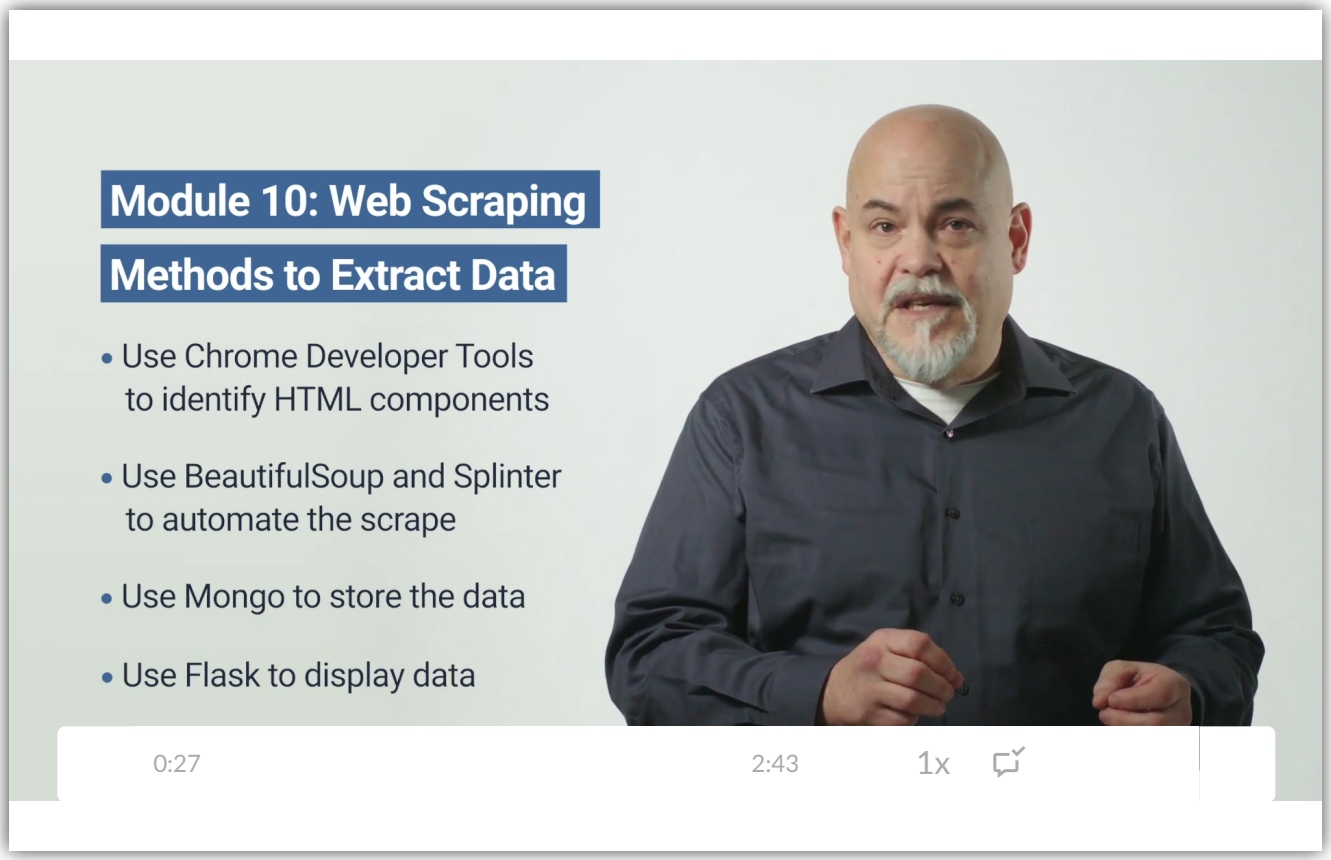
10.0.1: Web Scraping to Extract Online Data



10.0.2: Module 10 Roadmap

Looking Ahead

In this module, you'll automate a web browser to visit different websites to extract data about the Mission to Mars. You'll store it in a NoSQL database, and then render the data in a web application created with Flask. The completed work will be displayed in your portfolio, which you will also create.

Web scraping is a method used by organizations worldwide to extract online data for analysis. Large companies employ web scraping to assess their reputations or track their competitors' online presence.

On a smaller scale, web scraping automates tedious tasks for personal projects. For example, if you're collecting current news on a specific subject, web scraping can make it a simple process. Instead of visiting each website and copying an article, a web scraping script will perform those actions and save the scraped data for later analysis.

What You Will Learn

By the end of this module, you will be able to:

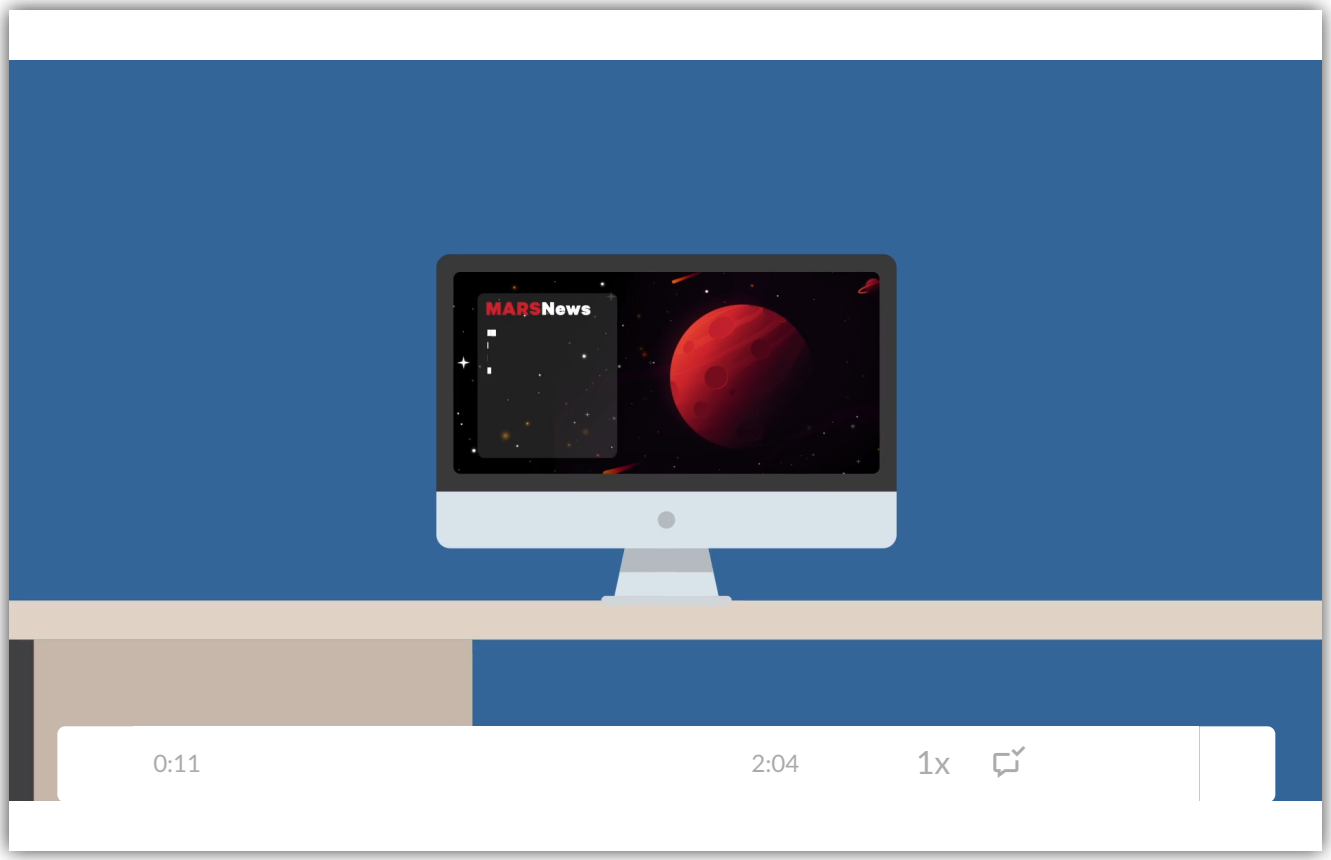
- Gain familiarity with and use HTML elements, as well as class and id attributes, to identify content for web scraping.
- Use BeautifulSoup and Splinter to automate a web browser and perform a web scrape.
- Create a MongoDB database to store data from the web scrape.
- Create a web application with Flask to display the data from the web scrape.
- Create an HTML/CSS portfolio to showcase projects.
- Use Bootstrap components to polish and customize the portfolio.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction (15 minutes)
- Get Started Using Web Scraping Tools (1 hour)
- Open the Window to the Internet (2 hours)
- Automate a Web Browser and Perform a Web Scrape (3 hours)
- Access Data in MongoDB (1 hour)
- Display Data With Flask (3 hours)
- Make It Pretty (3 hours)
- Show It Off (2 hours)
- Application (5 hours)

10.0.3: Tools for Scraping



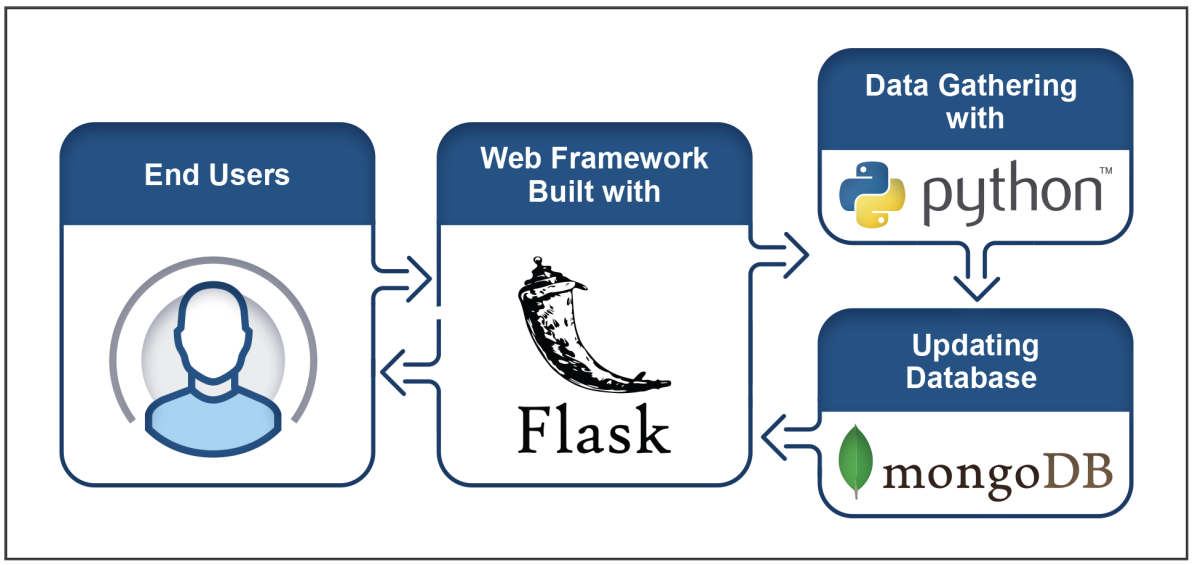
10.1.1: Install Your Tools

Robin is pretty excited about putting together this web-scraping project. Being able to get the latest news and updates with the click of a button? That's a really useful tool for someone who wants to keep up with the Mission to Mars.

First things first, though—preparation. Robin needs to download a few libraries and tools that she'll need when she's ready to start scraping data: Splinter to automate a web browser, BeautifulSoup to parse and extract the data, and MongoDB to hold the data that has been gathered.

With all of the information available on the web, people are able to stay up-to-date with almost every subject out there. What if a person wants to narrow their focus to a single topic? Are there tools that would make gathering the latest data easier?

Robin, who loves astrology and wants to work for NASA one day, has decided to use a specific method of gathering the latest data: web scraping. Using this technique, she has the ability to pull data from multiple websites, store it in a database, then present the collected data in a central location: a webpage.



> **Important**
> Before installing new tools, open your terminal and make sure your Python coding environment is active.

Let's get the tools we'll need to complete this web-scraping project: Splinter, BeautifulSoup, and MongoDB.

If you are using Windows, you'll need to take a few extra steps during this setup. Instructions are provided in the following sections.

Splinter

Splinter is the tool that will automate our web browser as we begin scraping. This means that it will open the browser, visit a webpage, and then interact with it (such as logging in or searching for an item). To do all of this, we'll need to install Splinter and ChromeDriver.

To install Splinter, open your terminal and make sure your coding environment is active. Then, run the command `pip install splinter`. Once that installation is complete, we'll install ChromeDriver.

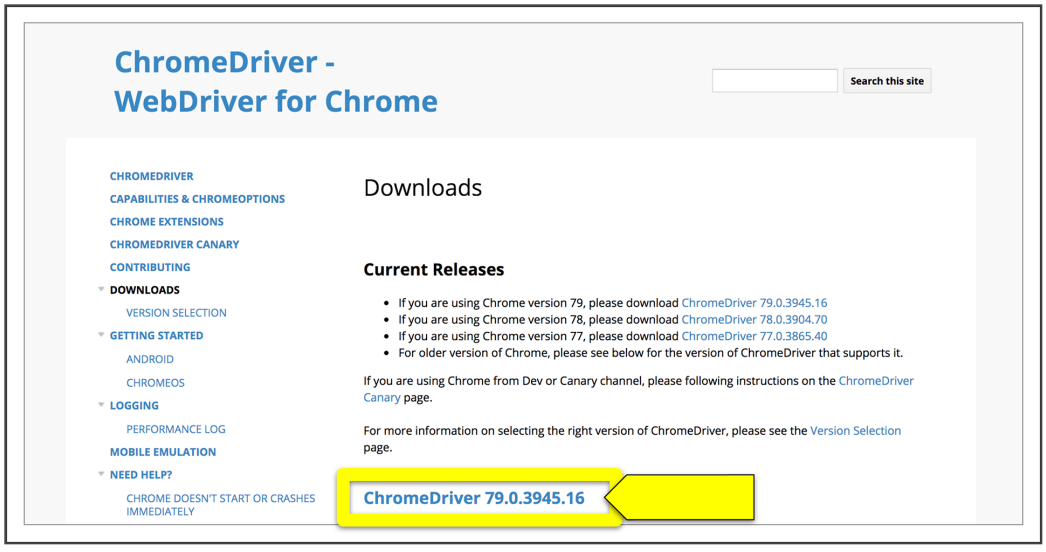
> **Important**
>
> To successfully use the ChromeDriver and scrape website data, Chrome will need to be installed and used as the primary browser.

You'll find current installation instructions for all operating systems on **Splinter's website (https://splinter.readthedocs.io/en/latest/index.html)** .

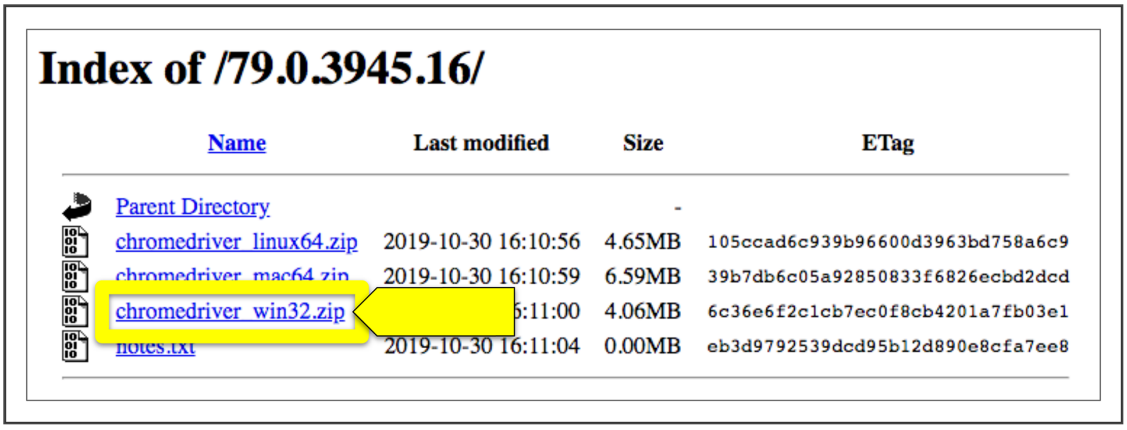
Install ChromeDriver on Windows

Visit the **ChromeDriver (https://sites.google.com/a/chromium.org/chromedriver/downloads)** webpage. Note that ChromeDriver updates really often, so the exact version you are using might be slightly different than the screenshots in these instructions. The screenshots below are for users running Chrome version 79. Make sure match your download to the version of Chrome you're currently using. Otherwise, you'll run into an error. It will still work. Follow these steps:

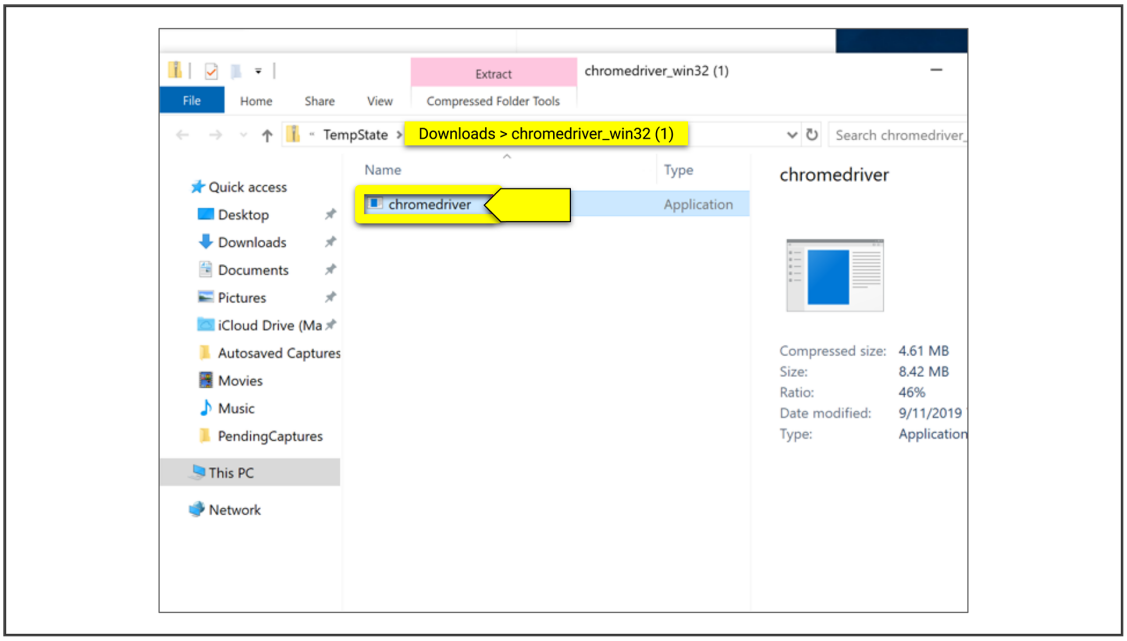
1. Click ChromeDriver 79.0.3945.16 (or the file that matches your version of Chrome).



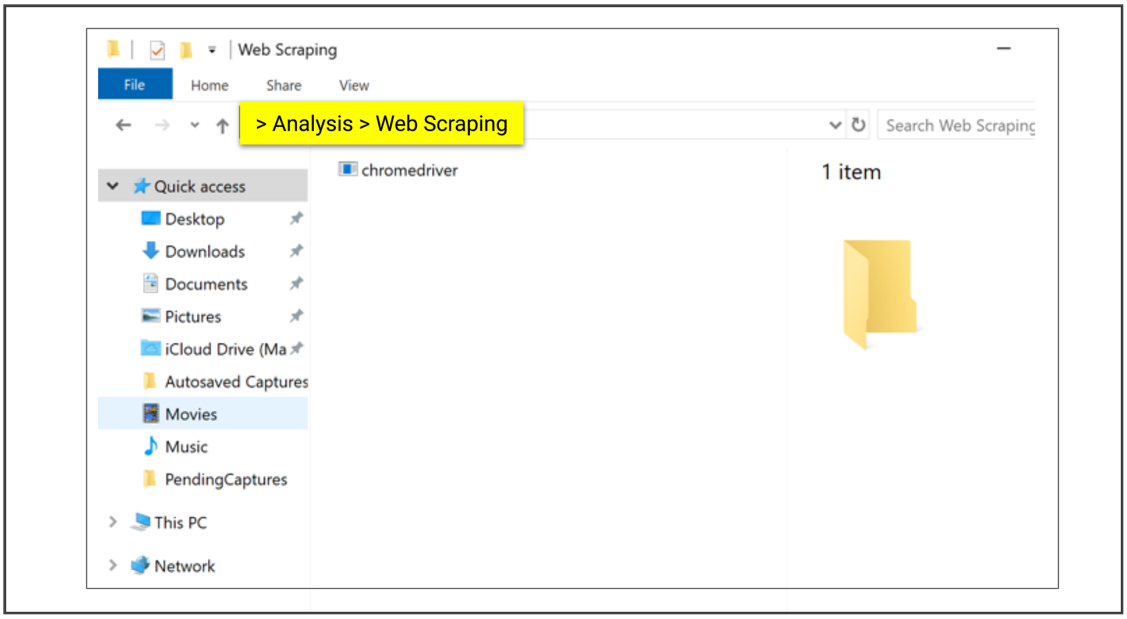
2. Click "chromedriver_win32.zip" to download ChromeDriver for Windows.



3. Extract the executable program file.



4. Place it in the same folder as your Python script.



Install ChromeDriver on macOS

The easiest way to install ChromeDriver on a Mac is through Homebrew. You can verify your installation by running `brew -v` in your terminal. If Homebrew is installed, simply run `brew install chromedriver` from the terminal. If you get an error instead of a version number, visit the **Homebrew website (https://brew.sh/)** to install it, and then run the command.

If Homebrew is installed, simply run `brew cask install chromedriver` from the terminal.

Verify your installation by running `chromedriver --version`.

BeautifulSoup

To install BeautifulSoup, run the command `pip install bs4` in your terminal. Make sure the environment you plan to work from is active first.

MongoDB

MongoDB (also known as Mongo) is a document database that thrives on chaos. Well, maybe it's not that extreme, but it is far more flexible when it comes to storing data than a structured database such as SQL. It's able to handle smaller, more personal projects as well as larger-scale projects that a company might require. For this module, Mongo is a better choice than SQL because the data we'll scrape from the web isn't going to be uniform. For example, how would we break down an image into rows and columns? We can't. But Mongo will store and access it as a document instead.

Let's first open a terminal window (make sure your virtual environment is active) and execute the "pip install pymongo" command. PyMongo is the tool that allow

developers to use Python with Mongo.

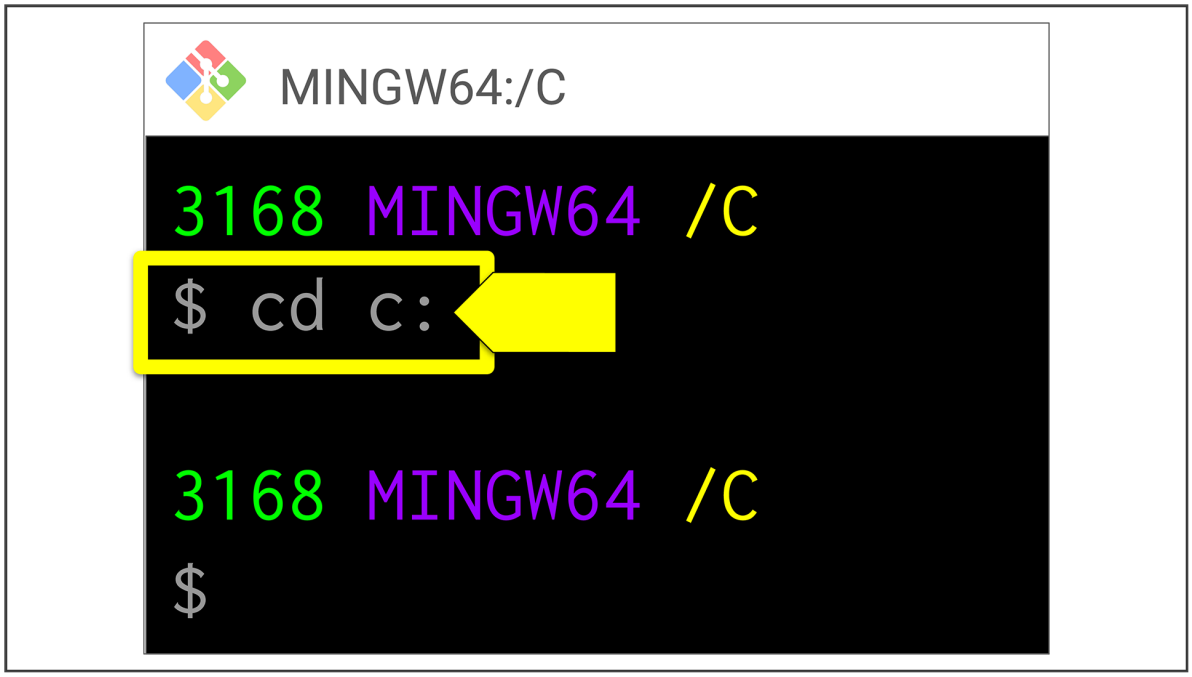
Now let's install MongoDB.

Windows

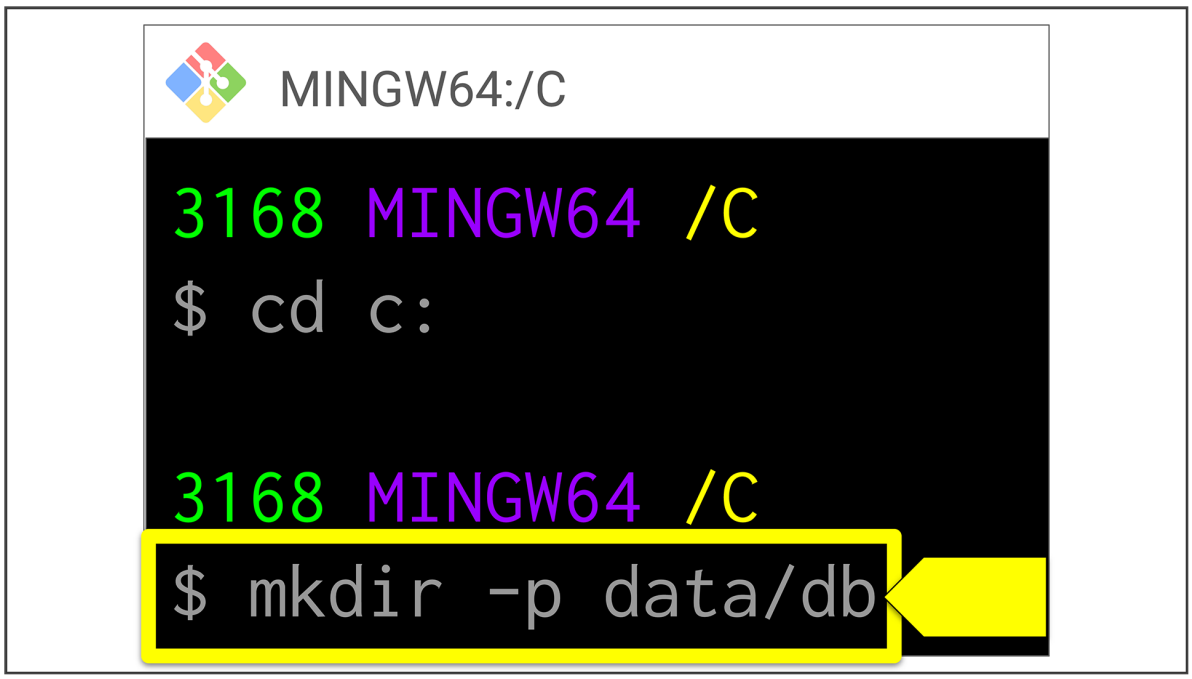
To install Mongo on your Windows computer, follow the instructions in the **official documentation (https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/)**. Be sure to follow all of the steps listed for installing the MongoDB Community Edition.

Adding a Data Directory to Root

You'll also need to add the data directory to your root directory on your computer; otherwise, MongoDB won't run. In your terminal, navigate to your root directory with `cd c:`.



Next, we want to make a directory with the command `mkdir -p data/db`.



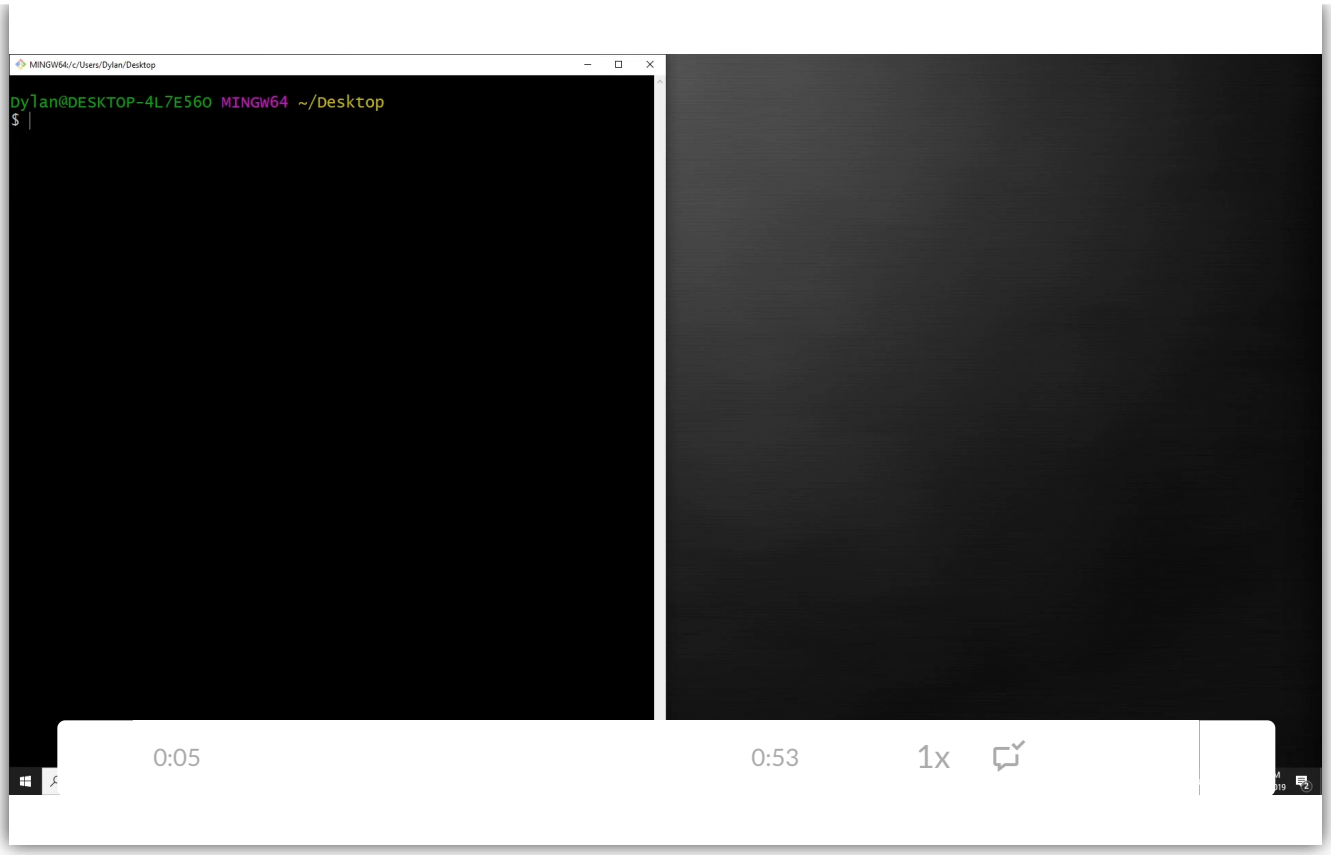
This is the default location for Mongo's databases. Without this, Mongo won't have a designated spot to create collections and store data. It will refuse to run, no matter how many cookies we offer it.

