have gone through some of the most useful magic commands, but there are many more.

- Use the `%lsmagic` command to get a full listing of the magic commands.
- Use `? %command_name` to get help on any specific magic command.
- For general documentation on magic commands, run `%magic` and `%quickref`.

Getting Help

There are several ways to get help in JupyterLab:

1. `object?` or `? object` – Get short documentation on an object. Try:

```
import datetime
datetime.datetime?
```

2. `object??` or `?? object` – Get extended documentation on an object. Try:

```
datetime.datetime??
```

3. `help()` – Python interactive help.
4. `help(object)` – Get standard Python documentation on an object. Try:

```
help(datetime.datetime)
```

5. Press **Shift+Tab** while inside the parentheses of a function to bring up help on that function.

```
1  datetime.datetime()
   Init signature: datetime.datetime(self, /, *args, **kwargs)
   Docstring:
   datetime(year, month, day[, hour[, minute[, second[, microsecond[,tzinfo]]]]])

   The year, month and day arguments are required. tzinfo may be None, or an
   instance of a tzinfo subclass. The remaining arguments may be ints.
   File:           c:\program files (x86)\python38-32\lib\datetime.py
   Type:           type
   Subclasses:     datetime
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Conclusion

In this lesson, you have learned to get around Jupyter notebooks.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

LESSON 2
NumPy

Topics Covered

☑ The purpose of NumPy.

☑ One-dimensional NumPy arrays.

☑ Two-dimensional NumPy arrays.

☑ Using boolean arrays to create new arrays.

> Its grey front stood out well from the background of a rookery, whose cawing tenants were now on the wing: they flew over the lawn and grounds to alight in a great meadow, from which these were separated by a sunk fence, and where an **array** of mighty old thorn trees, strong, knotty, and broad as oaks, at once explained the etymology of the mansion's designation.
>
> *– Jane Eyre, Charlotte Bronte*

Introduction

NumPy is a Python package for working with arrays and matrices (two-dimensional arrays) of numbers. Although you can do most everything NumPy can do with Python's core features, NumPy makes working with large quantities of numeric data easier and more efficient. We'll start by demonstrating how fast NumPy runs and then we'll give an overview of its most useful features.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 6: Demonstrating Efficiency of NumPy

20 to 30 minutes

Let's compare the time it takes to create a NumPy array of the first 10 squares (i.e., 0, 1, 4, 9, 16, 25, 36, 49, 64, 81) to the time it takes to create a standard Python list of those same squares.

❖ Install NumPy

If you haven't installed NumPy yet, you'll need to install it now:

1. If you have JupyterLab running, open the command prompt from which you started it and press **Ctrl+C** to stop JupyterLab. You may need to press it more than once. If you do not have JupyterLab running, open a command prompt at the `Webucator/Python/data-analysis` folder, and activate the virtual environment.
2. Run the following to install NumPy:

```
(.venv) PS …\Python\data-analysis> pip install numpy
```

3. Restart JupyterLab:

```
(.venv) PS …\Python\data-analysis> jupyter lab
```

❖ Comparing Straight Python with NumPy

We'll start with the Python list and use a list comprehension to create it.

1. Create a new notebook in `numpy/Exercises` and name it `efficiency.ipynb`.
2. In the first cell, enter and run:

```
[x**2 for x in range(10)]
```

The output should be:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

3. Generally, `NumPy` is imported like this:

```
import numpy as np
```

In a new cell, create a NumPy one-dimensional array, which is similar to a Python list, of the first 10 squares:

```
import numpy as np
np.arange(10) ** 2
```

The output should be:

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81], dtype=int32)
```

4. Now, let's use `timeit` to compare the two. IPython includes a magic `%timeit` command for using the `timeit` module. In a new cell enter and run:

```
%timeit -n 1000 [x**2 for x in range(10)]
%timeit -n 1000 np.arange(10) ** 2
```

This will output something like:

```
3.1 µs ± 221 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1.38 µs ± 110 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

As you can see, the NumPy array is created more than twice as quickly. And the difference is even more noticeable with large objects. Check out the difference when we create a list/array with 1,000 squares:

```
%timeit -n 1000 [x**2 for x in range(1000)]
%timeit -n 1000 np.arange(1000) ** 2
```

The NumPy array is created more than 100 times as quickly:

```
311 µs ± 9.81 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
2.72 µs ± 135 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

NumPy Arrays

Jupyter Notebook File

The examples in the section are included in the `numpy/Demos/numpy-basics.ipynb` notebook. Please refer to that notebook as you read through this section.

