WEB SCRAPING BASE CODE DOCUMENTATION

Code File: Base Code w_o Tool v3.py

Updated on: 11/09/2019

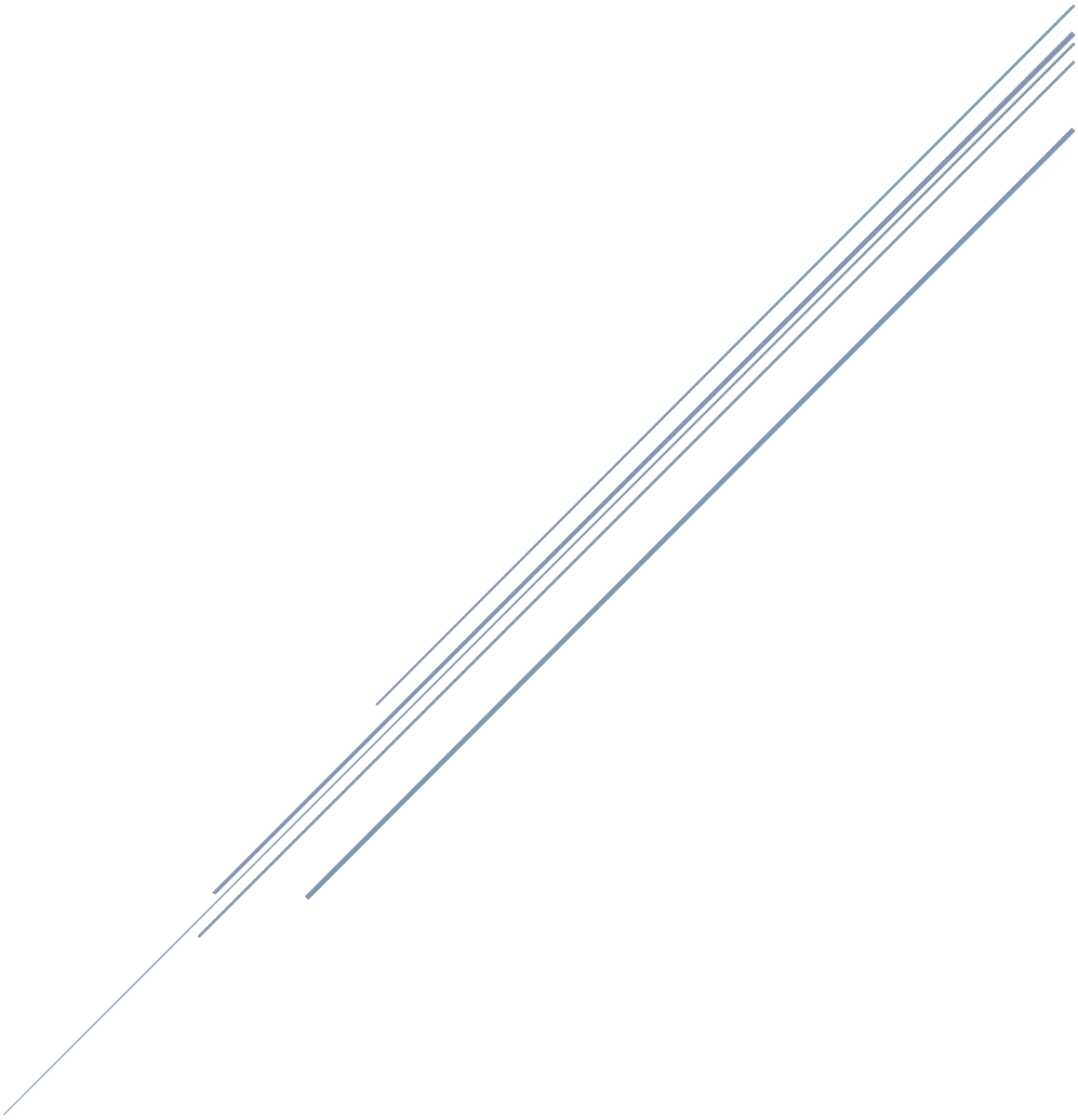

Table of Contents

1. Overview: 2
2. Package Modules 3
3. Major Methods 4
   - 3.1 main() 4
   - 3.2 multi_pool() 5
   - 3.3 collect_links() 5
   - 3.4 collect_data() 7
4. Other Methods 13
   - 4.1 checkNone(), HttpCheck(), Unique() 13
   - 4.2 convertLeast(), cleanWord(), convertNum() 13
   - 4.3 convertDuration() 14
5. Regular Expression 16
6. Webdriver / Selenium Package module 18

1. Overview:

This documentation is written according to the Base Code w_o Tool v3.py file at 11/09/2019. As the code may be improved over time, it is expected that this documentation to be updated accordingly to reflect the updated changes for reference purposes. The main purpose for this documentation is to provide a section by section understanding of the code either for newcomers or for future improvements.

