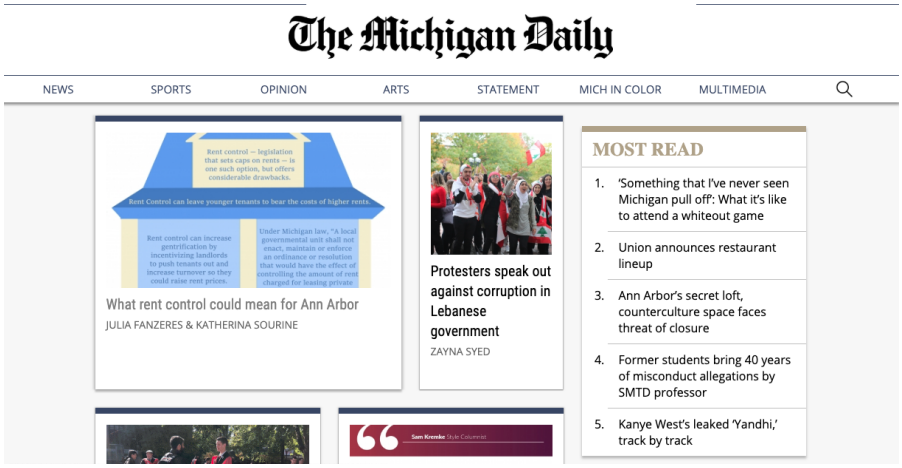
Project 2: Web Scraping: 200 Points

Due Nov 14

Introduction

Often in the world of data science, there isn't a neat prepackaged dataset for the problem that we're interested in solving.

When this occurs, we're often forced to compile and process a dataset from scratch so that we can do data analysis and answer the questions that interest us.



One powerful way to do that is through web scraping. Web scraping is the process of taking messy data from the web, processing it, cleaning it, and turning it into something useful for analysis.

The ability to do web scraping well is a powerful tool, since a large majority of data science work is often cleaning up messy data.

In this project...

You will be scraping data taken from The Michigan Daily, cleaning it, and extracting information from it. You will need to use the beautifulsoup library to parse through the HTML documents.

We have provided 2 static documents for you to use, but you will need to scrape some live content as well.

In order to expose you to using multiple data cleaning methods at once, you will need to combine beautifulsoup with regex and write output to a .csv file to complete this assignment.

After you've implemented all of the required functions, you will need to write test cases for each one except for main() and the get_soup…() functions. In order to write good test cases, you will need to open the websites, explore, and get a sense for what your data should actually look like.

The code

You will need to write several functions and test cases for them. Start from the stub code provided, which looks like the following:

```
def get_headlines_from_search_results(filename):
    """
    Write a function that calls get_soup_from_file() on "article1.html". Parse
    through the BeautifulSoup object and return a list of headlines for each
    search result in the following format:

    ['Headline 1', 'Headline 2'...]
    """

def get_most_read(url):
    """
    Write a function that calls get_soup_from_url() on
    "https://www.michigandaily.com". Parse through the BeautifulSoup object
    and return a list of URLs for each article in the "Most Read" section in
    the following format:

    ['https://somelink.com', 'https://anotherlink.com'...]
    """

def get_most_read_data(url_list):
    """
    Write a function that takes in a list of URLs (i.e. the one returned by
    get_most_read()). For each URL in the list, call get_soup_from_url(),
    parse through the BeautifulSoup object, and capture the article title,
    author, and date. Make sure to strip() any whitespace from the date.

    This function should return a list of tuples in the following format:
    (url, title, author, date)
```

```
    For example:
    [('https://one.com', 'A Great Title', 'Some Author', 'Thursday, October 17, 2019 -
    6:14pm')]

    HINT: Using BeautifulSoup's select() method may help you here.
    You can easily capture CSS selectors with your browser's inspector window.
    """

def get_names_and_years(filename):
    """
    Write a function to get the names and attending years of the alumni
    mentioned in "article2.html". You need to use regex to accomplish
    this. This function should call get_soup_from_file(). You will return a
    list of (Name, Year) tuples. If there are multiple years, add additional
    elements to the end of the tuple for each year.

    For example, if the article contains: "Matt Whitehead, attended in 2018
    and 2019", you should append ("Matt Whitehead", "2018", "2019") to your
    list of tuples.
    """

def write_csv(data, filename):
    """
    Write a function that takes in a list of tuples called data (i.e. the one that is
    returned by get_names_and_years()), writes the data to a csv file, and saves it to
    the passed filename.

    The first row of the csv should contain "Name" and
    "Year" respectively. For each tuple in data, write a new row to
    the csv, placing each element of the tuple in the correct column.
    If the tuple has multiple years, you should write them both into the year
    like so: "2018 & 2012".

    When you are done your CSV file should look like this:

    Name,Year
    Some Student,2019 & 2018
    Another Student,2012
    Yetanother Student,2010
```

```
    This function should not return anything.
    """

class TestCases(unittest.TestCase):
    """
    For each function you wrote above you should write a non-trivial test case
    with at least one additional assert statement to make sure that your
    function works properly.

    Remember that websites are dynamic and their content can change at any
    time, any function that calls get_soup_from_url() should not use
    assertEqual statements with hard coded values. You don't want to write a
    test case that passes today and fails tomorrow.
    """
```

Grading

Function	***points***
`def get_headlines_from_search_results(filename):`	30
`def get_most_read(url):`	30
`def get_most_read_data(url_list):`	30
`def get_names_and_years(filename):`	30
`def write_csv(data, filename):`	30
`TestCases`	50
Total	**200**
`def get_recent_si_articles():`	15 pts extra credit

We will be checking to make sure that you've implemented each function correctly. You will need to make sure that you are returning data in the format specified in the docstrings to get full credit. You will also need to make sure that you are calling your other functions when directed to do so.

We have provided some sample test cases to help you get started and make sure that you are on track. You will need to expand these. For each test case you write, you will receive 5 points for writing a valid test case with at least 1 assert statement, and another 5 points for writing a test case that is not trivial.

A non-trivial test case should do more than check that a constant is equal to a constant (`self.assertEqual(1, 1)`) or check that something obvious is true (`self.assertTrue(type([1, 2, 3]), list)`). Instead, you should be checking that your functions return data that falls within a set of expected constraints. Good examples of test cases might include: checking that a year is made up of exactly 4 digits, checking that a URL starts with “http” or “https”, or checking that a list of articles you retrieved matches the number of articles you see when you open the website in your browser. It is up to you to decide exactly what to test for, but your test cases should be of similar merit.

Tips

Work on one function at a time. Choose the one that you think is the easiest, and work on it until you can get all the tests related to that function to pass. This is a great strategy, since ***the solution to some functions can be used to quickly complete other functions.***

Extra Credit: Parsing Search Results - 15 points

Write a function called *get_recent_si_articles()* it should call *get_soup_from_url()* on the search results page for “School of Information”. Build a list of tuples in the following format for each article on the **first** page: (title, url, year, month, day). Sort the list so that the **most recent** articles appear first and then return it. Finally, write a non-trivial test case for this function and add it to your unit tests. You will not be given any starter code for this function. It is up to you to decide how to define and call it.

Your titles should not have any newline characters (\n) in them. The URLs must be valid and the year, month, and day columns should be integers.

HINT: Looking up “Python secondary sort” may help you if you get stuck.