Configuration

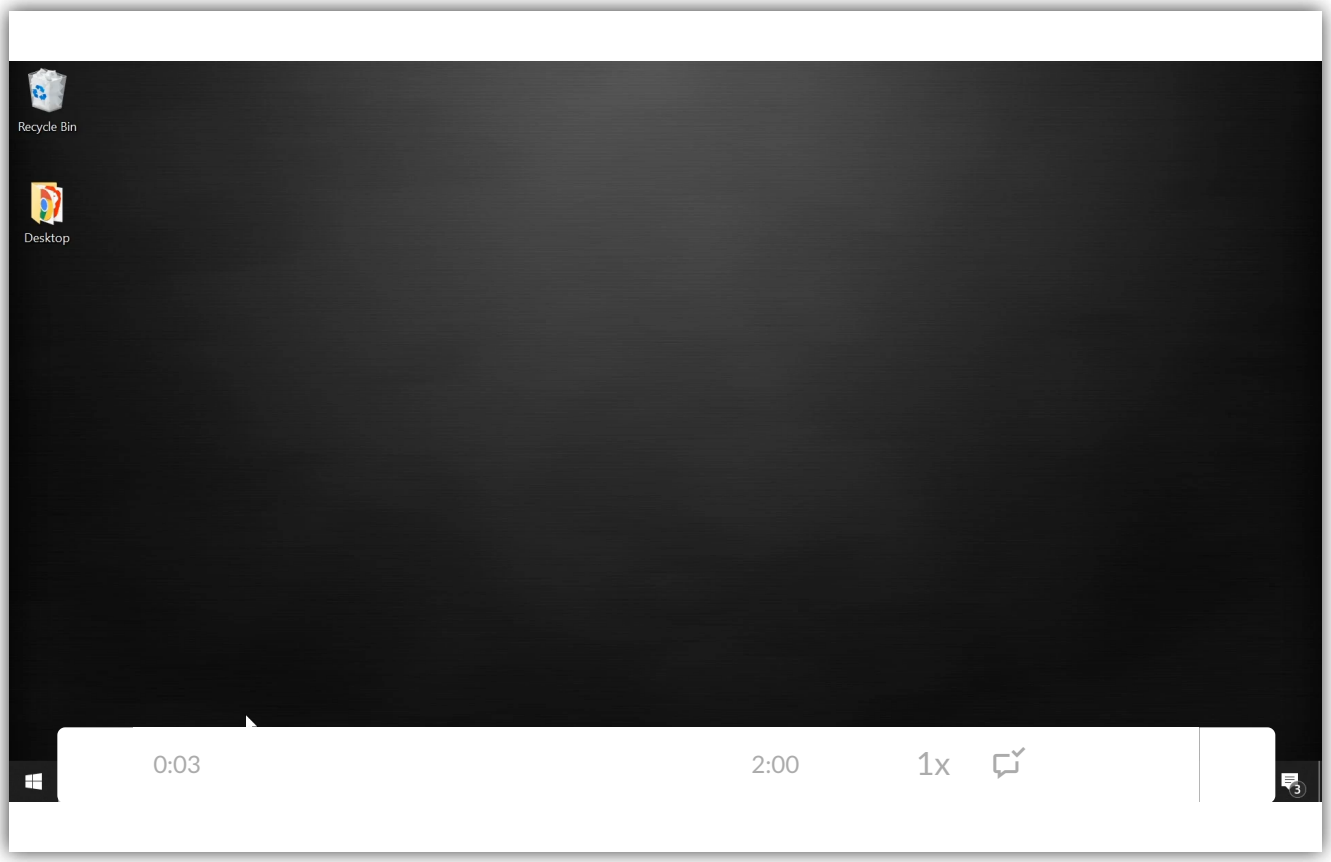
You'll also want to add MongoDB's path to the PATH environment variables for your computer so that you can run and launch MongoDB easily from the Bash command line.

First, locate the directory where you installed MongoDB. This is likely `C:\\Program Files\\MongoDB\\Server\\\\bin`. Copy this directory to your clipboard with CTRL + C.

The following video will aid in locating Mongo in your directory, then copying its location on your computer to add to the PATH.



Once the file path is highlighted, we can add it to our Path environment variable. To update your Path, dig into your security settings on your computer. See the instructions below.



To test if this worked, close your current Git Bash window, and then reopen it and run this command: `mongod`. If Mongo continues to run in the terminal, it's been successfully installed—great job!

> **Hint**
> There is no "b" in the `mongod` command.

Troubleshooting

If `mongod` didn't run and, instead, your Bash threw a "command not found" error, make sure you added MongoDB's **bin** directory to your PATH variable. When you navigate through your Windows Explorer to the MongoDB folder, make sure you click all the way through into the **bin** folder, then copy the path. Then, close out Git Bash and try running `mongod` again.

If `mongod` starts but closes after a series of prompts, make sure you created the `/data/db` directory in your root. MongoDB cannot run without this directory.

macOS

Installation

To install Mongo on your macOS computer, follow the instructions in the **official documentation (https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/)**. Be sure to follow all of the steps listed for installing the MongoDB Community Edition.

Here are a few important tips during this installation:

- It's best to use the Homebrew `brew` package for this installation. In your terminal, the installation command will start with `brew tap`.

  Once Mongo is installed, we want to run it as a **macOS service** because doing so will automatically set system **ulimit (https://docs.mongodb.com/manual/reference/ulimit/#ulimit-settings)** values correctly.

Hint: Mac users will use this line to create a database instance: `brew services start mongodb-community@4.2` instead of `mongod`.

> ### Create a Repo
>
> Navigate to GitHub and create a new repository to hold the code for this module. Name the new repo “Mission-to-Mars” and clone it into your class folder. Remember to add, commit, and push your code as you work through the module.

10.2.1: Use HTML Elements

Robin has gotten all of her tools installed and tested Mongo to make sure it's ready for data, but before she can really start pulling it off of the web, she needs to be able to identify where the data is stored within the HTML code.

Every webpage is built using hypertext markup language, more commonly known as HTML. Some sites are more sophisticated than others, but they all have the same basic structure. Each element of a page, such as a title or a paragraph, is wrapped in a tag. These tags are specific to the element they are holding, and there are many different types and layouts in existence.

Let's take a closer look at HTML tags.

Think of a webpage as a window into the internet. HTML is the glass, boards, and blinds on that window. Just like there are many sizes and shapes to windows, each webpage has been customized to present users with a view into a different topic. Consider a weather report delivered through a weather site. Think of a news source or social media platform. Each of these examples are all built using custom HTML. Our first step will be to explore that design so that we can write a script that knows what it's looking at when it interacts with a webpage.

Open VS Code and create a file named `index.html`. This file can be saved to your desktop because it's just for practice.

In this blank HTML file put an exclamation point on the first line and press Enter. This should autofill the editor to contain everything we need for a basic HTML page.

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <meta http-equiv="X-UA-Compatible" content="ie=edge">
 <title>Document</title>
</head>
<body>
 </body>
</html>
```

In this code, each line of code is wrapped in a tag, such as `<title />`.

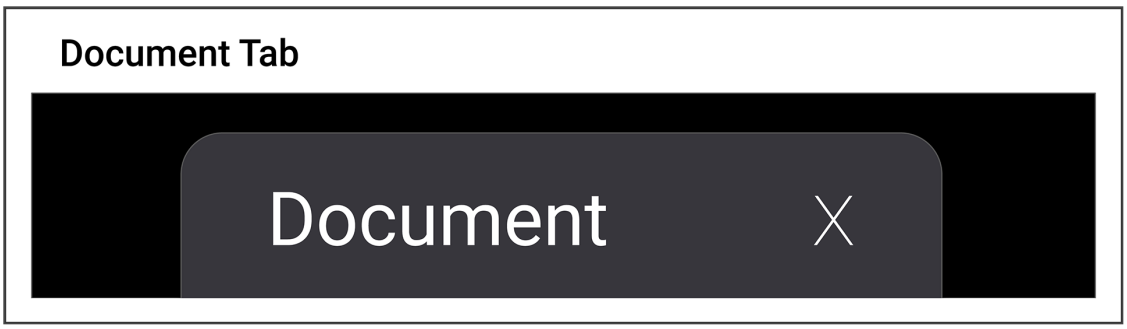
> **Rewind**
> HTML is a coding language used for creating webpages. It's built using specific tags and arranging them in a nested order, a bit like building blocks. For example, if we wanted a header and a paragraph in the same section of a webpage, we would nest `<h1 />` and `<p />` tags inside a `<div />` tag, with the `<div />` tag acting as a box to hold the other pieces.
>
> ```
> 
> <h1>Hello, world!</h1>
> This is a great beginning.
> 
> ```
>
> Each element has an opening tag and an ending tag. They look almost

> identical with the only difference being the slash (/) that signals the end of that particular line of HTML code.

These tags are what define each element of this webpage. We can open this page right now, but it will be blank because we haven't added anything to it yet. Let's take a closer look at how these different elements work together.

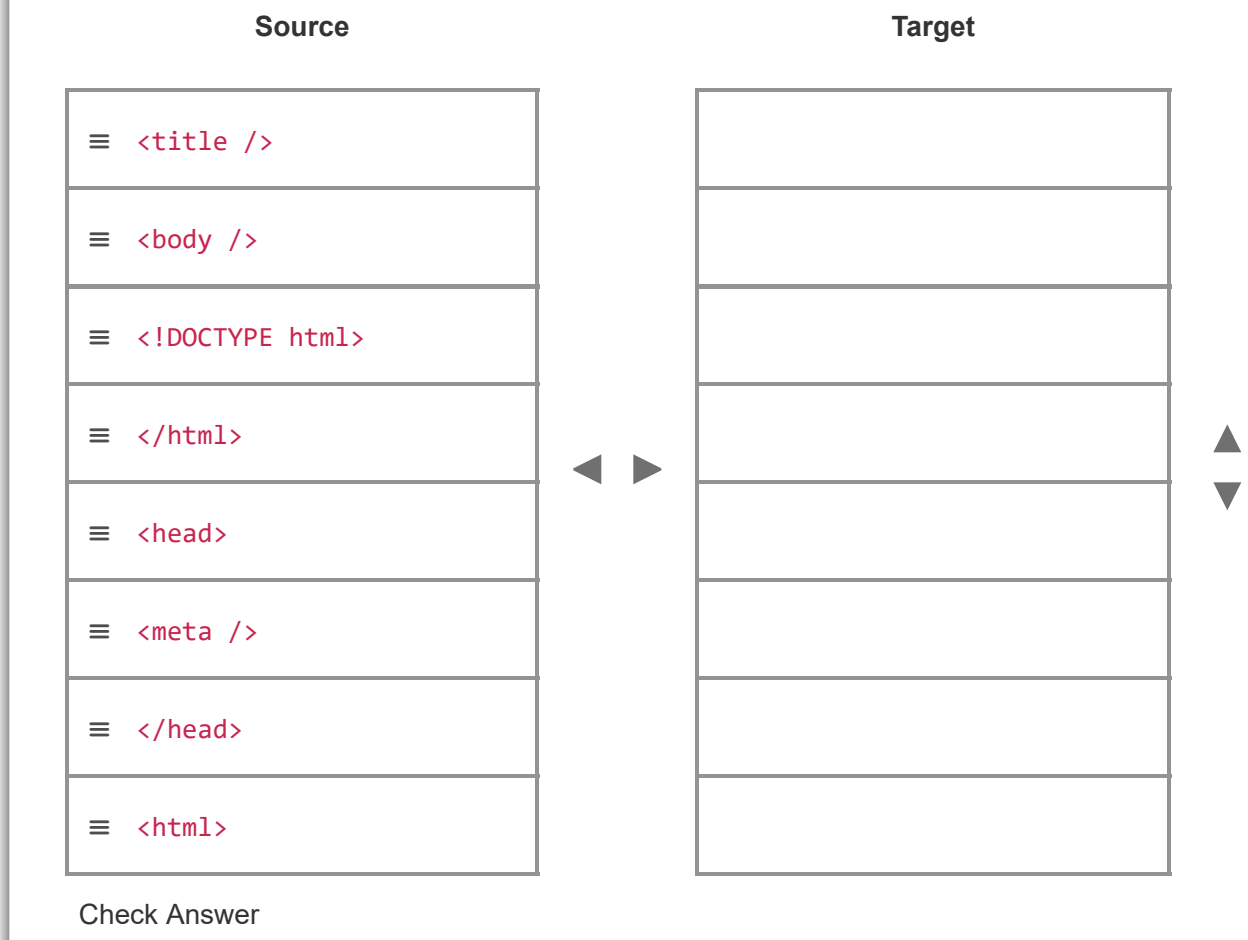
Let's define each HTML tag shown in the graphic:

1. `<!DOCTYPE html>` is a declaration, not a tag. It tells web browsers in which HTML version the document is written. This should always be the first line in an HTML document.
2. `<head>` is the opening tag that serves as a container for the setup elements. Jupyter Notebook imports occur in the top cell whereas Python imports occur at the top of the code. HTML imports (e.g., a stylesheet or a library) will be within the `<head>`
3. `<meta>` is short for "metadata" and tells the web browser basic information, such as page width.
4. `<title>` and `</title>` are the opening and closing tags that serve as a container for the page title displayed on the tab at the top of your web browser. In the example above, the title is "Document" and would appear like so in the browser:



5. `</head>` is the closing tag for the `<head>` tag, much like the end of a code block in Python.
6. `<body>` and `</body>` are opening and closing tags. They also serve as a container, but for data we can see (navigation menus, lists, and paragraphs).
7. `<html lang=”en”>` and `</html>` are opening and closing tags that serve as a container for all elements within an HTML page.

Arrange the following items in order:



IMPORTANT

Nesting is when HTML elements are contained within other elements. Picture a set of nesting dolls with each nested in proper order, by design, into the largest doll. It is the same for HTML tags—they must be in the correct order to not break the design of the webpage.

Keeping code clean and easy to read is an important part of being a developer. How would you keep your HTML in good visual shape?

- ○ Lots of white space between lines of code. This helps separate the sections.
- ○ Keep all lines at the same level of indentation; that way, everything's neatly lined up.
- ○ Use indentation to keep the tags in order—this helps show how and where elements are nested.

Check Answer

Finish ▸

An easy way to keep the tags in visual order is by using indentation. Containers nested within other containers are indented by two to four spaces. This helps to keep our code clean and easy to understand.

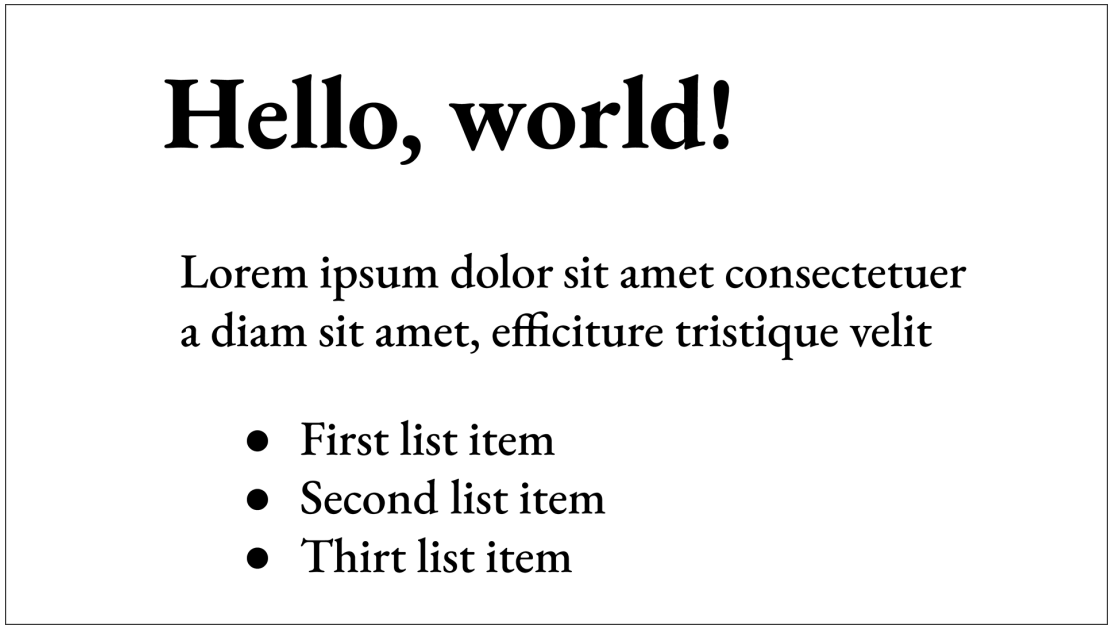
Let's take another look at this webpage, only with a few more elements added to it:

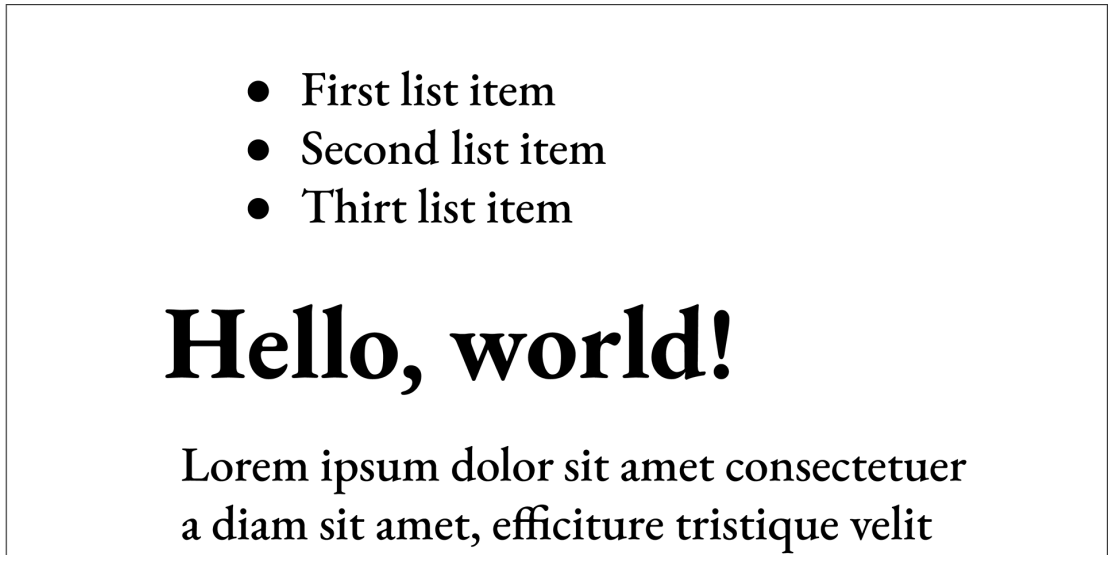
```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0"
   <meta http-equiv="X-UA-Compatible" content="ie=edge" />
   <title>Document</title>
 </head>
```