This code contains two main methods to scrape data from the university websites: collect_data() and collect_links(). Both these methods are ran within a main method: main(). This document will cover the major methods required for the code to run. There may be other methods that are present in the code and are not covered in detailed as they are mainly methods for formatting or calculation purposes and these methods often change depending on website layout. However, you can refer to Section 4 for a brief overview of the purposes of these methods.

2. Package Modules

```
import multiprocessing
import time
import timeit
import csv
import re
import requests
from collections import deque
from selenium import webdriver
from bs4 import BeautifulSoup, Tag
from fake_useragent import UserAgent
```

Package modules	Explanation
Multiprocessing	Used for multiprocessing.
Time, timeit	User for timer purposes.
csv	For usage with .csv files.
re	Necessary for Regular Expression / Regex functions.
requests	Used to load web pages. Does not need to open any web browsers.
Collections, deque	For iterating the array list containing URLs.
Selenium, webdriver	An alternative method to load web pages. Each process will open a web browser window to load the page. Much slower compared to requests method.
Bs4	For usage of soup functions in order to extract elements from a page.
Fake_useragent	Used to mask the browser that opens links. Can be used to bypass being blocked on some university pages when web scraping.

3. Major Methods

3.1 main()

This section will explain each section of the main() method. This method forms as the basis for the entire program and will execute the necessary methods in order to perform web scraping of the selected university website.

```
start = timeit.default_timer()
```

This shows the time for how long the code was executed for.

```
# Create the UniqueLinkList file.
with open('E:/Scrape/Canada/Concordia_ED/UniqueLinkList.csv', 'wt') as Linklist:
    Linklist.close()
```

Creates a file called UniqueLinkList_UG.csv at the directory that was specified. 'wt' signifies writing.

```
while(len(Url)) != 0:
    collect_links(Url.pop())  # Call this to collect the links first.
```

Used to ensure the collect_links() method will be ran for as long as there are URLs inside the 'Url' array. If there is only one Url for the collect_links() to be executed, the while loop may be removed. Refer to Section 3.2 for further information regarding collect_links() method.

```
# Create an output file
with open('E:/Scrape/Canada/Concordia_ED/ExtractedData.csv', 'wt', encoding='utf-8-sig', newline='') as website:
    writer = csv.writer(website)
    writer.writerow(['Course Name', 'Level', 'Faculty', 'Duration', 'Duration Type', 'URL', 'Description', 'ALLTXT'])
```

Creates a file '_ExtractedData_all.csv' at the directory that was specified. This file shall be the final output of the code containing all university scraped details in table format.

```
# Get links from UniqueLinkList.csv
with open('E:/Scrape/Canada/Concordia_ED/UniqueLinkList.csv', 'rt', encoding="utf-8") as course_link:
    reader = csv.reader(course_link)
    course_url_list = []
    for row in reader:
        course_url_list.append(row[0])
print('Total number of Course Links: ' + str(len(course_url_list)))
```

Reads the created 'UniqueLinkList.csv' file that have collected all URL links that contains the university information pages to be harvested. The links read from the file are stored in 'course_url_list'.

```
# Multiprocessing Collect_Data()
# all_data should contain all 'details' that was returned from collect_data().
all_data = multi_pool(collect_data, course_url_list, 10)            # Multiprocessing collect_data(), paramenter
with open('E:/Scrape/Canada/Concordia_ED/ExtractedData.csv', 'at', encoding="utf-8", newline='') as website:
    writer = csv.writer(website)
    print("Writing details to CSV File now....")
    for a in all_data:
        writer.writerow(a)
print("Total number of rows written to ExtractedData.csv file: " + str(len(all_data)))
```

multi_pool() is used to manage the multi-processing of collect_data() in order to optimize the information scraping process. The end result of multi_pool() returns a list containing all scraped data based on collect_data() iterated over a list of iterables 'course_url_list'. The list is assigned to a variable 'all_data' which is then written into the ExtractedData.csv file.

```
stop = timeit.default_timer()
time_sec = stop - start
time_min = int(time_sec / 60)
time_hour = int(time_min / 60)

time_run = str(format(time_hour, "02.0f")) + ':' + str(
    format((time_min - time_hour * 60), "02.0f") + ':' + str(format(time_sec - (time_min * 60), "^-05.1f")))
print("This code has completed running in: " + time_run)
```

Code for showing the time that was taken to complete all processes. Typically no change required to this section.

