Automatic Treatment of Catalog Information

Thesis Prospectus

Augusta University

Caleb Acree

**Introduction**

The Augusta University course catalog presents students at Augusta University with many issues when using the web portal. Upon investigating the web page each course seems to be a sort of embed into the actual page rather than a link to the course details webpage. This can cause issues when trying to work with and load the different classes and their details. To combat this, this thesis is to design and create a program that automatically grabs the information from the website and exports it to a file which should be easier for the end user to use and read.

Firstly, the improved accessibility for academic planning will be a significant advantage. With a more intuitive and user-friendly interface, users will be able to navigate the system, locating the information they need with ease.

The program’s scalable architecture is designed this way so if it needs modification to work with the future catalogs, it can be quickly edited. This flexibility means that as the university's needs evolve, the system can be easily adapted and expanded to accommodate new requirements. This feature ensures that the solution remains relevant and effective over time.

The development of this innovative tool aims to transform the course catalog experience for the Augusta University community. By providing a reliable, and easily usable program, it supports the academic community’s information needs in a meaningful way.

**Methodology**

**Overall Description**

This prospectus will see the creation of a program using the scripting language Python. Python is a simple but robust and scalable language that will work well for this thesis prospectus. However, natively python cannot look at webpages or parse through data on websites. Modules or more commonly known as libraries can be imported to python to increase the functionality of a couple lines of code

**Program Development**

At the time of writing this prospectus the program currently uses two libraries. They are Playwright and BeautifulSoup (BS). These are necessary for the functionality of the program. Playwright allows for the program to open a headless browser so that it can navigate to the course catalog and access the different classes. Once it grabs the classes it then uses BS to parse through the webpage and the webpage source. It is an HTML parser.

As of writing this, the program currently utilizes three functions, they are as follows: “run”, “extract\_course\_info”, and “extract\_course\_details”. The “run” function sets up and opens the headless browser, navigates to the web page, and handles some errors for the process of gathering the information off the website. After it navigates to the specified URL it waits for the courses to load on the web page. After this completes it calls the function that handles the actual gathering of the information from the website. The function that handles the actual gathering is “extract\_course\_info”.

The purpose of the “extract\_course\_info” function is to gather the detailed information for each course listed in the course catalog, process it, and save the data into a text file. The first thing that this function does is open a new tab in the headless browser and it uses it to navigate to the course catalog page. Next, it identifies all the course links on the page by using the CSS selector 'table.table\_default > tbody > tr > td.width > a[href^="preview\_course\_nopop.php"]'. This means that the links are filtered to only include those that match the specified pattern for individual course pages. This is so when trying to aggregate the course information it does not try to navigate to any other links that may be on the page. It then creates a folder in the same directory, if it does not exist already, and then generates the output file that has a timestamped name to store the data. The next section of code iterates through the individual course links listed on the page and extracts the course name and URL for the specified course. Once it has the link and the course name it opens a new browser tab and navigates to the course entry. Then it calls the “extract\_course\_details()” function to actually extract the data using BS. Once the data is gathered it saves it to the output file. Every time it tries to gather the information on a certain class the statements are wrapped in try/except statements which work as error handling, so that the program does not crash if there is an error. It also makes sure that after each class’s data is gathered the browser tab is closed. While it is iterating through the courses it prints progress reports to the console every time it processes ten courses. Also, once all the data has been gathered it prints a completion message to the console to let the end user know.

The next and last function currently in the program is the “extract\_course\_details()” function. The purpose of this function is to parse through individual course web pages to extract specific academic details using HTML analysis provided by the BS library. This function is the data extraction workhorse that pulls all the course-specific information off the web pages. It does this by capturing the full page’s HTML code and uses BS to find the information that is needed. BS targets the “#course\_preview\_title” element and extracts the first “<p>” tag and text that follows. If there is nothing there or it cannot find the correct tags it outputs that the specific item cannot be found. It then uses regex patterns to find the following labels in “<strong>” tags: Prerequisite(s)?, Lecture Hours, Repeat Status, Grade Mode, and Schedule Type. Once it finds those strings, it extracts the strings or data immediately following them. Each field is isolated by Try/Except blocks to handle one field errors to not crash the whole process. If the program cannot find any data in the following areas it defaults to “Not Found” instead of sending “Null” messages. The regex patterns and filtering is a flexible way of looking for this data because it allows the program to ignore the case of the characters in the gathered data.