```
</head>
<body>
  <h1>Hello, world!</h1>
  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin aliqu
    iaculis lorem non sollicitudin. Fusce elementum ac elit finibus auct
    Curabitur orci sem, accumsan a diam sit amet, efficitur tristique ve
  </p>
  <ul>
    <li>First list item</li>
    <li>Second list item</li>
    <li>Third list item</li>
  </ul>
</body>
</html>
```

There are several more tags within the `<body />` container. Add this new code to your `index.html` file and save it. Then, open the file by navigating to it in your repo folder and double-clicking it. Now you have a simple static webpage open in your browser, built from scratch. It's not super exciting yet, but that's okay. It's the innards of the page we're focusing on right now.

Which of the following images look like the webpage you've just opened?





- First list item
- Second list item
- Thirt list item

Lorem ipsum dolor sit amet consectetuer a diam sit amet, efficiture tristique velit

Hello, world!

Check Answer

Finish ▸

Let's review the new tags:

- `<h1 />` is a first-level header. The text in this tag will be displayed bigger and bolder than the rest of the page's text. There are many different headers available to use, from `h1` to `h5`, with `h1` returning the largest text.
- `<p />` is a paragraph tag, currently holding **lorem ipsum** sentences. (lorem ipsum is dummy text used to stage websites). More can be read about it on the **Lorem Ipsum reference website (https://www.lipsum.com/)**.
- `<ul />` is an **unordered list**.
- `<li />` is a **list item**.

What does it mean when the `<li />` tags are inside the `<ul />` tags?

- It means the tags are nested. The `<ul />` tags are a container for the `<li />` tags.
- It's just there to make the code look nice.
- The tags are nested, but it doesn't really matter which order they're nested in. Nesting is mainly for formatting purposes.

Check Answer

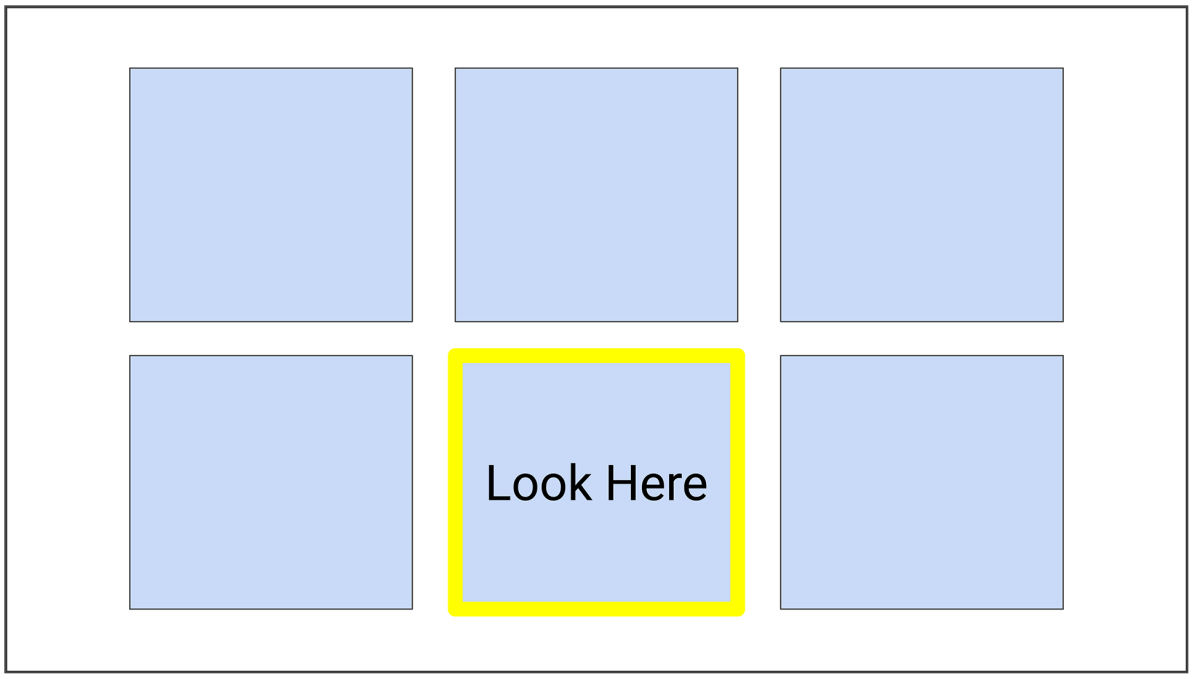
Finish ▸

This is only a small taste of how many tags exist out there. Remember, these tags are all part of website customization. Without the variety available to use, websites would look plain and uninspired. The sites that Robin intends to scrape data from are far more sophisticated, using many more combinations of tags than what we've discussed here. Understanding the basic layout and how nesting and containers work is an important part of successful web scraping.

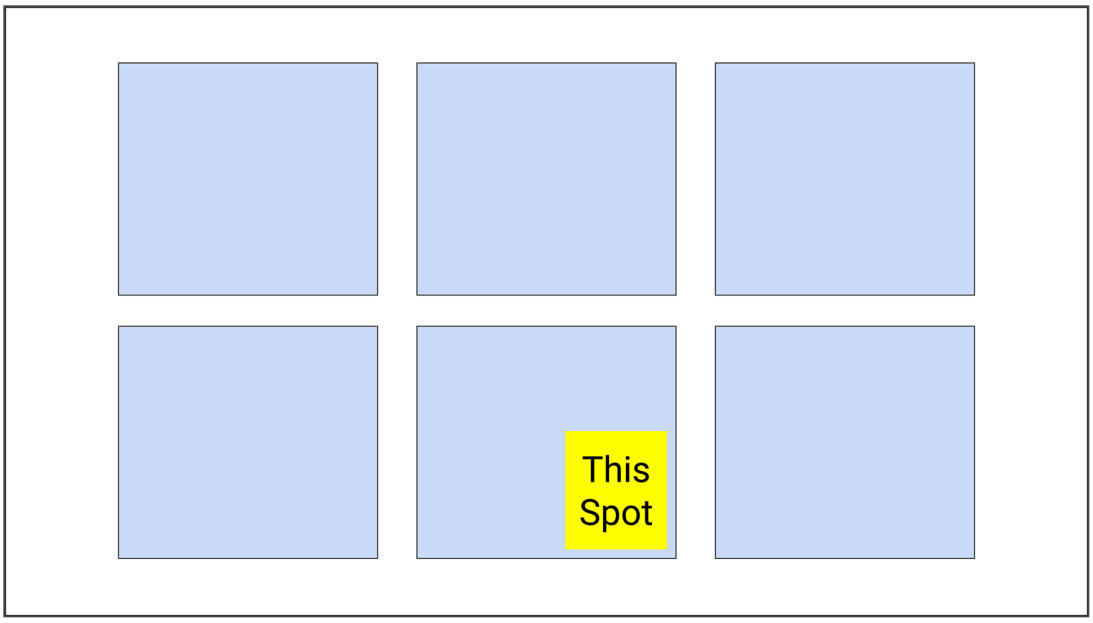
We know that when we scrape data from the web, we're simply pulling specific data from websites we've chosen. How do we specify the data? Let's say we want

the latest news article from a Mars website. Before we can program our script to pull that data, we have to tell it where to look. Basically, our script would say, “Look in this `<div />` tag, then look inside that for a `<p />` tag.”

For example, if the webpage was a window, we would use our script to direct it to a certain pane.



Once it found that pane, we can also tell it to look even closer, such as at the bottom-center pane.



That’s a simple way of putting it, but we’ll dive more deeply into how web scraping works soon. Visit W3Schools’ developer site for an extensive list of **HTML tags (https://www.w3schools.com/tags/tag_comment.asp)** .

10.2.2: Using Chrome Developer Tools

Robin has installed all of her tools and researched different HTML tags in preparation for her web-scraping project. She's really ready to jump in and start gathering data, but even with her initial research, she realized she wasn't quite ready to start scraping. The HTML components on the first site she visited quickly became more complex than she expected.

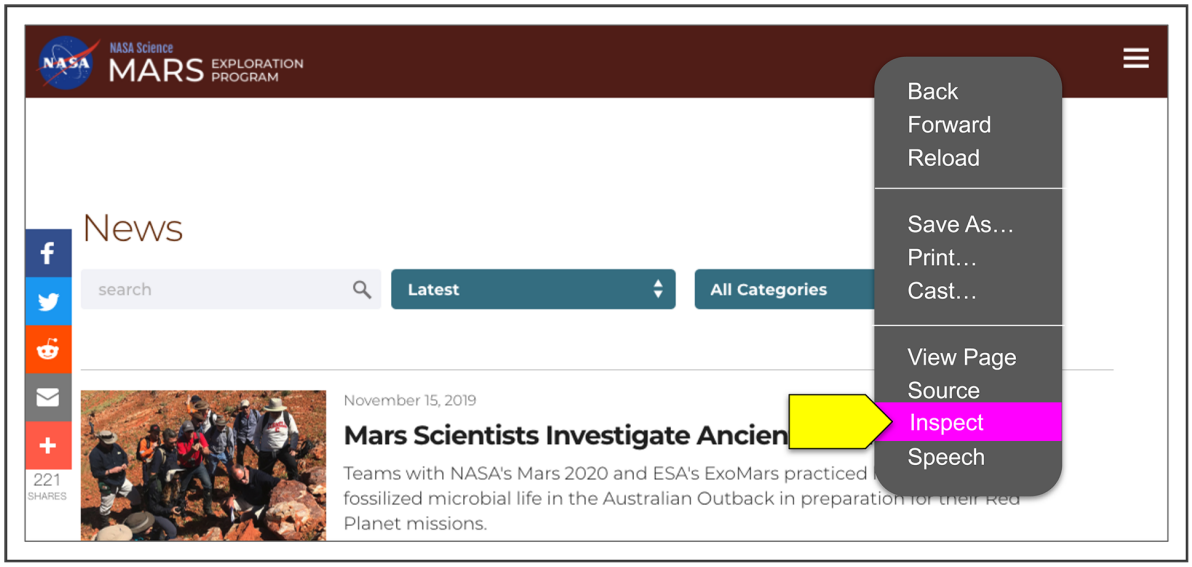
Instead, Robin has decided to practice identifying specific data using Chrome Developer Tools (also known as DevTools). This tool allows developers to look at the structure of any webpage. Not only that, but there's a search function as well. This should help make more sense of the tags and components that hold the data she's looking for.

Let's visit one of the websites Robin plans to use and take a peek at its structure, then practice finding different components.

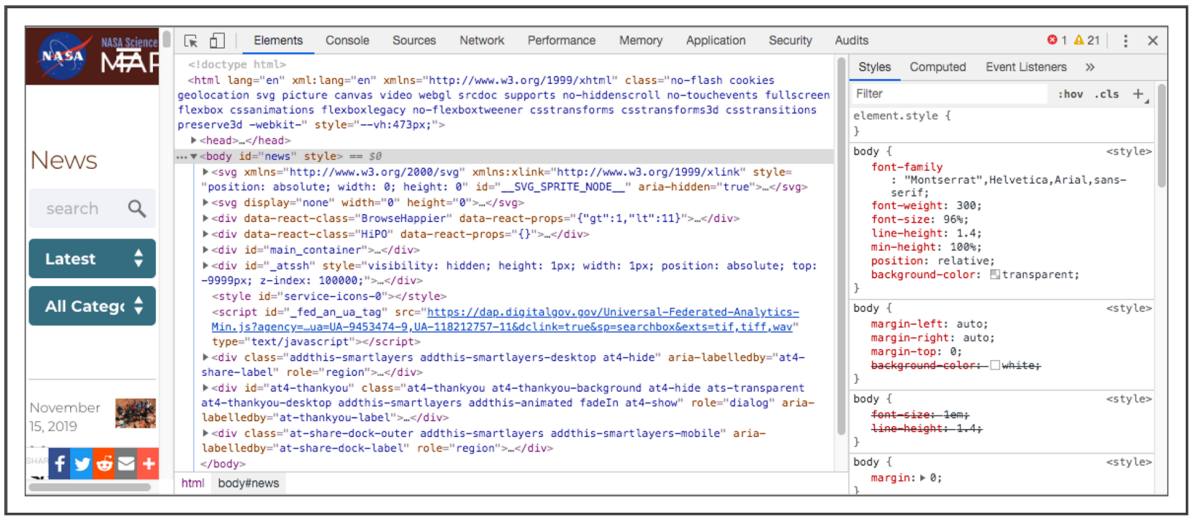
Robin wants to be kept up to date with different Mars news, and she's enjoyed the articles published on the **NASA news website (https://mars.nasa.gov/news/?page=0&per_page=40&order=publish_date+desc%2Ccreated_at+desc&search=&category=19%2C165%2C184%2C204&blank_scope=Latest)** . For her project specifically she

would like to extract the most recently published article's title and summary. Let's find the HTML components in the page so we can help her with that.

Start with opening the news site in a new browser window. At first glance, we can see that there are article titles and a quick sentence describing each article. Open the DevTools by right-clicking anywhere on the page, then click "Inspect" from the pop-up menu.



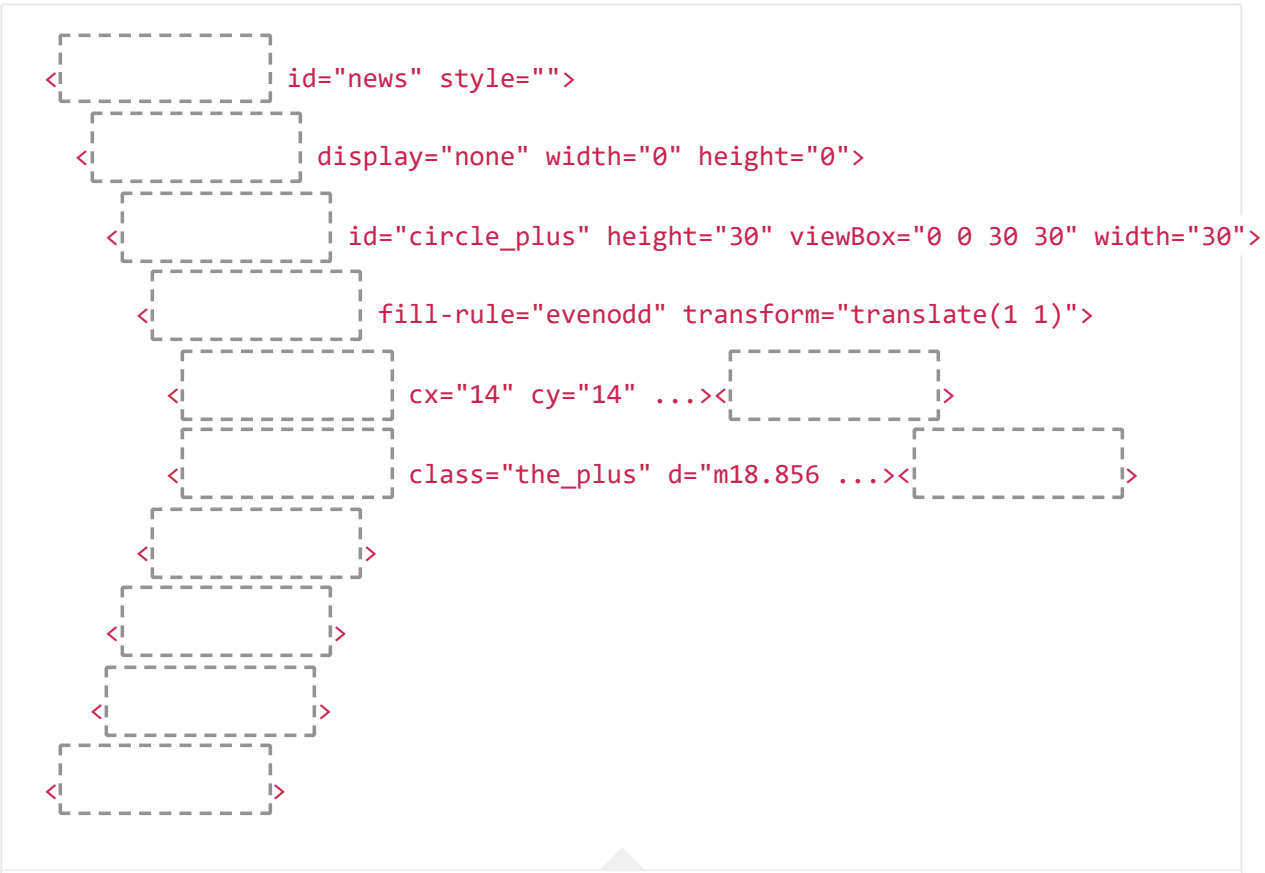
After clicking "Inspect," a new window should open under the webpage. This new window is docked to the webpage itself—it's part of the webpage, it's attached to the webpage, but it has a different job.



There is a lot going on in this site. What we're currently looking at is how this news site is assembled. The `<html lang="en" …>` line should look familiar, as well as the `<head />` and `<body />` tags, but what is all of this other stuff? And the stuff

inside the familiar tags? It's a good thing Robin wanted to take a deeper look at this webpage before trying to extract the data from it.

Drag the HTML tags to the correct positions in the code.



symbol | g | /g | /symbol | svg | path | circle | /path | /circle | body | /body | /svg

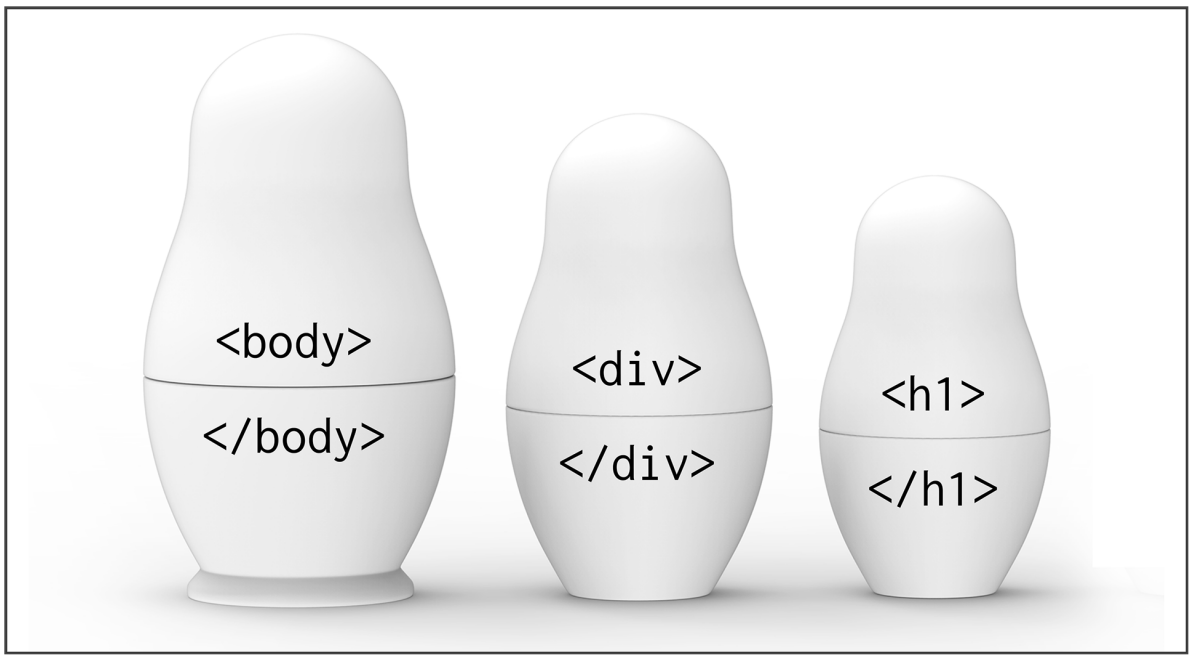
Check Answer

Finish ▸

Let's start breaking this down a bit. Remember how we spoke about containers? For example, the `<body />` tag is a container for every visual component of a webpage, such as headers and paragraphs. Inside that `<body />` tag are other

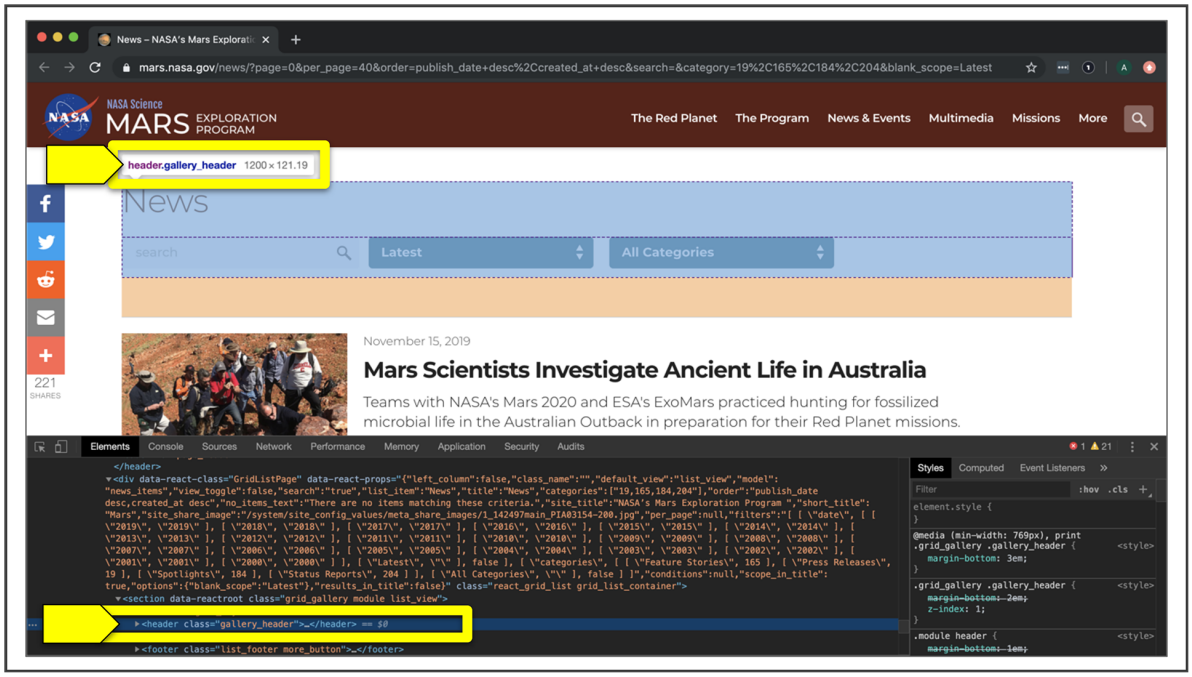
containers, which are nested much like a nesting doll. In the case of this website (and most websites), these other containers inside the body are `<div />` tags.

Take a look at the image below. Each container is nested within another. There can be multiple levels of nesting, depending on how elaborate the website is.



Another perk of the DevTools is that if you hover over the code displayed in the window below, the connected visual is highlighted in the page above at the same time. This is helpful because it shows us which code is specifically tied to features of the website above.

There is a lot of custom code included in this website, so instead of scrolling through all of it to find a certain element, we will search for it instead. In your DevTools, press “ctrl + f” or “command + f” to bring up the search function. Input “gallery_header” into the search bar then press enter. Make sure the line “header class=”gallery_header” is selected, then hover over it with your mouse pointer. This will highlight the header section of the page: the title and its container element.



Hover over the next line of code, `<h2 class="module-title">News</h2>`. Notice how the highlighted portion of the above site has become smaller? That's because we're now looking at an element that is nested inside of a container, instead of the full container.

This is a great way to pinpoint where on the website we want our web scraping code to pull data from. We can't just tell the code to grab a div or a header though, because there could be *many* of these on the website when we only want one. This is where the **class** and **id** attributes come into play.

HTML Classes and IDs

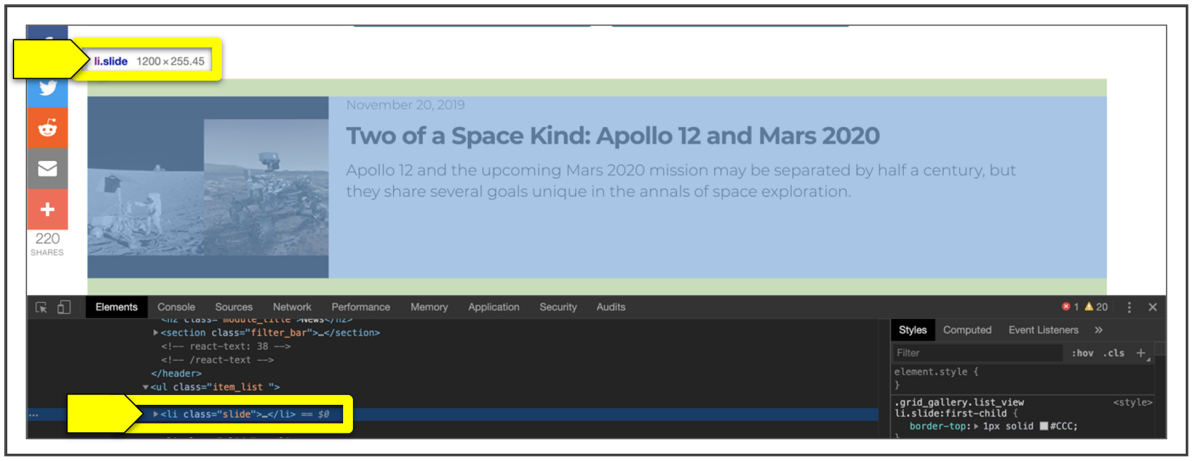
Because of how quickly HTML code can get bloated and confusing, it's important to keep specific containers unique. With everything contained within HTML code, it can be really difficult to find what we're looking for.

But how are developers able to distinguish one `<div />` from another? By adding attributes unique to each container or element. That's another reason to practice using DevTools. We can use it to search for these attributes. How exactly do they work?

Think of it like a litter of puppies. They all look pretty similar, but they each have a personality quirk or trait that makes them act a little differently from their siblings. By adding a different color collar to each puppy, we can now tell them apart just by looking. HTML class and id attributes are like those collars.

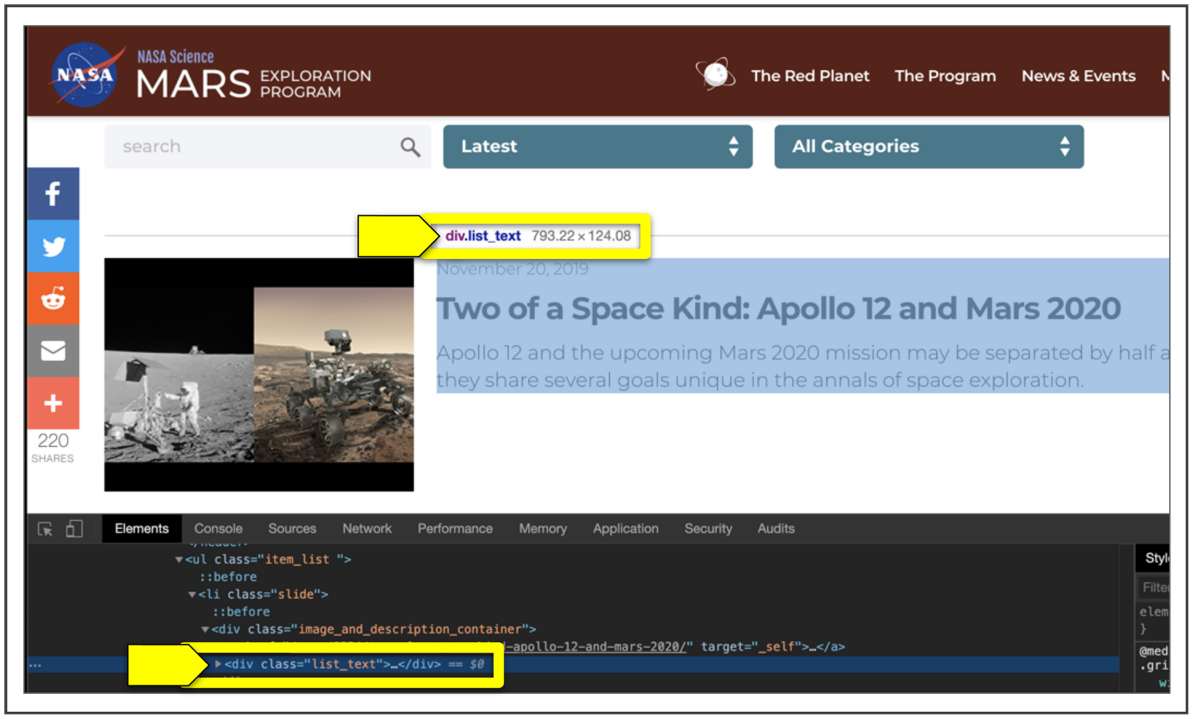
Robin knows that she will want to pull the top article and summary sentence. How do we identify those components, though? Let's look at our DevTools again. This time, let's drill further down into the nested components—we want to find the element that highlights only the top article on the page.

The first `<li />` element with a class of "slide" highlights the top article on the page.



The section we're aiming for (the article title and text) is nested further in, and there are quite a few steps we'll need to take to get there.

First, click the drop-down arrow on the `<li class="slide">` element. From there, we're directed to another element: a div with the class of "image_and_description_container." Click that drop-down arrow as well. Within that, we have another element, `<div class="list_text">`.



This final container holds the information Robin will want: the article title and summary. With the use of DevTools, we've been able to sift through the nested HTML code to find the exact tags we'll need to reference in our scraping script. This process is something we'll be following with each additional webpage we want to scrape: visit the page, identify the data, then shift through the HTML code to pinpoint its location on the webpage.

Which tag is used for the article title?

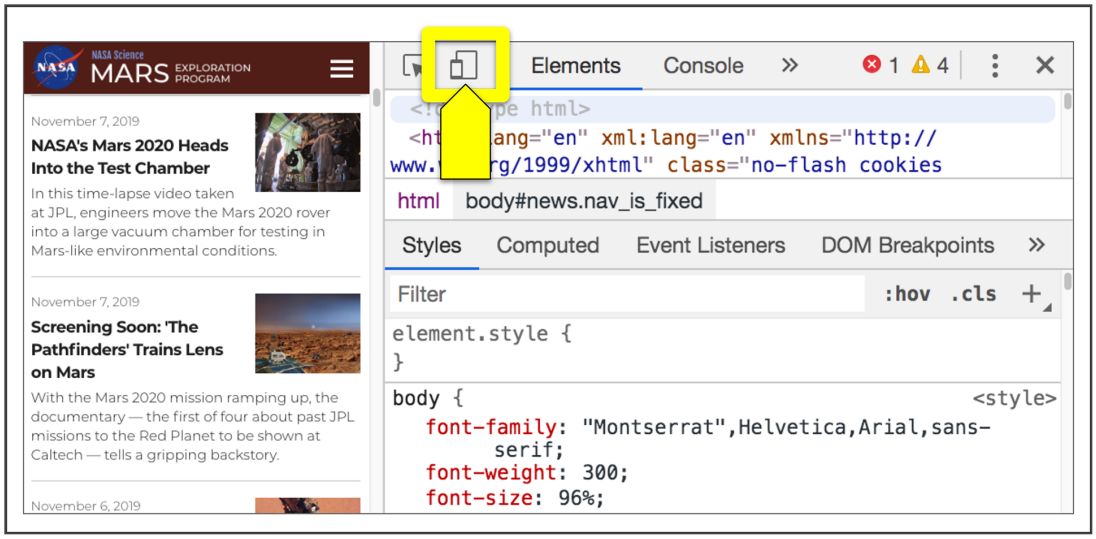
- <a>
- <p>
-

Check Answer

Finish ▸

Mobile Device Preview

DevTools also comes with a feature that allows us to view webpages as we would if using a phone or tablet. Not only that, but there are specific device models we can use to test the page. Let's look at the DevTools again—this time at the Device icon.



This button toggles the device selector. When clicked, the webpage we're viewing automatically adjusts to the height and width of a responsive mobile device. The drop-down menu at the top left of the screen holds a selection of devices to choose from. Switching between devices alters the webpage to reflect how it interacts with each device.

When you're ready to view the webpage as it normally is shown on your computer, you can toggle the responsive view with the same button (the device icon).

We'll use this later in the module when we build a portfolio.

10.3.1: Use Splinter

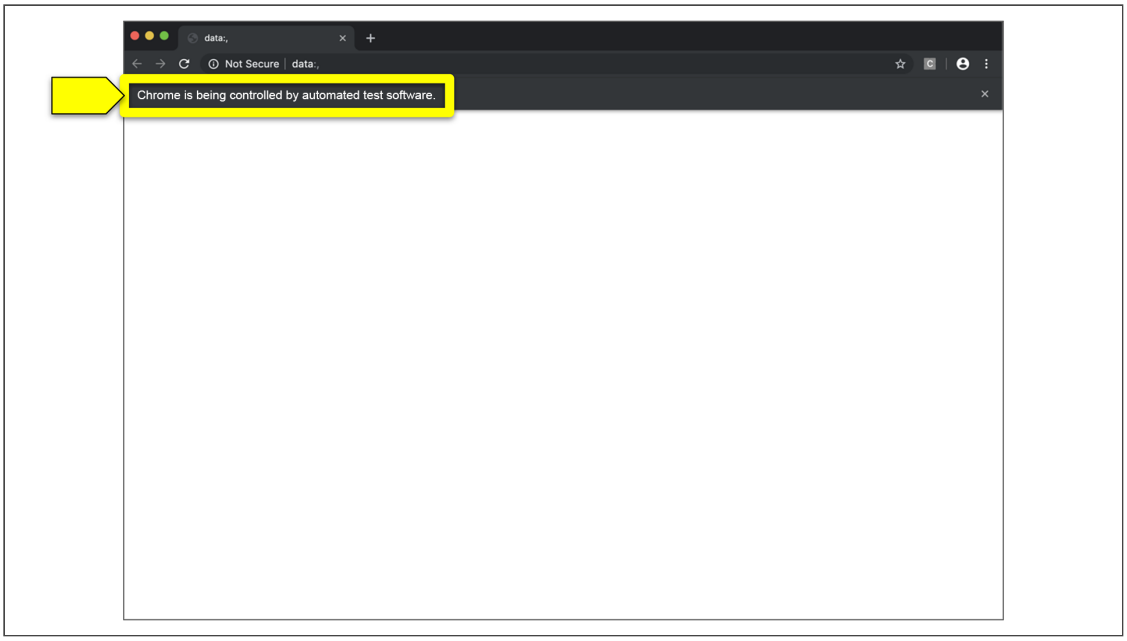
Robin is a bit more familiar with HTML tags and how they fit together to create a webpage, which is a great first step. She also has the necessary tools installed to get started with the scraping, so she's eager to dive in.

The next part is to use Splinter to automate a browser—this is pretty fun because we'll actually be able to watch a browser work without us clicking anywhere or typing in fields, such as using a search bar or next button.

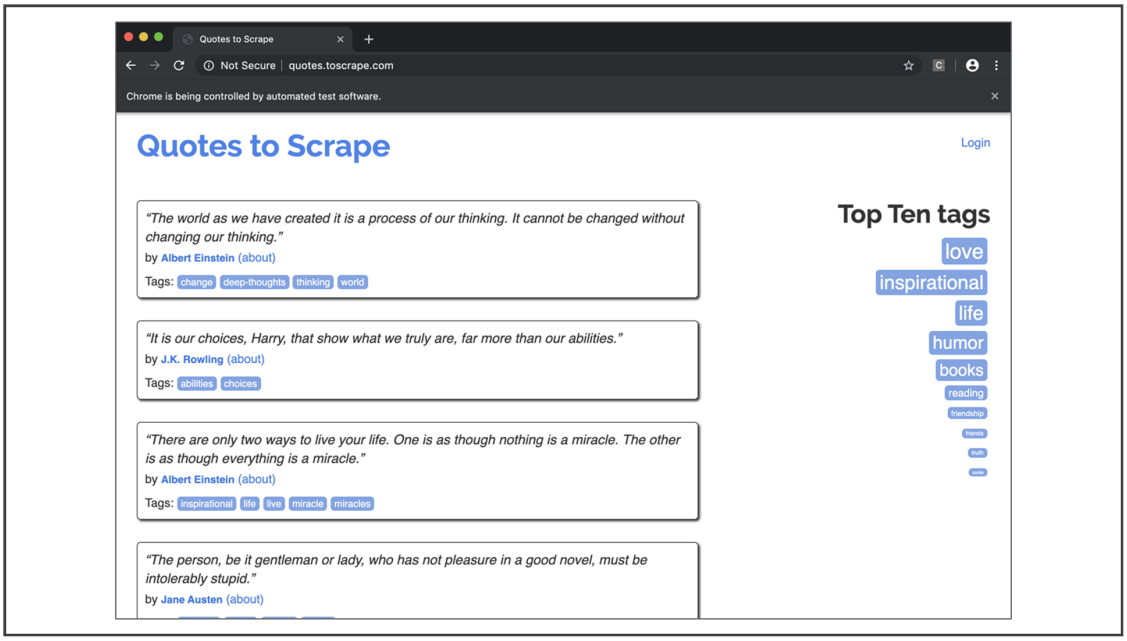
After we help Robin get Splinter rolling, we'll actually scrape data using BeautifulSoup. This is where our practice with HTML tags comes in. To scrape the data we want, we'll have to tell BeautifulSoup which HTML tag is being used and if it has an attribute such as a specific class or id.

One of the fun things about web scraping is the automation—watching your script at work.

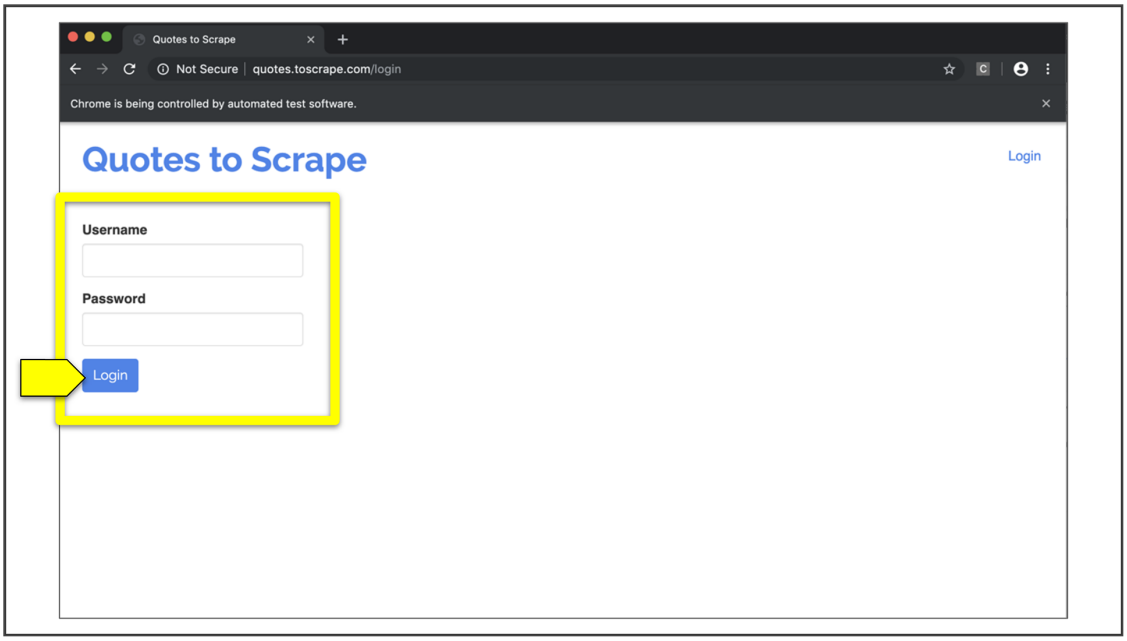
1. Once you execute your completed scraping script, a new Chrome web browser will pop up with a banner across the top that says "Chrome is being controlled by automated test software."



2. This message lets you know that your Python script is directing the browser. The browser will visit websites and interact with them on its own.



3. Depending on how you've programmed your script, your browser will click buttons, use a search bar, or even log in to a website.



Navigate to your Mission-to-Mars folder using the terminal. Then go ahead and activate Jupyter Notebook. Create a new `.ipynb` file to get started—this is where we'll begin our web scraping work. Let's name it "Practice." It can be deleted when we're done, or used as a reference later on. It's not necessary, but you can add it to your GitHub repo and to your `.gitignore` file so that it's hidden from public view.

> **Rewind**
>
> `.gitignore` is a text file that contains the names of files you don't want the public to see, such as configuration files, or files that aren't necessary for the completed project, but you want to keep for reference.

In the very first cell, we'll import the tools Splinter and BeautifulSoup.

```
from splinter import Browser
from bs4 import BeautifulSoup
```

Because of the installation differences between operating systems, the next two cells will differ between macOS and Windows users. For a Mac computer, we'll verify that the ChromeDriver is installed after importing the web scraping tools. In the next cell, type the following:

```
# Path to chromedriver
!which chromedriver
```

The output after this line should read: `/usr/local/bin/chromedriver`. If it doesn't, that's okay -- yours was installed in a different location. Make sure to use the path from your output instead of the one here. This is the path to the executable file we'll be using to automate our browser. This line isn't vital to our code—it would work just fine without it (in fact, Windows users shouldn't add this code at all). However, it does let us know that `chromedriver` is installed, ready for use, and where it is stored on your computer.

Next, we'll set the executable path and initialize a browser.

Mac users will use the path from the output of the previous cell to set up the executable path:

```
# Mac users
executable_path = {'executable_path': '/usr/local/bin/chromedriver'}
browser = Browser('chrome', **executable_path, headless=False)
```

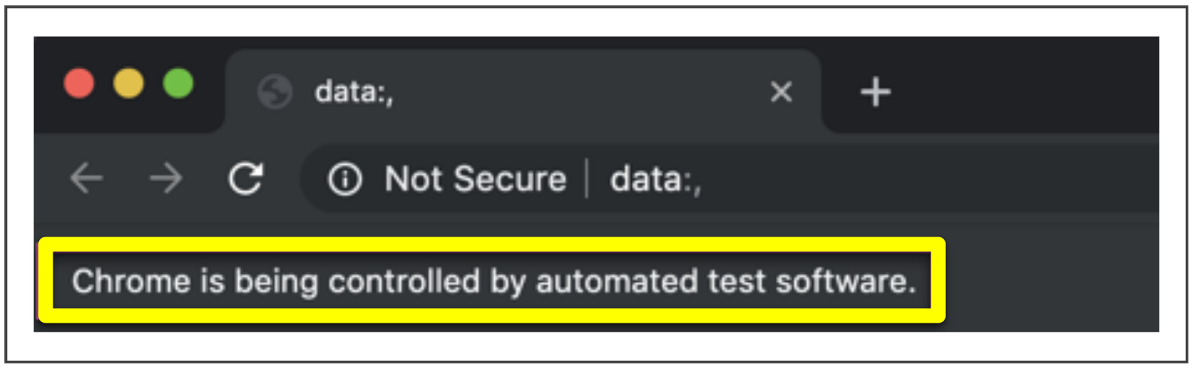
Setting up the path on a Windows computer is similar.

```
# Windows users
executable_path = {'executable_path': 'chromedriver.exe'}
browser = Browser('chrome', **executable_path, headless=False)
```

With these two lines of code, we are creating an instance of a Splinter browser. This means that we're prepping our automated browser. We're also specifying that we'll be using Chrome as our browser. `**executable_path` is unpacking the dictionary we've stored the path in – think of it as unpacking a suitcase. `headless=False` means that all of the browser's actions will be displayed in a Chrome window so we can see them.

You should have three cells ready to be run; go ahead and execute them. The third cell that initiates a Splinter browser may take a couple of seconds to finish, but an

empty webpage should automatically open, ready for instructions. You'll know that it's an automated browser because it'll have a special message stating so, right under the tab, as shown below:



This browser now belongs to Splinter (for the duration of our coding, anyway). It's a lot of fun to watch Splinter do its thing and navigate through webpages without us physically interacting with any components. It's also great a great way to make sure our code is working as we want it to. While the window can be closed at anytime, it's generally not a good idea to shut down the browser without ending the session properly – there's an excellent chance your code will fail or an error will be generated.

> **Note**
> Splinter provides us with many ways to interact with webpages. It can input terms into a Google search bar for us and click the Search button, or even log us into our email accounts by inputting a username and password combination.

10.3.2: Practice with Splinter and BeautifulSoup

Robin is feeling more comfortable with the different HTML components used to build webpages, and she knows that the data she wants to scrape will be nested within different HTML tags. An HTML page can get very confusing very quickly, so Robin would like to practice on a less sophisticated site first.

There are several sites available specifically for newly minted web scrapers to practice and hone their skills with Splinter and BeautifulSoup. These practice sites contain several different components that we'll encounter out in the wild: buttons to navigate, search bars, and nested HTML tags. It's a great introduction to how the tools we'll use work together to gather the data we want.

Before we start scraping things directly from a Mars website, let's practice on another, less-involved site first. Below is how our notebooks should currently look:

[1]:
```
# Import Splinter and BeautifulSoup
from splinter import Browser
from bs4 import BeautifulSoup
```

[2]:
```
# Path to chromedriver
!which chromedriver
```