NumPy arrays are of type `numpy.ndarray`, which stands for *n-dimensional array*. The most straightforward way to create an `ndarray` is with the `array()` method, which can take any sequence. Here's a simple example:

```
nums = [1, 2, 3, 4, 5]
np.array(nums)
------------------------
array([1, 2, 3, 4, 5])
```

❖ Getting Basic Information about an Array

The following code shows how to get basic information about an array:

```
ar = np.array([1,2,3,4,5])

print(f'''Object type: {type(ar)}
Array size: {np.size(ar)},
Array dimensions: {ar.ndim},
Array data type: {ar.dtype}''')
```

This will output:

```
Object type: <class 'numpy.ndarray'>
Array size: 5,
Array dimensions: 1,
Array data type: int32
```

1. `np.size()` – The number of elements in the array.
2. `np.ndim` – The number of dimensions in the array.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

3. `np.dtype` – The data type of the elements within the array.

❖ np.arange()

As we saw earlier, an `ndarray` can also be created with the `np.arange()` method. Like Python's built-in `range()` method, `np.arange()` takes `start`, `stop`, and `step` parameters:

```
np.arange(stop)
np.arange(start, stop)
np.arange(start, stop, step)
```

The following examples illustrate:

```
stop
np.arange(10)
------------------------
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
start and stop
ar2 = np.arange(5, 11)
------------------------
array([ 5,  6,  7,  8,  9, 10])
```

```
start, stop, and step
np.arange(0, 13, 3)
------------------------
array([ 0,  3,  6,  9, 12])
```

```
start, stop, and negative step
np.arange(4, -4, -1)
------------------------
array([ 4,  3,  2,  1,  0, -1, -2, -3])
```

In addition, the `arange()` method allows for fractional steps. For example:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
np.arange(1, 5, .5)
```

will produce the following array:

```
array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

All Elements Must Be the Same Data Type

All elements in `ndarrays` must be of the same data type. In the array above, notice that there are decimal points at the end of the whole numbers. That indicates that these are floats. Take a look at the following:

```
ar5 = np.arange(1, 5, 1)
ar6 = np.arange(1, 5, .5)
ar7 = np.arange(1, 5, dtype=float)

type(ar5[0]), type(ar6[0]), type(ar7[0])
------------------------
(numpy.int32, numpy.float64, numpy.float64)
```

Notice that `ar5` is an array of integers and `ar6` is an array of floats. For those two arrays, Python implicitly determined the data type:

- If any of the arguments is a float, then it creates an array of floats.
- If all of the arguments are integers, then it creates an array of integers.

To explicitly set the data type, pass in a `dtype` as we do for `ar7` above. Here it is again:

```
ar7 = np.arange(1, 5, dtype=float)
```

If you try to set an element in an array of integers to a float, the value will be coerced into an integer. Consider the following:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
ar = np.arange(1, 10)
ar[5] = 1.5

type(ar[0]), type(ar[5]), ar
------------------------
(numpy.int32, numpy.int32, array([1, 2, 3, 4, 5, 1, 7, 8, 9]))
```

Notice that we tried to set `ar[5]` to `1.5`, but it was coerced into the integer `1`.

❖ Similar to Lists

In some ways, `ndarray`s are like `list`s. For example, they are iterators:

```
my_list = [x**2 for x in range(10)]
for i in my_list:
    print(i, end=' ')

print('\n' + '='*25) # Separator

my_array = np.arange(10) ** 2
for i in my_array:
    print(i, end=' ')
------------------------
0 1 4 9 16 25 36 49 64 81
=========================
0 1 4 9 16 25 36 49 64 81
```

❖ Different from Lists

In other ways, `ndarray`s behave differently from `list`s. For example, consider the following operations on a list:

```
my_list = [1, 2, 3, 4, 5]
print(my_list)
print(my_list + my_list) # Appends list to itself.
print(my_list * 3) # Repeats list 3 times.
------------------------
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

1. When two `list`s are “added together” with the plus (+) operator, one list is appended to the other.
2. When a list is “multiplied” by a number with the asterisk (*) operator, the list is repeated that number of times.

Compare this to how the same operations work with `ndarray`s:

```
ar = np.arange(1, 6)
print(ar)
print(ar + ar) # Adds ar[0] to ar[0], ar[1] to ar[1], etc.
print(ar * 3) # Multiplies each element of array by 3.
-------------------------
[1 2 3 4 5]
[ 2  4  6  8 10]
[ 3  6  9 12 15]
```

1. When two `ndarray`s are “added together” with the plus (+) operator, each element of the first array is added to the corresponding element of the second array to create a new array of the same length.
2. When an `ndarray` is “multiplied” by a number with the asterisk (*) operator, that scalar is *broadcast* across all element of the array, meaning that each element of the array is multiplied by that number to return a new array of the same length.

