How to Easily Read HTML Tables into Pandas DataFrames

Authored by

stats writer

December 1, 2025

Pandas is recognized globally as the preeminent library for data manipulation and analysis within the Python ecosystem. Its utility extends far beyond standard CSV or Excel files, possessing robust capabilities for integrating data directly from the web. Specifically, Pandas provides elegant solutions for Web scraping tasks, making the process of extracting structured data from websites remarkably straightforward. A core feature enabling this is the `read_html()` method, designed to parse and interpret HTML tables embedded within web pages.

When executed, the `read_html()` function automatically scans a specified URL or a string of raw HTML code, identifies all present table structures, and converts them into a list of DataFrame objects. This seamless transition from complex web structure to analytical format drastically reduces the effort required for initial data acquisition. This detailed guide will walk through the precise steps necessary to utilize this powerful method, demonstrating how to extract, filter, and prepare web-based tabular data for subsequent processing using Python.

The ability to quickly ingest data from publicly available web sources empowers data scientists and analysts. Whether retrieving economic statistics, sports scores, or complex configuration matrices, converting the raw output of HTML tables into manipulable DataFrame structures is the critical first step in many data processing pipelines. We will use a real-world example involving NBA team data hosted on Wikipedia to illustrate these techniques clearly.

You can use the read_html() function from pandas to read HTML tables into a list of pandas DataFrame objects.

This function utilizes the following basic syntax, pointing to the target URL:

df = pd.read_html('https://en.wikipedia.org/wiki/National_Basketball_Association')

The following sections provide a comprehensive example showing how to use this function to read in a table of NBA team names from the specified Wikipedia URL.

Prerequisites: Necessary Libraries and Installation

While Pandas handles the high-level data structure creation, the underlying process of parsing complex HTML often requires specialized engines. The `read_html()` function relies on external dependencies, primarily lxml, which is a high-performance library crucial for robust HTML document traversal and parsing.

Before attempting to execute any code utilizing the `read_html()` function, you must ensure that lxml is installed in your Python environment. Failure to install this dependency will result in a `ValueError` indicating that no suitable parser could be found. Installation is achieved using the standard Python package installer, `pip`.

Execute the following command in your terminal or command prompt environment to install the required dependency:

pip install lxml

Note: If you are working within an interactive environment such as a Jupyter notebook, it is highly recommended that you restart the kernel immediately after performing this installation. This ensures the newly installed library is correctly loaded into the session path, preventing potential import errors when calling the function.

Example 1: Extracting All Tables from a URL

To begin the process of data extraction, we first import the necessary libraries. For this example, we require `pandas`, but often `numpy` and data visualization libraries like `matplotlib` are imported alongside it for a typical data analysis workflow. Once imported, we call `pd.read_html()` using the target URL.

Executing the function on a complex page like the NBA Wikipedia entry will often yield a surprisingly high number of tables. This is because Pandas parses all structures marked with HTML `<table>` tags, which may include layout elements, sidebars, and navigation boxes in addition to core data tables.

The following code snippet performs the extraction and then checks the length of the resulting list to determine the total number of tables found:

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from unicodedata import normalize

# read all HTML tables from specific URL
tabs = pd.read_html('https://en.wikipedia.org/wiki/National_Basketball_Association')

# display total number of tables read
len(tabs)

44

We can see that a total of 44 HTML tables were found on this specific page. This demonstrates the necessity of filtering the results to isolate the specific data required, as manually inspecting 44 separate DataFrames is time-consuming.

Refining Extraction: Using the 'match' Argument

To efficiently handle a large volume of extracted tables, the `pd.read_html()` function provides a powerful argument called `match`. This argument accepts a string or a regular expression pattern and ensures that only tables whose content matches that pattern are returned in the final list. This is the most effective way to perform targeted Web scraping.

In our example, we are interested in the table listing the teams sorted by their organizational divisions. We can assume that this relevant table contains the keyword "Division." By utilizing the `match` argument, we drastically reduce the number of resulting DataFrames, isolating the essential data structure without requiring manual indexing or inspection.

The following code snippet repeats the extraction, but this time limits the results to only those tables that contain the text 'Division':

 # read HTML tables from specific URL with the word "Division" in them
tabs = pd.read_html('https://en.wikipedia.org/wiki/National_Basketball_Association', match='Division')

# display total number of tables read
len(tabs)

1

The output confirms that only 1 table was returned, validating the efficiency of the `match` argument. The relevant data is now stored in the single-element list `tabs`, ready for focused post-processing.

Post-Processing the Data: Inspection and Selection

After isolating the desired table, which is accessible via `tabs`, the next step is to inspect and clean the resulting DataFrame. Data scraped from the web, particularly from sources like Wikipedia, often results in multi-level column headers due to the inherent complexity of the original HTML table structure.

We first define our target DataFrame and then list its column names. This step is critical as it reveals the exact structure of the headers inherited during the parsing process, which may include hierarchical tuple names:

 # define table
df = tabs

list all column names of table

list(df)

For our purposes, we are only interested in the first two columns, 'Division' and 'Team'. To accurately select these columns, especially when dealing with complex multi-level headers, using the positional indexer `.iloc` is often the most reliable method in Pandas.

Refining the DataFrame Structure and Column Renaming

To create a clean, functional DataFrame, we utilize `.iloc` to slice the DataFrame, selecting all rows (`:`) and columns from index 0 up to (but not including) index 2 (`0:2`). This isolates the desired 'Division' and 'Team' data.

Following selection, it is essential to rename the complex column headers to simple, single-level strings for easier manipulation. This is done by assigning a list of new names directly to the `.columns` attribute of the final DataFrame.

 # filter DataFrame to only contain first two columns
df_final = df.iloc

rename columns
df_final.columns =

# view first few rows of final DataFrame
print(df_final.head())

Division Team
0 Atlantic Boston Celtics
1 Atlantic Brooklyn Nets
2 Atlantic New York Knicks
3 Atlantic Philadelphia 76ers
4 Atlantic Toronto Raptors

The final output confirms that the extraction and refinement process was successful. The resulting DataFrame is clean, properly structured, and ready for further analytical tasks such to merging with other datasets or aggregating data by division.

Summary of Workflow and Key Takeaways

The process of reading web data using `pd.read_html()` is highly effective, provided the necessary dependencies like lxml are installed. Key takeaways from this workflow include the

understanding that the function returns a list of DataFrame objects, not a single DataFrame.

For maximizing efficiency, utilizing the `match` argument is the most important technique when targeting specific tables on content-heavy pages. This prevents unnecessary processing of dozens of irrelevant HTML structures.

The following tutorials explain how to read other types of files in pandas: