Henan Cable' s Innovative Application—Automatic Scraping and Submission of Daily Report Data Postprint

Authors: Su Benguo, Zhang Weiyun

Date: 2023-10-08T00:00:00+00:00

Abstract

This paper describes the complete process of using Python to scrape report data from dynamically loaded web pages and update Excel template data, thereby implementing automated population of corporate periodic report data. It focuses on presenting solutions to two key problems: first, how to retrieve data from JavaScript dynamically loaded pages; second, how to partially update data in Excel templates.

Full Text

An Innovative Application at Henan Cable: Automated Scraping and Reporting of Daily Data

Journal: ChinaXiv Partner Journal
Abstract: This paper describes the complete process of using Python to scrape report data from dynamically loaded web pages and update Excel template data, thereby achieving automated periodic reporting for the company. The focus is on solutions to two key challenges: first, how to obtain data from JavaScript-driven pages, and second, how to partially update data in Excel templates without disrupting existing formulas.

Keywords: Python; selenium; Excel; automated reporting
Classification: TP391
Document Code: A
Article ID: 1671-0134(2019)12-060-03
DOI: 10.19483/j.cnki.11-4653/n.2019.12.016

Authors: Su Benguo, Zhang Weiyun

1. Background

The Information Support Department of Henan Cable frequently creates various Excel templates based on ad-hoc requests from company leadership, manually selecting and entering report data from multiple system platforms into these temporary templates. This work is low in complexity but highly repetitive. Particularly at the beginning of each month, approximately 30 templates require completion, with each requiring data from over 10 different report pages. This manual process consumes the entire day of six staff members.

As McKinsey notes, data has permeated every industry and business function, becoming a critical production factor. In this era, corporate decision-makers and operators must observe business operations and patterns through data—without it, progress becomes impossible. While system reports provide various foundational data, their dimensions and formats are fixed, unable to flexibly meet our temporary needs. Consequently, manual filling of customized and variable operational daily (weekly/monthly) reports is a routine task that all business units must continuously perform.

To address this, we sought an automated method for data acquisition and reporting. We ultimately selected Python, the top-ranked open-source development tool. After a period of study and research, we developed executable programs for web scraping and Excel data population using this tool, successfully achieving our goal of automated data reporting.

2. Overview of Approach and Process

The typical steps for writing a web scraper in Python can be divided into four phases: sending requests, retrieving data, parsing data, and storing data. However, in our actual implementation, because our scraping method directly obtained a list of data, we eliminated the data parsing step, retaining only three phases: sending requests, retrieving data, and storing data. The detailed processing steps for each phase are illustrated in Figure 1 [Figure 1: see original paper].

The following sections explain the problems encountered and corresponding solutions during each step of application development.

3. Implementation Details

3.1 Sending Requests

The first operation in data scraping is simulating a browser to send requests to the web server. The pages we needed to scrape were internal CRM system report pages that did not require authentication, but contained numerous request parameters and dynamically loaded JavaScript-generated data.

3.1.1 Three Approaches for Requesting Pages We evaluated three methods for sending requests:

3.1.1.1 Basic Approach: Using urllib' s request module
The `urlopen()` method in this module can perform simple request operations and obtain responses. However, constructing parameterized requests with this method is relatively complex.

3.1.1.2 Advanced Approach: Using the requests library
Methods in the requests library easily enable parameterized requests, cookie handling, login authentication, and proxy settings. However, the results differed from what was visible in the browser: display data visible in the browser was absent from the requests results. This occurs because requests only retrieves the raw HTML document, while browser-displayed data is generated after JavaScript processing.

3.1.1.3 Browser Simulation: Using the selenium library
To solve the problem of obtaining JavaScript-generated dynamic page data, we selected the browser automation library Selenium after reviewing relevant literature. Selenium is an automated testing tool that can drive a browser to perform specific actions such as clicking and scrolling, while simultaneously retrieving the page source code as currently rendered by the browser—enabling "what you see is what you get" scraping. Note: Before using this method, besides installing the Selenium library, you must properly install the target browser (e.g., Chrome) and configure ChromeDriver.

3.1.2 Page Analysis and Request Construction To analyze request parameters, we opened the report page and pressed F12 to launch Developer Tools. The Query String Parameters revealed numerous URL parameters, including Chinese characters (city information). For such complex parameter structures, we reconstructed the URL using a "base address + parameter information" approach before sending requests with Selenium. Key details and solutions encountered during this process are summarized in Table 1 .

When constructing URLs, we concatenated parameters with the base address to form a usable URL. Since parameters contained special characters like "/" , we used `quote()` and `unquote()` for encoding, and employed `urlencode()` with the `safe` parameter (e.g., `urlencode(params, 'utf-8', safe='/')`) to handle special characters properly.