3.2 multi_pool()

This method is used to handle multi-processing of collect_data() method. Each time collect_data() method is executed, a list containing the information of a university is returned. Essentially, multi_pool will be obtaining multiple lists that is returned from parallel processing of collect_data() and these lists will be compiled into one. The end result of multi_pool() is returning a single compiled list that consists of all lists gathered from collect_data().

```
def multi_pool(func, url, procs):               # Defines method to handle multiprocessing of collect_data()
    templist = []                               # Stores the data to be returned from this method.
    counter = len(url)                          # Number counter for total links left.
    pool = multiprocessing.Pool(processes=procs)
    print('Total number of processes: ' + str(procs))
    for a in pool.imap(func, url):              # Loop each collect_data() execution.
        if a is not None:
            templist.append(a)                      # Puts the details row from collect_data() inside templist
            print('Number of links left: ' + str(counter - len(templist)))
    pool.terminate()
    pool.join()
    return templist
```

The method requires three parameters to be specified upon used. ‘func’ refers to the method that will be multi-processed. In this case, collect_data(). ‘url’ refers to the parameter which the multiple processing is iterated upon. ‘procs’ is the number of parallel processes allowed to be executed at a time.

Multiprocessing.Pool is the main method for handling multi-processing in this method. ‘pool’ variable is used to handle the call for multiprocessing.pool methods.

As previously described, since there are multiple lists returned as a result of multi-processing collect_data(), a ‘for’ loop is used to collect each list and compiled it into ‘templist’. This list is then returned at the end of the method.

3.3 collect_links()

This method is used to collect the university course link pages. These pages should contain individual course information which should ideally include the following: Course name, course level, course duration, course description, page URL. Refer to the Documentations for Scraping for information. Note: the example code shown here may differ to the one in Base Code w_o Tool file but the logic and reasoning for the code is the same. Since each university has different html tags and locations for links, it is only natural that each code for each university will look different.

```
# This function is to collect the links from the website.
def collect_links(link):
    # Only change the executable_path to your path. Leave the chrome_options.
    user = ua.random
    print(user)
    headers = {'User-Agent': user}

    start_req = requests.get(str(link), headers=headers)
    start_soup = BeautifulSoup(start_req.content, 'lxml')
```

This section forms the basic parameters that will be required to start collecting url links from a website. This method takes a single argument variable when called. In this case, the variable is labelled as 'link'. From the line 'user = ua.random' to 'headers = …', this section involves defining the parameters for using the Fake_useragent package module. Refer to Section 2 for further information regarding Fake_useragent does.

Then, requests and beautiful soup package module is utilized to obtain information from the url and as such variables for utilizing the modules were created.

This section of the code typically does not require any changes to be made. However, if one intends to use Selenium/Webdriver to obtain the information from a website instead of using requests, then one must make some changes to the code here.

```
navigation_container = start_soup.find('article', {'class': 'o-degree-finder__content'})
course_url_list = []

num = 0
for reslist in navigation_container.find_all('tr'):
    if 'http' not in reslist.find('a').get('href'):
        course_link = 'https://www.adelaide.edu.au' + reslist.find('a').get('href')
    else:
        course_link = reslist.find('a').get('href')
    if Methods.HttpCheck(course_link) & Methods.Unique(course_link):  # Check the link to make sur
        with open('E:/Scrape/Australia/AdelaideUni/UniqueLinkList_UG.csv', 'at', encoding='utf-8',
                  newline='') as Linklist:
            writer = csv.writer(Linklist)
            u = (str(course_link).split('\n'))
            writer.writerow(u)
        Linklist.close()
        print(course_link)
        course_url_list.append(course_link)
        num += 1
print("================================\n" + str(num) + " the courses have been collected")
```

This section of the code will vary depending on the university website. The basic function of this section is to locate the tags containing the 'href' attribute as this attribute contains the 'URL' links that is the goal of utilizing this method. By utilizing soup commands the tags can be located accordingly.

In the picture example, the navigation container variable was used to store the primary tag 'article' that contains the tag we want nested inside. From there, find_all is used to find a list of all 'tr' tags within 'article'. 'a' tag contains the 'href' attribute which we need and is a child of each 'tr' tag. Once the location of the 'href' attribute is determined, the information is then stored in course_link. The 'course_link' variable is then checked using methods that will determine if the URL found is unique to eliminate duplicated links. The links that passes the check will then be finally placed in the UniqueLinkList.csv file that was created in Section 3.1.