```
/usr/local/bin/chromedriver
```

[3]:
```
# Set the executable path and initialize the chrome browser in splinter
executable_path = {'executable_path': '/usr/local/bin/chromedriver'}
browser = Browser('chrome', **executable_path)
```

What are each of the following cells doing? Drag and drop the correct answer into the blanks below.

Cell 1:

Cell 2:

Cell 3:

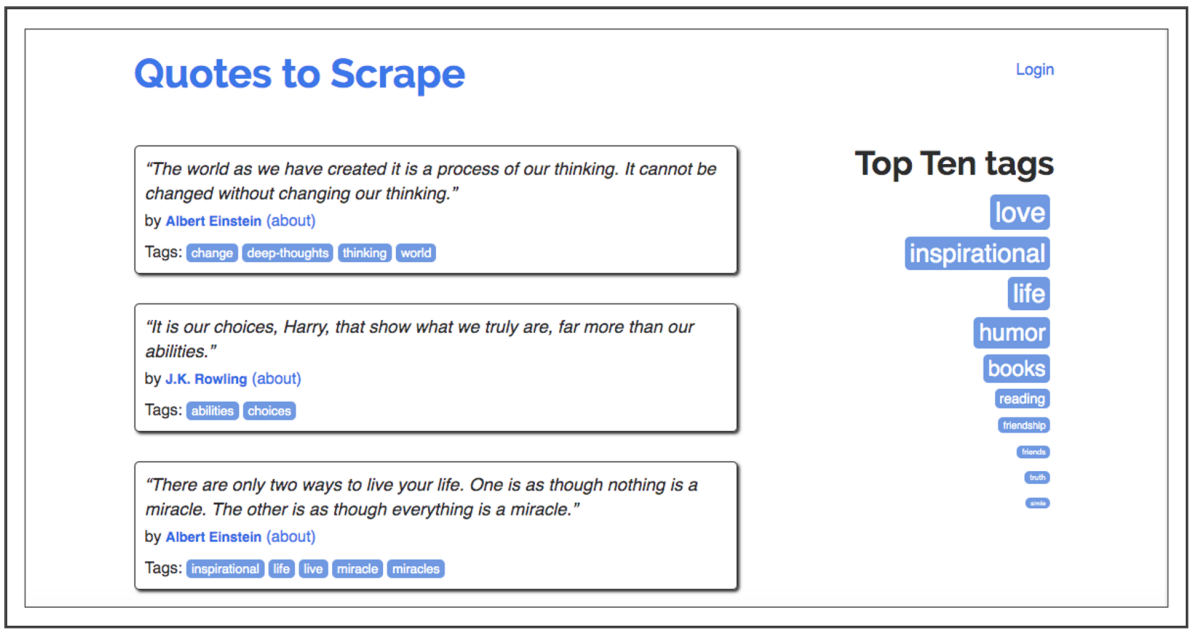
Checks to make sure the ChromeDriver is installed and to see where the path to it is.

Imports the required dependencies to successfully scrape the webpage.

Sets the executable path and initializes the Chrome browser in Splinter.

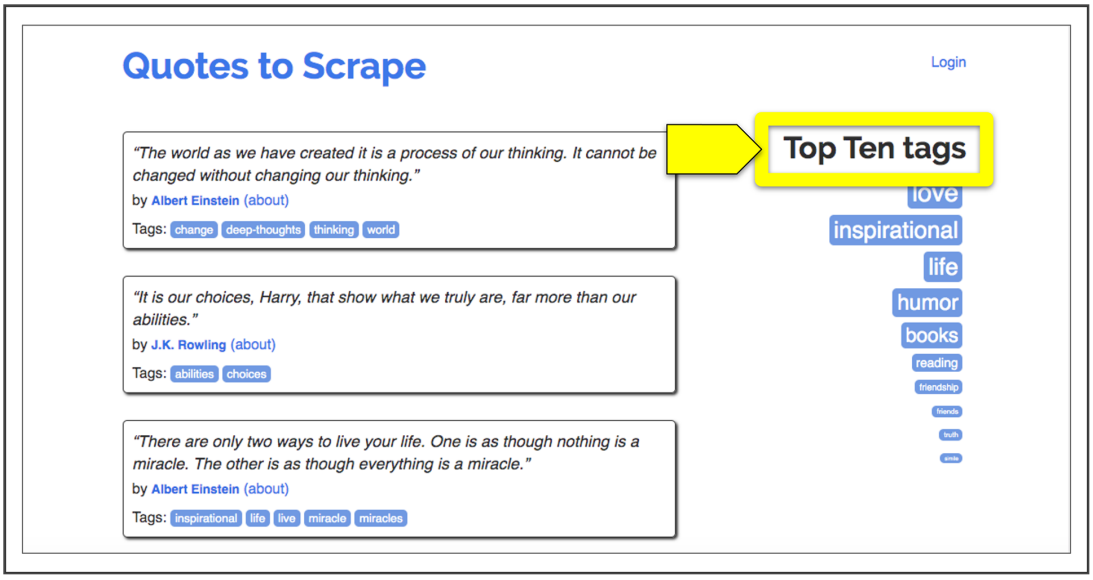
Check Answer

In the fourth cell, we'll scrape data from a website specifically created for practicing our skills: **Quotes to Scrape (http://quotes.toscrape.com/)**. Open the website in a browser and familiarize yourself with the page layout.



Scrape the Top Ten Tags

Interacting with webpages is Splinter's specialty, and there are lots of things to interact with on this one, such as the login button and tags. Our goal for this practice is to scrape the "Top Ten tags" text.



What is the process to follow when discovering what HTML tags contain our data?

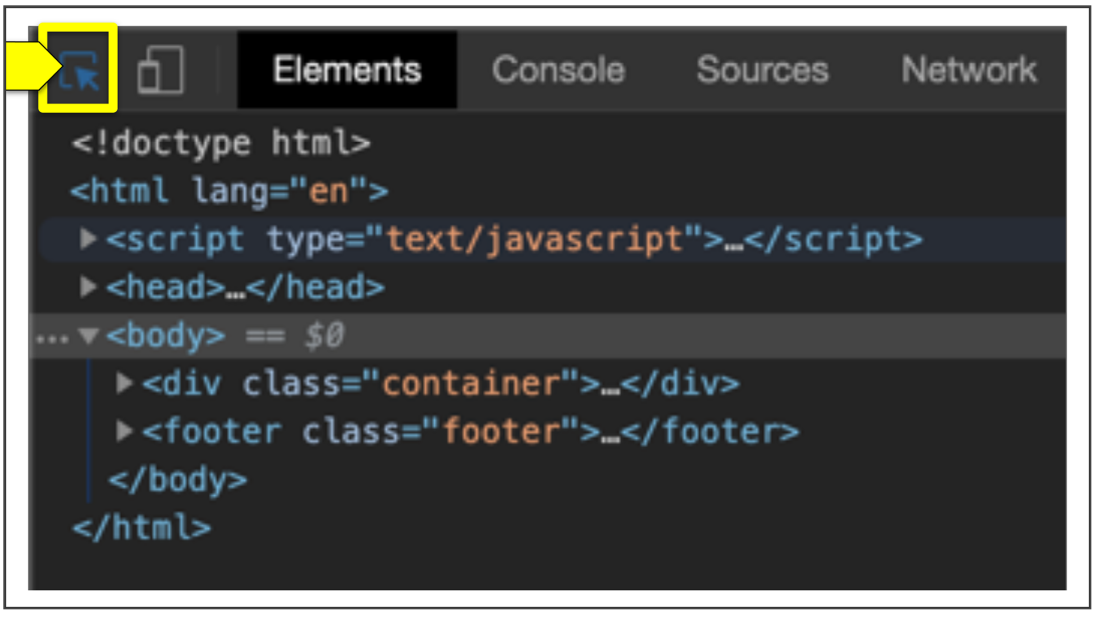
- Inspect the page to bring up DevTools, then sift through all of the tags on the page until you find the one you're looking for.
- Locate the data to scrape on the page, use "Inspect" to bring up DevTools, then select an element on a page to inspect it.
- Download the webpage as an HTML file, open it with VSCode, and search for the item you want using the search function.

Check Answer

Finish ▸

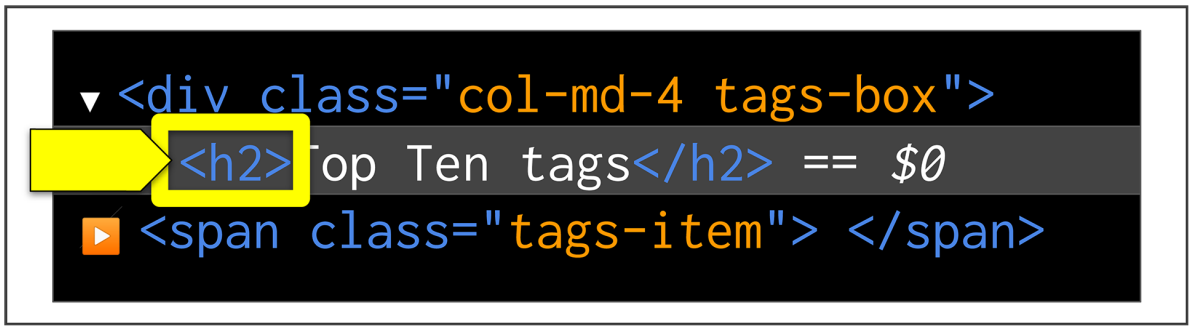
Before we start with the code, we'll want to use the DevTools to look at the details of this line. Right-click the webpage and select "Inspect." From the DevTools window, we can actually select an element on the page instead of searching through the tags.

First, select the inspect icon (the one to the far left).



Then, click the element you want to select on the page, such as the humor tag. This will direct your DevTools to the line of code the humor tag is nested in.

That was really quick. Sometimes we'll still have to dig through the tags to find the ones we want, but being able to select items directly from the webpage helps scale down time immensely. So with this shortcut, we've been able to select the `<h2 />` tag holding the text we want.



With this, we know that our data is in an `<h2 />` tag, and that's great! We've narrowed down where our data is hanging out. But what if there is more than one `<h2 />` tag on the page? When scraping one particular item, we will often need to be more specific in choosing the tag we're scraping from. We can narrow this down even further by using the search function in our DevTools.

So we're using DevTools to search through the HTML of the webpage. We know we want text that's inside an `<h2 />` tag. What else could we do to get more specific?

- While DevTools is active, press Command + F or CTRL + F to activate the search function, then search for the tags or text you're looking for.
- While on the webpage, press Command + F or CTRL + F to activate the search function, then search for the text you're looking for.
- Open DevTools and scroll through the code, looking for the tags and text you want.

Check Answer

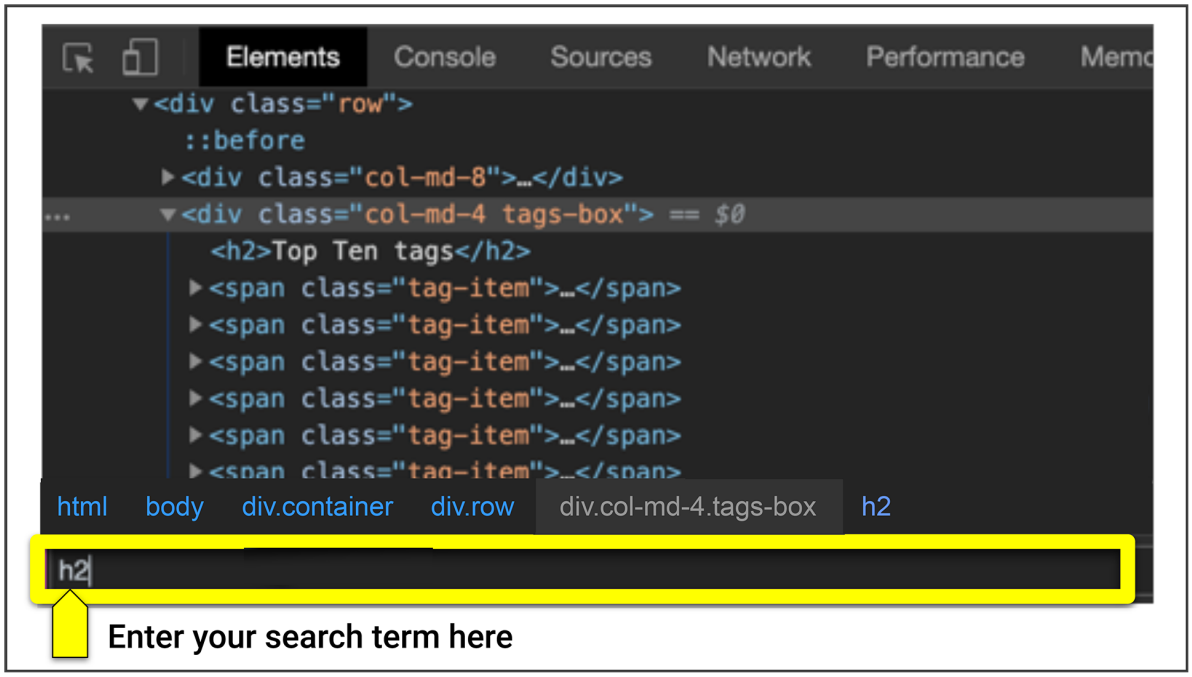
Finish ▸

Search for Elements

Searching within the HTML code is another useful way to quickly find items we're looking for. Earlier, we were able to select a particular component from the page with the select tool. But there are times where we need to know how many of a certain type of tag are in the page. For example, the title we want to scrape is in an `<h2 />` tag, but there are several others on the page as well. Knowing this, we can expect to tailor our code to pull only the `<h2 />` tag we want. First, let's practice searching in the HTML.

With the DevTools still active, press Command + F if you use a Mac, or CTRL + F if you use a Windows computer. This activates the search functionality, only instead of searching the webpage, we're searching the HTML of the webpage. So if we search for all of the `<h2 />` tags in the document, we'll know if we need to make our search more specific by adding attributes such as a class name or an id.

In the search bar that we just activated, type "h2" and then press Enter on your keyboard.



The result of our search immediately shows us two things: that the first tag we've searched for is highlighted, and also the number of those tags in the document.

Because there is only "1 of 1" h2 tags in the document, we know that we can scrape for an `<h2 />` without being more specific. In most other cases, we'll need to include a class or id, so we'll practice that in a little bit.

Scrape the Title

Now let's scrape that title. In the next cell, type the following:

```
# Visit the Quotes to Scrape site
url = 'http://quotes.toscrape.com/'
browser.visit(url)
```

This code tells Splinter which site we want to visit by assigning the link to a URL. After executing the cell above, we will use BeautifulSoup to parse the HTML. In the next cell, we'll add two more lines of code:

```
# Parse the HTML
html = browser.html
soup = BeautifulSoup(html, 'html.parser')
```

Now we've parsed all of the HTML on the page. That means that BeautifulSoup has taken a look at the different components and can now access them. In our next cell, we will find the title and extract it.

```
# Scrape the Title
title = soup.find('h2').text
title
```

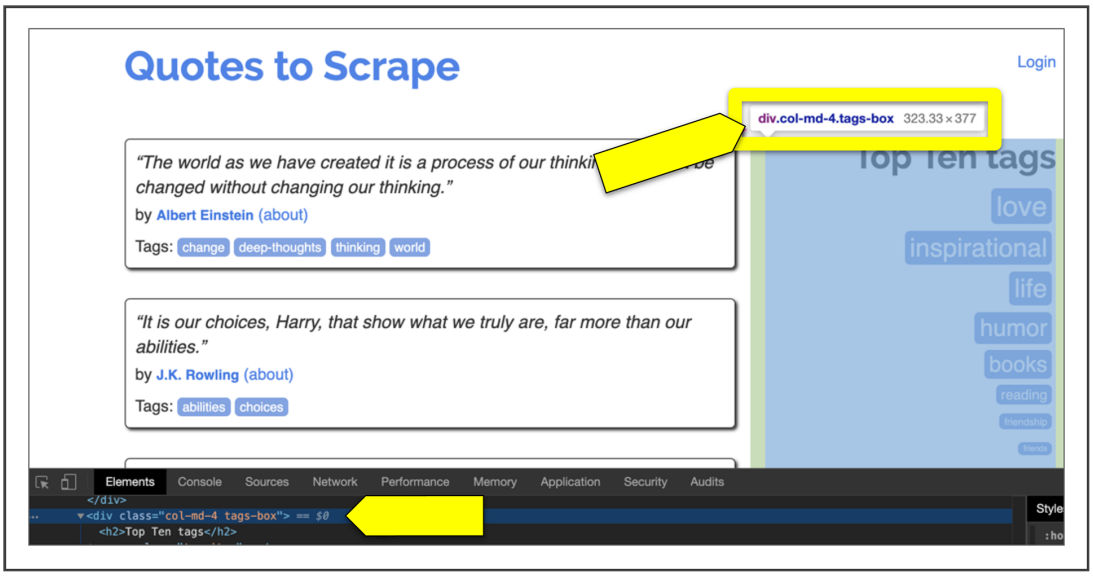
What we've just done in the last two lines of code is:

1. We used our soup object we created earlier and chained `find()` to it to search for the `<h2 />` tag.
2. We've also extracted only the text within the HTML tags by adding `.text` to the end of the code.

We've completed our first actual scrape. Let's practice again, this time using Splinter to scrape the actual tags to go with the title we just pulled.

Scrape All of the Tags

Using our DevTools again, look at the code for the tags. We want all of the tags instead of just one, so we want to first use our select tool to highlight the `<div />` container that holds all of the tags.



Notice that the `<div />` container holding all of the tags has two classes. The "`col-md-4`" class is a Bootstrap feature. Bootstrap is an HTML and CSS framework that simplifies adding functional components that look nice by default. In this case, the "`col-md-4`" means that this webpage is using a grid layout, and it's a common class that many webpages use. We'll dive into that more later.

The other class, "`tags-box`," looks custom, though. Let's make sure first by searching for it using our search box.

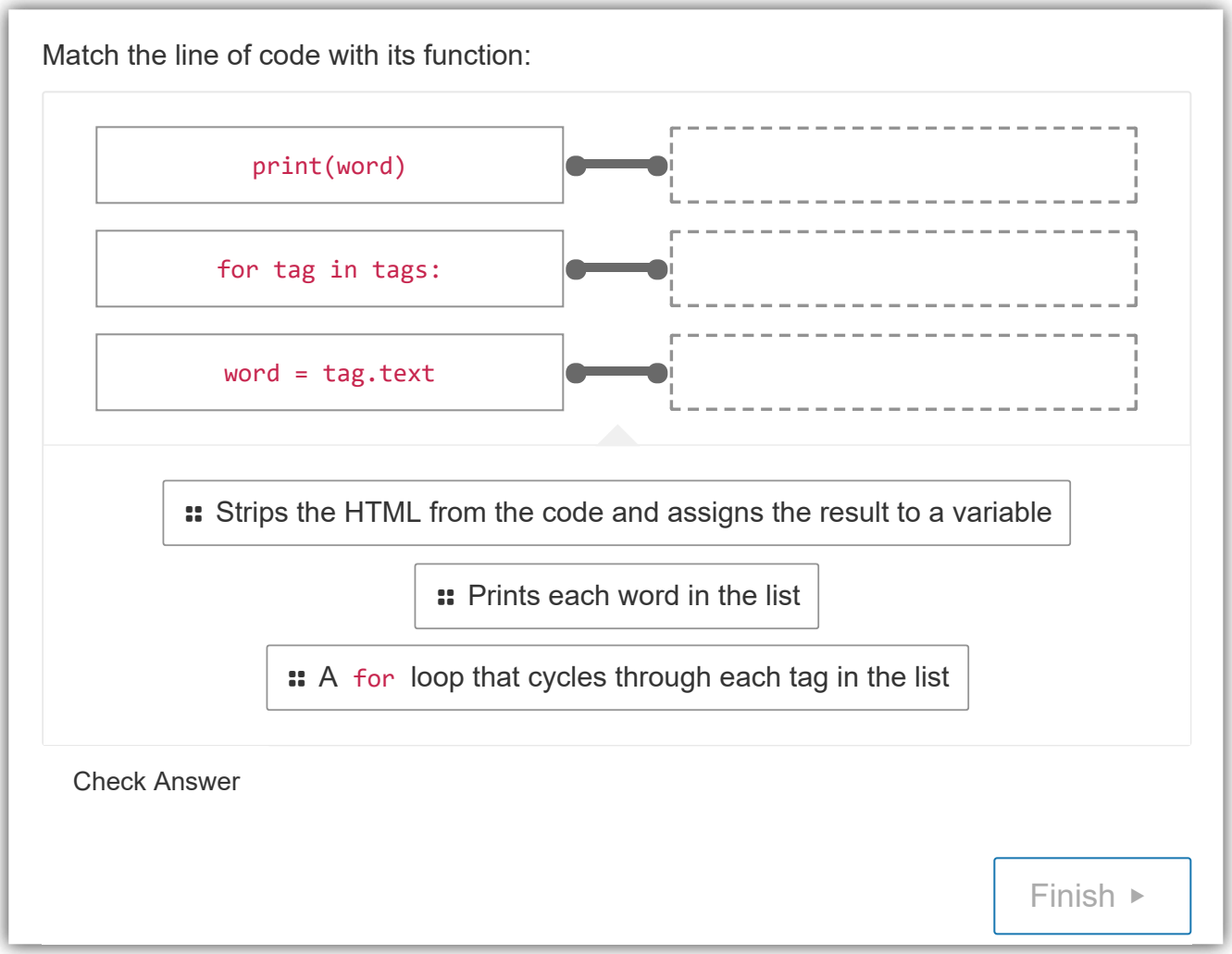
From here, we can see that each `<span />` inside the `tags-box` `<div />` has the data we're looking to scrape. Let's make sure that there are only 10 `<span>` elements with a class of `tag-item` now.

Looks great! Let's scrape each of them. In the next cell of your Jupyter Notebook, type the following:

```
# Scrape the top ten tags
tag_box = soup.find('div', class_='tags-box')
# tag_box
tags = tag_box.find_all('a', class_='tag')

for tag in tags:
    word = tag.text
    print(word)
```

This code looks really similar to our last, but we've upped the difficulty a bit by incorporating a `for` loop. This `for` loop cycles through each tag in the list we scraped, strips the HTML code out of it, and then prints only the text of each tag.



Scrape Across Pages

Now that we've practiced scraping items from a single page, we're going to up the ante by scraping items that span multiple pages. Our next section of code will scrape the quotes on the first page, click the "Next" button, then scrape more quotes and so on (five pages worth of quotes).

When is it best to use a `for` loop?

- It is best to use a `for` loop when we want code to be repeated indefinitely (e.g., to scrape all of the quotes on the page).
- It is best to use a `for` loop when we want code to be repeated for a certain number of iterations.
- It is best to use a `for` loop to scrape all of the quotes off of the page, then we would create another `for` loop to scrape the next page of quotes.

Check Answer

Finish ▸

In the next cell of your practice notebook, type the following:

```
url = 'http://quotes.toscrape.com/'
browser.visit(url)
```

These two lines do two things: They assign an actual URL to the variable of the same name and tell Splinter to visit that webpage. Go ahead and execute this cell. This will cause the automated browser to navigate there.

In the next cell, we'll create a `for` loop to collect each quote, "click" the "Next" button, then collect the next set of quotes.

```
for x in range(1, 6):
    html = browser.html
    soup = BeautifulSoup (html, 'html.parser')
    quotes = soup.find_all('span', class_='text')
    for quote in quotes:
        print('page:', x, '----------')
        print(quote.text)
    browser.click_link_by_partial_text('Next')
```

Now let’s take a look at what the code is doing.

Match the code to its description:

```
for x in range(1, 6):
```

```
    html = browser.html
```

```
    soup = BeautifulSoup (html, 'html.parser')
```

```
    quotes = soup.find_all('span', class_='text')
```

```
    for quote in quotes:
        print('page:', x, '----------')
        print(quote.text)
```

```
browser.click_link_by_partial_text('Next')
```

- A `for` loop with five iterations (pages 1 - 6)
- Use BeautifulSoup to find all `<span />` tags with a class of "text"
- Use BeautifulSoup to parse the `html` object

It's important to note that there are many ways that BeautifulSoup can search for text, but the syntax is typically the same: We look for a tag first, then an attribute. We can search for items using only a tag, such as a `<span />` or `<h1 />`, but a **class** or **id** attribute makes the search that much more specific.

Create a line of HTML code that includes the following: a `<div />` with a class of `"container"` and an id of `"box"`.

Check Answer

Finish ▸

By including an attribute, we have a far better chance of scraping the data we want.

Go ahead and run the code in this cell. Thanks to our print statements, five pages worth of quotes should be right at our fingertips.

What would happen we ran `soup.find_all('div', class_='quote')` instead of `soup.find_all('span', class='text')`?

- We would scrape the parent element and grab everything instead of just the quotes.
- We would scrape only the text from the quotes anyway.
- There would be an error because scraping the entire element would return too many items.

Check Answer

Finish ▸

Now test your skills in the following Skill Drill.

Skill Drill

Stretch your scraping skills by visiting **Books to Scrape (http://books.toscrape.com/)** and scraping the book URL list on the first page.

10.3.3: Scrape Mars Data: The News

After getting all of the tools set up and practicing all of our scraping skills, Robin's ready to start gathering data specific to her project, the Mission to Mars. She's really eager to create her application, and she has a list of websites she plans to use for scraping.

She's really excited about actually scraping live data. The script we're building is designed to scrape the most recent data—that means that each time we run the script, we'll pull the newest data available. As long as the website continues to be updated with new articles, which is likely, we'll have a constant influx of new information at our fingertips.

Robin has carefully staked out the websites she wants to routinely scrape data from. It's her dream to eventually work for NASA, so that's part of why she's chosen two of its sites as a source. Another reason is that NASA has a very friendly Terms of Service (or ToS, also known as Terms of Use) when it comes to web scraping.

Many websites don't want automated browsers visiting their sites and snagging data. If there are too many visits, the server hosting the site could get overloaded and shut down. Administrators can then ban the IP address of the person doing the scraping, making it more difficult to even manually visit the site to view data.

> **Important**
> Terms of Service and Terms of Use bring up an ethical issue when gathering data. Many websites don't allow automated browsing and scraping—some of the scraping scripts out there are designed to gather data quickly, and the constant traffic can overload web servers and disable a website.

With those precautions out of the way, Robin's ready to start scraping Mars news.

In Jupyter Notebook, navigate back to your home folder and create a new `.ipynb` file. This time, name it "Mission_to_Mars" and begin with importing Splinter and BeautifulSoup in the first cell.

```
# Import Splinter and BeautifulSoup
from splinter import Browser
from bs4 import BeautifulSoup
```

Remember, set your executable path in the next cell, then set up the URL (**NASA Mars News (https://mars.nasa.gov/news/)**) for scraping. Remember that the executable path is different between Mac and Windows operating systems, so choose the line of code that will work with your computer.

Mac users start with this block of code:

```
# Set the executable path and initialize the chrome browser in splinter
executable_path = {'executable_path': '/usr/local/bin/chromedriver'}
browser = Browser('chrome', **executable_path)
```

Windows users use this block of code instead:

```
# Set the executable path and initialize the chrome browser in splinter
executable_path = {‘executable_path’: ‘chromedriver’}
browser = Browser('chrome', **executable_path)
```

In the next cell of your Jupyter notebook, we'll assign the url and instruct the browser to visit it.

```
# Visit the mars nasa news site
url = 'https://mars.nasa.gov/news/'
browser.visit(url)
# Optional delay for loading the page
browser.is_element_present_by_css("ul.item_list li.slide", wait_time=1)
```

The optional delay is useful because sometimes dynamic pages take a little while to load, especially if they are image-heavy. With the following line, we are telling our browser to wait a second before searching for components:

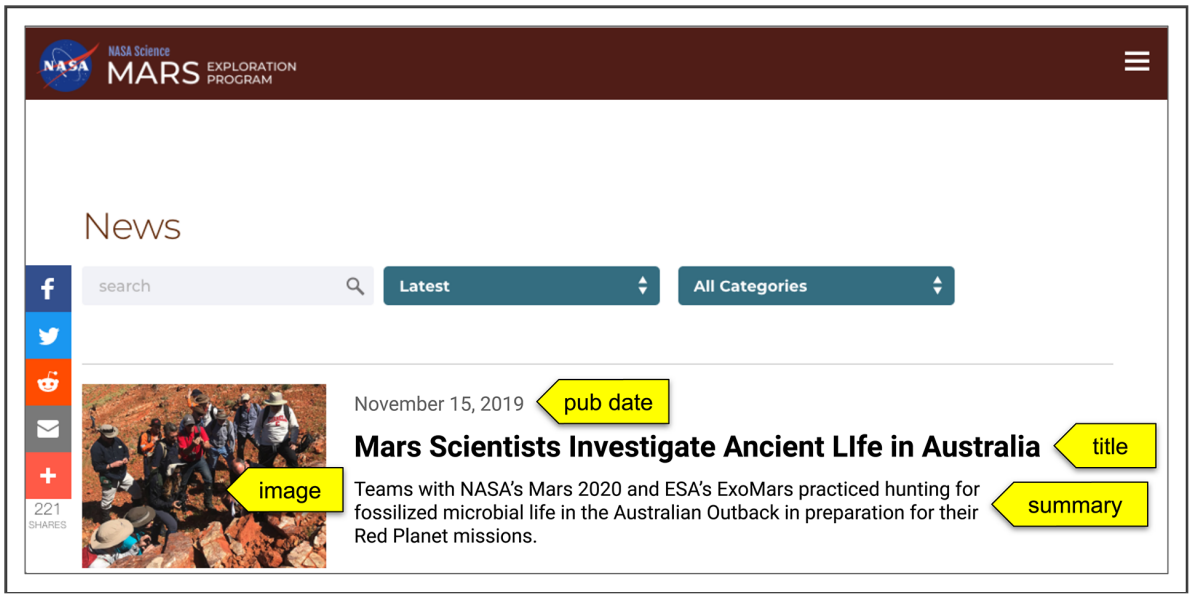
```
browser.is_element_present_by_css("ul.item_list li.slide", wait_time=1)
```

In the next empty cell, we'll set up the HTML parser:

```
html = browser.html
news_soup = BeautifulSoup(html, 'html.parser')
slide_elem = news_soup.select_one('ul.item_list li.slide')
```

Notice how we've assigned `"slide_elem"` as the variable to look for the `<ul />` tag and its descendent (the other tags within the element), and `<li />` tags? This is our **parent element**. This means that this element holds all of the other elements within it, and we'll reference it when we want to filter search results even further. The . is used for selecting classes, such as "item_list," so the code `'ul.item_list li.slide'` pinpoints the tag with a class of "item_list", and the tag with the class of "slide."

The data Robin wants to collect from this particular website is the most recent news article along with its summary. Remember, the code for this will eventually be used in an application that will scrape live data with the click of a button—this site is dynamic and the articles will change frequently, which is why Robin is removing the manual task of retrieving each new article.



After opening the page in a new browser, right-click to inspect and activate your DevTools. Then search for the HTML components you'll use to identify the title and paragraph you want.

Which HTML attribute will we use to scrape the article's title?

- `class = "article_teaser_body"`
- `div = "content_title"`
- `class = "content_title"`
- `id = "content_title"`

Check Answer

Finish ▸

We'll want to assign the title and summary text to variables we'll reference later. In the next empty cell, let's begin our scraping. Type the following:

```
slide_elem.find("div", class_='content_title')
```

In this line of code, we chained `.find` onto our previously assigned variable, `"slide_elem."` When we do this, we're saying, "This variable holds a ton of information, so look inside of that information to find *this* specific data." The data we're looking for is the content title, which we've specified by saying, "The specific data is in a `<div />` with a class of `'content_title`."

Go ahead and run this cell. The output should be the HTML containing the content title and anything else nested inside of that `<div />`.

The title is in that mix of HTML in our output—that's awesome! But we need to get just the text, and the extra HTML stuff isn't necessary. In the next cell, type the following:

```
# Use the parent element to find the first `a` tag and save it as `news_t
news_title = slide_elem.find("div", class_='content_title').get_text()
```

Although the code block above is similar to the last, what are the differences?

- We've added `.get_text()` to our code and rerun it to return only the text of the title.
- We updated the code to search for the content title and we've also added `.get_text()` to remove the extra HTML tags and attributes.
- We have created a new variable for the title, added the `get_text()` method, and we're searching within the parent element for the title.

Check Answer

Finish ▸

Once executed, the result is the most recent title published on the website. When the website is updated and a new article is posted, when our code is run again, it will return that article instead.