Mathematical operations can also be performed on arrays and lists together. Consider the following three operations:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Multiplication

```
ar1 = np.arange(1, 6)
multiplier = 2
my_list = [2, 2, 2, 2, 2]
ar2 = np.array(my_list)

result1 = ar1 * multiplier
result2 = ar1 * my_list
result3 = ar1 * ar2

result1, result2, result3
------------------------
(
    array([ 2,  4,  6,  8, 10]),
    array([ 2,  4,  6,  8, 10]),
    array([ 2,  4,  6,  8, 10])
)
```

In each case, each element of the original array (`ar1`) is multiplied by 2 to create a new array of the same length.

Other mathematical operations work in the same way:

Original Array

```
ar = np.arange(1, 6)
```

Addition

```
ar + 2 # Add 2 to each element of array.
------------------------
array([3, 4, 5, 6, 7])
```

Subtraction

```
ar - 2 # Subtract 2 from each element of array.
------------------------
array([-1,  0,  1,  2,  3])
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
Division
ar / 2 # Divide each element of array by 2.
------------------------
array([0.5, 1. , 1.5, 2. , 2.5])
```

```
Squares
ar ** 2 # Square each element of array.
------------------------
array([ 1,  4,  9, 16, 25], dtype=int32)
```

```
Square Roots
ar ** .5 # Get square root of each element of array.
------------------------
array([1.        , 1.41421356, 1.73205081, 2.        , 2.23606798])
```

❖ Universal Functions

Imagine you have two ranges and you want to multiply all the elements in one range by the corresponding elements in the other range. One way to do this is to use the built-in `zip()` function, which takes any number of iterables and creates an iterator of tuples from the elements at the same index in each of the iterables. For example, zipping the following two lists:

```
[1, 2, 3]
[4, 5, 6]
```

will create an iterator with the following tuples:

```
(1, 4), (2, 5), (3, 6)
```

The following code shows how to use `zip()` to create a list of products of elements in two ranges:

```
Zipping two iterables
r1 = range(1, 10)
r2 = range(10, 1, -1)
zip(r1, r2)
------------------------
<zip at 0x134dea08>
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Displaying the zip as a list

```
list(zip(r1, r2))
------------------------
[(1, 10), (2, 9), (3, 8), (4, 7), (5, 6), (6, 5), (7, 4), (8, 3), (9, 2)]
```

Creating a list of multiples from the zip

```
[i1 * i2 for i1, i2 in zip(r1, r2)]
------------------------
[10, 18, 24, 28, 30, 30, 28, 24, 18]
```

There are many functions in NumPy that behave like this by default. Functions that operate on `ndarrays` one element at a time (i.e., element-wise) are called *universal functions*. NumPy includes many such built-in methods. For example, the following code multiplies one array by another element-wise:

Multiplying two ndarrays

```
ar1 = np.arange(1, 10)
ar2 = np.arange(10, 1, -1)
np.multiply(ar1, ar2)
------------------------
array([10, 18, 24, 28, 30, 30, 28, 24, 18])
```

NumPy's universal functions can take `ndarrays` or *array-like* objects. For example, `np.add()` can be used to add the elements of a list to the elements of a range, like this:

Adding array-like objects