3.4 collect_data()

This method contains the code for the scraping of the required university information from the website. Similarly to collect_links() the code involving locating the html tags may differ for each university however, the basic structure should be the same. Basically, the code will create an array called 'details' and will store all relevant information within this array according to the column names that was written inside the created file 'ExtractedData.csv'. (Refer to Section 3.1 to see what column names were made when ExtractedData.csv was created) Once the 'details' array is populated with the required data, it will be written into the ExtractedData.csv file.

```
# This will get the keywords from faculty file and put it into a dictionary
with open('C:/Users/Anon/Dropbox/Scrapping/Others/faculty.csv', 'rt', encoding='utf-8') as List:
    reader = csv.reader(List)
    mydict = {rows[0]: rows[1] for rows in reader}
```

This reads the faculty file in the shared Dropbox folder. This file contains a list of keywords that will be used to determine the Faculty of a course. mydict is a variable that is used to hold the information from the faculty file.

```
while True:
    num_loop = 1
    finished = 0
    retry_count = 1
    while finished != 1:
        try:
            if retry_count == 5:
                print('This link cannot be opened ' + course_url_list)
                break
            else:
                time.sleep(1)  # buffer period
                # print('OPENING ' + course_url_list)
                # print('Retry Counter ' + str(retry_count))
                req = requests.get(course_url_list, timeout=10, headers=headers)
                if req.status_code == 404:
                    print(req.status_code)
                    req.raise_for_status()
                    break
                if req:
                    finished = 1
        except requests.exceptions.HTTPError as e:
            print(e)
            print(course_url_list)
            retry_count += 1
            continue
        except requests.exceptions.ReadTimeout as e:
            print(e)
            print(course_url_list)
            retry_count += 1
        except requests.exceptions.ConnectionError as e:
            print(e)
            print(course_url_list)
            retry_count += 1
    soup = BeautifulSoup(req.content, "lxml")
```

This part of the code involves a loop check for attempting to open a website link as well as handling errors.

If a website link from 'course_url_link' is unable to be opened, the error will be handled by the exceptions and the 'retry_count' will be increased by one. The loop will repeat for each link until the retry_count hits the value of 5 before the link will finally be skipped as the link is considered broken.

If HTTPError occurs, this usually means there is a problem with opening the page or that the page does not exist. In this case, the 'continue' statement will be ran and the loop will repeat to try to open the page again.

If ReadTimeOut or ConnectionError occurs, this usually means that the webpage could not be loaded due to poor or unstable Internet connection. Try switching to a different wifi network that is more stable before running the code again to avoid this error.

```
# details['Course Name', 'Level', 'Faculty', 'Duration', 'Duration Type', 'URL', 'Description', 'ScrapeAll']
details = ['', '', '', '', '', '', '', '']
```

This creates a variable ‘details’ with the array type to store the respective data. Each course information will occupy one column in the array and may be referred to with indexes. The first column as details[0] where Course Name should be entered, the second column as details[1] where Course Level should be entered and so on.

```
# Course Name
name = ''
if soup.find('h1', {'class': 'banner__heading'}) is not None:
    name = soup.find('h1', {'class': 'banner__heading'}).get_text()
    name = name.replace('\n', '')
    name = re.sub(r'[^\x00-\x7f]', r'', name)
    details[0] = " ".join(name.split())
```

This section involves collecting the data for Course name and assigning it to the first row of ‘details’. The first row of details can be indicated using the index number 0, hence details[0]. In this example, the tag containing the course name is first located and checked if present on the website using the ‘if’ statement. If found, the ‘get_text()’ soup method is then called to obtain the course name. Regular expression is then used to format the name into a legible form before assigning it to details[0].

```
# Levels
word = details[0]
lock = 0
for level, key in level_key.items():
    for each in key:
        for wd in word.split(" "):
            if each.lower() == wd.lower():
                details[1] = level
                lock = 1
                break
        if lock == 1:
            break
    if lock == 1:
        break
```

This section is for collecting the Course Level information. The example shows that the course level information can be obtained from the Course Name which was assigned to details[0] previously. For example: if the course name is Bachelors of Computer Science, the course level Bachelor can be found. Hence, details[0] is assigned to word. After this, a loop is created in order to match the appropriate level based on the keywords matched with the ‘word’ variable. Level_key is a pre-defined array that should be near the top of the Base Code with a list of keywords corresponding to the Course Level.