```
[7]  # Use the parent element to find the first a tag and save it as
     `news_title`
     news_title = slide_elem.find("div", class_='content_title').get_text()
     news_title

     'NASA Sets Sights on May 5 Launch of InSight to Mars'
```

If you aren't seeing this exact title, that's fine, because the webpage has likely been updated since the screenshot was taken. If you're not seeing a title at all, then check to make sure that you have selected the correct tag and attribute (such as `"div"` and `"class"`) in your find function.

We have the title we want, and that's a great start. Next we need to add the summary text. This next block of code will be very similar to our last one.

What will we need to change in the following line of code to scrape the article summary instead of the title:

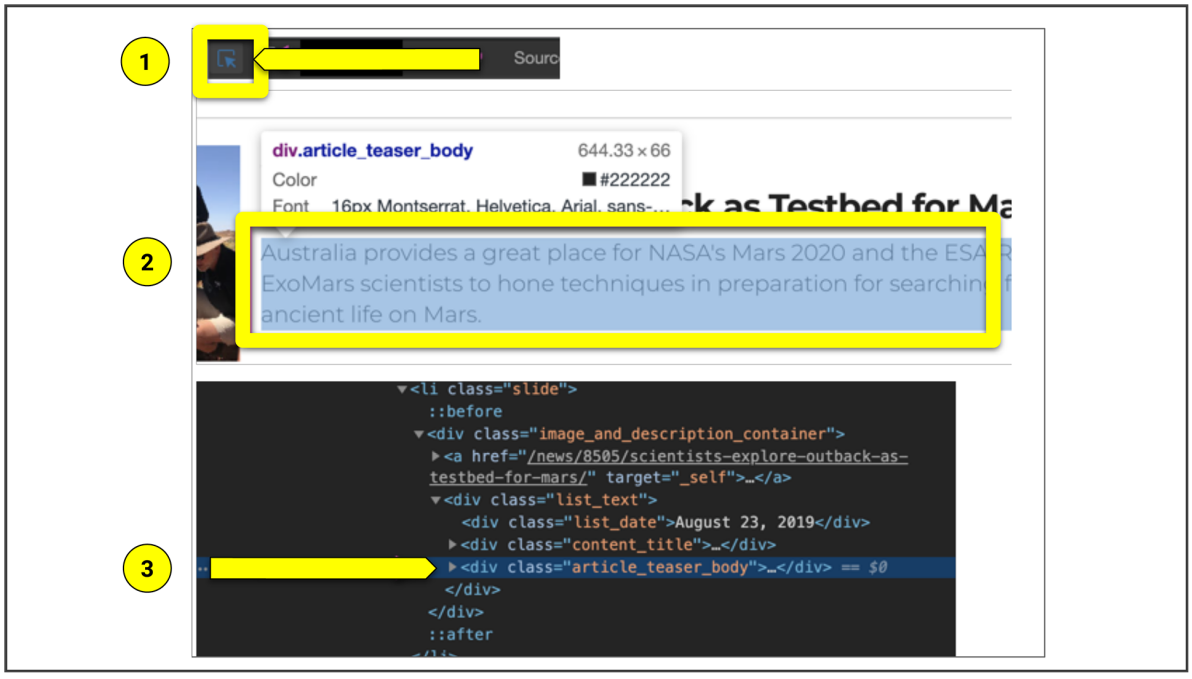
```
slide_elem.find("div", class_='content_title').get_text()
```

- Nothing, we just need to filter through all of the other tags nested inside the first `<div />`.
- We'll need to change the class to `"article_teaser_body."`
- We'll need to switch from using a `"class"` to using an `"id"` because the summary and the title have different attributes.

Check Answer

Finish ▸

Before we can update our code, we'll need to use our DevTools to make sure we're scraping the right tag and class. Use the DevTools selector tool and select the article summary (teaser), then check to see which tag is highlighted.



We know that `"article_teaser_body"` is the right class name, but when we search for it, there is more than one result. What now?

That's okay. There will be many matches because this webpage is updated often and each new article is put on the top of the article list. We want to pull the first one on the list, not a specific one, so more than 40 results is fine. In this case, if our scraping code is too specific, we'd pull only that article summary instead of the most recent.

Because new articles are added to the top of the list, and we only need the most recent one, our search leads us to the first article. *New* news only, please!

> **IMPORTANT**
>
> There are two methods used to find tags and attributes with BeautifulSoup:
>
> - `.find()` is used when we want only the first class and attribute we've specified.
> - `.find_all()` is used when we want to retrieve *all* of the tags and attributes.
>
> For example, if we were to use `.find_all()` instead of `.find()` when pulling the summary, we would retrieve all of the summaries on the page instead of just the first one.

In the next empty cell in your Jupyter Notebook, type the following:

```
# Use the parent element to find the paragraph text
news_p = slide_elem.find('div', class_="article_teaser_body").get_text()
news_p
```

When you run this cell, your output should only be the summary of the article.

Example Summary Output

```
'NASA’s next mission to Mars, InSight, is scheduled to launch Saturday, May 5, on a first-ever mission to study the heart of the Red Planet.'
```

10.3.4: Scrape Mars Data: Featured Image

Robin has finished her first Mars-related scrape. All of the practice and study is starting to pay off. It's a fantastic first step to creating her web application. The next step is to scrape the featured image from another NASA website. Once the image is scraped, we'll want to add it to our web app as well.

Ultimately, with each item we scrape, we'll also save and then serve it on our own website. We're basically using pieces from other websites to piece together our own website, with news and images custom tailored to Robin's taste.

She knows that adding a finished, functional web application to her portfolio will go a long way toward helping her achieve her goal of working for NASA.

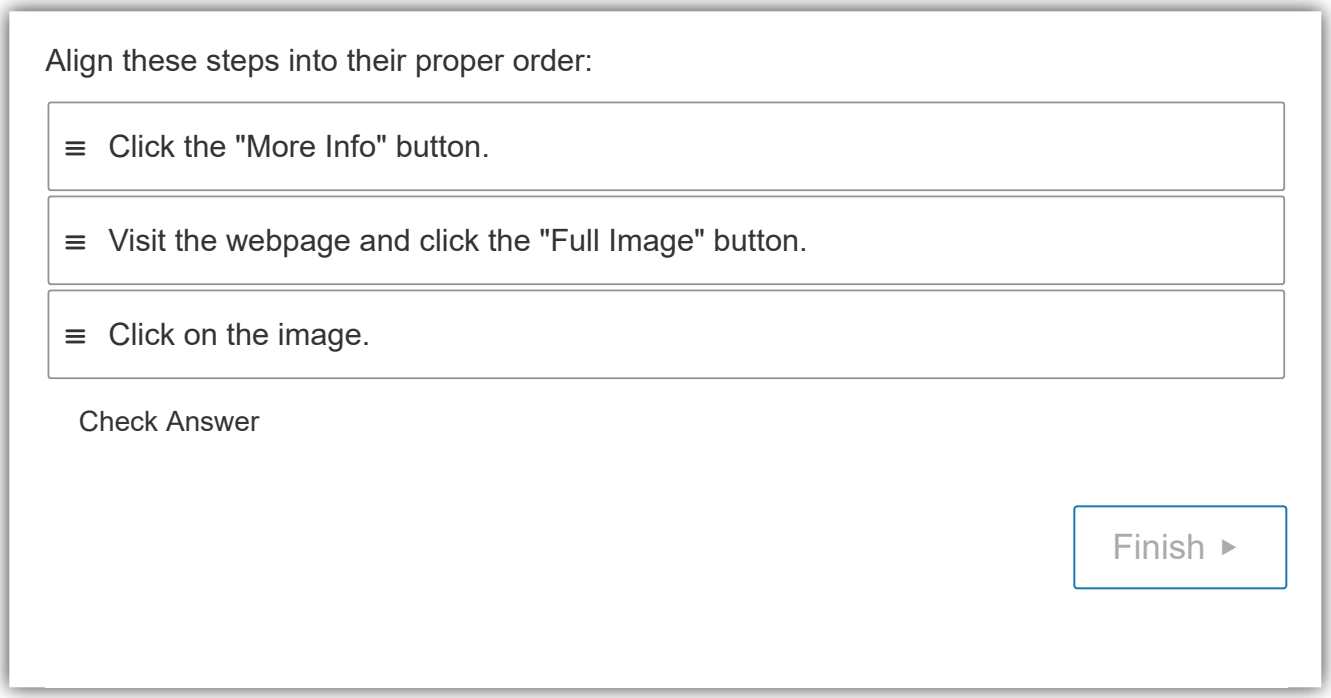
Robin's next step scraping code will be to gather the featured images from NASA's **Space Images (https://www.jpl.nasa.gov/spaceimages/?search=&category=Mars)** webpage. In your Jupyter notebook, use markdown to separate the article scraping from the image scraping.

In the next empty cell, type `"### Featured Images"` and change the format of the code cell to `"Markdown."` The cell below this one is where we'll begin our

scraping. First, let's check out the webpage.

The first image that pops up on the webpage is the featured image. Robin wants the full-size version of this image, so we know we'll want Splinter to click the "Full Image" button. From there, the page directs us to a slideshow. It's a little closer to getting the full-size feature image, but we aren't quite there yet.

Click the "More Info" button on the page. This takes us to an article about the image, but it's still *not quite* where we want it. We'll need to click the image itself one more time to be directed to the actual full-size version.



This is a lot of clicking to get to the image we want. Let's start getting our code ready to automate all of the clicks.

In the next notebook cell, set up the URL:

```
# Visit URL
url = 'https://www.jpl.nasa.gov/spaceimages/?search=&category=Mars'
browser.visit(url)
```

Run this code to make sure it's working correctly. A new automated browser should open to the featured images webpage.

Next, we want to click the "Full Image" button. Let's take a look at the button's HTML tags and attributes with the DevTools.

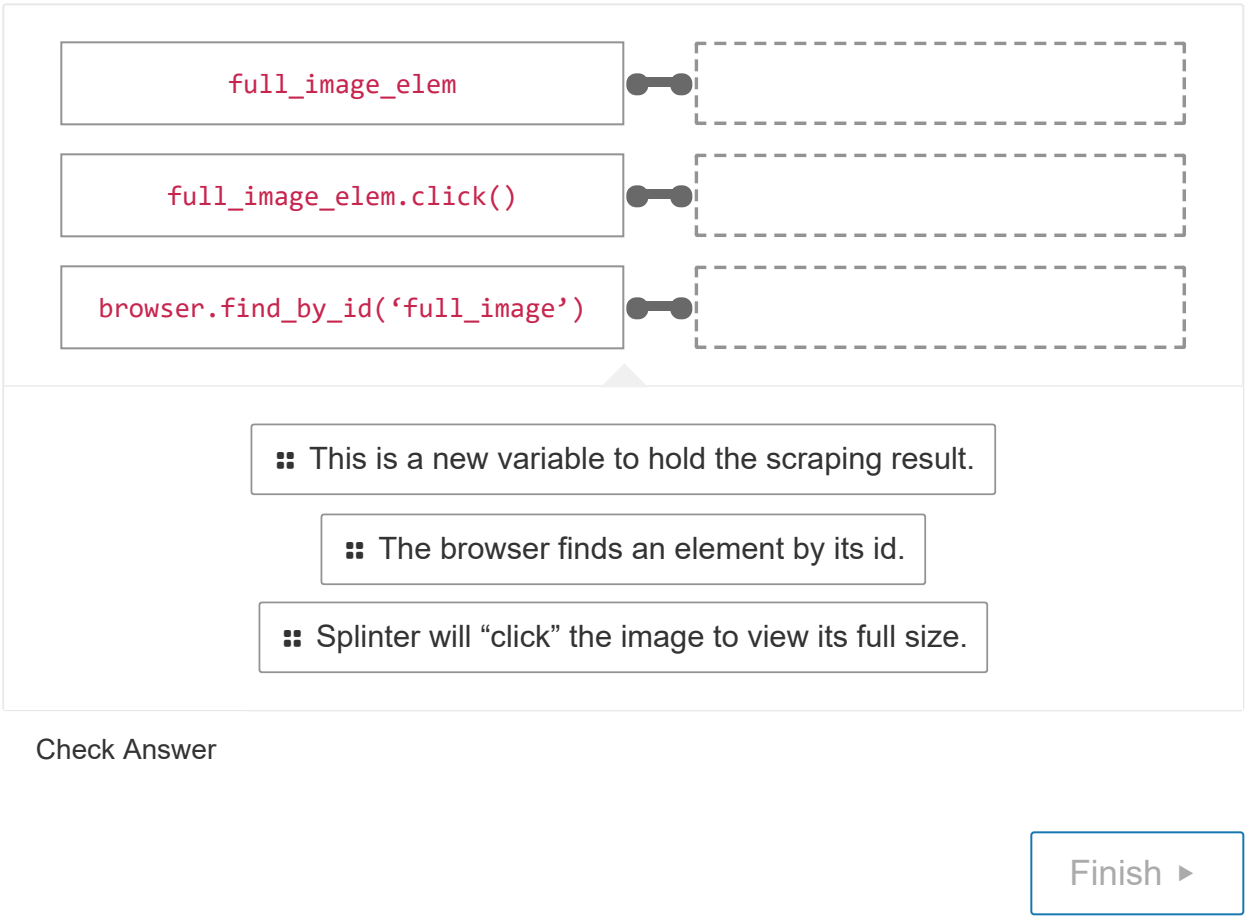
There is quite a lot of stuff going on within that `<a />` tag. Buried down toward the bottom of the bunch is `id="full_image"`. This is significant because in HTML, an id is a completely unique identifier. Often, a class is used as a unique identifier, but only for other similar HTML tags. For example, when we were scraping the articles, we saw that all of the articles had the same class. None of the other components of that webpage had that class, though. An id, on the other hand, can only be used one time throughout the entire page.

> **Skill Drill**
> Use the search function in DevTools to verify that the `full_image` id is unique.

Because we want to click the full-size image button, we can go ahead and use the id in our code. In the next cell, let's type the following:

```
# Find and click the full image button
full_image_elem = browser.find_by_id('full_image')
full_image_elem.click()
```

Match the code to its description:



Go ahead and run this code. The automated browser should automatically "click" the button and change the view to a slideshow of images, so we're on the right track. We need to click the More Info button to get to the next page. Let's look at the DevTools again to see what elements we can use for our scraping.

This looks a little confusing because there aren't any really unique classes here and no ids at all. This brings us to another useful Splinter functionality: the ability to search for HTML elements by text. In the next available cell, try using Splinter's ability to find elements using text.

```
# Find the more info button and click that
browser.is_element_present_by_text('more info', wait_time=1)
more_info_elem = browser.find_link_by_partial_text('more info')
more_info_elem.click()
```

Let's break down this code.

- First, the code makes sure an element is present using text. This tells Splinter to search through the HTML for the specific text `"more info."` We can verify that it does exist in the HTML by using the DevTools to search for it. We also add a wait time of one second to make sure that everything in the page is finished loading before we search for it.
- Next, we create a new variable, `more_info_elem`, to find the link associated with the `"more info"` text.
- Finally, we tell Splinter to click that link by chaining the `.click()` function onto our `more_info_elem` variable.

All together, these three lines of code check for the `"more info"` link using only text, find the link using the same text, then click the link.

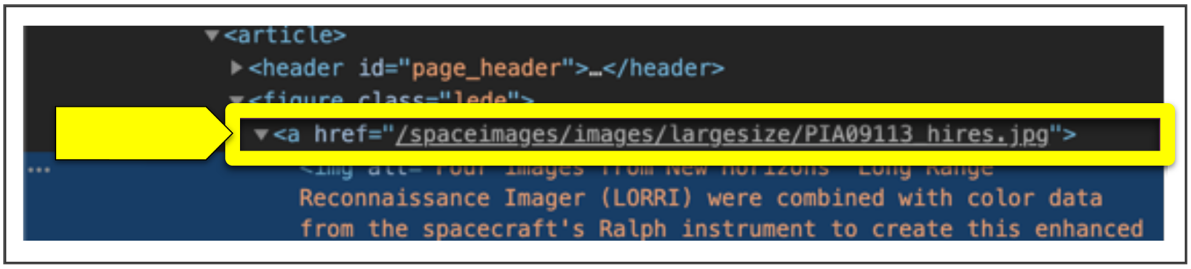
Go ahead and run this cell in your notebook to watch the browser navigate to the next page.

With the new page loaded onto our automated browser, it needs to be parsed so we can continue and scrape the full-size image URL. In the next empty cell, type the following:

```
# Parse the resulting html with soup
html = browser.html
img_soup = BeautifulSoup(html, 'html.parser')
```

Now we need to find the relative image URL. In our browser (make sure you're on the same page as the automated one), activate your DevTools again. This time, let's find the image link for that image. This is a little more tricky. Remember,

Robin wants to pull the most recently posted image for her web app. If she uses the image URL below, she'll only ever pull that specific image when using her app.

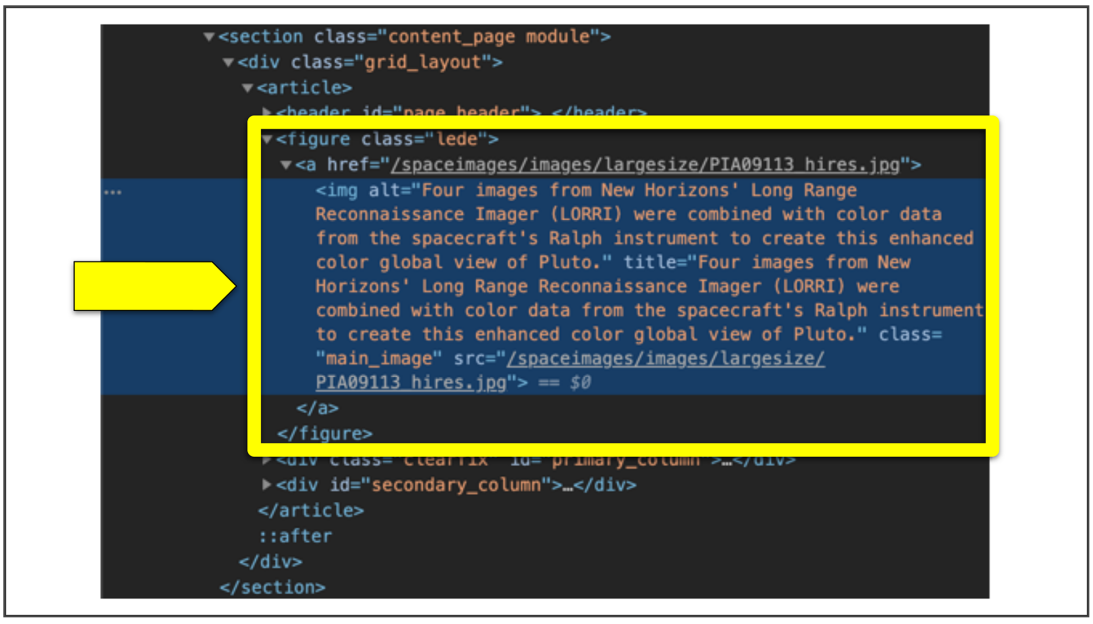


What tags can we use to find the most recent image?

- The `<figure />` and `<a />` tags have the image link nested within them.
- The `<div>` element with a class of `"grid_layout."`
- The `<article>` element.

Check Answer

Finish ▸



We'll use all three of these tags (`<figure />`, `<a />`, and `<img />`) to build the URL to the full-size image. Let's go back to Jupyter Notebook to do that.

```
# Find the relative image url
img_url_rel = img_soup.select_one('figure.lede a img').get("src")
img_url_rel
```

We've done a lot with that single line.

Let's break it down:

- `figure.lede` references the `<figure />` tag and its class, `lede`.
- `a` is the next tag nested inside the `<figure />` tag.
- `.get("src")` pulls the link to the image.

What we've done here is tell BeautifulSoup to look inside the `<figure class="lede" />` tag for an `<a />` tag, and then look within that `<a />` tag for an `<img />` tag. Basically we're saying, "This is where the image we want lives—use the link that's inside these tags."

Run the notebook cell to see the output of the link.

URLs on the featured image page are updated often and will not exactly match yours

```
'/spaceimages/images/largesize/PIA16105_hires.jpg'
```

This looks great! We were able to pull the link to the image by pointing BeautifulSoup to where the image will be, instead of grabbing the URL directly. This way, when NASA updates its image page, our code will still pull the most recent image.

But if we copy and paste this link into a browser, it won't work. This is because it's only a partial link, as the base URL isn't included. If we look at our address bar in the webpage, we can see the entire URL up there already; we just need to add the first portion to our app.

https://www.jpl.nasa.gov/spaceimages/details.php?id=PIA09113

Let's add the base URL to our code.

```
# Use the base URL to create an absolute URL
img_url = f'https://www.jpl.nasa.gov{img_url_rel}'
img_url
```

Python rewind! Identify the following line of code and explain how it works:
`img_url = f'https://www.jpl.nasa.gov{img_url_rel}'`

This variable hold our f-string.

This is an f-string, a type of string formatting used for print statements in Python.

The curly brackets hold a variable that will be inserted into the f-string when it's executed.

`f'https://www.jpl.nasa.gov` `{img_url_rel}'` `img_url`

Check Answer

Finish ▸

We're using an f-string for this print statement because it's a cleaner way to create print statements; they're also evaluated at run-time. This means that it, and the variable it holds, doesn't exist until the code is executed and the values are not

constant. This works well for our scraping app because the data we're scraping is live and will be updated frequently.

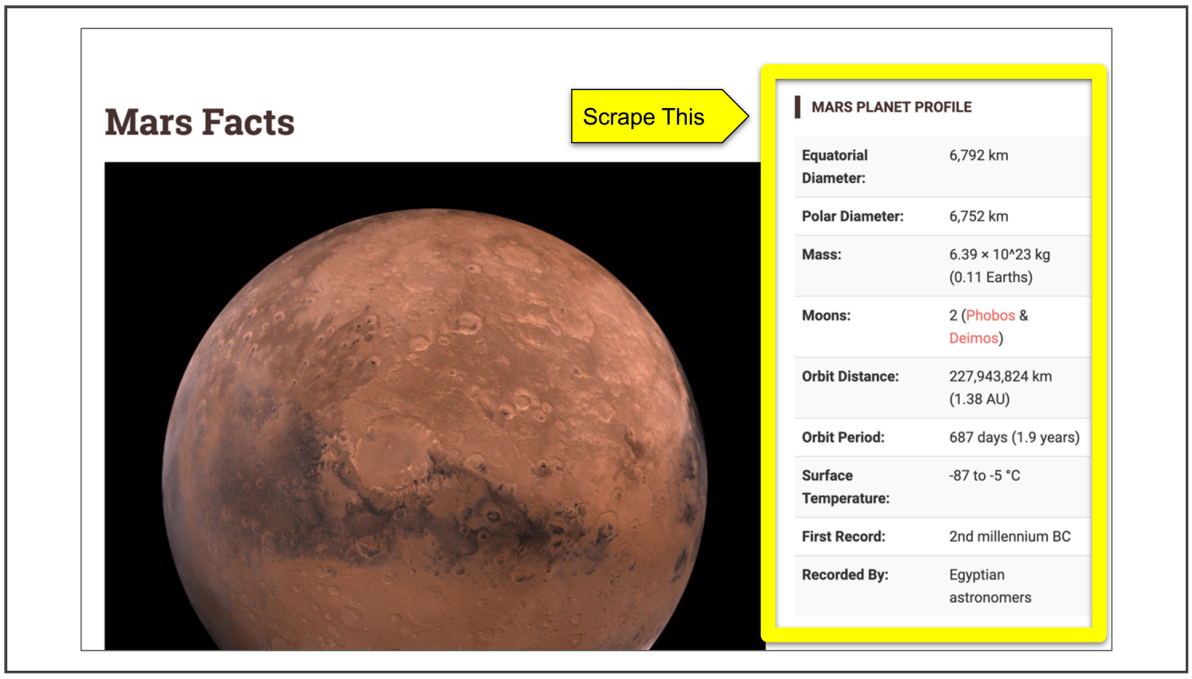
10.3.5: Scrape Mars Data: Mars Facts

We have completed an awesome bit of programming so far. We've been able to automate visiting a website to scrape the top news article (title and summary) and, with just a few lines of code, we have automated the task of visiting a website and navigating through it to find a full-size image, and then we've extracted a link based on its location on the page. Our app will always pull the full-size featured image.

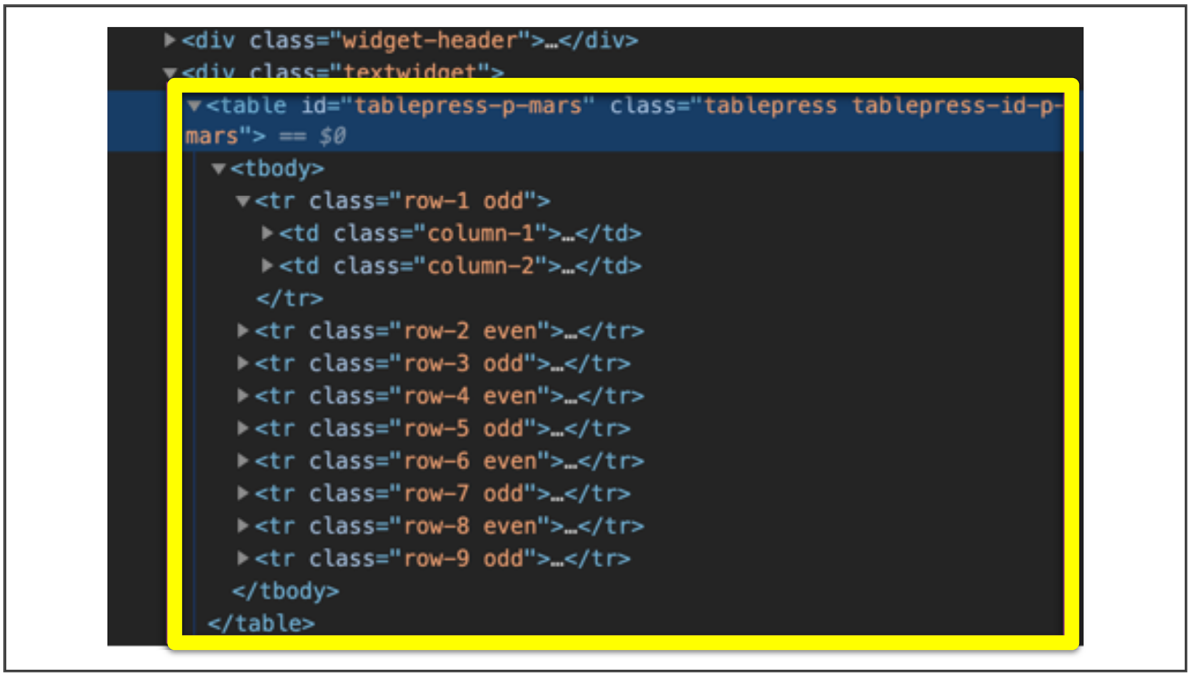
The next bit of information Robin wants to have included in her app is a collection of Mars facts. With news articles and high-quality images, a collection of facts is a solid addition to her web app.

She already has decided which webpage she'll use for fact scraping, but the information is held in a table format. Even though it's in a slightly different format, we can still use DevTools to pinpoint where the tags are, and then extract the table based on its particular tag and attribute pairing set in the HTML code. For this task, we'll just be copying the table's information from one page and helping Robin place it into her application.

Robin has chosen to collect her data from **Mars Facts** (http://space-facts.com/mars/) , so let's visit the webpage to look at what we'll be working with. Robin already has a great photo and an article, so all she wants from this page is the table. Her plan is to display it as a table on her own web app, so keeping the current HTML table format is important.

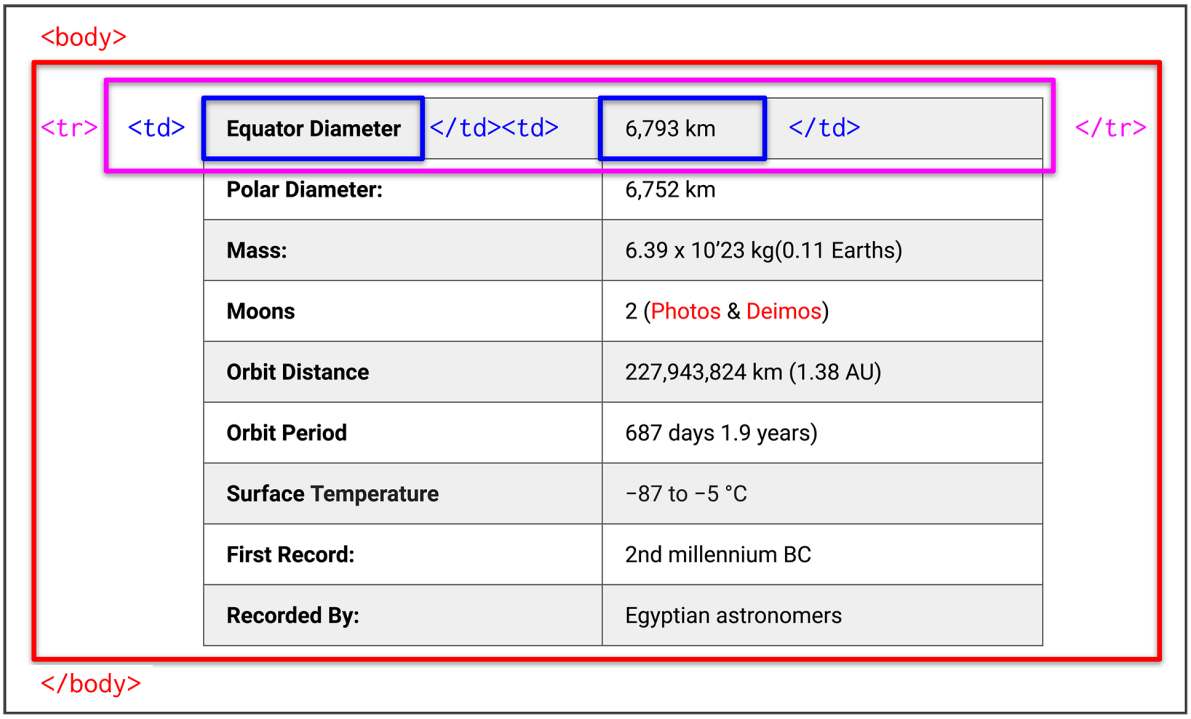


Let's look at the webpage again, this time using our DevTools. All of the data we want is in a `<table />` tag. HTML code used to create a table looks fairly complex, but it's really just breaking down and naming each component.



Tables in HTML are basically made up of many smaller containers. The main container is the `<table />` tag. Inside the table is `<tbody />`, which is the body of the table—the headers, columns, and rows.

`<tr />` is the tag for each table row. Within that tag, the table data is stored in `<td />` tags. This is where the columns are established.



Instead of scraping each row, or the data in each `<td />`, we're going to scrape the entire table with Pandas' `.read_html()` function.

At the top of your Jupyter Notebook, add `import pandas as pd` to the dependencies and rerun the cell. This way, we'll be able to use this new function without generating an error.

Back at the bottom of your notebook, in the next blank cell, let's set up our code.

```
df = pd.read_html('http://space-facts.com/mars/')[0]
df.columns=['description', 'value']
df.set_index('description', inplace=True)
df
```

Now let's break it down:

- `df = pd.read_html('http://space-facts.com/mars/ (http://space-facts.com/mars/) ')[0]` With this line, we're creating a new DataFrame from the HTML table. Pandas looks at the HTML and automatically parses the table. By specifying an index of 0, we're telling Pandas to pull only the first table it encounters. Then, it turns the table into a DataFrame.
- `df.columns=['description', 'value']` Here, we assign columns to the new DataFrame for additional clarity.
- `df.set_index('description', inplace=True)` By using the `.set_index()` function, we're turning the Description column into the DataFrame's index. `inplace=True` means that the updated index will remain in place, without having to reassign the DataFrame to a new variable.

Now, when we call the DataFrame, we're presented with a tidy, Pandas-friendly representation of the HTML table we were just viewing on the website.

description	value
Equatorial Diameter:	6,792 km
Polar Diameter:	6,752 km
Mass:	6.42 x 10^23 kg (10.7% Earth)
Moons:	2 (Phobos & Deimos)
Orbit Distance:	227,943,824 km (1.52 AU)
Orbit Period:	687 days (1.9 years)
Surface Temperature:	-153 to 20 °C
First Record:	2nd millennium BC
Recorded By:	Egyptian astronomers

This is exactly what Robin is looking to add to her web application. How do we add the DataFrame to a web application? Robin's web app is going to be an actual webpage. Our data is live—if the table is updated, then we want that change to appear in Robin's app also, so we can't simply stick an `<img />` tag in there.

Thankfully, Pandas also has a way to easily convert our DataFrame back into HTML-ready code using the `.to_html()` function. Add this line to the next cell in your notebook and then run the code.

```
[23]  df.to_html()
```

```
'<table border="1" class="dataframe">\n  <thead>\n    <tr style="text-align:
right;">\n      <th></th>\n      <th>value</th>\n    </tr>\n    <tr>\n
<th>description</th>\n      <th></th>\n    </tr>\n  </thead>\n  <tbody>\n
<tr>\n      <th>Equatorial Diameter:</th>\n      <td>6,792 km</td>\n    </tr>\n
<tr>\n      <th>Polar Diameter:</th>\n      <td>6,752 km</td>\n    </tr>\n
<tr>\n      <th>Mass:</th>\n      <td>6.42 x 10^23 kg (10.7% Earth)</td>\n
</tr>\n    <tr>\n      <th>Moons:</th>\n      <td>2 (Phobos &amp;amp; Deimos)</td>\n
</tr>\n    <tr>\n      <th>Orbit Distance:</th>\n      <td>227,943,824 km (1.52
AU)</td>\n    </tr>\n    <tr>\n      <th>Orbit Period:</th>\n      <td>687 days
(1.9 years)</td>\n    </tr>\n    <tr>\n      <th>Surface Temperature:</th>\n
<td>-153 to 20 °C</td>\n    </tr>\n    <tr>\n      <th>First Record:</th>\n
<td>2nd millennium BC</td>\n    </tr>\n    <tr>\n      <th>Recorded By:</th>\n
<td>Egyptian astronomers</td>\n    </tr>\n  </tbody>\n</table>'
```

The result is a slightly confusing-looking set of HTML code—this means success. After adding this exact block of code to Robin's web app, the data it's storing will be presented in an easy-to-read tabular format.

Now that we've gathered everything on Robin's list, we can end the automated browsing session. This is an important line to add to our web app also. Without it, the automated browser won't know to shut down—it will continue to listen for instructions and use the computer's resources (it may put a strain on memory or a laptop's battery if left on). We really only want the automated browser to remain active while we're scraping data. It's like turning off a light switch when you're ready to leave the room or home.

In the last empty cell of Jupyter Notebook, add `browser.quit()` and execute that cell to end the session.

