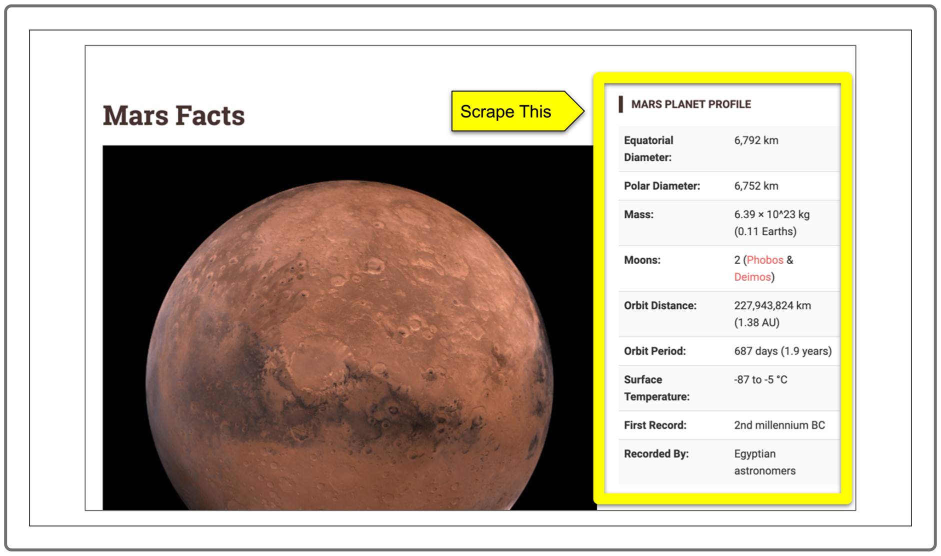
10.3.5 Scrape Mars Data: Mars Facts

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.

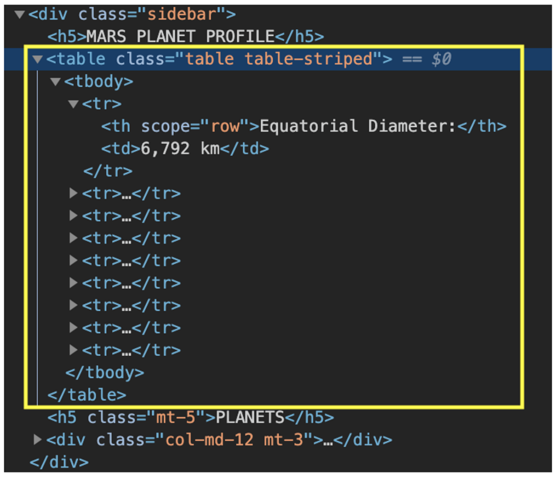
Robin has chosen to collect her data from **Mars Facts** **(https://galaxyfacts-mars.com)** , so let's visit the webpage to look at what we'll be working with. Robin already has a great photo and an article, so all she wants from this page is the table. Her plan is to display it as a table on her own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



MARS PLANET PROFILE

Equatorial Diameter:	6,792 km
Polar Diameter:	6,752 km
Mass:	6.39 × 10^23 kg (0.11 Earths)
Moons:	2 (Phobos & Deimos)
Orbit Distance:	227,943,824 km (1.38 AU)
Orbit Period:	687 days (1.9 years)
Surface Temperature:	-87 to -5 °C
First Record:	2nd millennium BC
Recorded By:	Egyptian astronomers