3.1.3 Sending Requests Based on the parameter analysis, we used the Selenium library to simulate Google Chrome in sending requests to the server. The script for retrieving cash flow data from various cities is as follows:

```
def get_{cash}(p_r_{name}, p_r_{id}, std, edd, c_{id}, p_{type}, p_{id}, p_{name}, p_r_{type
    browser = webdriver.Chrome()  # Initialize browser
    base_{url} = 'http://.../bossreport25/frameset?'  # Base address
    params = {'参数 1': p_r_{name},  # Pass front-end input
```

```
            # ... other parameters ...
            '参数 n': p_r_{type}}
    url = base_{url} + urlencode(params, 'utf-8', safe='/')  # Reconstruct parameterized ad
    browser.get(url)  # Send request
```

3.2 Retrieving Data

3.2.1 Page Analysis and Data Extraction After opening the report page and pressing F12 to access Developer Tools, we selected the target element, right-clicked, and chose "Inspect Element" to locate its node position. Since the element lacked a clear node ID and had many similar nodes, we traversed upward to find the nearest node with a clear ID, "___{{{bookmark2}}_{ ",}} to enable CSS selector positioning of the target data. The data extraction script is as follows:

```
# Find specified element
bookmark_2 = browser.find_{{{element}}_{{by}}}_{{{css}}_{{selector}}}("#__{bookmark}2")
# Find data elements contained within this element
data_{elements} = bookmark_2.find_{{{elements}}_{{by}}}_{{{css}}_{{selector}}}("tr")
```

3.2.2 Handling Delayed Waiting During testing, we observed that report pages automatically opened but closed quickly without retrieving target data. Research indicated that the response from Selenium' s `get()` method to the browser might not contain the target data if the page hadn' t fully loaded. Therefore, we needed to implement delayed waiting. Waiting can be explicit or implicit; we adopted explicit waiting by adding the `WebDriverWait()` function to our control statements. This approach loads specified nodes within a defined time period—if loading completes, it returns the located node; otherwise, it throws a timeout exception. The control script is as follows:

```
wait = WebDriverWait(browser, 20)  # Wait for specified time
wait.until(EC.presence_{{{of}}_{{all}}}_{{{elements}}_{{located}}}((By.ID, '__{bookmark}2'))
```

3.3 Storing Data

3.3.1 Five Methods for Writing Data to Excel from Python Python boasts a powerful standard library and extensive third-party modules, offering multiple approaches to any task. Only by selecting the appropriate method can one achieve their goals. When writing retrieved data to Excel, we experimented with numerous methods but could not achieve our objective of "updating without damaging the template." We identified five methods for writing to Excel from Python, including four documented online plus our own solution:

Method	Library	Description	Platform	Limitations
to_{excel}	pandas	Exports DataFrame data to Excel tables	Cross-platform	Cannot modify existing files
XlsWriter	xlsxwriter	Creates Excel 2007+ XLSX files	Cross-platform	Cannot read or modify existing files
xlrd&xlwt	xlrd, xlwt	Provides separate read/write modules	Cross-platform	XLS format only; formula loss
OpenPyXL	openpyxl	Handles XLSX and XLS files	Cross-platform	Can only append sheets, not update cells with formulas
Microsoft Excel API	pywin32	Communicates directly with Excel process via COM components, enabling all Excel functionalities	Windows+Excel	Windows platform required

Among these, OpenPyXL could only append sheets but not update cells without affecting existing formulas. However, Microsoft Excel API could modify specific cell data without impacting original formulas. Since our application required both reading and updating partial data in Excel documents without modifying original formulas, we had to use Microsoft Excel API, specifically the win32com component.

3.3.2 Using win32com Components to Modify Partial Excel Data
The complete process for writing to Excel documents involves calling the win32com component to launch an independent Excel process, opening the Excel template file, and using `sheet.Cells(i, j).Value` to assign values to the cell at row i, column j. The relevant script is as follows:

```
excel = win32com.client.DispatchEx('Excel.Application')  # Launch independent Excel process
cash_{book} = excel.Workbooks.Open('F:/模板.xlsx', ReadOnly=False)  # Open template
sht1 = cash_{book}.Worksheets('sheet1')  # Open target sheet

# 18 branches require reading 18 rows of data
for i in range(0, 18):
    # Each branch requires 6 data items
    for j in range(0, 6):
```

```
# Update data starting from row 5, column 4; data source is the retrieved list
sht.Cells(i+5, j+4).Value = cash_{list}[8*i+j]
```

4. Results and Demonstration

Through the three steps described above, a complete data scraping and reporting program was developed. With the addition of user-friendly parameter input and progress notification interfaces, we used pyinstaller to compile the program into an executable file. By placing the executable and templates in the production environment and running the file periodically, users can visually observe the page opening process and Excel data refresh process. Figure 2 [Figure 2: see original paper] shows the sequential opening of target pages, while Figure 3 [Figure 3: see original paper] illustrates the automatic opening of Excel documents. As Excel data refreshes, existing calculation formulas automatically recalculate, and upon completion of data writing, the target data becomes ready for distribution.

Using Python' s extension libraries and modules—including selenium, urllib, win32com, and datetime—proved straightforward for achieving our objectives. This solution successfully addresses the automated reporting problem for routine reports, saving labor costs while improving work efficiency and data accuracy.

References

[1] Zhang Junhong. *Learning Python Data Analysis Through Comparison with Excel* [M]. Beijing: Publishing House of Electronics Industry, 2019.

[2] Cui Qingcai. *Python3 Web Crawler Development in Practice* [M]. Beijing: Posts & Telecom Press, 2018.

[3] CSDN Blog: https://blog.csdn.net/SvJr6gGCzUJ96OyUo/article/details/78967060.

(Author Affiliation: Information Support Department, Henan Cable Television Network Group Co., Ltd.)

Note: Figure translations are in progress. See original paper for figures.