```
my_list = [1, 3, 5, 7]
r = range(0, 4)
np.add(my_list, r)
------------------------
array([ 1,  4,  7, 10])
```

The following lists show some (but not all[1]) of these methods. Some of these methods are demonstrated in the `numpy/Demos/numpy-basics.ipynb` Jupyter notebook under "Some Examples of Universal Functions":

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

1. See `https://numpy.org/doc/stable/reference/ufuncs.html` for additional documentation on NumPy's universal functions.

Element-wise Mathematical Methods

1. `np.add(ar1, ar2)` – Same as `ar1 + ar2`.
2. `np.subtract(ar1, ar2)` – Same as `ar1 - ar2`.
3. `np.multiply(ar1, ar2)` – Same as `ar1 * ar2`.
4. `np.divide(ar1, ar2)` – Same as `ar1 / ar2`.
5. `np.floor_divide(ar1, ar2)` – Same as `ar1 // ar2`.
6. `np.negative(ar)` – Same as `ar * -1`.
7. `np.power(ar1, ar2)` – Same as `ar1 ** ar2`.
8. `np.remainder(ar1, ar2)`, `np.mod(ar1,ar2)` – Same as `ar1 % ar2`.
9. `np.absolute(ar)` – Same as `abs(ar)`.
10. `np.rint(ar)` – Rounds each element of `ar` to the nearest integer.
11. `np.sign(ar)` – Returns 1 for positive numbers, -1 for negative numbers, and 0 for 0.
12. `np.exp(ar)` – Calculates the exponential of each element in `ar`.
13. `np.log(ar)` – Returns the natural logarithm of each element in `ar`.
14. `np.log2(ar)` – Returns the base-2 logarithm of each element in `ar`.
15. `np.log10(ar)` – Returns the base 10 logarithm of each element in `ar`.
16. `np.sqrt(ar)` – Return the positive square-root of each element in `ar`.
17. `np.square(ar)` – Return the square of each element in `ar`.
18. `np.reciprocal(ar)` – Return the reciprocal of each element in `ar`.
19. `np.floor(ar)` – Returns the floor of each element.
20. `np.ceil(ar)` – Returns the ceiling of each element.
21. `np.trunc(ar)` – Returns the truncated value of each element.

Element-wise Trigonometric Methods

1. `np.sin(ar)` – Returns the sine of each element.
2. `np.cos(ar)` – Returns the cosine of each element.
3. `np.tan(ar)` – Returns the tangent of each element.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

4. `np.arcsin(ar)` – Returns the inverse sine of each element.
5. `np.arccos(ar)` – Returns the inverse cosine of each element.
6. `np.arctan(ar)` – Returns the inverse tangent of each element.
7. `np.hypot(ar1, ar2)` – Given "legs" of a right triangle, returns hypotenuse.
8. `np.deg2rad(ar)` – Converts degrees to radians for each element.
9. `np.rad2deg(ar)` – Converts radians to degrees for each element.

Element-wise Boolean Methods

A *boolean* `ndarray` is an array that just contain boolean (`True`/`False`) values.

1. `np.isreal(ar)` – Returns a boolean `ndarray`: `True` if element is real.
2. `np.isnan(ar)` – Returns a boolean `ndarray`: `True` if element is `NaN`.

Element-wise Comparison Methods

Comparison methods also return boolean `ndarrays`.

1. `np.greater(ar1, ar2)` – Same as `ar1 > ar2`.
2. `np.greater_equal(ar1, ar2)` – Same as `ar1 >= ar2`.
3. `np.less(ar1, ar2)` – Same as `ar1 < ar2`.
4. `np.less_equal(ar1, ar2)` – Same as `ar1 <= ar2`.
5. `np.not_equal(ar1, ar2)` – Same as `ar1 != ar2`.
6. `np.equal(ar1, ar2)` – Same as `ar1 == ar2`.
7. `np.logical_and(ar1, ar2)` – Compares each element of `ar1` to the corresponding element in `ar2` and returns an `ndarray` of booleans: `True` if *they are both true*, `False` otherwise.
8. `np.logical_or(ar1, ar2)` – Compares each element of `ar1` to the corresponding element in `ar2` and returns an `ndarray` of booleans: `True` if *at least one is true*, `False` otherwise.
9. `np.logical_xor(ar1, ar2)` – Compares each element of `ar1` to the corresponding element in `ar2` and returns an `ndarray` of booleans: `True` if *one and only one is true*, `False` otherwise.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

NumPy: `all()` and `any()`

These two methods are not element-wise. They return a single boolean value.

1. `np.all(ar)` or `ar.all()` – Returns `True` if *all* elements in array are `True`.
2. `np.any(ar)` or `ar.any()` – Returns `True` if *any* elements in array are `True`.

NumPy: `max()` and `min()`

These two methods are not element-wise. They return a single value.

1. `np.max(ar)` or `ar.max()` – Returns the largest element in the array.
2. `np.min(ar)` or `ar.min()` – Returns the smallest element in the array.

Some Examples of Universal Functions

Setup

```
ar_ints = np.arange(1, 6)
ar_floats = np.arange(1, 7.5, 1.5)
ar_ints, ar_floats
------------------------
(array([1, 2, 3, 4, 5]), array([1. , 2.5, 4. , 5.5, 7. ]))
```

The following table shows how the two arrays line up:

`ar_ints`	`ar_floats`
1.	1
2.5	2
4.	3
5.5	4
7.	5

np.add()

```
np.add(ar_ints, ar_floats)
------------------------
array([ 2. ,  4.5,  7. ,  9.5, 12. ])
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

np.negative()

```
np.negative(ar_ints)
------------------------
array([-1, -2, -3, -4, -5])
```

np.square()

```
np.square(ar_ints)
------------------------
array([ 1,  4,  9, 16, 25], dtype=int32)
```

np.hypot()

```
np.hypot(ar_ints, ar_ints)
------------------------
array([1.41421356, 2.82842712, 4.24264069, 5.65685425, 7.07106781])
```

np.greater()

```
np.greater(ar_ints, ar_floats)
------------------------
array([False, False, False, False, False])
```

np.ceil()