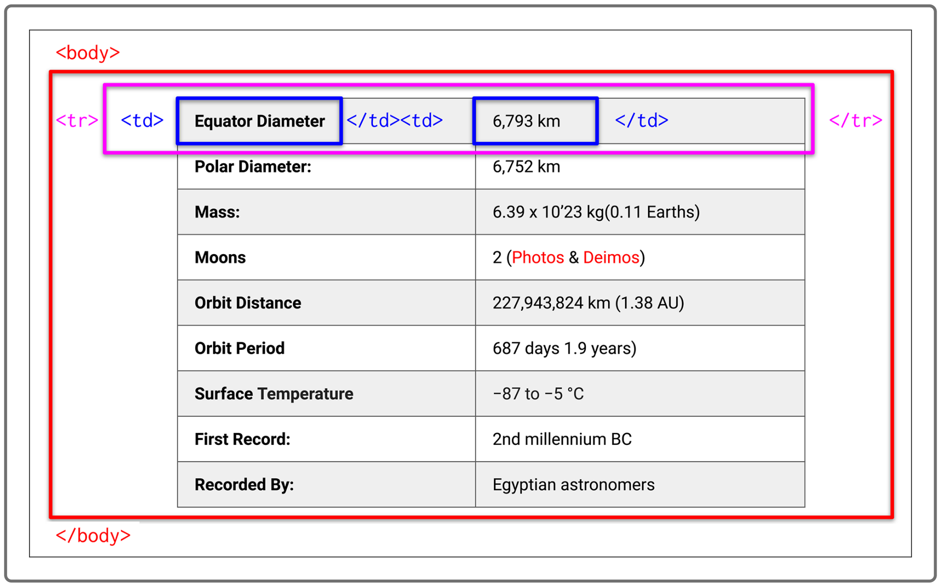
Let's look at the webpage again, this time using our DevTools. All of the data we want is in a `<table />` tag. HTML code used to create a table looks fairly complex, but it's really just breaking down and naming each component.



```
▼<div class="sidebar">
   <h5>MARS PLANET PROFILE</h5>
 ▼<table class="table table-striped"> == $0
   ▼<tbody>
     ▼<tr>
         <th scope="row">Equatorial Diameter:</th>
         <td>6,792 km</td>
       </tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     ▶<tr>…</tr>
     </tbody>
   </table>
   <h5 class="mt-5">PLANETS</h5>
 ▶<div class="col-md-12 mt-3">…</div>
 </div>
```

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.



Equator Diameter	6,793 km
Polar Diameter:	6,752 km
Mass:	6.39 x 10'23 kg(0.11 Earths)
Moons	2 (Photos & Deimos)
Orbit Distance	227,943,824 km (1.38 AU)
Orbit Period	687 days 1.9 years)
Surface Temperature	-87 to -5 °C
First Record:	2nd millennium BC
Recorded By:	Egyptian astronomers

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.

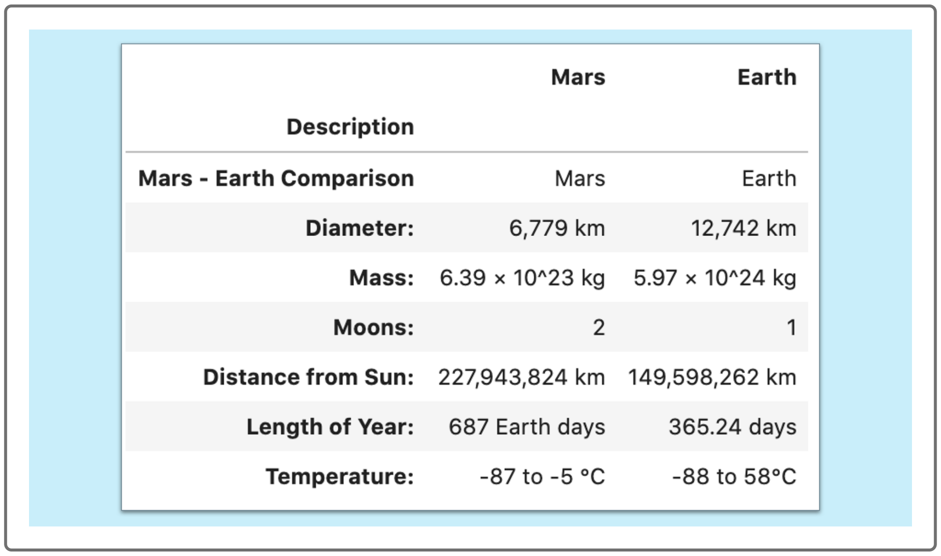
```
df = pd.read_html('https://galaxyfacts-mars.com')[0]
df.columns=['description', 'Mars', 'Earth']
df.set_index('description', inplace=True)
df
```

Now let's break it down:

- `df = pd.read_htmldf = pd.read_html('https://galaxyfacts-mars.com')[0]` With this line, we're creating a new DataFrame from the HTML table. The Pandas function `read_html()` specifically searches for and returns a list of tables found in the HTML. By specifying an index of 0, we're telling Pandas to pull only the first table it encounters, or the first item in the list. Then, it turns the table into a DataFrame.
- `df.columns=['description', 'Mars', 'Earth']` 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.