Each.lower() refers to the keywords from the level_key array while wd.lower() refers to the course name assigned to ‘word’. When the keywords are found inside the ‘word’ variable, the appropriate level is assigned to details[1] based on the keyword that was matched. Note that the comparison character ‘ == ‘ is used meaning the keyword has to be an exact match in order for a level to be assigned. NOTE: If the word has ‘Bachelors’ but the keyword only has ‘Bachelor’, it is considered a mismatch and no assignment will be made.

There is typically no change in the code for this section except for cases where course names have no direct reference to the level. In cases like these, either use soup to find the Course Level from the website page and assign to details[1] or manually hardcode the level based on the website information. Otherwise, just add the respective keywords to the level_key array in order for the keywords to be matched to the Course level. Ensure that the keywords added are specific enough for that case and cannot be applied falsely in any other cases.

```
# Faculty
# Nothing to be changed here.
loop_must_break = False
for a in details[0].split():
    for fac, key in mydict.items():
        for each in key.split(','):
            if each.replace("'", '').title() in a:
                print("\t\t\t" + each + '  in  ' + details[0] + ' from ' + course_url_list)
                details[2] = fac
                loop_must_break = True
                break
            elif each.replace("'", '') in a.lower():
                print("\t\t\t" + each + '  in  ' + details[0] + ' from ' + course_url_list)
                details[2] = fac
                loop_must_break = True
                break
        if loop_must_break:
            break
    if loop_must_break:
        break
```

This section is for collecting and assigning the Faculty for the course. Similarly to the Course Level code, keywords will be used to match against the Course name where applicable with the final output being the faculty name. In this example, details[0] is the course name and 'a' is a variable representing the list of course names which will be compared with the keyword 'each' from 'key'. 'fac' is the faculty name while 'key' is the keywords from the corresponding faculty name. The faculty name and keywords are both located inside a faculty file in the shared Scrapping folder in Dropbox. The faculty file serves as a universal library that is updated in real-time to ensure that the library is expansive. If the faculty is not assigned due to lacking keywords, please ensure the following steps are checked before adding keywords to the library:

1) Is the course name in English words?
2) Is the directory for the faculty file pointing to the file in the shared Dropbox folder?
3) Is the faculty name listed elsewhere in the website page outside of the course name? If so, point towards the faculty name instead of details[0]. If it's a single exception, then just add an exception code via 'if' 'else' to handle this exception website page while the rest can continue using details[0]

If (3) could not be done and the answer to (1) and (2) is yes, then proceed to add keywords to the faculty file. Take note not to add keywords that are too general and crosscheck with the IT Scraping leader / team to ensure that the keywords used are always relevant. Remember that this is a live document and it needs to maintain its integrity for future university websites so we have to be quite specific with the keywords.

```
# Duration
if soup.find('div', {'class': 'acc__content'}) is not None:
    dur_found = 0
    a = soup.find('div', {'class': 'acc__content'})
    dur_text = str(a.get_text())
    dur_text = " ".join(dur_text.split())
    for pattern in dur_regex:  # Locate Duration terms from Duration Regex Pattern Library, dur
        if re.search(pattern, dur_text) is not None:
            dur_text = str(re.findall(pattern, dur_text))
            for c in dur_text.split(','):
                if 'year' or 'month' or 'week' or 'day' or 'hour' in c.lower():
                    if convertDuration(c) != 'WRONG DATA' and convertDuration(c) is not None:
                        details[3] = convertDuration(c)[0]
                        details[4] = convertDuration(c)[1]
                        dur_found = 1
                        break
            break
    if dur_found == 0:
        details[3] = 'null'
        details[4] = 'null'
```

This section is for collecting and assigning the duration for the course. The main logic of the duration code utilizes a combination of locating the duration tags on the website along with convertDuration() method. Refer to Section 4.3 for the convertDuration() for further information regarding the method. In this example, the location tags were first found using soup, then the text is filtered using Regular Expression in order to obtain the exact duration information which is finally passed through convertDuration() before assigning to the respective 'details' arrays. dur_text is the variable assigned to obtain the duration text information from the tag. It is then passed through a 'for' loop for Regular expression matching to extract the specific duration information. For instance, if the duration is hidden inside a paragraph of text, a regular expression that matches the term 'x years full time' could be written inside dur_regex which is then matched with dur_text such that the output is 'x years full time'. Refer to Section 5 for further information regarding Regular Expression used. Finally, the regexed dur_text is then passed through convertDuration() method to ensure the right duration type and number format before assigning to 'details' arrays. dur_found is a variable used as a condition for ensuring no blank cells in final output. Details[3] and details[4] should always be populated with either the duration data or 'null'. As dur_found is only 1 whenever a duration is successfully written in the 'details' array, dur_found == 0 can be used for all other cases where the duration could not be found / written.

```
# URL
details[5] = course_url_list
```