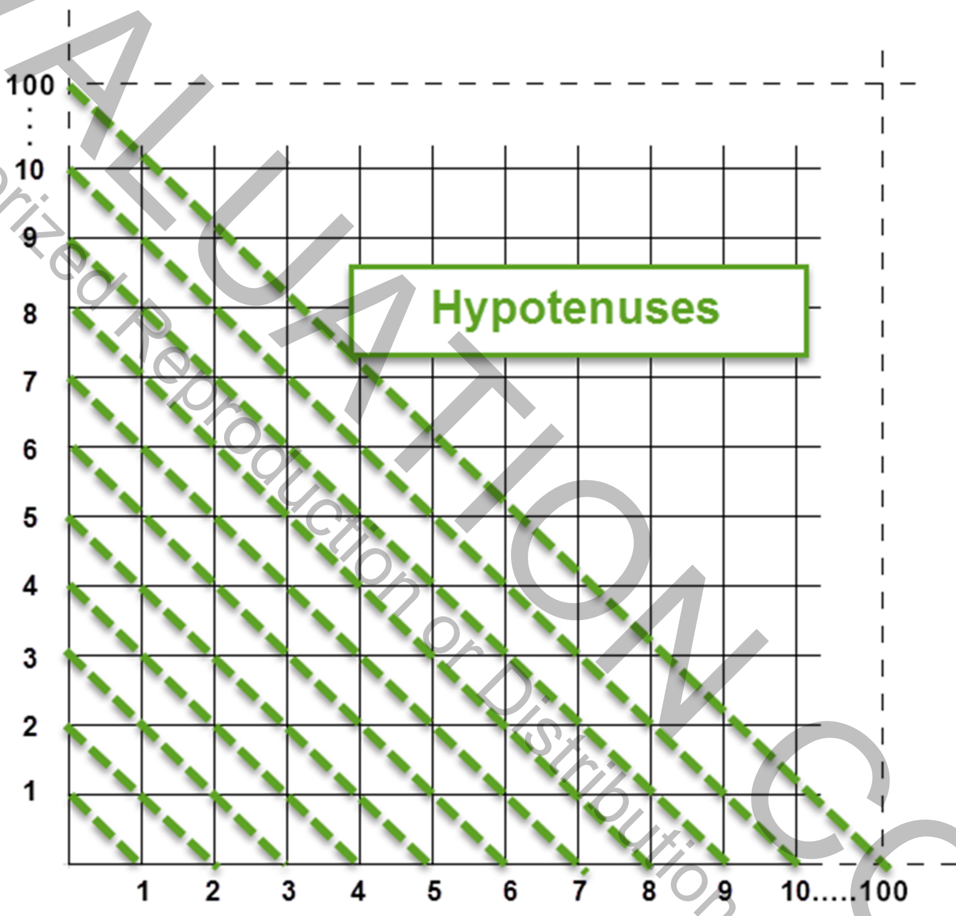
```
np.ceil(ar_floats)
------------------------
array([1., 3., 4., 6., 7.])
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 7: Multiplying Array Elements

15 to 25 minutes

In this exercise, you will calculate the hypotenuses shown in the following diagram first without NumPy and then with NumPy, and then you will compare the efficiency of each method:



The formula to calculate the hypotenuse of a right triangle is:

$$hypotenuse = \sqrt{leg1^2 + leg2^2}$$

If `leg1` and `leg2` are equal, as in the diagram above, the formula can be simplified to:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

$$hypotenuse = \sqrt{2 * leg^2}$$

In Python, this can be written like this:

```
import math
math.sqrt(2 * leg**2))
```

1. In JupyterLab, navigate to `numpy/Exercises` and open the `hypotenuses.ipynb` notebook.
2. The first cell contains the necessary `import` statements.
3. In the second cell, write code to create a list of 100 hypotenuses for right triangles with legs of lengths between 1 to 100. Do not use NumPy.
4. In the third cell, using NumPy, write code to create an array of 100 hypotenuses for right triangles with legs of lengths between 1 to 100.
5. Once you have tested your code above, use `timeit` to compare the efficiency of the two solutions. Prefix `timeit` with two percentage signs (`%`) to test all the code in a cell, like this:

   ```
   %%timeit -n 1000
   # Your code
   # goes here
   ```

   The NumPy solution should run much faster.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Solution

numpy/Solutions/hypotenuses.ipynb

Import Modules

