David Cavaceci

Meteo 496

September 11, 2009

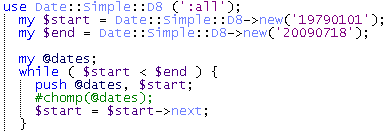
**Report: First Three Milestones**

|  |  |  |
| --- | --- | --- |
| **Milestone** | **Task** | **Code** |
| *I* | *Download the data from the weather station web page, and save it to file.* | *WWW::Mechanize module*  *File creation/writing*  *While flow control*  *Regular Expressions* |
| *II* | *Format the data to be Fortran-friendly, and parse it to an array* | *Fortran::Format module*  *If/then construct*  *Storage to array* |
| *III* | *Extend code to cover entire POR* | *Date::Simple module*  *List control* |

The first three objectives of this project involve the automated collection, storage, and formatting of State College temperature data. Such information is available on the Penn State Weather Station web site back to 1896. The Perl language is very useful in completing this task with the use of regular expressions, and the CPAN community modules WWW::Mechanize and DateTime::Simple. These modules made the tasks of creating a date array for the period of record and using an automated agent to browse the web very easy.

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

The line above uses a *pragma*, or special instruction to the Perl interpreter to use when running the program. *Strict* is used to ensure that all of my variables are declared throughout the code. The *warnings* pragma tells the interpreter to warn me about various syntax and variable type errors found throughout.



*Date::Simple* is a module available in the Comprehensive Perl Archive Network (CPAN) for download and use for creating simple date objects. In the case of this project, the module creates each date from January 1, 1979 to July 18, 2009 and loads them into the array *@dates*. The *:D8* argument at the end of each function call tells the module to create the dates in the *YYYYMMDD* format, which is how the weather station form accepts input. The basic flow of the block above is setting a start date, and creating and loading each proceeding date to the list until the end date is reached. Originally I achieved this list with a nested if loop and special exceptions for leap years. This module accounts for such things, and thus it is advantageous to use it.



This line opens the file *hlai.txt* to which the collected data will be written. This will allow for loading in Fortran after proper formatting of the data. If the file fails to be created, and error message will be written to the screen. Note that I define a specific directory to write the file to, though the final product will simply write the file to the working directory. This will allow my code to be run on machines other than my own.



Here the *WWW::Mechanize* module is introduced. Mechanize, or *Mech*, makes automated interaction with a website extremely easy. Visiting a page, submitting a query, filling out a form, using the back and forward commands, downloading page content, and many other tasks are compacted into usable functions by this object. This community-developed module undoubtedly saved me hours of teaching and coding. This specific block tells the interpreter to use *Mech*, defines our URL (the weather station page), uses the function *new* to initiate a new *Mech* object (which is defined as *$mech*), and the *get* function to fetch the page. All lines utilizing *$mech->* hereafter are telling the interpreter to use our defined object to complete a task.



Following the *Mech* initialization, we enter a while loop to submit each of our dates to the online form and begin scraping for data. This code block says, “For each of the dates in my list, fill in the *dtg* field on the first form of the page.”

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

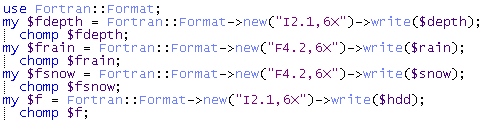
Next, the scalar variable *$page* stores the page content returned by the form submission above. The *Mech* function *content* downloads the content and stores it in the variable, and the parameter *format->’text’* makes sure the result does not contain html markup or other properties that will make scraping more difficult. The next line utilizes one of the most valuable traits of the Perl language and the key reason why it was used in this project, regular expressions. The *=~* operator tells Perl to look for the exact pattern to the right within the variable *$page*. Remember that *$page* was previous defined to hold the content of the page returned by the form submission. The parenthesis within the pattern tells Perl that the enclosed pattern is what I am interested in extracting and saving. So in the example above, anytime the pattern

***“Temperature (one or more spaces): (one or more spaces) (dd)”***

is found within downloaded page content, the *dd* (two digits) are stored in the variable implicit to the pattern matching operator. This is how the high, low, and average temperatures were collected for each date in the list. In the same way, the thirty-year high and low temperature normals were scraped from the website. The high example is shown below.



Included with this block is an *if* block ensuring that any *Not Applicable* values were changed to *99.99*. This allows for proper formatting and manipulation in Fortran. The same procedure was done for the thirty-year low normals, as well as the high and low normals for 10 years and maximum years. (Note: Other data such as rainfall amounts, hdd’s, snow depth, etc. were scraped from the site as well. This was for practice, and the code was left in the script for future reference. However, these values have yet to be used for analysis in Fortran or other).



Still inside the dates *while* loop (thus, each line pertains to each individual date submission), now the collected data are formatted for correct input to Fortran using the CPAN module *Fortran::Format*. Above, snow depth, rain and snow amount, and hdd’s are formatted. C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

After formatting, the collected data are stored to an array. This is done to make printing easier. As seen above, the date, high, low, and average temperature (data[0-2]), max, thirty, and ten year normals, snow depth, rain and snow amounts, and hdd’s are stored.

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

Now the data array is printed to the screen, and to the file created for writing. The handle *OUT* was declared when the file was opened, prior to entering the while loop. This method allows for the monitoring of collection progress and determining if/when an error occurs during scraping.

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

After writing the stored data, the *sleep* command (also a *Mech* function) is used to make sure the agent doesn’t overload the server and get timed-out. Although, I did experience connectivity issues at various times throughout development, pausing the program for .1 seconds is thought to help minimize this problem. Bandwidth is an important concern when accessing different web addresses either in terms of respecting the host, or appearing as a human user. The *back* function acts just as the “Back” button in a browser. Thus, at this point, the agent returns to the date query page. *Shift* removes the previously used date from the list, leaving the next in line. The ending curly brace closes the while loop initiated back just before the form submission.

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

The final step is closing the written file after all the dates have been submitted, and their corresponding temperature and precipitation data collected. Thus, after closing the file, data from the period of record defined in the *Date::Simple* module has been written line-by-line to the screen, and to the data file.

The full program (*ms\_i\_ii\_ii.pl*) is available at *www.personal.psu.edu/dac5039*