```
[24]    browser.quit()
```

IMPORTANT

Live sites are a great resource for fresh data, but the layout of the site may be updated or otherwise changed. When this happens, there's a good chance your scraping code will break and need to be reviewed and updated to be used again.

For example, an image may suddenly become embedded within an inaccessible block of code because the developers switched to a new JavaScript library. It's not uncommon to revise code to find workarounds or even look for a different, scraping-friendly site all together.

10.3.6: Export to Python

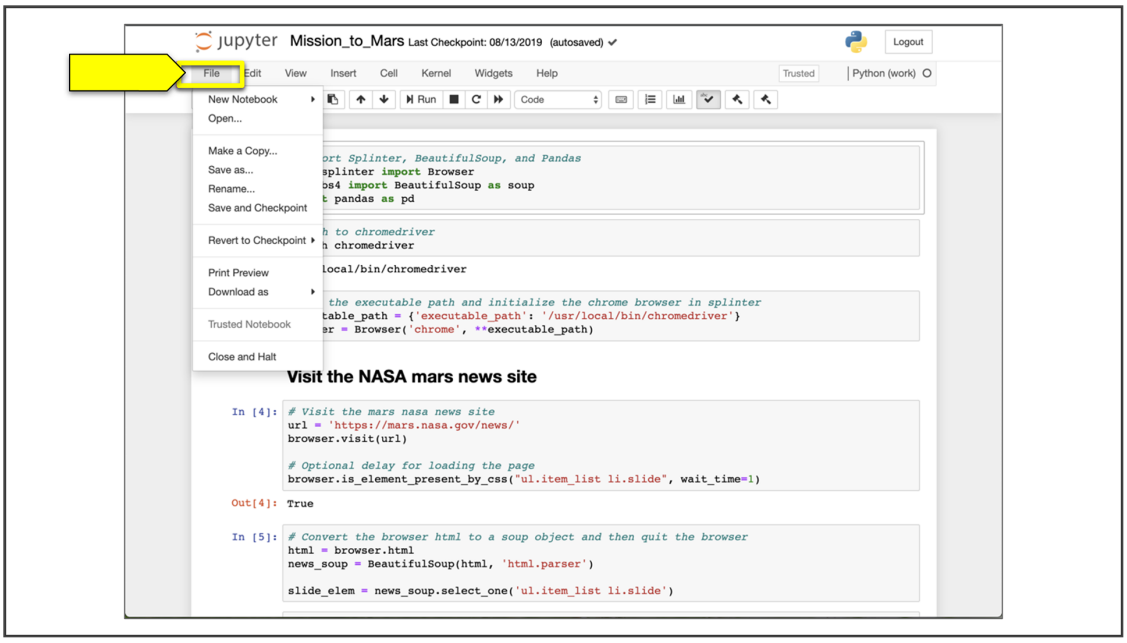
All of the scraping work is coming together. Robin's code can pull article summaries and titles, a table of facts, and a featured image. This is awesome. And Jupyter Notebook is the perfect tool for building a scraping script. We can build it in chunks: one chunk for the image, one chunk for the article, and another for the image.d Each chunk can be tested and run independently from the others. However, we can't automate the scraping using the Jupyter Notebook. To fully automate it, it will need to be converted into a `.py` file.

The next step in making this an automated process is to download the current code into a Python file. It won't transition over perfectly, we'll need to clean it up a bit, but it's an easier task than copying each cell and pasting it over in the correct order.

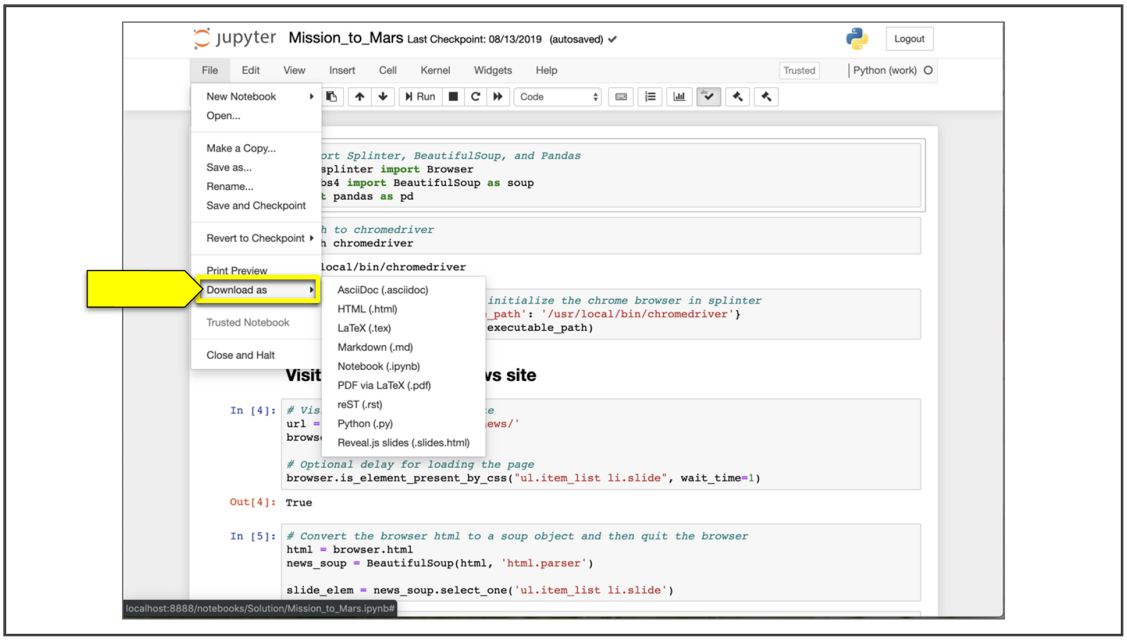
The Jupyter ecosystem is an extremely versatile tool. We already know many of its great functions, such as the different libraries that work well with it and also how easy it is to troubleshoot code. Another feature is being able to download the notebook into different formats.

There are several formats available, but we'll focus on one by downloading to a Python file.

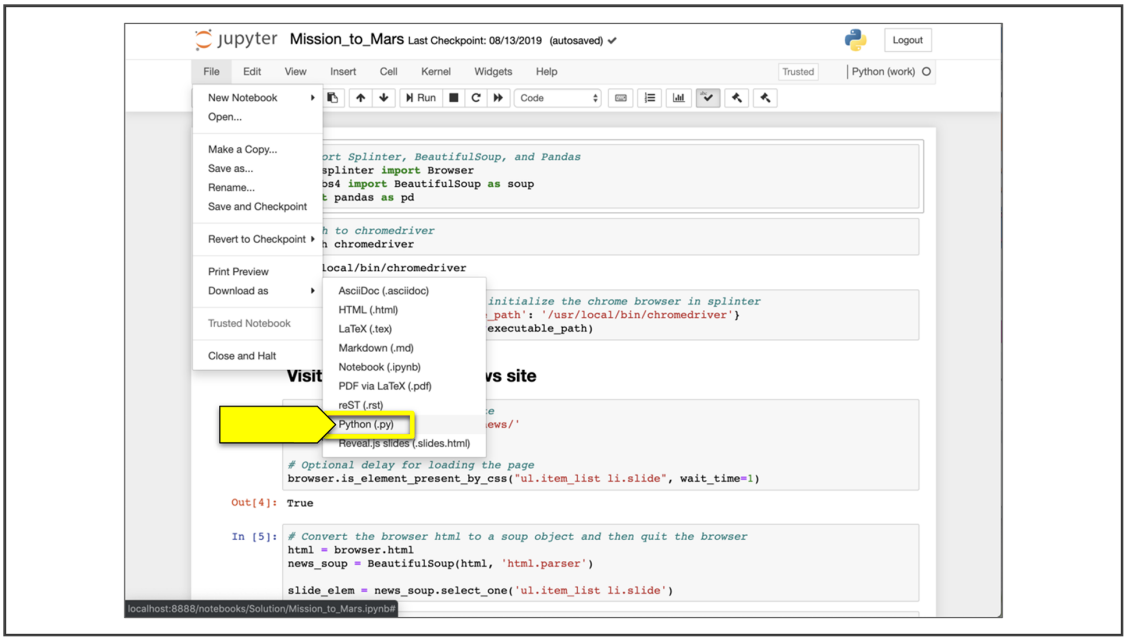
1. While your notebook is open, navigate to the top of the page to the Files tab.



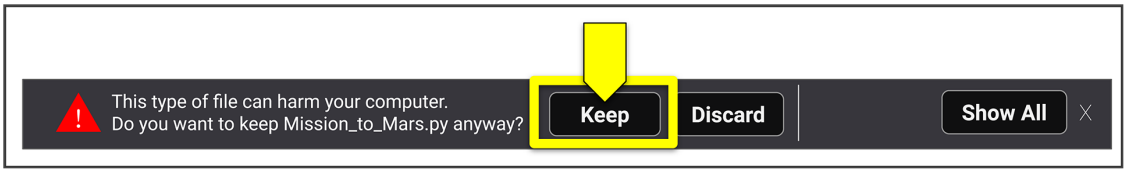
2. From here, scroll down to the “Download as” section of the drop-down menu.



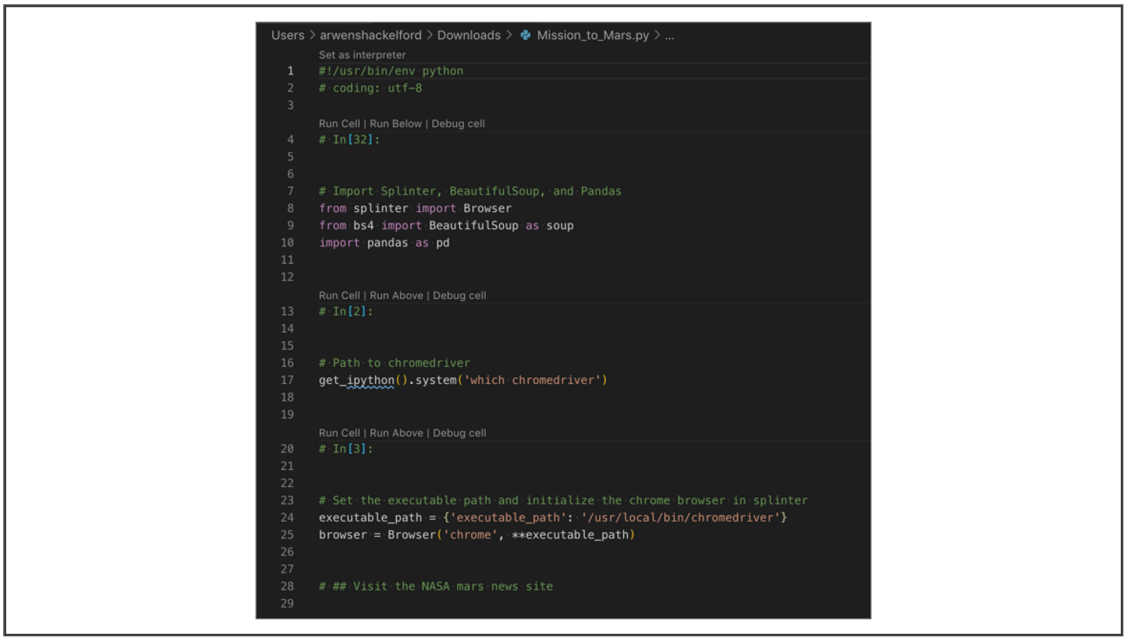
3. Select “Python (.py)” from the next menu to download the code.



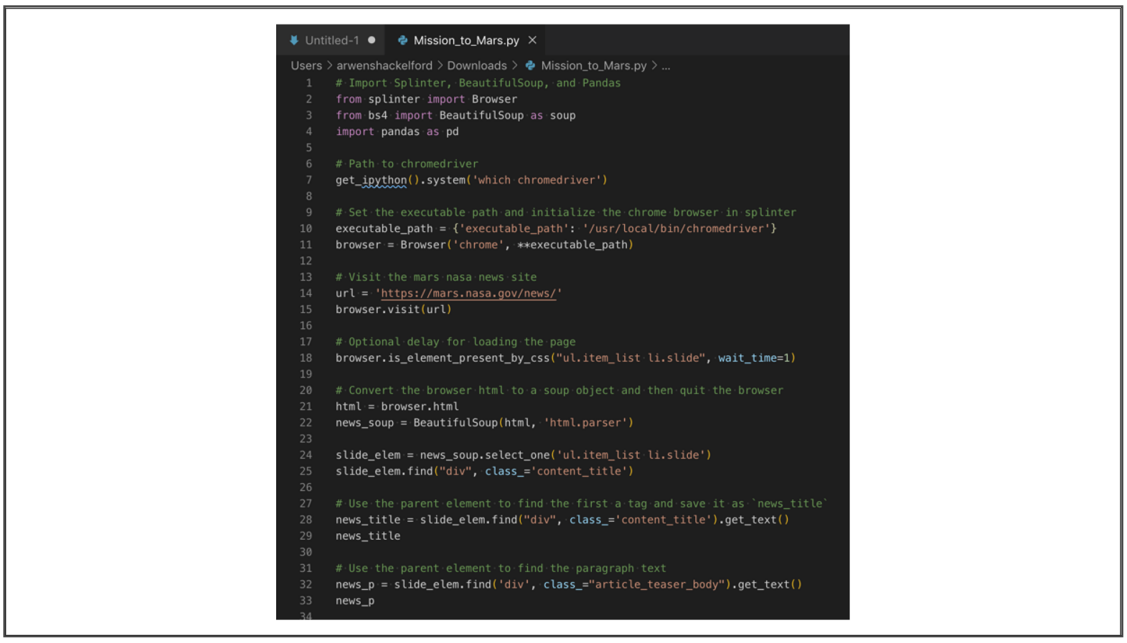
4. If you get a warning about downloading this type of file, click “Keep” to continue the download.



5. Navigate to your Downloads folder and open the new file. A brief look at the first lines of code shows us that the code wasn’t the only thing to be ported over. The number of times each cell has been run is also there, for example.



6. Clean up the code by removing unnecessary blank spaces and comments.



Note: When you're done tidying up the code, make sure you save it in your working folder with your notebook code as scraping.py. You can also test the script by running it through your terminal.

Awesome! Although it was a little tedious to clean up the code, at least you didn't have to rewrite everything or copy and paste the code cell by cell. Also, Jupyter Notebook can be converted to other file types as well, such as markdown and text, so this is a really useful skill to have.

10.4.1: Store the Data

Robin is pretty excited about all of the data we've managed to scrape. And the code is designed to grab the most recent data, so if it's run at a later time, all of the results will have been updated—without us needing to alter the code.

Now that she has the results she wants, she needs to store them in a spot where they can be easily accessed and retrieved as needed. SQL isn't a good option because it works with tabular data, and only one of the items we scraped is presented in that format. Even then, it's all condensed into a block of HTML code.

What Robin will need to use is a database that works differently from SQL with its neatly ordered tables and relationships. Mongo, a NoSQL database, is designed for exactly this task. While Robin is familiar with SQL databases, Mongo is completely new and we'll need to practice with it before loading in our scraped data.

The data Robin has gathered for her web app is great. She's been able to pull a great image, the most recent news article summary, and even an HTML table. Each data type is different, though, with text and images and HTML all together. Compared to SQL's orderly relational system, where each table is linked to at least

one other by a key, the data we've helped Robin gather is a bit chaotic. This is where a non-relational database comes in.

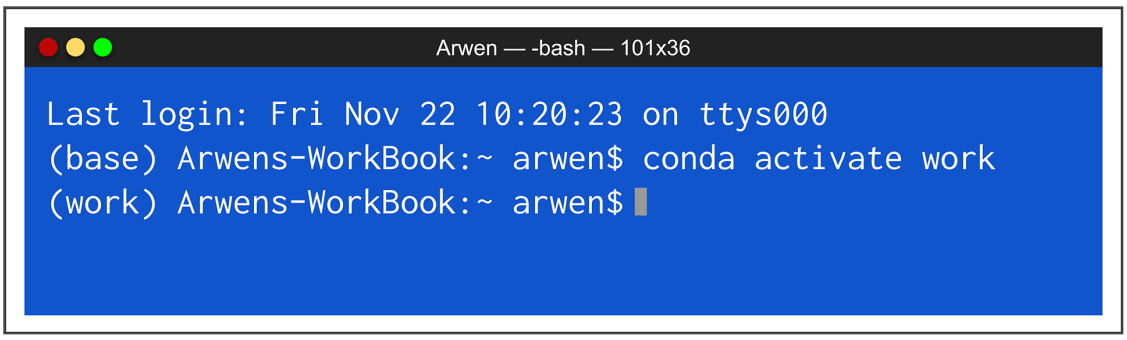
MongoDB (Mongo for short) is a non-relational database that stores data in Binary JavaScript Object Notation (JSON), or BSON format. We'll access data stored in Mongo the same way we access data stored in JSON files. This method of data storage is far more flexible than SQL's model. If you'd like to dig into MongoDB at a deeper level, check out the **official documentation (https://docs.mongodb.com/)**.

> **Rewind**
> JSON, JavaScript Object Notation, is a method that sorts and presents data in the form of an data:value pairs. It looks much like a Python dictionary and can be traversed through using list notation.

A Mongo database contains collections. These collections contain documents, and each document contains fields, and fields are where the data is stored.

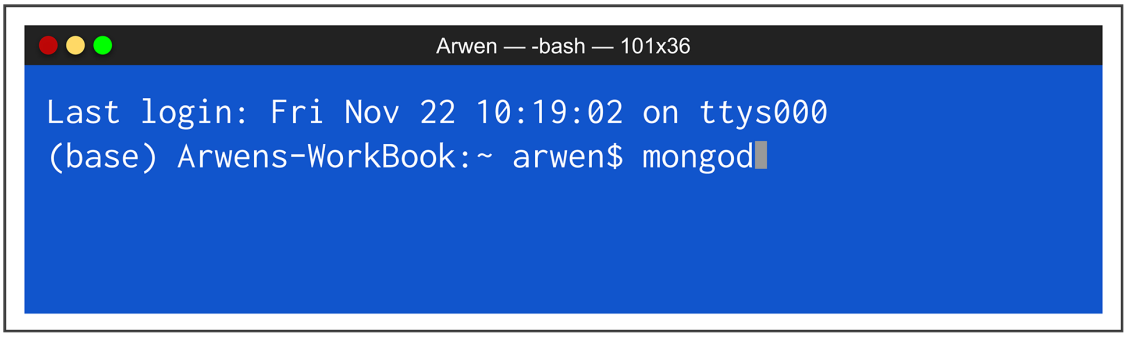
While Mongo and SQL are both databases, that's where the similarities end. They handle documents differently, the storage model isn't even close, and we even interact with them in very different ways.

1. To get started with Mongo, first open a new terminal window, but make sure your working environment is activated. Note that your environment does not need to have the same name as the one in the image.

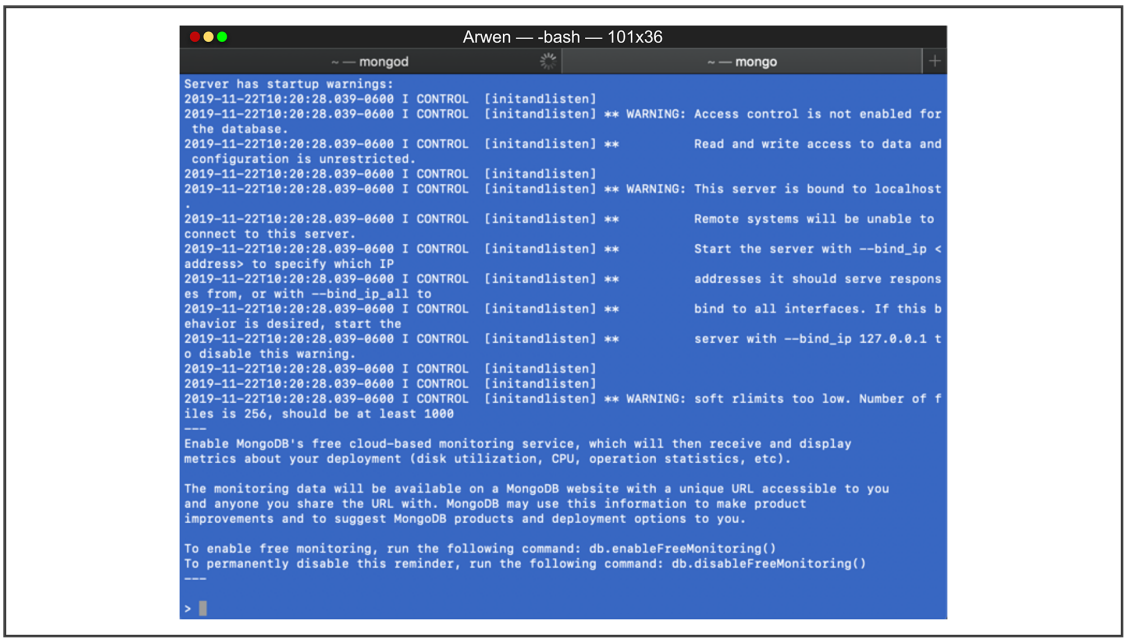


2. Then, to start an instance, type `mongod` into the first line of your terminal and press return or enter on your keyboard. Some Mac users may not need to run this command as Mongo is already running in the background

We need to keep this tab open and active so that the Mongo instance continues to run. While Mongo does have a GUI, similar to pgAdmin for Postgres, we'll be using a command line interface (CLI) to make connections within the database.



3. In our terminal, create a second window or tab to use for working in Mongo. Again, make sure your environment is active. On the first line of this new window, type "mongo."



Note: Either process can be easily stopped and exited from by pressing Command + C or CTRL + C on your keyboard.

Create a Database

We don’t have a fancy GUI to use while navigating through creating a database and inserting data, but that doesn’t mean that our commands will be very complex. Let’s create a new practice database to get used to some of the more common commands.

In the terminal, type “use practicedb” and then press Enter. This creates a new database named “practicedb” and makes it our active database.

Our New Database

```
> use practicedb
switched to db practicedb
>
```

If you're not sure which database you’re using, type “db” in the terminal and press Enter.

What happens when you type “db” into a terminal connected to Mongo?

- It lets us create a new database. We just have to pick a name next.
- It shows us a list of all the databases we’ve made so far.
- It doesn’t do anything on its own. It needs a database name as well.
- It returns the name of the database you’re currently working in.

Check Answer

Finish ▸

After typing “db” into the terminal and pressing Enter, the name of the current active database is returned. This is a quick check to make sure we’ll be saving data to the right spot.

Current Active Database

```
> db
practicedb
>
```

You can also see how many databases are stored locally by typing “show dbs” in your terminal. There should be a few already there by default, so don’t be alarmed if more than one appears that you didn’t create yourself.

There is also a way to check to see what data, or collections, are already in the database. Type “show collections” into the shell, or terminal, then press Enter.

Nothing came up after that, right? That’s a good thing. We haven’t entered any data yet. We’ll practice doing that next.

Insert Data

Now that we’ve confirmed we’re in the right database, we can practice the commands to insert data or a document.

The syntax follows: `db.collectionName.insert({key:value})`. Its components do the following:

- `db` refers to the active database, test.
- `collectionName` is the name of the new collection we’re creating (we’ll customize it when we practice).
- `.insert({ })` is how MongoDB knows we’re inserting data into the collection.
- `key:value` is the format into which we’re inserting our data; its construction is very similar to a Python dictionary.

In short, we're saying, "Hey, Mongo, use the database we've already specified, and insert a document into this collection. If there's not a collection named that, then create one."

Let's explore how this works a bit by adding some zoo animals to our collection.

In the shell, type:

```
db.zoo.insert({name: 'Cleo', species: 'jaguar', age: 12, hobbies: ['sleep
```

After pressing Enter, the next line in your terminal should read `WriteResult({ 'nInserted" : 1 })`. This means that we've successfully inserted Cleo into the database.

Now let's add another animal. In your shell, type the following:

`db.zoo.insert({name: 'Banzai', species: 'fox', age: 1, hobbies: ['sleeping', 'eating', 'playing']})`.

This time we've added a fox to our collection, but the code is very similar to when we added Cleo.

> **Skill Drill**
> Add three more animals to your database, then type "show collections" in your shell.

Now that we've added data to our collection, when we type "show collections" we'll actually see a result: zoo. The name of our new collection is returned.

Documents can also be deleted or dropped. The syntax to do so follows: `db.collectionName.remove({})`.

So, if we wanted to remove Cleo from the database, we would update that line of code to:

`db.zoo.remove({"name": "Cleo"})`.

We can also empty the collection at once, instead of one document at a time. For example, to empty our pets collection, we would type: `db.zoo.remove({})`. Because the inner curly brackets are empty, Mongo will assume that we want everything in our pets collection to be removed.

Additionally, to remove a collection all together, we would use `db.zoo.drop()`. After running that line in the shell, our pets collection will no longer exist at all.

And to remove the test database, we will use this line of code: `db.dropDatabase()`.

> **Skill Drill**
> Remove the animals you added earlier with the `remove({})` method, then drop the database.

Now test your code as you answer the following question:

What is returned when you execute the code to drop a database?

- `{ “dropped” : “test”, “ok” : 1 }`
- `WriteResult({ "nRemoved" : 1 })`
- `WriteResult({ "nRemoved" : 0 })`
- `true`

Check Answer

Finish ▸

You can quit the Mongo shell by using keyboard commands: Command + C for Mac or CTRL + C for Windows. This stops the processes that are actively running and frees up your terminal. Remember to quit both the server and the shell when

you’re done practicing. Otherwise, they’ll continue to run in the background and use system resources, such as memory, and slow down the response time of your computer.

Skill Drill

Create a new database named “mars_app” to hold the Mars data we scrape.

10.4.1: Store the Data

Robin is pretty excited about all of the data we've managed to scrape. And the code is designed to grab the most recent data, so if it's run at a later time, all of the results will have been updated—without us needing to alter the code.

Now that she has the results she wants, she needs to store them in a spot where they can be easily accessed and retrieved as needed. SQL isn't a good option because it works with tabular data, and only one of the items we scraped is presented in that format. Even then, it's all condensed into a block of HTML code.

What Robin will need to use is a database that works differently from SQL with its neatly ordered tables and relationships. Mongo, a NoSQL database, is designed for exactly this task. While Robin is familiar with SQL databases, Mongo is completely new and we'll need to practice with it before loading in our scraped data.

The data Robin has gathered for her web app is great. She's been able to pull a great image, the most recent news article summary, and even an HTML table. Each data type is different, though, with text and images and HTML all together. Compared to SQL's orderly relational system, where each table is linked to at least

one other by a key, the data we've helped Robin gather is a bit chaotic. This is where a non-relational database comes in.

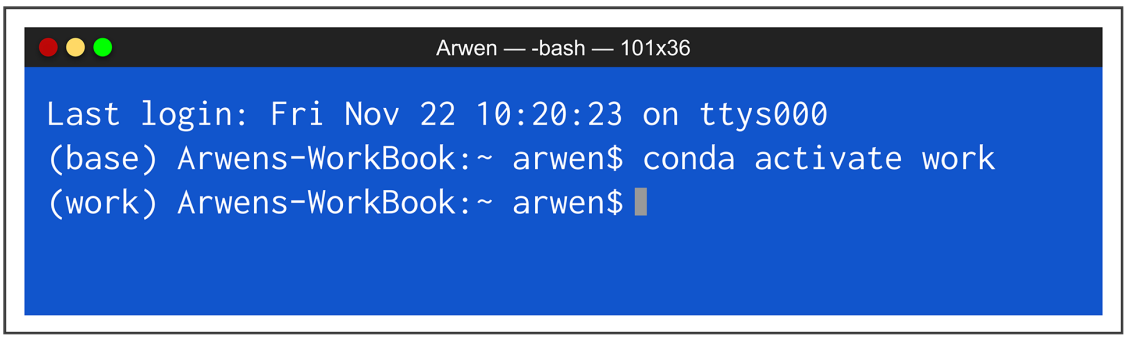
MongoDB (Mongo for short) is a non-relational database that stores data in Binary JavaScript Object Notation (JSON), or BSON format. We'll access data stored in Mongo the same way we access data stored in JSON files. This method of data storage is far more flexible than SQL's model. If you'd like to dig into MongoDB at a deeper level, check out the **official documentation (https://docs.mongodb.com/)**.

> **Rewind**
> JSON, JavaScript Object Notation, is a method that sorts and presents data in the form of an data:value pairs. It looks much like a Python dictionary and can be traversed through using list notation.

A Mongo database contains collections. These collections contain documents, and each document contains fields, and fields are where the data is stored.

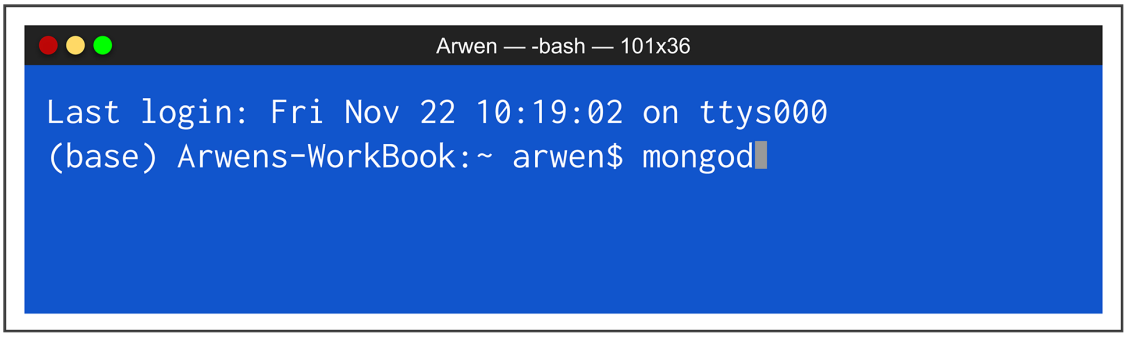
While Mongo and SQL are both databases, that's where the similarities end. They handle documents differently, the storage model isn't even close, and we even interact with them in very different ways.

1. To get started with Mongo, first open a new terminal window, but make sure your working environment is activated. Note that your environment does not need to have the same name as the one in the image.

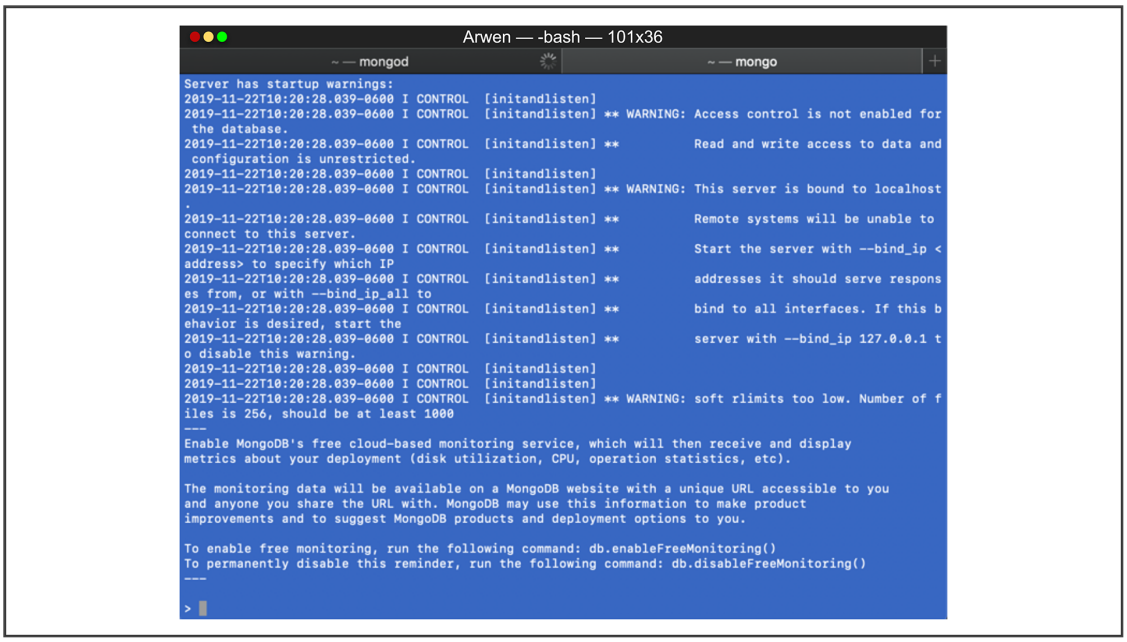


2. Then, to start an instance, type `mongod` into the first line of your terminal and press return or enter on your keyboard. Some Mac users may not need to run this command as Mongo is already running in the background

We need to keep this tab open and active so that the Mongo instance continues to run. While Mongo does have a GUI, similar to pgAdmin for Postgres, we'll be using a command line interface (CLI) to make connections within the database.



3. In our terminal, create a second window or tab to use for working in Mongo. Again, make sure your environment is active. On the first line of this new window, type "mongo."



Note: Either process can be easily stopped and exited from by pressing Command + C or CTRL + C on your keyboard.

Create a Database

We don’t have a fancy GUI to use while navigating through creating a database and inserting data, but that doesn’t mean that our commands will be very complex. Let’s create a new practice database to get used to some of the more common commands.

In the terminal, type “use practicedb” and then press Enter. This creates a new database named “practicedb” and makes it our active database.

Our New Database

```
> use practicedb
switched to db practicedb
>
```

If you're not sure which database you’re using, type “db” in the terminal and press Enter.

What happens when you type “db” into a terminal connected to Mongo?

- It lets us create a new database. We just have to pick a name next.
- It shows us a list of all the databases we’ve made so far.
- It doesn’t do anything on its own. It needs a database name as well.
- It returns the name of the database you’re currently working in.

Check Answer

Finish ▸

After typing “db” into the terminal and pressing Enter, the name of the current active database is returned. This is a quick check to make sure we’ll be saving data to the right spot.

Current Active Database