```
import math
import numpy as np
```

Calculate Hypotenuses without NumPy

```
legs = range(1, 101)
hypotenuses = [(math.sqrt(2 * leg**2)) for leg in legs]
------------------------
[1.4142135623730951,
 2.8284271247461903,
 4.242640687119285,
 5.656854249492381,
 7.0710678118654755,
 8.48528137423857,
 9.899494936611665,
 11.313708498984761,
 12.727922061357855,
…]
```

Calculate Hypotenuses with NumPy

```
r = np.arange(1, 101)
hypotenuses = np.hypot(r, r)
hypotenuses
------------------------
array([  1.41421356,   2.82842712,   4.24264069,   5.65685425,
         7.07106781,   8.48528137,   9.89949494,  11.3137085 ,
        12.72792206, …]
```

Compare Efficiency: Without NumPy

```
%%timeit -n 1000
legs = range(1, 101)
hypotenuses = [(math.sqrt(2 * leg**2)) for leg in legs]
------------------------
49.3 µs ± 469 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Compare Efficiency: With NumPy

```
%%timeit -n 1000
r = np.arange(1, 101)
hypotenuses = np.hypot(r, r)
------------------------
4.59 µs ± 311 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The NumPy code runs ten times more quickly.

Multi-dimensional Arrays

Jupyter Notebook Examples

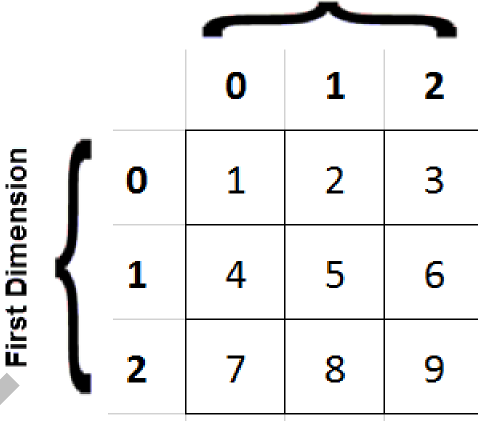
Open `numpy/Demos/multi-dimensional-arrays.ipynb` to follow along with the examples in this section.

NumPy arrays are n-dimensional, meaning they can be one, two, three, or more dimensions. However, you will mostly work with one-dimensional or two-dimensional arrays. Two-dimensional arrays (also called *matrices*) are sequences of sequences. The following code shows the most basic way of creating a two-dimensional array:

```
mda = np.array([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ])
```

The following diagram provides a visual of the two-dimensional array created above:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



The following list shows some properties for getting quick information about an array:

- `mda.size` – The number of elements in the array:

```
mda.size
------------------------
9
```

- `mda.ndim` – The number of dimensions in the array:

```
mda.ndim
------------------------
2
```

- `mda.shape` – The shape of the array (i.e., the length of each dimension):

```
mda.shape
------------------------
(3, 3)
```

- `np.size(mda, 0)` – Get the number of rows using `np.size()`, passing in the array as the first argument and the first dimension (`0`) as the second argument:

```
np.size(mda, 0)
------------------------
3
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

- `np.size(mda, 1)` – Get the number of columns using `np.size()`, passing in the array as the first argument and the second dimension (1) as the second argument:

```
np.size(mda, 1)
------------------------
3
```

The following list shows examples for accessing elements of an array:

1. The rows are the first dimension. To get row `r` of two-dimensional array `mda`, use `mda[r]`. A row of a two-dimensional array is a one-dimensional array:

```
row2 = mda[1]
row2
------------------------
array([4, 5, 6])
```

2. A "cell" within a row is just an element within a one-dimensional array. To get the second element in the third row, get the third row and then get the second element in that row:

```
row3 = mda[2]
row3cell2 = row3[1]
row3cell2
------------------------
8
```

3. To get the "cell" in one step, use `mda[r, c]` (e.g., `mda[2, 1]`):

```
row3cell2 = mda[2, 1]
row3cell2
------------------------
8
```

4. The cells within the rows (or the "columns") are the second dimension. Getting the *nth* "column" is the equivalent of getting the *nth* element of every row:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

A. To do this, we use slicing syntax. First, let's get all the rows:

```
all_rows = mda[0:len(mda)]
all_rows
------------------------
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

B. In the code above, we specify `0` as the `start` index and `len(mda)` as the `stop` index. But we can leave both of these out as they are the default values. So, the following will also get all the rows:

```
all_rows = mda[:]
all_rows
------------------------
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

C. To get only the second column, get the second element (element `1`) of every row. That means limiting the second dimension:

```
col2 = mda[:, 1]
col2
------------------------
array([2, 5, 8])
```

That can look a little confusing, so we'll make it a little bigger:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