Some benefits of the way this program is currently written and designed is that it is modular in design, performance balancing, and many failure scenarios are handled with the Try/Except blocks. The modular design is beneficial because it allows for independent improvements to parsing, easily to add or remove fields, and it is simpler to test. The localized HTML parsing avoids multiple playwright queries and unnecessary network requests. The Failure scenarios that are handled in the current implementation are missing parent elements, unexpected tag structures, network delays and text encoding issues.

However, the main downside to the current implementation is that of position dependency. The program relies on consistent HTML structure because field labels must immediately precede their values and the description must follow the tile element. This is somewhat of a larger issue that will need to be worked out because the course catalog is notoriously not consistent throughout the entire catalog.

**Future Plans**

The current program is only a part of what this thesis aspires to achieve. In the future it is planned to do the following:

* PDF course catalog parsing
* PDF and online cross referencing and checking
* Separating the course title from the course number and identifier
* Navigating through all of the course catalog web pages
* Sortable output file
* Degree pathways mapped in output file
* Easily useable and accessible for end user
* Split the three main functions into smaller functions for readability and scalability
* Inform the registrar if any inconsistencies are found

**Timeline**

* April 2025: Complete Prospectus
* May 2025: Spring Semester Ends
* June 2025: Complete Cadet Summer Training
* July 2025: Complete Viceroy Maven Internship
* August 2025: Fall Semester Begins and start Thesis semester

**Glossary:**

* API (Application Programming Interface): a set of protocols, routines, and tools that enable communication between different software applications. It serves as an intermediary layer, allowing programs to exchange data and functionality without exposing their internal workings.
* Console: A text-based interface for interacting with software, used for inputting commands or displaying system output
* Crash: An abrupt termination of a program due to unforeseen errors or system failures, often resulting in data loss.
* CSS (Cascading Style Sheets): Style sheet language controlling visual presentation of HTML documents through layout, colors, and fonts.
* CSS Selector: Pattern syntax for selecting HTML elements to style, using identifiers like classes, IDs, and element hierarchies.
* Data: Structured or unstructured information processed by computer systems, often stored in files or databases.
* Directory: File system structure organizing digital assets hierarchically, analogous to physical folders.
* Encoding: System for representing characters/values through standardized schemes like UTF-8.
* End User: Final consumer of software applications or digital services.
* Error: Unexpected condition preventing normal program execution, handled through mechanisms like try/catch.
* Error Handling: Programming techniques (e.g., try/except blocks) for gracefully managing runtime exceptions.
* File: Discrete digital resource storing data persistently on storage media.
* Function: Reusable code block performing specific tasks, accepting inputs and returning outputs.
* Function Calling: Programming practice of invoking predefined operations with specific parameters.
* Headless Browser: Graphical User Interface (GUI)-less web browser controlled programmatically for automated testing/scraping.
* HTML (HyperText Markup Language): Standard markup language structuring web content through tags like <p> and <strong>.
* HTML Tags: Syntax elements (<element>) defining document structure and semantics.
* Library: Precompiled code collection providing reusable functionality through APIs.
* Localized: Software adapted for specific linguistic/cultural contexts through translated content.
* Module: Self-contained code unit implementing specific functionality within larger systems.
* Network: Interconnected system infrastructure enabling data communication between devices.
* Network Request: Data packet transmission between clients and servers via protocols like HTTP.
* Null: Special value representing intentional absence of meaningful data.
* Null Messages: Empty data packets or communication signals with no payload.
* Output: Information produced by programs through computation or processing.
* Parsing: Analysis of structured data (e.g., HTML/JSON) to extract meaningful information.
* PDF (Portable Document Format): File format preserving document layout across platforms.
* Python: High-level scripting language emphasizing code readability through whitespace.
* Regex (Regular Expressions): Pattern-matching syntax for text search/manipulation operations.
* Regex Patterns: Specific character sequences defining search criteria in text processing.
* Scripting Language: Interpreted programming language automating software environment tasks (e.g., Python).
* Statements: Basic executable units in programming languages performing actions.
* String: Data type representing sequences of Unicode characters.
* <strong> Tag: HTML element indicating text importance, typically rendered in bold.
* <p> Tag: HTML element defining paragraph blocks.
* Text Encoding: Character encoding schemes (e.g., ASCII, UTF-8) mapping bytes to text symbols.
* Try/Except Statements: Error handling constructs catching exceptions during code execution.
* URL (Uniform Resource Locator): Web address specifying protocol, domain, and resource path.