```
> db
practicedb
>
```

You can also see how many databases are stored locally by typing “show dbs” in your terminal. There should be a few already there by default, so don’t be alarmed if more than one appears that you didn’t create yourself.

There is also a way to check to see what data, or collections, are already in the database. Type “show collections” into the shell, or terminal, then press Enter.

Nothing came up after that, right? That’s a good thing. We haven’t entered any data yet. We’ll practice doing that next.

Insert Data

Now that we’ve confirmed we’re in the right database, we can practice the commands to insert data or a document.

The syntax follows: `db.collectionName.insert({key:value})`. Its components do the following:

- `db` refers to the active database, test.
- `collectionName` is the name of the new collection we’re creating (we’ll customize it when we practice).
- `.insert({ })` is how MongoDB knows we’re inserting data into the collection.
- `key:value` is the format into which we’re inserting our data; its construction is very similar to a Python dictionary.

In short, we're saying, "Hey, Mongo, use the database we've already specified, and insert a document into this collection. If there's not a collection named that, then create one."

Let's explore how this works a bit by adding some zoo animals to our collection.

In the shell, type:

```
db.zoo.insert({name: 'Cleo', species: 'jaguar', age: 12, hobbies: ['sleep
```

After pressing Enter, the next line in your terminal should read `WriteResult({ 'nInserted" : 1 })`. This means that we've successfully inserted Cleo into the database.

Now let's add another animal. In your shell, type the following:

`db.zoo.insert({name: 'Banzai', species: 'fox', age: 1, hobbies: ['sleeping', 'eating', 'playing']})`.

This time we've added a fox to our collection, but the code is very similar to when we added Cleo.

> **Skill Drill**
> Add three more animals to your database, then type "show collections" in your shell.

Now that we've added data to our collection, when we type "show collections" we'll actually see a result: zoo. The name of our new collection is returned.

Documents can also be deleted or dropped. The syntax to do so follows: `db.collectionName.remove({})`.

So, if we wanted to remove Cleo from the database, we would update that line of code to:

`db.zoo.remove({"name": "Cleo"})`.

We can also empty the collection at once, instead of one document at a time. For example, to empty our pets collection, we would type: `db.zoo.remove({})`. Because the inner curly brackets are empty, Mongo will assume that we want everything in our pets collection to be removed.

Additionally, to remove a collection all together, we would use `db.zoo.drop()`. After running that line in the shell, our pets collection will no longer exist at all.

And to remove the test database, we will use this line of code: `db.dropDatabase()`.

> **Skill Drill**
> Remove the animals you added earlier with the `remove({})` method, then drop the database.

Now test your code as you answer the following question:

What is returned when you execute the code to drop a database?

- `{ “dropped” : “test”, “ok” : 1 }`
- `WriteResult({ "nRemoved" : 1 })`
- `WriteResult({ "nRemoved" : 0 })`
- `true`

Check Answer

Finish ▸

You can quit the Mongo shell by using keyboard commands: Command + C for Mac or CTRL + C for Windows. This stops the processes that are actively running and frees up your terminal. Remember to quit both the server and the shell when

you’re done practicing. Otherwise, they’ll continue to run in the background and use system resources, such as memory, and slow down the response time of your computer.

Skill Drill

Create a new database named “mars_app” to hold the Mars data we scrape.

10.5.1: Use Flask to Create a Web App

We've really come a long way in helping Robin prepare to build her web application. After familiarizing ourselves with HTML and its attributes, we've created code to scrape live data from scraping-friendly websites. Once the application is complete, we'll get the latest featured image, news article and its summary, and fact table at the push of a button.

Robin has also studied and practiced with Mongo, a NoSQL database that she'll be using to display the scraped data we've pulled. The next part is actually building the framework for the app using Flask and Mongo together.

One really great part about how we interact with Mongo through the terminal is that it works really well with Python script and Flask.

> **Rewind**
> Flask is a web microframework that helps developers build a web application. The Pythonic tools and libraries it comes with provide the means to create anything from a small webpage or blog or something large enough for commercial use.

In your code editor, first create a subfolder named “apps,” and then create a new `.py` file named `app.py`. This is where we’ll use Flask and Mongo to begin creating Robin’s web app. Let’s begin by importing our tools. In our new Python file, add the following lines of code:

```
from flask import Flask, render_template
from flask_pymongo import PyMongo
import scraping
```

Let’s break down what this code is doing.

- The first line says that we’ll use Flask to render a template.
- The second line says we’ll use PyMongo to interact with our Mongo database.
- Finally, the last line says that to use the scraping code, we converted from Jupyter to Python.

Under the imports, let’s add another line of code to set up Flask:

```
app = Flask(__name__)
```

We also need to tell Python how to connect to Mongo using PyMongo. In the code editor, add these lines:

```
# Use flask_pymongo to set up mongo connection
app.config["MONGO_URI"] = "mongodb://localhost:27017/mars_app"
mongo = PyMongo(app)
```

- `app.config["MONGO_URI"]` tells Python that our app will connect to Mongo using a URI, a uniform resource identifier similar to a URL.
- `"mongodb://localhost:27017/mars_app"` is the URI we’ll be using to connect our app to Mongo. This URI is saying that the app can reach Mongo through our localhost server, using port 27017, using a database named mars_app”.

Set Up App Routes

The code we create next will set up our Flask routes: one for the main HTML page everyone will view when visiting the web app, and one to actually scrape new data using the code we've written.

> **Rewind**
> Flask routes bind URLs to functions. For example, the URL "`ourpage.com/`" brings us to the homepage of our web app. The URL "`ourpage.com/scrap`" will activate our scraping code.
>
> These routes can be embedded into our web app and accessed via links or buttons.

First, let's define the route for the HTML page. In our script, type the following:

```
@app.route("/")
def index():
   mars = mongo.db.mars.find_one()
   return render_template("index.html", mars=mars)
```

This route, `@app.route("/")`, tells Flask what to display when we're looking at the home page, index.html. This means that when we visit our web app's html page, we will see the home page.

Within the `def index():` function the following is accomplished:

`mars = mongo.db.mars.find_one()` - This line uses PyMongo to find the "mars" collection in our database, which we created when we converted our Jupyter scraping code to Python Script. We also assigned that path to a variable for use later.

`return render_template("index.html"` tells Flask to return an HTML template using an index.html file. We'll create this file after we build the Flask routes.

`, mars=mars)` tells Python to use the "mars" collection in MongoDB.

This function is what links our visual representation of our work, our web app, to the code that powers it.

Our next function will set up our scraping route. This route will be the "button" of the web application, the one that will scrape updated data when we tell it to from the homepage of our web app. It'll be tied to a button that will run the code when it's clicked.

Let's add the next route and function to our code. In the editor, type the following:

```
@app.route("/scrape")
def scrape():
   mars = mongo.db.mars
   mars_data = scraping.scrape_all()
   mars.update({}, mars_data, upsert=True)
   return "Scraping Successful!"
```

Let's look at these six lines a little closer.

The first line, `@app.route("/scrape")` defines the route that Flask will be using. This route, `"/scrape"`, will run the function that we create just beneath it.

If we were to visit `"index.html/scrape"` (instead of just `"index.html"`), the code within the function would run without us clicking the button from the homepage of the app. For example, we would visit localhost:5000/scrape to activate our scraping function.

The next lines allow us to access the database, scrape new data using our `scraping.py` script, update the database, and return a message when successful. Let's break it down.

First, we define it with `def scrape():`.

Then, we assign a new variable that points to our Mongo database: `mars = mongo.db.mars`.

Next, we created a new variable to hold the newly scraped data: `mars_data = scraping.scrape_all()`.

Now that we've gathered new data, we need to update the database using `.update()`. Within that, we'll add `upsert=True`, which tells Mongo to create a new document if one doesn't already exist. This way, we'll always save our new data, even if there's not an existing document for it. The entire line of code looks like this: `mars.update({}, mars_data, upsert=True)`.

Finally, we will add a message to let us know that the scraping was successful: `return "Scraping successful!"`.

The final bit of code we need for Flask is to tell it to run. Add these two lines to the bottom of your script and save your work:

```
if __name__ == "__main__":
   app.run()
```

> **Add, Commit, Push**
>
> Don't forget to commit and push your work!

10.5.2: Update the Code

Robin's almost ready to launch her web app. She's created scraping code and a template, and she's defined the two Flask routes her web app will be using. She's completed a ton of work and made it a really long way—her dreams of working at NASA seem that much closer with all she's accomplished.

Before her code is ready for deployment, she'll need to integrate her scraping code in a way that Flask can handle. That means updating it to include functions and even some error handling. This will help with our app's performance and add a level of professionalism to the end product.

We've already downloaded our Jupyter Notebook code and converted it to a Python script, but it's not quite ready to be used as part of our Flask app yet. The bulk of our code will remain the same—we know it works and will successfully pull the data we need. There are two big things we want to update in our code: we want to refactor it to include functions, and we will be adding some error handling into the mix.

> **Rewind**
> Functions are a very necessary part of programming. They allow developers to create code that will be reused as needed, instead of needing to rewrite

> the same code repeatedly.

In our case, we want our code to be reused, and often, to pull the most recent data. That's what web scraping is all about, right? Pulling in the live data at the click of a button. Functions enable this capability by bundling our code into something that is easy for us (and once it's deployed, whoever else we share the web app with) to use and reuse as needed.

Also, because the intention is to reuse this code often, we need to update our `scraping.py` script to use functions. Each major scrape, such as the news title and paragraph or featured image, will be divided into a self-contained, reusable function. Let's take a look at our code.

News Title and Paragraph

Our first scrape, the news title and paragraph summary, currently looks like this:

```
# Visit the mars nasa news site
url = 'https://mars.nasa.gov/news/'
browser.visit(url)
# Optional delay for loading the page
browser.is_element_present_by_css("ul.item_list li.slide", wait_time=1)

# Convert the browser html to a soup object and then quit the browser
html = browser.html
news_soup = soup(html, 'html.parser')
slide_elem = news_soup.select_one('ul.item_list li.slide')
slide_elem.find("div", class_='content_title')

# Use the parent element to find the first 'a' tag and save it as 'news_t
news_title = slide_elem.find("div", class_='content_title').get_text()
news_title

# Use the parent element to find the paragraph text
news_p = slide_elem.find('div', class_="article_teaser_body").get_text()
news_p
```

```
news_p
```

Next, we will go back to the top of that block of code and define our function. Let's call it `mars_news`.

```
def mars_news():

    # Visit the mars nasa news site
    url = 'https://mars.nasa.gov/news/'
    browser.visit(url)
    # Optional delay for loading the page
    browser.is_element_present_by_css("ul.item_list li.slide", wait_time=1

    # Convert the browser html to a soup object and then quit the browser
    html = browser.html
    news_soup = soup(html, 'html.parser')
    slide_elem = news_soup.select_one('ul.item_list li.slide')
    slide_elem.find("div", class_='content_title')
    # Use the parent element to find the first <a> tag and save it as  `ne
    news_title = slide_elem.find("div", class_='content_title').get_text()
    news_title
    # Use the parent element to find the paragraph text
    news_p = slide_elem.find('div', class_="article_teaser_body").get_text
    news_p
```

To complete the function, we need to add a return statement.

What is the purpose of a return statement?

- When a return statement is included in a function, it's used as a print statement.
- A return statement tells Python that the function is complete.
- The return statement provides the developer with feedback to show if the function is working correctly or not.

Check Answer

Finish ▸

Instead of having our title and paragraph printed within the function, we want them printed at the end of our function. This is because the code within is self-contained, but we want the data available as part of the larger script. When code is self-contained within a function, it serves as its own miniscript that can run by itself. Even though it's self-contained, in our script it relies on other functions to operate to its full potential.

At the bottom of our function, add a return statement that includes the news title and paragraph.

```
def mars_news():

    # Visit the mars nasa news site
    url = 'https://mars.nasa.gov/news/'
    browser.visit(url)
    # Optional delay for loading the page
    browser.is_element_present_by_css("ul.item_list li.slide", wait_time=1

    # Convert the browser html to a soup object and then quit the browser
    html = browser.html
    news_soup = soup(html, 'html.parser')
    slide_elem = news_soup.select_one('ul.item_list li.slide')
    # Use the parent element to find the first <a> tag and save it as `new
    news_title = slide_elem.find("div", class_='content_title').get_text()
    # Use the parent element to find the paragraph text
    news_p = slide_elem.find('div', class_="article_teaser_body").get_text
```

```
    news_p = slide_elem.find('div', class_='article_teaser_body').get_text

    return news_title, news_p
```

This function is looking really good. There are two things left to do. First, we need to add an argument to the function.

Update your function like this:

```
def mars_news(browser):
```

When we added the word "`browser`" to our function, we're telling Python that we'll be using the **browser** variable we defined above. All of our scraping code utilizes an automated browser, and without this section, our function wouldn't work.

The finishing touch is to add error handling to the mix. This is to address any potential errors that may occur during web scraping. Errors can pop up from anywhere, but in web scraping the most common cause of an error is when the webpage's format has changed and the scraping code no longer matches the new HTML elements.

We're going to add a try and except clause addressing AttributeErrors. By adding this error handling, we are able to continue with our other scraping portions even if this one doesn't work.

In our code, we're going to add the `try` portion right before the scraping:

```
    # Add try/except for error handling
    try:
        slide_elem = news_soup.select_one("ul.item_list li.slide")
        # Use the parent element to find the first 'a' tag and save it as
        news_title = slide_elem.find("div", class_="content_title").get_t
```

```
        # Use the parent element to find the paragraph text
        news_p = slide_elem.find("div", class_="article_teaser_body").get
```

After adding the `try` portion of our error handling, we need to add the `except` part. After these lines, we'll immediately add the following:

```
    except AttributeError:
        return None, None
```

By adding `try:` just before scraping, we're telling Python to look for these elements. If there's an error, Python will continue to run the remainder of the code. If it runs into an AttributeError, however, instead of returning the title and paragraph, Python will return nothing instead.

Let's update our featured image the same way.

Featured Image

The code to scrape the featured image will be updated in almost the exact same way we just updated the `mars_news` section. We will:

1. Declare and define our function.

   ```
   def featured_image(browser):
   ```

2. Remove print statement(s) and return them instead.

   In our Jupyter Notebook version of the code, we printed the results of our scraping by simply stating the variable (e.g., after assigning data to the `img_url` variable, we simply put `img_url` on the next line to view the data).

We still want to view the data output in our Python script, but we want to see it at the end of our function instead of within it.

```
return img_url
```

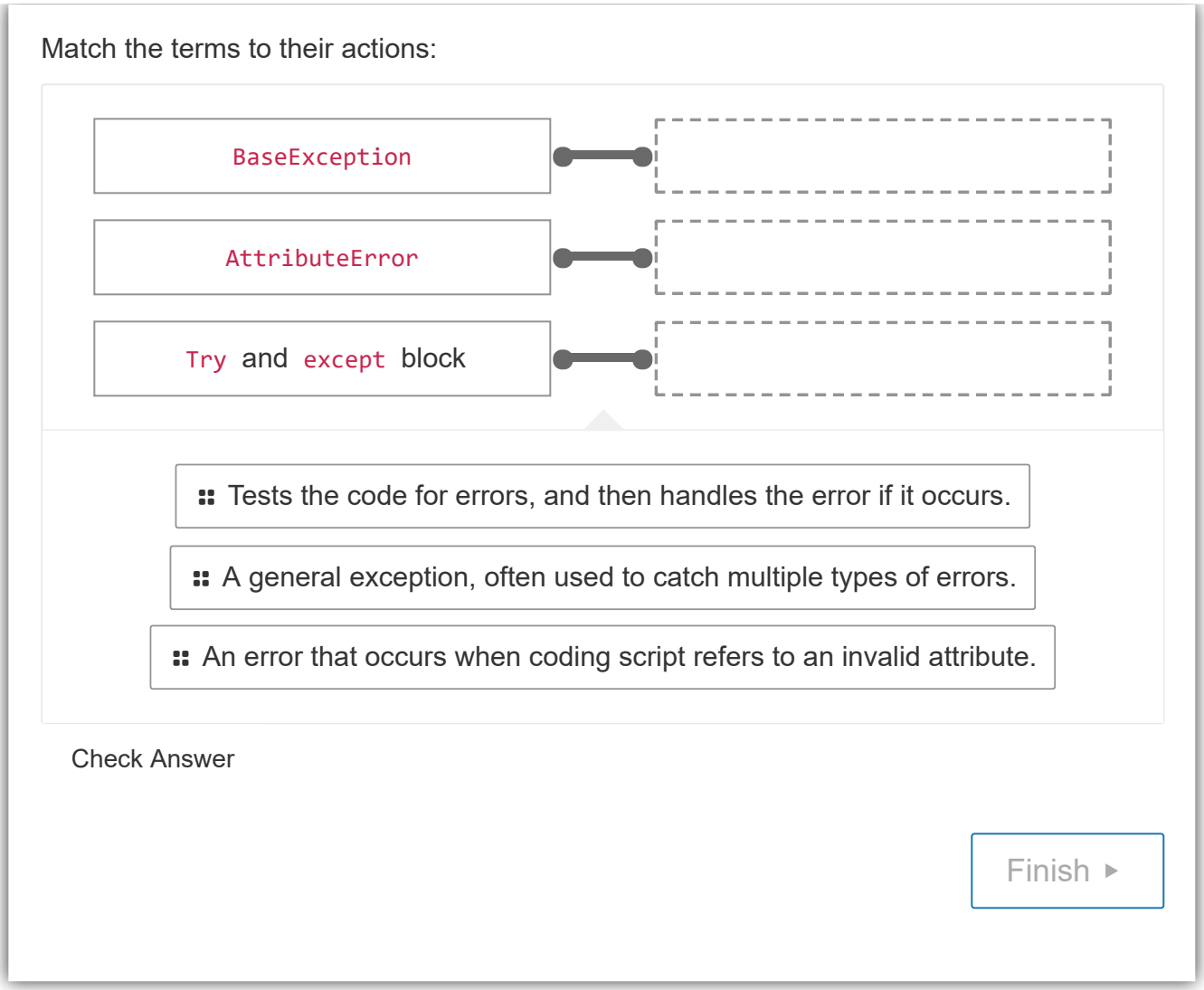
3. Add error handling for `AttributeError`.

```
try:
    # find the relative image url
    img_url_rel = img_soup.select_one('figure.lede a img').get("src")

except AttributeError:
    return None
```

Mars Facts

Code for the facts table will be updated in a similar manner to the other two. This time, though, we'll be adding BaseException to our except block for error handling.



A `BaseException` is a little bit of a catchall when it comes to error handling. We're using it here because we're using Pandas' `read_html()` function to pull data, instead of scraping with BeautifulSoup and Splinter. The data is returned a little differently and can result in errors other than AttributeErrors, which is what we've been addressing so far.

Let's first define our function:

```
def mars_facts():
```

Next, we'll update our code by adding the `try` and `except` block.

```
    try:
        # use 'read_html" to scrape the facts table into a dataframe
```

```
        df = pd.read_html('http://space-facts.com/mars/')[0]
    except BaseException:
        return: None
```

As before, we've removed the print statements. Now that we know this code is working correctly, we don't need to view the DataFrame that's generated.

The code to assign columns and set the index of the DataFrame will remain the same, so the last update we need to complete for this function is to add the return statement.

```
    return df.to_html()
```

The full `mars_facts` function should look like this:

```
def mars_facts():
    # Add try/except for error handling
    try:
        # Use 'read_html' to scrape the facts table into a dataframe
        df = pd.read_html('http://space-facts.com/mars/')[0]

    except BaseException:
        return None
    # Assign columns and set index of dataframe
    df.columns=['Description', 'Mars', 'Earth']
    df.set_index('Description', inplace=True)
    # Convert dataframe into HTML format, add bootstrap
    return df.to_html()
```

Now you're ready to integrate Mongo.

10.5.3: Integrate MongoDB Into the Web App

And now to add the very last bit of code before the coat of HTML paint. Robin has refactored her code so that it separates each scraping section into its own function, which will make reusing the code a much simpler task. She has already built out the Flask routes as well, which is an integral part of scraping—without the routes, the web app simply wouldn't function.

Robin has also set up a Mongo database to hold the data that gets scraped. The next step is to integrate Mongo into the web app. She wants the script to update the data stored in Mongo each time it's run. We need to add just a little bit more code to our `scraping.py` script to establish the link between scraped data and the database.

Before we make our website look pretty (you never know when NASA is looking for its new analyst.), we need to connect to Mongo and establish communication between our code and the database we're using. We'll add this last bit of code to our `scraping.py` script.

At the top of our `scraping.py` script, just after importing the dependencies, we'll add one more function. This function differs from the others in that it will:

1. Initialize the browser.

2. Create a data dictionary.
3. End the WebDriver and return the scraped data.

Let's define this function as "`scrape_all`" and then initiate the browser.

```
def scrape_all():
   # Initiate headless driver for deployment
   browser = Browser("chrome", executable_path="chromedriver", headless=T
```

When we were testing our code in Jupyter, **headless** was set as False so we could see the scraping in action. Now that we are deploying our code into a usable web app, we don't need to watch the script work (though it's totally okay if you still want to).

Next, we're going to set our news title and paragraph variables.

```
   news_title, news_paragraph = mars_news(browser)
```

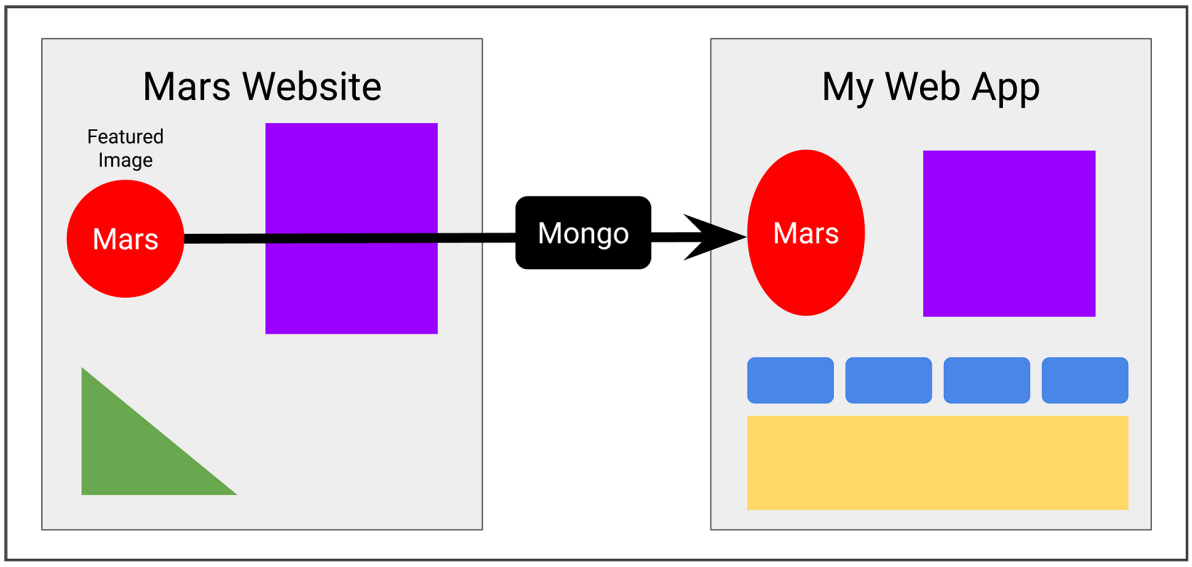
This line of code tells Python that we'll be using our `mars_news` function to pull this data.

Now that we have our browser ready for work, we need to create the data dictionary.

```
# Run all scraping functions and store results in dictionary
data = {
      "news_title": news_title,
      "news_paragraph": news_paragraph,
      "featured_image": featured_image(browser),
      "facts": mars_facts(),
      "last_modified": dt.datetime.now()
}
```

This dictionary does two things: It runs all of the functions we've created—`featured_image(browser)`, for example—and it also stores all of the results. When we create the HTML template, we'll create paths to the dictionary's values, which lets us present our data on our template. We're also adding the date the code was run last by adding `"last_modified": dt.datetime.now()`. For this line to work correctly, we'll also need to add `import datetime as dt` to our imported dependencies at the beginning of our code.

For example, we're collecting the path to the featured image, then storing it in our database, then placing that link on our web application for everyone to see. We're basically finding the link to the image page and then reusing it on our own page.



The last step we need to add is similar to the last code block in our `app.py` file. At the bottom of our `scraping.py` script, add the following:

```
if __name__ == "__main__":
    # If running as script, print scraped data
    print(scrape_all())
```

This last block of code tells Flask that our script is complete and ready for action. The print statement will print out the results of our scraping to our terminal after executing the code.

Add, Commit, Push

Don’t forget to save your code and push it to your repo.

10.6.1: Customize the Appearance

The code you just created covers a lot of ground. Not only does it scrape new data on command, but it does it by incorporating your other code. By importing the script you wrote to scrape code, you were able to condense our new code into fewer lines, which is easier to read and troubleshoot.

You also added Flask, which allows you to put all of the data you've gathered into an easy-to-view web application. With each successfully completed step on the way to a full web app, Robin feels more confident about boosting her portfolio and eventually working for NASA.

The default Flask template isn't very interesting or inspiring, though, even with all of the neat data you've gathered. Robin really wants her web app to wow people, so to add a bit of extra polish, you'll use Bootstrap components to enhance the HTML and CSS of the `index.html` file.

Now that our code includes Flask, we need to create an HTML template for it.

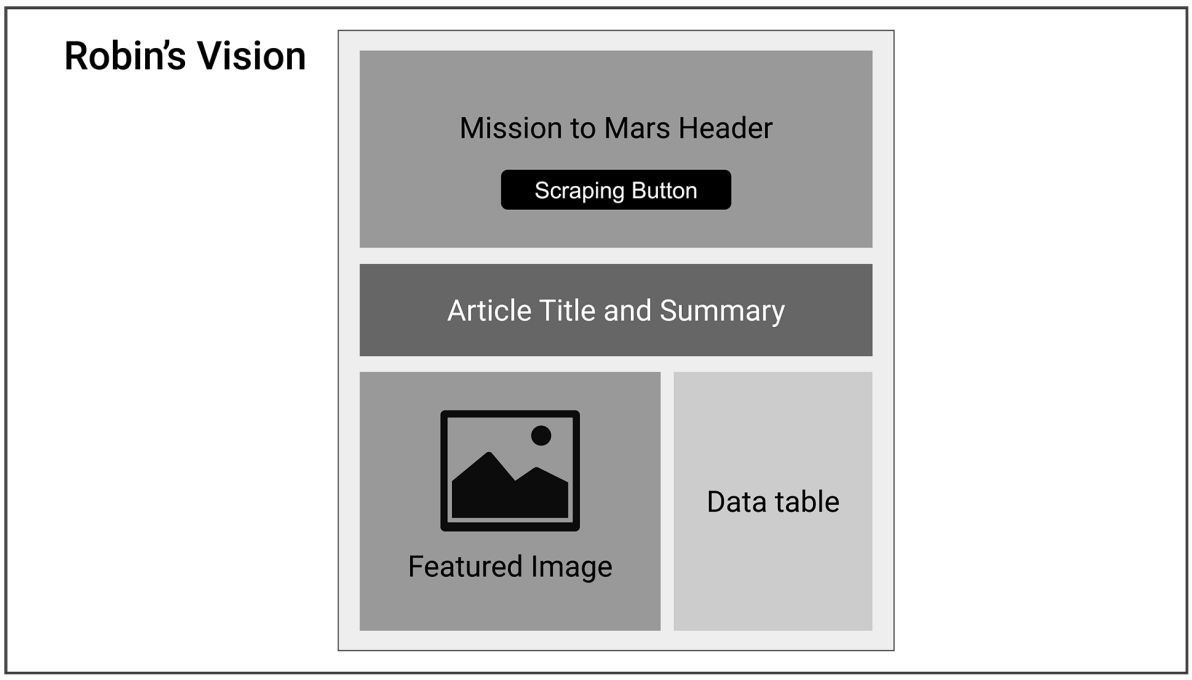
Create HTML Template

In our apps folder, create another subfolder named “templates”—this is where our HTML file will be saved. It’s important to name and arrange these folders and files as we’ve lined out here. Flask templating works because Flask already “knows” what to look for (such as a “templates” folder and not a “Templates” folder); so, if capitalization or location are off, Flask won’t recognize them.

Go ahead and create a file named `index.html` while we’re here.

> **Skill Drill**
>
> Open a new, blank `index.html` file with VSCode. Create a basic HTML template by using the exclamation point shortcut.

Let’s think about how we want the page to look for a moment. We have three sections we want displayed on the web app: a news article title and summary, a featured image, and the table of Mars facts. We’ll also want a nice header at the top to really help the app stand out—something like so:



Not only will we have to think about how we're laying these pieces out, but we'll also be adding Bootstrap in to really add some polish.

A benefit of Bootstrap is that it is responsive. By responsive, we mean webpages that react to different screen sizes. For example, webpages that resize images based on a screen size is considered responsive. Another example is when content is reorganized based on screen size: three items in a horizontal line on a large screen becomes three items stacked on top of each other when viewed on a small screen.

It's extremely popular and comes with loads of built-in components that makes designing a webpage a much simpler process. Before we can do anything, we'll need to add Bootstrap's content delivery network (CDN) to our HTML.

A CDN is a group of servers that work together to provide users with content. In Bootstrap's case, it means that we use the Bootstrap data that lives on those servers and pick and choose what we need for our project.

An alternative to using a CDN is to download all of the components to our computers, which takes up space on our machines. Since we're only using a few of the components, it's more efficient to use the CDN.

> **Important**
> Like many tech tools and coding languages, Bootstrap has many different versions of release. We'll be using Bootstrap 3 to customize Robin's web app because it's a stable release.
>
> When developers designate a version of their product as "stable," they're saying that this version has been thoroughly tested and is as bug-free as possible. Newer releases are often getting kinks worked out, which can often lead to more research and time spent debugging your code.

In our `index.html` file, we need to add a link for Bootstrap's stylesheet. This is where we can access all of Bootstrap's components without needing to download

the files themselves. This is a CDN: the content is delivered to our computer through their service.

Between the `<head />` tags, add this link to the bottom of the list:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.
```

Your `index.html` should look like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0"
   <meta http-equiv="X-UA-Compatible" content="ie=edge" />
   <title>Mission to Mars</title>
   <link
     rel="stylesheet"
     href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.
   />

</head>
</html>
```

Now that we have the stylesheet selected, we can start adding Bootstrap components. Components include buttons, thumbnails, alerts—there are many. One of the biggest features that Bootstrap offers is its grid system, which is what makes Bootstrap so responsive and popular. But before we start putting components together and into a grid, we need to set up the container.

Set Up Containers

Each HTML page with Bootstrap follows the same layout: The very first tag is a `<div />` with a class of `"container."` This is because a container is required for the grid system to work. In your HTML document, in the `<body />`, add the following:

```
<div class="container">
</div>
```

Everything added to the webpage from now on will be inside of this container. What is the best way to visually keep track of the code we are adding?

- Comment every line of code with an explanation.
- Create new containers for each block of code, so there are containers within containers.
- Nest the code neatly, using proper indentation.

Check Answer

Finish ▸

HTML can get unwieldy fairly quickly—especially when we continue to refine it by adding classes and ids—so keeping the code neat is important. Below is how the page should look now:

```
<head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0"
   <meta http-equiv="X-UA-Compatible" content="ie=edge" />
   <title>Mission to Mars</title>
   <link
     rel="stylesheet"
     href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.
   />
 </head>
 <body>
   <div class="container">
```

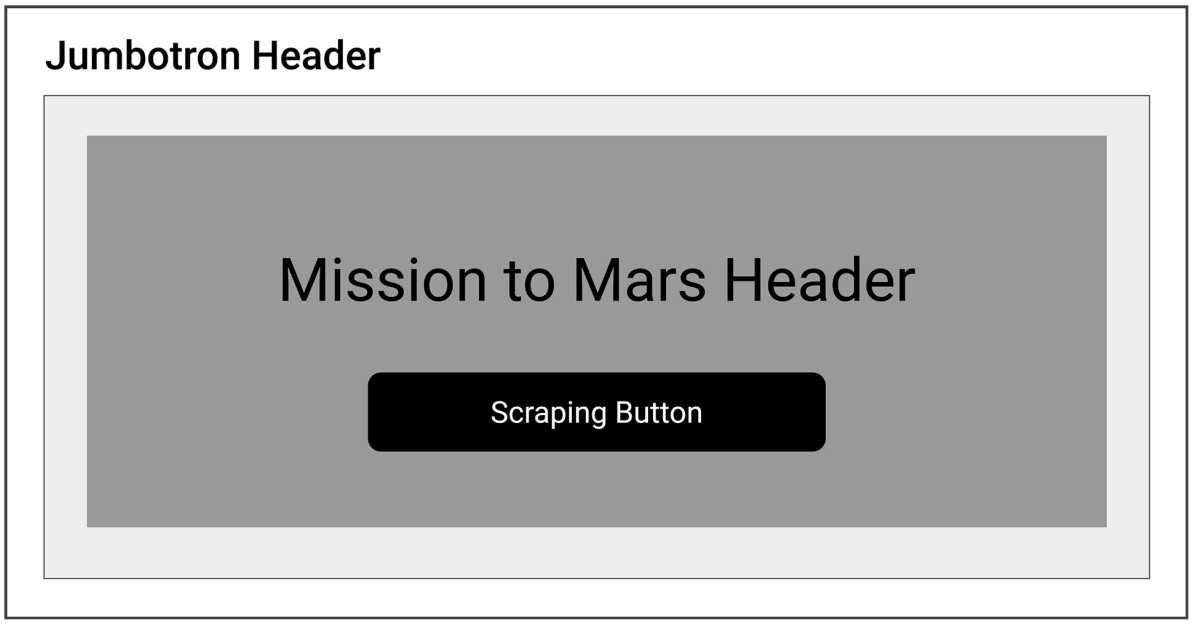
```
</div>
```

Now that we have the basic layout for the grid system ready to go, we can add components. Let's start with the header.

Add the Header Using Jumbotron

For our header we'll use the Bootstrap Jumbotron component. In a non-programming context, a Jumbotron is a very large video display screen used in venues like sports stadiums.

Similarly, Bootstrap's Jumbotron makes the header BIG. Here's where the Jumbotron header will appear in relation to other page elements.



This Jumbotron component enhances the older style `<h1 />` header without additional customizations through CSS (though we can certainly customize it further if we want to later). In the sample image above, we have our "Mission to Mars" title and Scraping button inside the Jumbotron. This way, both items are front and center on the web app.

We'll set it up by creating a set of `<div />` tags first. Once they're in place, we'll add `class="jumbotron text-center"` to the opening tag. By using these specific class attributes, we're adding the Jumbotron component and centering the text. This way, Bootstrap knows that these items are getting the Jumbotron enhancement. Now our code should look like the following:

```
<div class="container">
   <div class="jumbotron text-center">
   </div>
</div>
```

Now we've built our Jumbotron, let's add the main text. The next level of nesting, within the Jumbotron div, is where we'll add the main text and the button we'll use to run our scraping script.

We already know that we want the highest-level header, so let's add an `<h1 />` tag. We also know that Robin wants to scrape data with the click of a button, so we'll add that too.