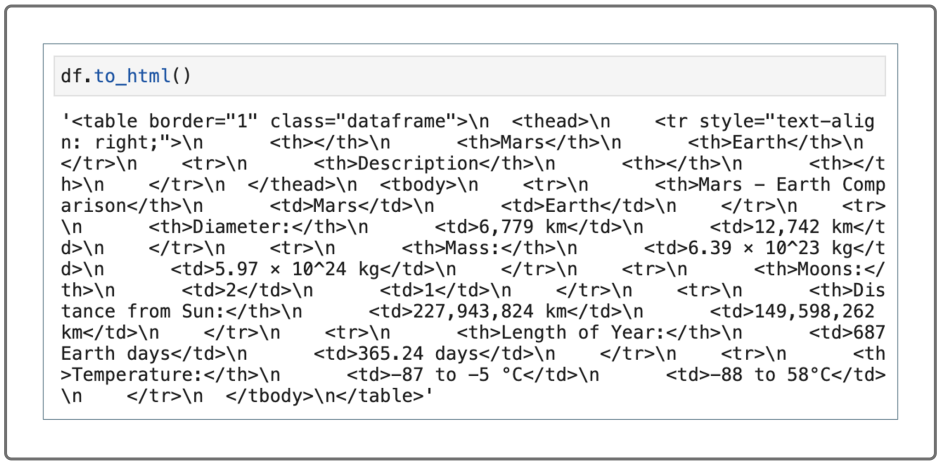
Now, when we call the DataFrame, we're presented with a tidy, Pandas-friendly representation of the HTML table we were just viewing on the website.



	Mars	Earth
Description		
Mars - Earth Comparison	Mars	Earth
Diameter:	6,779 km	12,742 km
Mass:	6.39 × 10^23 kg	5.97 × 10^24 kg
Moons:	2	1
Distance from Sun:	227,943,824 km	149,598,262 km
Length of Year:	687 Earth days	365.24 days
Temperature:	-87 to -5 °C	-88 to 58°C

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.

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.



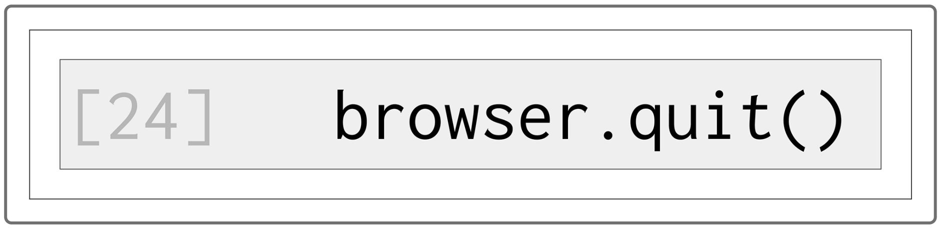
```
df.to_html()
```

```
'<table border="1" class="dataframe">\n  <thead>\n    <tr style="text-alig
n: right;">\n      <th></th>\n      <th>Mars</th>\n      <th>Earth</th>\n
</tr>\n    <tr>\n      <th>Description</th>\n      <th></th>\n      <th></t
h>\n    </tr>\n  </thead>\n  <tbody>\n    <tr>\n      <th>Mars - Earth Comp
arison</th>\n      <td>Mars</td>\n      <td>Earth</td>\n    </tr>\n    <tr>
\n      <th>Diameter:</th>\n      <td>6,779 km</td>\n      <td>12,742 km</t
d>\n    </tr>\n    <tr>\n      <th>Mass:</th>\n      <td>6.39 × 10^23 kg</t
d>\n      <td>5.97 × 10^24 kg</td>\n    </tr>\n    <tr>\n      <th>Moons:</
th>\n      <td>2</td>\n      <td>1</td>\n    </tr>\n    <tr>\n      <th>Dis
tance from Sun:</th>\n      <td>227,943,824 km</td>\n      <td>149,598,262
km</td>\n    </tr>\n    <tr>\n      <th>Length of Year:</th>\n      <td>687
Earth days</td>\n      <td>365.24 days</td>\n    </tr>\n    <tr>\n      <th
>Temperature:</th>\n      <td>-87 to -5 °C</td>\n      <td>-88 to 58°C</td>
\n    </tr>\n  </tbody>\n</table>'
```

The result is a slightly confusing-looking set of HTML code—it's a `<table />` element with a lot of nested elements. 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.