5. To get the last two columns of the matrix, take a slice of the second dimension:

```
cols2and3 = mda[:, 1:]
cols2and3
------------------------
array([[2, 3],
       [5, 6],
       [8, 9]])
```

You can also use negative indexing:

```
cols2and3 = mda[:, -2:]
cols2and3
------------------------
array([[2, 3],
       [5, 6],
       [8, 9]])
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Exercise 8: Retrieving Data from an Array

15 to 30 minutes

The purpose of this exercise is to give you practice retrieving data from a NumPy array.

If you haven't installed `ipywidgets` yet, you should install it now (see page 4). It is used to create the quiz questions.

1. In JupyterLab, navigate to `numpy/Exercises` and open the `quiz.ipynb` notebook.**Be sure to open the notebook and not the Python (`.py`) file.**
2. Run the first cell to set up the notebook. This will:
    A. Import `numpy`.
    B. Create a NumPy array called `ar`.
    C. Generate the quiz.
3. Run each cell to load the questions, do your work in the indicated cells, and answer the questions. Use the hints if (and only if) you need them.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Solution

numpy/Solutions/quiz.ipynb

Here are some possible ways of arriving at the answers. There may be others:

1. How many elements are in the array? **120**

```
ar.size
```

2. How many dimensions is the array? **2**

```
ar.ndim
```

3. How many rows are in the array? **15**

```
np.size(ar, 0)
```

4. How many columns are in the array? **8**

```
np.size(ar, 1)
```

5. What is the sum of all the elements in the 4th row? **4161**

```
ar[3].sum()
```

6. What is the sum of all the elements in the 5th column? **359600**

```
ar[:, 4].sum()
```

7. What is the value in the 4th column of the 5th row? **-4083**

```
ar[4, 3]
```

8. What is highest value in the array? **69620**

```
ar.max()
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

More on Arrays

❖ Modifying Parts of an Array

We saw earlier that we can operate on arrays on an element-wise basis. We can do the same with subsets of an array. For example, the following code shows how to add `2` to the last two elements of the last two rows. We use negative indexing to do this:

```
mda = np.array([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ])

mda[-2:, -2:] += 2
```

This will only change the last two elements of the array:

```
------------------------
array([[ 1,  2,  3],
       [ 4,  7,  8],
       [ 7, 10, 11]])
```

❖ Adding a Row Vector to All Rows

When you add (or multiply or subtract, etc.) a vector (or an array-like object) to a multi-dimensional array, the operation is broadcast through the whole array. The following code illustrates this:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
mda = np.array([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ])

v = [10, 20, 30]
mda + v
------------------------
array([[11, 22, 33],
       [14, 25, 36],
       [17, 28, 39]])
```

Notice that `10` was added to all the first-column values, `20` was added to the second-column values, and `30` was added to the third-column values.

❖ More Ways to Create Arrays

The following methods provide additional ways to create NumPy arrays:

1. `np.zeros(shape, dtype=float)` - Creates an array of zeros of specified shape, which must be a tuple:

```
ar1 = np.zeros(shape=(6, 3), dtype=float)
ar1
------------------------
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

2. `np.reshape(ar, newshape)` - `newshape` is a tuple containing the dimensions of the new array. Returns a reshaped array. Use `-1` for any of the dimensions in `newshape` to have the number be inferred based on the length of the array and the value of the other dimension:

```
ar1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
ar2 = np.reshape( ar1, (2, -1) )
ar2
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

This will reshape the array, so that the first dimension (the rows) has a length of 2 and the second dimension is as long as it has to be to fit all the values:

```
------------------------
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

A. The `reshape()` method can also be called on an `ndarray` object: `ar.reshape(rows, cols)`. It returns a new array and does not modify the original array:

```
ar1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
ar3 = ar1.reshape(-1, 2)
ar3
------------------------
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])
```

3. `np.linspace(start, stop, num)` - Creates a `num`-element one-dimensional array of evenly distributed numbers between (and including) `start` and `stop`:

```
ar4 = np.linspace(4, 6, 5)
ar4
```