```
<div class="container">
   <div class="jumbotron text-center">
      <h1>Mission to Mars</h1>
      <p><a class="btn btn-primary btn-lg" href="/scrape"
                  role="button">Scrape New Data</a></p>
   </div>
</div>
```

The `<h1 />` tag is pretty straightforward. We don't need to dress it up at all because the Jumbotron is already doing that for us. The button is a little more complex, though.

First, we added a `<p />` tag to the Jumbotron, without any classes. This is because it's only there to help with formatting; it helps styling the content within the tag. Then we've added an `<a />` tag with a few different classes included.

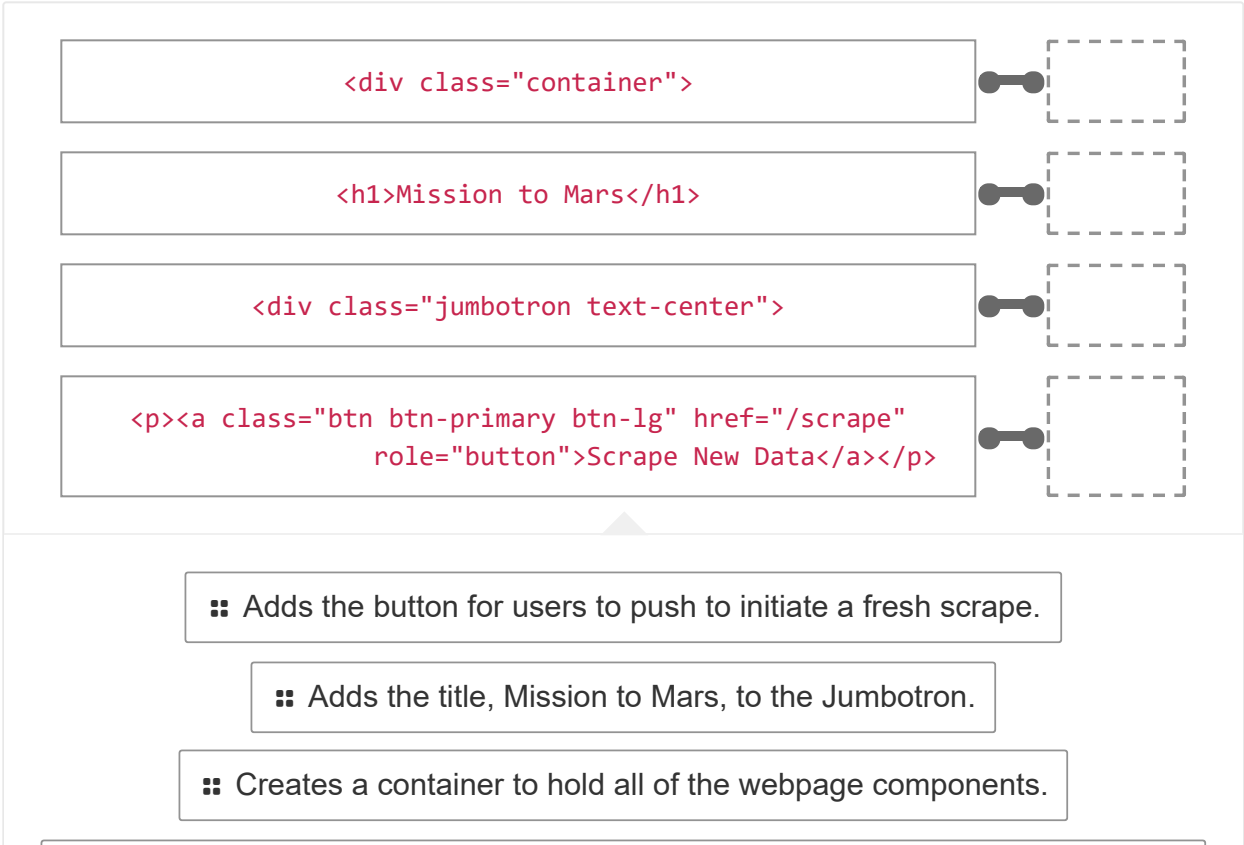
The `"btn btn-primary btn-lg"` classes are also part of Bootstrap's button component. There are three classes here because we're telling Bootstrap that we're using a button, it's the primary color, and it's large.

An `<a />` tag is specifically used to link to other things, so we've also included the href (hypertext reference). The href is the link to another document or webpage. In our case, we're linking it to another component of our page, `"/scrape"`.

We've added a "button" role so that the webpage knows this link functions as a button, not as a regular hyperlink. This just boils down to aesthetics, though.

The text in our button is "Scrape New Data," and then we complete the code by closing both tags.

Match the HTML snippet to the action it's taking:



Check Answer

Finish ▸

Add the Mars News Article and Summary

Next we'll add in the news article and summary and use the grid system that Bootstrap offers.

Bootstrap Grid is a very useful tool because it provides extremely flexible and customizable layouts for all of your HTML components. It consists of a column and row layout with up to 12 columns per row. We're not required to use every column available, and we can mix and match how many items fit into a row.

It's also responsive, which means that it will adapt to whatever browser is being used to view content. Whether the app is viewed on a laptop, a desktop computer, or a tablet, the webpage content will adjust to best fit the screen size. We'll experiment with this as we continue to build the web app.

For the article and its summary, we'll want it to sit right under the header and span across the page. To span the entire page, we'll utilize all 12 of the grid columns. Let's set the grid up first.

Under the header, add a comment for clarity. This way we'll know where the code for each component is.

```
<!-- Mars News -->
```

Now we can add the code, which will be more complex than what we've written so far in HTML. There are a lot of nested tags, so we'll add one at a time and discuss them as we go.

First, create a new div with a class of `"row"` and an id of `"mars-news"`.

```
<div class="row" id="mars-news">
</div>
```

By adding a class of `"row"` we're telling Bootstrap that we're about to use the grid system and everything within this div will be horizontally aligned. The id `"mars-news"` helps us select it later if we want to customize it later with CSS.

Next we'll add a tag that specifies how many columns this component will be using. We want the article and summary to span the width of the page, so we'll use all columns. Update your code as follows:

```
<div class="row" id="mars-news">
<div class="col-md-12">

    </div>
</div>
```

The `md` part of our class is telling Bootstrap that we'll be optimizing for a midsize resolution. The other options are `sm` and `lg` for small and large, respectively.

Now we'll add the media portion: the article and summary. Because it's a news source, we want it to be displayed as media. There are two more tags we'll add to achieve this, so let's update our code with them.

```
<div class="row" id="mars-news">
   <div class="col-md-12">
    <div class="media">
<div class="media-body">

        </div>
    </div>
   </div>
</div>
```

With this update, we've nested in two more tags: one with a class of `"media"` and one with a class of `"media-body."` First, this tells Bootstrap that we want our content to be displayed as media, and then we insert the media within the body. Bootstrap already has custom CSS set up for these classes. Now our setup is complete and we can add the article title and summary.

First, we'll add a header, this time a second-level one, so this section has a title.

```
<h2>Latest Mars News</h2>
```

Next, we need to add the name of the article we scraped. This is where Flask comes into play. We'll use a fourth-level header for the title of the article.

```
<h4>{{mars.news_title}}</h4>
```

How is the title being rendered?

- The HTML extracts data from the Python code and inserts it into the title for us.
- The curly brackets are part of the templating; we're just telling Flask which data we want inserted.
- It won't; we actually need to use parentheses to extract the data and present it here.

Check Answer

Finish ▸

Let's add the article summary the same way, only using paragraph tags.

```
<p>{{ mars.news_paragraph }}</p>
```

We didn't add any classes or ids to this paragraph tag because the default text is fine. If we want to change its appearance later, we can add our own CSS stylesheet.

All together with proper nesting, our code should look like this:

```
<!-- Mars News -->
<div class="row" id="mars-news">
     <div class="col-md-12">
       <div class="media">
         <div class="media-body">
           <h2>Latest Mars News</h2>
           <h4 class="media-heading">{{mars.news_title}}</h4>
           <p>{{mars.news_paragraph}}</p>
         </div>
       </div>
```

```
    </div>
  </div>
```

Add the Featured Image

Now we'll add the featured image that we scraped. This time, we want the image and the table of Mars facts to be next to each other, so we won't use the full 12 columns on the image. First, let's add a comment so we know which piece of data we're inserting.

```
<!-- Featured Image -->
```

Now we'll add the code that'll insert the image.

Which of the following options is the correct format to add the image?

```
<div class="row" id="mars-featured-image">

  <div class="col-md-8">

    <h2>Featured Mars Image</h2>

    <img src="{{ mars.featured_image }}" class="img-responsive" alt="Respo
nsive image"/>

  </div>

</div>
```

```
<div class="row" id="mars-featured-image">

<div class="col-md-8">

<h2>Featured Mars Image</h2>

<img src="{{ mars.featured_image }}" class="img-responsive" alt="Responsiv
e image"/>

</div>

</div>
```

```
<div class="row" id="mars-featured-image">

  <div class="col-md-8">

    <h2>Featured Mars Image</h2>

    <img src="mars.featured_image" class="img-responsive" alt="Responsive
image"/>

  </div>

</div>
```

Check Answer

Finish ▸

Let’s walk through these lines and how they’ve been assembled.

Each new item we're inserting now is on its own row, so the very first div will have a class of `"row."`

```
<div class="row" id="mars-featured-image">

</div>
```

We also added an id for later customization.

Next, we'll add the div for the rows; 8 this time, with 4 remaining for the facts table we'll add later.

```
<div class="row" id="mars-featured-image">
   <div class="col-md-8">

   </div>
</div>
```

Now we're going to add the code to insert the image. Add the following to your code:

```
<img src="{{ mars.featured_image }}" class="img-responsive" alt="Responsi
```

This line inserts an image using the `<img />` tag, but the tag alone won't actually insert the image. We also need to include the source, or link to where the image is. We do this by adding `src="{{ mars.featured_image }}."` The source in this case is our `mars` collection and the `featured_image` document in Mongo.

When we add `class="img-responsive,"` we're using another built-in Bootstrap component that makes the image responsive. That means that the size of the image varies depending on the browser used, without us having to add extra code to do so.

The last portion, `alt="Responsive image,"` adds alt-text to our image. Alt-text is just text that will appear if the image doesn't load, or will be read by a screen

reader if one is used.The benefit of alt-text lies in accessibility: Visually impaired users will have the opportunity to better understand a webpage without actually viewing the image on it.

Add the Mars Facts

We've gotten two of the three items we've scraped into our HTML. Let's add the third, our table of Mars facts.

Remember how the featured image only used 8 of the 12 columns? This is where we'll use the remaining 4. They're on the same row, though, so we need to be careful about where we place the new tags.

To display two different items on a webpage, such as an image and a fact table, how would the tags be nested?

- The code for the image and the fact table both need to be nested inside the `<div class="row"/>` tags.
- The fact table will need its own row, so there will be a new set of `<div />` tags added to the container.

Check Answer

Finish ▸

Let's take a look at the code we've created so far, starting with the featured image.

```
<!-- Mars Featured Image -->
<div class="row" id="mars-featured-image">
    <div class="col-md-8">
        <h2>Featured Mars Image</h2>
        <img src="{{ mars.featured_image }}" class="img-responsive" alt=
    </div>
</div>
```

When we add the table of facts, we want it to be on the same row as the featured image. This means that we want it to be nested inside the first level of `<div />` tags, but at the same level as the second set. In the code below, note the comments indicating the first and second level elements.

```
    <!-- Mars Featured Image →
    <!-- First level -->
    <div class="row" id="mars-featured-image">
      <!-- Second level -->
      <div class="col-md-8">
        <h2>Featured Mars Image</h2>
        <img src="{{mars.featured_image | default('static/images/error.pr
      </div>
    </div>
```

We'll add new code after the closing tags of the second `<div />`. In our editor, let's add a new column class, this time with four columns.

```
<!-- Mars Featured Image -->
<div class="row" id="mars-featured-image">
     <div class="col-md-8">
         <h2>Featured Mars Image</h2>
         <img src="{{ mars.featured_image }}" class="img-responsive" alt=
     </div>
     <!-- Mars Facts -->
     <div class="col-md-4">
     </div>
</div>
```

By adding the new set of div tags this way, we're completing the row and have used all 12 columns worth of space. Inside of these new tags, we'll insert the table.

Skill Drill

> Add the table of Mars facts to the web app. Use a fourth-level header to add a “Mars Facts” title, then add the table under the header.

Let’s take a look at our entire page to see how it’s coming together.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Mission to Mars</title>
    <link
      rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootst
    />
  </head>
  <body>
    <div class="container">
      <!-- Add Jumbotron to Header -->
      <div class="jumbotron text-center">
        <h1>Mission to Mars</h1>
        <!-- Add a button to activate scraping script -->
        <p><a class="btn btn-primary btn-lg" href="/scrape" role="butt
      </div>
```

> **Important**
>
> The line `{{ mars.facts | safe }}` tells the web browser that this code doesn’t contain anything malicious. See the **official Template Designer Documentation (https://jinja.palletsprojects.com/en/2.10.x/templates/#working-with-automatic-escaping)** page for more information.

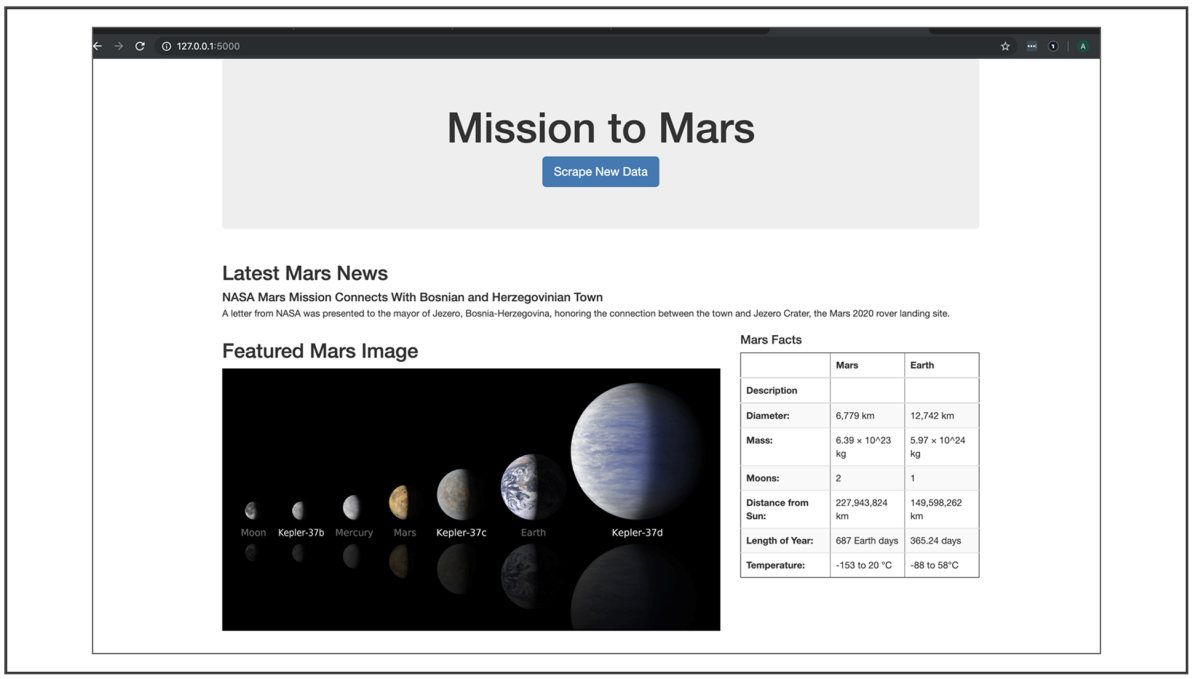
Adding only three items has created many, many lines of code. That’s why

following style guidelines is so important. Imagine if none of the tabs were aligned so neatly—it would be very difficult to see what's going on in there.

After saving the file, let's open it to see how it looks in a web browser. Without MongoDB running, it's just a shell.

But even a shell, we can see how it will all fit together like puzzle pieces cleanly clicking into place. After we click the Scrape New Data button, you'll see the terminal working and updating, and eventually a "Scraping successful!" message will appear on a blank page.

From here, we can navigate back to our localhost to see the updated webpage.



This is amazing! Everything we've been working on scraping has been collected into one location and presented in a clear and pleasing way. Bootstrap has made it easy to organize the data, too, without too much extra legwork on our part.

10.7.1: Create a Portfolio

The effort Robin has put into creating this web app really shows her motivation and enthusiasm for the subject. In fact, completing the web app has only driven Robin's NASA ambitions to new heights! This means she's ready to market her skills and get her name out there as a reputable data scientist.

To do so, she has planned to build a portfolio to showcase this web app, as well as other projects she's proud of. She wants her portfolio to serve as an introduction to her work, feature her favorite project, and include smaller images to capture other projects.

In addition to showcasing her work, Robin's portfolio should have a clean and professional presentation. If the portfolio is cluttered or otherwise unappealing, even her most stellar projects won't receive the recognition she'd like. Additionally, a portfolio will be responsive—it will adjust to different screen sizes (also known as viewports) as needed.

Create a Portfolio

There are two options for creating a portfolio:

1. Customize a template to show off your work. Include images, links to GitHub repos, and contact information (LinkedIn profile, email address, etc.).
2. Use a website service to create your portfolio.

Make sure the portfolio is easily viewed on phones, tablets, and computers.

A portfolio is a tool used to showcase your talents. When a clean, clear, and well-organized portfolio is included with a resume, it makes an impression. Robin is well aware that to become an employee of NASA, she needs to market herself to her best ability, and that includes creating a portfolio.

Creating one from a blank slate is a little daunting, so it's a good idea to use a template when available. Click the following link to download the portfolio templates. Note that you will need to unzip the file to access the templates.

templates.zip (https://courses.bootcampspot.com/courses/138/files/14562/download?wrap=1)

Open the HTML file in your web browser and explore it. Note it has all the hallmarks of a good portfolio:

- It's clean, with no distracting clutter.
- It captures your projects with eye-catching images and quick summaries.
- There are links to your presence everywhere—LinkedIn, GitHub, email, etc.
- It's personable, and after adding your personal touch, it will be more so.

Feel free to use this template to start building a portfolio. Follow the prompts within the code to customize it with your own images and links. The basic portfolio here is a great start, but feel free to make adjustments or add your own flair. Fonts, colors, image sizes—there are few limitations on what you can do.

Customize the Portfolio

This portfolio template already links to Bootstrap 3.3.7, but it also has a custom stylesheet included. To change the background of the body, navigate to that style

sheet and add this line:

```
body {
    margin: 5px;
    padding-bottom: 70px;
    background-color: darkgray;
}
```

Save the stylesheet, then refresh your browser. Notice that the background is dark gray instead of white? Each element in your HTML can be adjusted this way.

Here are some tips to help with customizing:

- Your featured images should be `.pngs` generated from the project you're featuring, such as a graph made in Matplotlib, a visual you've created that represents the project, or an interesting screenshot of your work.
- Sometimes the Bootstrap CSS customization needs to be overwritten. To ensure your stylesheet's code takes precedence, add `!important;` to the line of code you want changed, like in the code below.

  ```
  body {
  margin: 5px;
  padding-bottom: 70px;
  background-color: darkgray !important;
  }
  ```

- Images can be used as backgrounds. Just be careful that they don't distract from your featured projects.
- Background images and colors aren't the only way to add personal touch—free fonts (such as **Google Fonts (https://developers.google.com/fonts/docs/getting_started)**) are also great.

There are loads of great resources on CSS customization out on the web, too, for any other specific questions you may have.

Add Your Portfolio to GitHub

Your portfolio is something you'll want several other projects linked to. As such, it's best to keep it in its own repository.

From your GitHub homepage, create a new repository named "portfolio" and clone it into the master folder containing all of your class projects. Download one of the portfolio templates into this folder, then customize it to fit you—your personality, goals, and favorite projects.

Module 10 Challenge

Submit Assignment

Due Mar 22 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Robin's really happy with how her web app—and her portfolio—turned out. After some thought, she has decided her web app needs a bit more…oomph. She would like to add high resolution images for each of **Mars' hemispheres (https://astrogeology.usgs.gov/search/results?q=hemisphere+enhanced&k1=target&v1=Mars)** . The quality of the images will bring an extra level of aesthetic appeal to her app.

In this challenge, you will scrape additional content for the web app. This time, the focus is to add more images to the app.

Background

Robin's web app is looking good and functioning well, but she wanted to add more polish to it. She had been admiring images of Mars' hemispheres and realized that the site is scraping-friendly. She would like to adjust the current web app to include all four of the hemispheres images. This requires additional scraping code to pull the high-resolution images, updating Mongo to include the new data, and altering the design of her web app to accommodate these images.

Objectives

- Use BeautifulSoup and Splinter to automate a web browser and scrape high-resolution images.

- Use a MongoDB database to store data from the web scrape.
- Update the web application and Flask to display the data from the web scrape.
- Use Bootstrap to style the web app.

Instructions

Visit the **Mars Hemispheres (https://astrogeology.usgs.gov/search/results?q=hemisphere+enhanced&k1=target&v1=Mars)** web site to view the hemisphere images and use DevTools to find the proper elements to scrape.

To complete this task, follow these steps:

1. Obtain high-resolution images for each of Mars's hemispheres.
   Note: You must click each hemisphere's link to access the full-resolution image's URL.
2. Save both the image URL string (for the full-resolution image) and the hemisphere title (with the name).
3. Use a Python dictionary to store the data using the keys `` `img_url` `` and `` `title` ``.`` ` ``
4. Append the dictionary with the image URL string and the hemisphere title to a list. This list will contain one dictionary for each hemisphere.

Extension

Bootstrap components are designed to make polishing your webpage easier. Take advantage of the different options Bootstrap provides to create a tidy web app. For an extra challenge, make the following updates to your web app. The goal is to present the gathered data in a format that is easy to read but also pleasing to the eye. Feel free to experiment and familiarize yourself with the different available components.

- Update your web app to use more Bootstrap 3 components. The grid system needs to be used so your website will respond to different devices (such as a phone, tablet, or computer). Many people turn to their mobile devices to browse webpages, so the app needs to be responsive and look good on any device. **Hint:** Remember to test this using DevTools.
- Include at least three other Bootstrap 3 components from **this list (https://getbootstrap.com/docs/4.0/components/alerts/)** . Examples include:
  - Adding a button that redirects users to the homepage of the web app.
  - Adding the hemisphere images as thumbnails
  - Customizing the facts table using a Bootstrap table

As you update your app, keep the following questions in mind.

- Is this page clean?
- Does the page stand out from other pages?

Submission

Make sure your repo is up to date and contains the following:

- A `README.md` file containing a screenshot of your completed portfolio
- The coding files created during the module:
  - `Mission_to_Mars.ipynb`
  - `scraping.py`
  - `app.py`
- The `index.html` template and its resources (images, stylesheet, etc.)
- A link to the GitHub repo for your portfolio

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Criteria	Ratings					Pts
Select html tags	**25.0 pts** **Mastery** Select appropriate html tags for all four hemisphere images, with no errors.	**18.75 pts** **Approaching Mastery** Select appropriate html tags for at least three hemisphere images, with one or two minor errors.	**12.5 pts** **Progressing** Select appropriate html tags for at least two hemisphere images, with one or two minor errors.	**6.25 pts** **Emerging** Select appropriate html tags for at least one hemisphere image, with significant errors.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty	25.0 pts
Scrape images	**25.0 pts** **Mastery** Apply BeautifulSoup and Splinter to correctly automate a web browser and perform a web scrape.	**18.75 pts** **Approaching Mastery** Apply BeautifulSoup and Splinter to automate a web browser and perform a web scrape, with one or two minor errors.	**12.5 pts** **Progressing** Apply BeautifulSoup and Splinter to automate a web browser or perform a web scrape, with one or two minor errors.	**6.25 pts** **Emerging** Apply BeautifulSoup and Splinter to automate a web browser or perform a web scrape, with significant errors.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty	25.0 pts
Create MongoDB	**25.0 pts** **Mastery** Create MongoDB database to correctly store data from web scrape, with no errors.	**18.75 pts** **Approaching Mastery** Create MongoDB database to correctly store data from web scrape, with one or two minor errors	**12.5 pts** **Progressing** Create MongoDB database to correctly store data from web scrape, with more than two minor errors	**6.25 pts** **Emerging** Create MongoDB database to store data from web scrape, with significant errors	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty	25.0 pts
Amend Flask Routes	**25.0 pts** **Mastery** Amend Flask routes to display all four hemisphere images, with no errors.	**18.75 pts** **Approaching Mastery** Amend Flask routes to display at least three hemisphere images, with one or two minor errors	**12.5 pts** **Progressing** Amend Flask routes to display at least two hemisphere images, with one or two minor errors.	**6.25 pts** **Emerging** Amend Flask routes to display at least one hemisphere image, with significant errors.	**0.0 pts** **Incomplete** No submission was received, or submission was empty or blank, or submission contains evidence of academic dishonesty	25.0 pts
						Total Points: 100.0

Module 10 Career Connection

Web Scraping Skills Are in Demand

Nice job this week. You learned how to web scrape libraries, which is an important set of skills to showcase when talking with prospective employers. This week we really want to emphasize why web scraping is a valuable skill to add to your resume and the various ways in which web scraping may show up in the technical interview.

In this section, you will:

- Learn how web scraping is used on the job.
- Learn how to showcase this new skill to be a more competitive candidate in the job market.
- Answer common interview questions about web scraping.

On-the-job web scraping involves using specialized web crawling tools to extract the desired data from other websites—usually, as you've learned in this module, this data will then be stored in a database and used for analysis. Companies that might use this technique of data extraction may use the data for competitor analysis, tracking market trends, price research on competitors, or simply to be a data-driven company to improve.

As a data analyst and engineer, you can use your web scraping skills to collect information for future employers—so don't be shy about showcasing that you know how to do this.

Employer Competitive Advantage

Adding web scraping to your resume's skills list will help you pass through the applicant screening filters. It also tells a potential employer that you can capably harvest the data they need. For more resume support, see Resources at Milestone 4.

Link to Milestone 4 in Canvas

Consider the following possible scenarios, for which you might find yourself working with web scraping in the professional world.

After reading each case study, you will see some common technical interview questions around the scenario you were presented. During a technical interview, you may receive one of two types of questions—you might even get both. Typically, you can expect the technical interview to fall into two different categories:

- Technical questions with short answers
- Broader, less-technical questions that require reflective thinking

Let's get started.

Technical Interview Preparation

Case Study No. 1: Price Comparison
You've just been offered that job at NASA you've been dreaming about, and it's time for that final technical interview. As part of the interview, they give you the following prompt to read and consider.

You recently started working at a major online clothing retailer (Company A) whose focus is on selling high volume to increase its profit margins. Because this e-retailer is not a boutique marketplace selling to a niche market, it depends on its product, and its competitive pricing, to reach the largest audience possible.

However, a few months ago, a major competitor (Company B) entered the online market, and Company A wants to keep an eye on its competitor's pricing so that it can offer the best possible prices to its own customers.

You've been hired to do the data analysis, but because Company B isn't going to just go ahead and release all of its pricing in a well-documented API for you (you wish!), you'll have to regularly scrape the data off its pages and maintain a database of the products, prices, and

any price changes.

After reading this prompt, the NASA technical interviewers ask the following questions:

1. Would you consider it legal to scrape data from your competitors?
    - *This is a relatively gray area. The short answer is yes, unless there's some sort of privacy/legal agreement on its website that specifically prohibits web scraping. However, even if it's legal to scrape data from the page, do consider how you then use that data may have legal implications of its own. In other words, you couldn't scrape data and then republish and represent it as your own.*
2. Can you scrape data behind a login page?
    - *Yes, you could. But it is significantly more difficult. You would need to provide the web application with valid credentials, and then navigate to the authenticated portion of the site and scrape—to do this, you could use some sort of browser automation tool like Selenium Web Driver. This process, though, is not readily recommended.*

Case Study No. 2: Airline Tickets

FlyCheap is a locally owned tech company with a big idea. Using its browser extension, customers can book flights to travel all over the world on essentially

any airline. However, it's facing a very real problem—its major competitors, Kayak and Google, change their flight prices multiple times per day. So how do FlyCheap customers know when the best time is to book their tickets?

Well, that's where you come in. You've just been hired to improve the functionality of the browser extension by allowing it to pop up with an alert, for those who have it installed, when the price of a flight has dropped. But because the Google Flights API was recently deprecated, and you can no longer just make an API request for it—you're going to have to get the information yourself.

Your Task

Of course, you can't sit there all day monitoring the prices of flights manually—there are just too many. So your first task on the job with FlyCheap is to write an application that scrapes data from Kayak, Google Flights, and other companies and monitors price variations. When a flight drops or increases in price, that information will get fed to the browser extension and then on to the end-user.

1. Can you extract data from sites not written in English?
    - *Of course. You can extract data in any language, even if it's not in a Roman-style alphabet (i.e., Chinese, Japanese, Korean), but obviously the material you scrape remains in the language you scrape it in.*
2. Can you republish data and/or information that you scraped from the web?
    - *Another gray area—maybe! Watch out for policies that explicitly forbid redistribution of material and/or the citation guidelines. You might be able to freely republish, not republish at all, or republish with limitations and credits to the original authors. If you're unsure, get in touch with the owner of the site you're scraping from.*

Case Study No. 3: Customer Review Sentiment Analysis

Companies and their products live and die on customer reviews—all hail the mighty five gold stars. Your current company sells its products on Amazon, but it needs access to comprehensive data analytics on the customer reviews.

Your Task

There's going to be two main steps to this tasks. First, you'll use web scraping to crawl the Amazon product pages for your company's products and extract the review text and numerical value, then using a sentiment analysis library like **VADER Sentiment Analysis (https://github.com/cjhutto/vaderSentiment)** to analyze the text to provide useful and actionable feedback for your company

1. How would you handle scraping data from HTML that has been dynamically generated using a client-side JavaScript framework such as ReactJS or Angular.
    - *You can, but again it's a challenging task. Imagine the site has an empty* `<div></div>` *tag where content is dynamically generated—if you try the ordinary web scraping approach, your data will come back empty because there's nothing there. So you have a couple of options: (a) reverse engineering the JavaScript that is dynamically generating the content or (b) force the page to generate the content and then scrape it—to do this, you can use Selenium WebDriver to navigate to and load the page, then trigger your scrape.*
2. Would you say that web scraping is ethical?
    - *This is not something we can really give you an answer to. In some cases it will be, and in others it won't be. Before scraping data, consider the ethical implications of doing so, especially what you are going to do with that data.*

Employer Competitive Advantage

Continue practicing your technical interview skills in one of our prep sessions. Register for our live workshops **here (https://careerservicesonlineevents.splashthat.com/)** .

Continue to Hone Your Interview Skills

If you'd like to learn more about the technical interviewing process and to practice algorithms in a mock interview setting, check out our **upcoming workshops. (https://careerservicesonlineevents.splashthat.com/)**