This collects the current page URL and assigns it to details[5]. Course_url_list holds the URLs collected from collect_links() method.

```
# Description
desc = ''
desc_found = 0
if soup.find('div', {'class': 'intro-df'}) is not None:
    desc = soup.find('div', {'class': 'intro-df'}).get_text()
    desc = re.sub(r'[^\x00-\x7f]', r'', desc).replace('\n', ' ')
    desc = " ".join(desc.split())
    if desc != '':
        desc_found = 1
if desc_found == 1:
    details[6] = desc
else:
    details[6] = 'null'
```

This section is for collecting and assigning the description for the course. Soup is used to locate the tags of the description and assigning the text found to a variable ‘desc’. Regular expression is then used to format ‘desc’ into a legible format before writing it into details[6].

```
    # Scrape All
    [s.extract() for s in soup(['style', 'script', '[document]', 'head', 'title'])]
    visible_text = repr(soup.get_text().replace(r'\\n', ' ').replace('\n', '').replace('\\', '').replace(', ', ''))
    visible_text = re.sub(r'[^\x00-\x7f]', r' ', visible_text)
    visible_text = ' '.join(visible_text.split())

    details[7] = str(repr(visible_text))
    print(details)
    num_loop += 1
    time.sleep(2)
    break
if req.status_code != 404:
    return details
```

This section just scrapes all text that is found on the website. Then print() statements to show the scraping progress when the script is running. At the end of the method, the details array will be returned as a result of this There is typically no changes required to this section of the code.

4. Other Methods

This section will cover the remaining methods used in the code in general. They are mainly used for data validation and formatting data to support the main methods listed in Section 3.

4.1 checkNone(), HttpCheck(), Unique()

```
class Methods:
    # Check if the variable datatype is None
    def CheckNone(link):
        if link != None:
            return True
        else:
            return False

    # Check if the passed string contains http or https
    def HttpCheck(link):
        if Methods.CheckNone(link):
            if ('https://' in link) or ('http://' in link):
                return True
            else:
                return False
        else:
            return False

    # Check if link is Unique
    def Unique(link):
        with open('E:/Scrape/Australia/AdelaideUni/UniqueLinkList_UG.csv', 'rt', encoding='utf-8') as Linklist:
            reader = csv.reader(Linklist)
            for url in reader:
                if Methods.CheckNone(link) & Methods.CheckNone(url):
                    if link == str(url).replace("['", '').replace("']", ''):
                        Linklist.close()
                        return False
                else:
                    Linklist.close()
                    return False
            return True
```

These 3 methods are used mainly for collect_links() method. CheckNone() is to check for empty links. HttpCheck() will determine if https or http is in the url link. These two methods will then be used with Unique() method to write the links inside the Uniquelinklist.csv file. Not all methods are used sometimes and it varies depending on university websites.

4.2 convertLeast(), cleanWord(), convertNum()

```
def convertLeast(x, y):
    return round(x * CLA[y - 1])

def cleanWord(word):
    return word.replace('(', ' ').replace

def convertNum(Text):
    return Text.replace('one', '1').repla
```

These 3 methods are for formatting data for further use. convertLeast() is to ensure the number is rounded up according to data format while convertNum() just turns numbers written in words into digits. These 2 methods are normally used for convertDuration() method.

cleanWord() is used for removing symbols and replacing them with either blanks or whitespaces and is typically used within collect_data() method.

4.3 convertDuration()

```
def convertDuration(duration):
    duration = duration.lower()
    duration = convertNum(duration)
    numbers = re.findall(r'\d+(?:\.\d+)?', duration)

    dur_type_list = []
    for word in duration.split():
        if 'credit' in word.lower() or 'term' in word.lower() or 'hour' in word.lower() or
            dur_type_list.append(word)
    # print(dur_type_list)

    nums = []
    for number in numbers:
        if number != '':
            nums.append(number)
    # print(nums)

    for number in nums:
        for dur in dur_type_list:
            if 'year' in dur:
                if '.' in number:
                    if re.findall(r'\d+', duration)[1] != 0:
                        return convertDuration(str(round(float(number) * 12)) + ' month')
                    return int(re.findall(r'\d+', duration)[0]), 'Years'
                else:
                    return int(number), 'Years'
            elif 'month' in dur:
                if '.' in number:
                    if re.findall(r'\d+', duration)[0] < 7:
                        return convertDuration(str(int(float(number) * 4)) + ' week')
                elif int(number) % 12 == 0:
                    return int(int(number) / 12), 'Years'
                else:
                    return int(round(float(number))), 'Months'
            elif 'week' in dur:
                return round(int(number)), ' Weeks'
            elif 'hour' in dur:
```

