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://redplanetscience.com/'
browser.visit(url)

# Optional delay for loading the page
browser.is_element_present_by_css('div.list_text', 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('div.list_text')
slide_elem.find('div', class_='content_title')

# Use the parent element to find the first 'a' tag and save it as 'news_titl
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
```

Next, we will revisit that code and insert it into a function. Let's call it `mars_news`. Begin the function by defining it, then indent the code as needed to adhere to function syntax. It should look like the code below:

```
def mars_news():
    # Visit the mars nasa news site
    url = 'https://redplanetscience.com/'
    browser.visit(url)

    # Optional delay for loading the page
    browser.is_element_present_by_css('div.list_text', 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('div.list_text')
    slide_elem.find('div', class_='content_title')

    # Use the parent element to find the first <a> tag and save it as  `news_
    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.

Instead of having our title and paragraph printed within the function, we want to return them from the function so they can be used outside of it.

We'll adjust our code to do so by deleting `news_title` and `news_p` and include them in the return statement instead, as shown below.

```
def mars_news():

   # Visit the mars nasa news site
   url = 'https://redplanetscience.com/'
   browser.visit(url)

   # Optional delay for loading the page
   browser.is_element_present_by_css('div.list_text', 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('div.list_text')

   # 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()

   # Use the parent element to find the paragraph 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 add the word "`browser`" to our function, we're telling Python that we'll be using the **browser** variable we defined outside the function. 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('div.list_text')
        # 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
        # Use the parent element to find the paragraph text
        news_p = slide_elem.find('div', class_='article_teaser_body').get_te
```

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.

The complete function should look as follows:

```
def mars_news(browser):

    # Scrape Mars News
    # Visit the mars nasa news site
    url = 'https://redplanetscience.com/'
    browser.visit(url)

    # Optional delay for loading the page
    browser.is_element_present_by_css('div.list_text', 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')

    # Add try/except for error handling
    try:
        slide_elem = news_soup.select_one('div.list_text')
        # Use the parent element to find the first 'a' tag and save it as 'n
        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_te

    except AttributeError:
        return None, None

    return news_title, news_p
```

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.find('img', class_='fancybox-image').get('src'

   except AttributeError:
      return None
   ```

All together, this function should look as follows:

```
def featured_image(browser):
    # Visit URL
    url = 'https://spaceimages-mars.com'
    browser.visit(url)

    # Find and click the full image button
    full_image_elem = browser.find_by_tag('button')[1]
    full_image_elem.click()

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

    # Add try/except for error handling
    try:
        # Find the relative image url
        img_url_rel = img_soup.find('img', class_='fancybox-image').get('src

    except AttributeError:
        return None

    # Use the base url to create an absolute url
    img_url = f'https://spaceimages-mars.com/{img_url_rel}'

    return img_url
```

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. It is raised when any of the built-in exceptions are encountered and it won't handle any user-defined exceptions. 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('https://galaxyfacts-mars.com')[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('https://galaxyfacts-mars.com')[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.