# Insurance Premium Web Scraping Readme

## Overview

This Python script is designed for web scraping insurance premium information from the major NZ insurance companies’ websites. It utilizes the Selenium library to automate interactions with the website and extract insurance premium data based on provided input data.

## Instructions

### Copy this folder onto your local device

* Copy the folder onto a location in your local device (your PC or any other PC you plan to run this scraper on)

Install Python

* Download Python: If you don't have Python installed, download it from the official website: [Python Downloads](https://www.python.org/downloads/).
  + Install Python: Follow the installation instructions for your operating system.
  + NOTE: **Make sure you add python to your file path** (this is an option in the python installer, if you don’t do this, then uninstall and reinstall with this enabled)

### Dependencies

Before running the script, ensure you have the necessary packages installed. You can install them using the following commands in the command line (cmd):

* pip install *selenium*
* pip install *webdriver\_manager*
* pip install *pandas*
* pip install *openpyxl*
* pip install fuzzywuzzy
* pip install *python-Levenshtein*

**Note: If you have not downloaded python on your device and added, pip will not work. Also if you did not add python to you file path, the pip command will not work**

### WebDriver

The script uses the Chrome WebDriver. *Make sure you have Google Chrome installed on your machine*. (Google chrome (the standard web browser) needs to be downloaded for ChromeDriver to function)

### ChromeDriver

ChromeDriver is required for Selenium to automate Chrome. The script uses ChromeDriverManager to automatically download the appropriate version of ChromeDriver.

* Chromedriver should be automatically downloaded with *webdriver\_manager*

### Input Data

The test data input files (for both home and auto insurance) need to have their file paths defined. This is done at the top of the python file under the header “File path definitions”

* Notes:
  + Two backslashes (\\) are required for to define a backslash character (\) in a python string. This is because \ is an escape character in python strings (meaning it allows us to note special character, such as \n newline character). Therefore the 1st \ is just defining that the 2nd \ is actually just a character.
  + Templates for the test data input files can be found here: [S:\Library\IQS\Test data](file:///S:\Library\IQS\Test%20data).

### Running the Script

1. Turn on VPN (IPVanish) to prevent IP from being blocked by the websites
2. Either…
   1. run the script from the command prompt (cmd) - <https://www.wikihow.com/Use-Windows-Command-Prompt-to-Run-a-Python-File>
   2. Or download a python coding environment such as VScode (download VScode: <https://code.visualstudio.com/download>) and then open the file with VScode (or any other environment) and press the play button in the top right.

A screenshot of a computer screen

Description automatically generated

1. The script will open a Chrome browser, navigate to each of the insurance websites, input the data for each potential individual from the spreadsheet, and scrape the insurance premiums.

### Output

Outputs to a csv called ‘scraped\_auto\_premium.csv’.

* **NOTE: MAKE SURE this file is NOT OPEN while the code is running, or the scraped premiums cannot be written to it. (If the file is open when the code attempts to write to the file then it throws up an error and does not write the data)**

### Notes

* The script uses explicit waits to handle dynamic page loading [time.sleep(‘seconds’)]. The wait times may need adjustment based on your internet speed.
* This script is provided as-is and may require adjustments based on changes to the AMI website structure. Use it responsibly and respect the website's terms of service.

# Detailed Code Explanation

## Important code details

### Interacting with the webpage

* The chrome ‘browser’ opened by the code is called the *driver*.
* We interact with the driver by applying various functions, from the selenium library to it
* HTML (Hypertext Markup Language) is a programming language which defines the blueprint of a webpage. It uses "tags" to define different parts of the website content, such as headings, paragraphs, images, and links, so browsers know how to organize and display them correctly.
  + HTML has a tree-like structure where elements are nested within one another, forming a hierarchy that represents the organization and relationship of content on a webpage.
* I use pythons’ *selenium package* to *find the specific html tags* that contain the content or perform the actions required by the scraper.
  + I have done this in two different ways in the code. Both having slightly different functions