There are two versions of convertDuration() method. The first one is as the picture above shows. The method will essentially find the keywords for duration type such as ‘year’ and ‘month’ before returning the appropriate result. The numbers are found using Regular Expression from the ‘numbers’ variable. This version of convertDuration() relies on the source data ‘duration’ to have the correct data and only one type of duration type. If ‘duration’ contains more than one duration data, it will always take the first duration which may or may not be accurate.

```
def convertDuration(duration):
    duration = duration.lower()
    duration = convertNum(duration)
    duration = cleanWord(duration)
    dur_type_key = ['hour', 'day', 'week', 'month', 'mester', 'term', 'academic', 'year']
    tmp_duration_list = []

    # In this loop, it will split the text by space, find number and when it finds a number
    # If the next word is a duration type, it will append both the number and duration type
    word_index = 0
    for word in duration.split():
        if is_number(word):
            for d_type in dur_type_key:
                if d_type in duration.split()[word_index + 1]:
                    tmp_duration_list.append(word + ' ' + duration.split()[word_index + 1])
        word_index += 1

    duration_list = []
    highest_index = 0
    value = ''
    if len(tmp_duration_list) > 1:
        for each in tmp_duration_list:
            for d_type in dur_type_key:
                if d_type in each:
                    if highest_index < dur_type_key.index(d_type):
                        highest_index = dur_type_key.index(d_type)
                        print('Largest duration type is ' + each)
                        value = each
                    else:
                        highest_index = dur_type_key.index(d_type)
                        value = each
        duration_list.append(value)
        print('The highest value is ' + value)
    else:
        duration_list = list(tmp_duration_list)
```

This is the 2nd version of convertDuration() method. This method will take the 'highest' duration type if 'duration' contains more than one duration type. For example: if 'duration' has '1 month part-time, 1 year full-time, this method will automatically select '1 year full-time' as 'year' is bigger than 'month'. Any of the two methods do work but it depends on university websites. Some are easier to use with the first but you must be sure that the data is specific and has one type of duration. You can try to Regex to ensure this. The 2nd convertDuration() on the other hand lets you deal with multiple duration types more accurately.

But typically, if you use Regex to obtain 'duration' before putting it through convertDuration(), then the first method is usually more than sufficient.

5. Regular Expression

This section addresses some of the regular expressions that are used within the code. This section is aimed to explain what the regular expressions used in the code are for and not to teach what regular expression is. If you wish to learn more it, please refer to the standard documentation which can be found online.

Regular expressions are used by importing the package module re and using the commands to filter information based on the pattern that was assigned as the filter. The most prevalent regular expression method used is to obtain duration information. The Regex (regular expression in short) patterns required may differ for every university website so one must understand why and how it is used to ensure information is obtained properly.

Firstly we defined an array called dur_regex.

```
# Duration Pattern Regex Library
dur_regex = [r"([0-9]+.years.full.time)",
             r"([0-9]+.year.full.time)",
             r"([0-9]+.months.full.time)",
             r"([0-9]+.month.full.time)",
             r"([a-zA-Z0-9]+.years.of.full.time)",
             r"([a-zA-Z0-9]+.months.of.full.time)",
             r"([a-zA-Z0-9]+.months)",
             r"([a-zA-Z0-9]+.years)"]
```

This array will hold the Regex patterns we wish to use to match with the information on the website. [0-9] indicates any digits from 0-9 while the '+' next to it indicates possibility of more than one occurrence. For example: a number '12' would require [0-9][0-9] to capture both digits but if we do [0-9]+ we can capture as many digits in the same phrase. [a-zA-Z0-9]+ refers to any alphabetical character, both upper and lower case and digits. This is used when the information may contain duration types in words or digits. Basically, the patterns in this array should be changed to match the duration format on the website.

[0-9]+.years.full.time will only match with information like '2 years full time', '5 years full time', '10 years full time'. But if '2 years Full Time' or '2 Years full time' is written on the website than the regular expression will fail to match as there are capital letters in the website info that is not present in the regex pattern. In this case you must add the patterns to match EXACTLY as the information on the websites such as the capital letters.