This gives us five numbers equally distributed between 4 and 6:

```
------------------------
array([4. , 4.5, 5. , 5.5, 6. ])
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Using Boolean Arrays to Get New Arrays

Jupyter Notebook Examples

Open `numpy/Demos/using-boolean-arrays.ipynb` to follow along with the examples in this section.

A common trick with NumPy arrays is to use a boolean array to filter an array. For example, suppose you have an `ndarray` of integers from which you want to create a new `ndarray` of its odd integers. Here's how you can do that:

1. Create an `ndarray` of random integers:

   ```
   np.random.seed(1)
   ar = np.random.randint(100, size=10)
   ar
   ------------------------
   array([37, 12, 72,  9, 75,  5, 79, 64, 16,  1])
   ```

2. Create a boolean `ndarrray` showing which values of the integer array are odd:

   ```
   ar_isodd = ar % 2 == 1
   ar_isodd
   ------------------------
   array([ True, False, False, True, True, True, True,False, False, True])
   ```

3. **Here's the magic**: Use the boolean array to get just the odd integers from the integer array:

   ```
   ar[ar_isodd]
   ------------------------
   array([37,  9, 75,  5, 79,  1])
   ```

   Notice that the new array only includes the numbers at indexes that contained `True` values.

And here a shortcut for doing the same thing:

```
ar[ar % 2 == 1]
------------------------
array([37,  9, 75,  5, 79,  1])
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Random Number Generation

Jupyter Notebook Examples

Open `numpy/Demos/random-sampling.ipynb` to follow along with the examples in this section.

The first step to generating random numbers with NumPy is to create a generator:

```
rng = np.random.default_rng()
```

Random Floats

Generate random floats between `0` and `1` using `rng.random(size=None)`. If `size` is passed in, an array is returned; otherwise, a single float is returned.

```
rng.random()
------------------------
0.34991410735415185
```

```
rng.random(5)
------------------------
array([0.83759985, 0.37047308, 0.77579737, 0.39871428, 0.44692712])
```

Different numbers will be generated each time you run the code.

Random Integers

Generate random integers between `min` (inclusive) and `max` (exclusive) with `rng.integers(min, max, size=None)`. If `size` is passed in, an array is returned; otherwise, a single integer is returned.

```
rng.integers(0, 100)
------------------------
43
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

```
rng.integers(0, 100, 5)
------------------------
array([56, 70, 46, 35, 25], dtype=int64)
```

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Random Choices

Generate random choices from an array with `rng.choice(ar, size=None, replace=True)`:

```
ar = np.array(['rock', 'paper', 'scissors'])
rng.choice(ar)
------------------------
'paper'
```

The `replace` parameter determines if values from the input array can be selected more than once. By default, `replace` is `True`, meaning values can be selected multiple times:

```
rng.choice(['rock', 'paper', 'scissors'], 4)
------------------------
array(['scissors', 'paper', 'scissors', 'paper'], dtype='<U8')
```

If `replace` is `False`, values from the passed-in array can only be selected once. In this case, the size cannot be larger than the length of the passed-in array:

```
rng.choice(ar, 3, replace=False)
------------------------
array(['rock', 'paper', 'scissors'], dtype='<U8')
```

The `size` Parameter

For the `random()`, `integers()`, and `choice()` methods, if `size` is an integer, a one-dimensional array of random numbers will be created. If it is a tuple, a multi-dimensional array will be created:

```
rng.random((3, 5))
------------------------
array([[66, 84, 60, 48,  2],
       [26, 19, 65, 53, 26],
       [42, 32, 44, 73, 22]], dtype=int64)
```

```
rng.integers(1, 100, (3, 5))
------------------------
array([[0.96423402, 0.10286424, 0.79970035, 0.02674376, 0.84124845],
       [0.55431763, 0.69622166, 0.13279084, 0.38528933, 0.60191166],
       [0.67754984, 0.29560972, 0.98600287, 0.40912323, 0.70336712]])
```

```
rng.choice(['rock', 'paper', 'scissors'], (5, 2))
------------------------
array([['rock', 'rock'],
       ['rock', 'paper'],
       ['scissors', 'scissors'],
       ['scissors', 'paper'],
       ['paper', 'rock']], dtype='<U8')
```

Regenerating the Same Random Numbers

Pass in a `seed` when creating the generator to be able to recreate the same random numbers. Run the following code multiple times. Each time, the same three values will be returned:

```
rng = np.random.default_rng(seed=42)
rng.random(), rng.integers(1, 100), rng.choice(['rock', 'paper', 'scissors'])
------------------------
(0.7739560485559633, 65, 'paper')
```

* * *

Exploring NumPy Further

There is much more you can do with NumPy arrays, including:

1. Combine arrays with `hstack()`, `vstack()`, `dstack()`, `concatenate()`, `column_stack()`, and `row_stack()`.
2. Split arrays with `split()`, `hsplit()`, `vsplit()`, and `dsplit()`.
3. Flatten arrays with `ravel()`.

See `https://numpy.org/doc/stable/reference/routines.array-manipulation.html` for documentation on these functions.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Conclusion

In this lesson, you have learned to work with one-dimensional and two-dimensional NumPy arrays to work efficiently with large amounts of data.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com

9-38-00324-000-03-04-21