Once the array is made and contains a list of patterns, this array is usually called upon in the finding duration section of the code, usually nested within collect_data().

```
# Duration
if soup.find('div', {'class': 'c-degree-finder__detail-grid'}) is not None:
    dur_found = 0
    a = soup.find('div', {'class': 'c-degree-finder__detail-grid'})
    dur_text = str(a.get_text())
    dur_text = " ".join(dur_text.split())
    for pattern in dur_regex:  # Locate Duration terms from Duration Regex P
        if re.search(pattern, dur_text) is not None:
            dur_text = str(re.findall(pattern, dur_text))
```

As dur_regex contains a list of patterns, we iterate through them using a 'for' loop. First, for each pattern, we check for pattern matches using re.search(). If a match is found, re.search() will return a matched object. If no match is found, it will return a None object. This is why a 'if' statement is used to ensure the code does not crash when a None type object is obtained. When a match is found, we use re.findall() to obtain the actual matched information and assign it to a variable called dur_text.

If you only want to use ONE regular expression pattern and do not require an array, then just replace the variable pattern with the single regular expression pattern you want and remove the 'for' loop.

6. Webdriver / Selenium Package module

Both Base Codes currently collect information from websites by utilizing requests Package module. Requests allows collection of information from websites without requiring a browser to be opened. In comparison, Web Driver / Selenium also can collect information from website but will require a browser to be automatically opened. As such, requests is usually preferred over Web driver due to requiring less computational resources due to not needing a web browser and is much faster as a result as it does not have to wait for a browser page to load.

However, this does not mean that there isn't a use case for the Web Driver / Selenium package module. Elements on the page that require interaction such as button clicks, scrolling down the page, selecting dropdown lists can be interacted with via Selenium while requests can't. For university web pages where the information is hidden behind elements that can be interacted with, Selenium should be used. The following code will show an example on how to use Selenium to collect website links using collect_links(). It could be applied to collect_data() as well if necessary.

```
# Setup Chrome display
options = webdriver.ChromeOptions()
options.add_argument('--ignore-certificate-errors')
options.add_argument("--test-type")
ua = UserAgent()
```

First ensure that the browser settings required is present. This set of code should be placed below the code where package modules are imported. UserAgent() is actually not part of the Selenium package but is for Fake_usragent package module but we normally use it together.

```
def collect_links(link):
    # Only change the executable_path to your path. Leave the chrome_options.
    user = ua.random
    print(user)
    options.add_argument(f'user-agent={user}')
    driver = webdriver.Chrome(chrome_options=options, executable_path=r'E:\Scrape\chromedriver.exe')
    driver.get(str(link))

    navigation_container = driver.find_element_by_class_name('o-degree-finder__content')
    course_url_list = []
    num = 0
    result_container = navigation_container.find_elements_by_class_name('c-table')
    for reslist in result_container:
        for b in reslist.find_elements_by_css_selector('a'):
            course_link = b.get_attribute('href')
            if Methods.HttpCheck(course_link) & Methods.Unique(course_link):  # Check the link to make sure
                with open('E:/Scrape/Australia/AdelaideUni/UniqueLinkList_UG.csv', 'at', encoding='utf-8',
                          newline='') as Linklist:
                    writer = csv.writer(Linklist)
                    u = (str(course_link).split('\n'))
                    writer.writerow(u)
                Linklist.close()
                print(course_link)
                course_url_list.append(course_link)
                num += 1
    print("================================\n" + str(num) + " the courses have been collected")
    driver.quit()
```

After the Chrome Settings code, a variable ‘driver’ is defined to utilize the chrome browser settings. An executable file for the chromedriver.exe will be required for the browser to open, so make sure the directory is pointing towards the executable file. After defining ‘driver = …..’, the links can be accessed by utilizing driver.get() which will prompt a browser to open up the URL link. Basically that’s all that is required to setup Selenium to open links.

The rest of the code are just codes for locating the tags containing ‘href’ elements similar to the collect_links() using requests as we are scraping for URL links. But the methods used are different since we are using Selenium. To understand the selenium commands used here, please refer to the online documentation for Selenium.

Overall, how to decide which package module to use? It’s simple. Does the website have information that you need hidden behind buttons or need scrolling? If yes, use Selenium. If no, use requests.

When using Selenium, you will need the Chrome settings as well as ‘driver’ definition as shown in the picture. Then you will use Selenium commands to find the elements you need.

When using requests, you just need to define a ‘req’ definition shown in collect_links() in Section 3.2. Then you will use Soup commands to find the elements you need.