1. driver.find\_element(By.ID, element\_id)
   * + Simply find an element, where ID attribute of the html tag has value element\_id.
     + We can search ‘By’ different methods, such as XPATHs (a method used extensively in the scraper) (see below for XPATH explanation)
2. WebDriverWait(driver, 10).until(EC.element\_to\_be\_clickable( (By.XPATH, "xpath") ) )
   * + This finds the html tag with the XPATH, ‘xpath’, similar to how find\_element does.
     + The main difference is that WebDriverWait allows the code to specify a maximum length of time (10 seconds as above) to wait *until* a certain Expected Condition (EC) is satisfied. (If the time runs out then a TimeoutException is raised)
     + In the code I have predefine a couple of expected conditions. I then use these throughout the code instead of putting the full definition in
3. Wait = WebDriverWait(driver, 3)
4. Wait10 = WebDriverWait(driver, 10)

#### XPATHS

XPATHs are instructions to find specific tags in the html based only on the structure of the html document. There are 2 types of XPATH

1. Absolute XPATHs give the full path from the root of the HTML tree (at the beginning of the document, to the element you're looking for)
2. Relative XPATHs define a specific point in the HTML tree to start from and then navigate to the element you're interested in using relationships between tags.

**Absolute XPATH example**

"/html/body/div[4]/main/div/div[2]/form/fieldset[1]/div/div/label[1]/span"

This absolute XPATH is constructed to target a specific element on a webpage based on its absolute position within the HTML structure:

* /html: Starts at the root of the HTML document.
* /body: Navigates to the body element.
* /div[4]: Moves to the fourth div element within the body.
* /main: Goes into the main element.
* /div/div[2]: Selects the second div element nested within the first one.
* /form: Continues to the form element.
* /fieldset[1]: Navigates to the first fieldset element within the form.
* /div/div: Moves through two nested div elements.
* /label[1]: Selects the first label element within the last div.
* /span: Finally, selects the span element within the first label.

In summary, this XPATH locates a span element within the first label element, which is nested within specific div elements, inside the first fieldset element within a form element, which is situated within specific div and main elements within the HTML document.

**Relative XPATH example**

"//\*[@id='quote']/fieldset[1]/div/div/label[2]/span"

This relative XPATH is constructed to target a specific element on a webpage based on its position relative to other elements in the HTML structure:

* //\*[@id='quote']: Starts the XPATH by selecting any element (//\*) with an id attribute equal to "quote".
* /fieldset[1]: Navigates to the first fieldset element within the previously selected element.
* /div/div: Continues by selecting two nested div elements within the first fieldset.
* /label[2]: Further narrows down to the second label element within the last selected div.
* /span: Finally, selects the span element within the second label.

In summary, this XPATH locates a span element within the second label element, which is nested within specific div elements, inside the first fieldset element with an id attribute equal to "quote".

### Error Handling

#### Error Handling in General

In Python, when a program encounters an error during execution, it raises an exception. An exception is basically an object that represents an error state. These exceptions can occur due to various reasons such as incorrect input, file not found, division by zero, etc.

To handle these exceptions gracefully and ensure that your program doesn't crash, Python provides a mechanism called "exception handling" using try and except blocks.

Here's how it works:

1. try block: This is where you place the code that you suspect might raise an exception. You enclose this code inside a try block.
2. except block: If an exception occurs within the try block, Python looks for a matching except block to handle it. An except block contains code that defines what to do if a specific type of exception occurs. You can specify the type of exception you want to catch by providing its name after the except keyword. If no exception name is provided, then the except block just catches all exceptions that occurred in the associated try block.

#### Error Handling within the web scraper code

In the code, I use try, except blocks to facilitate the scraping.

* This is because the code sometimes needs search for html tags that are not always present.
  + For example, on AMI’s website, it sometimes asks whether or not a car has an immobiliser. However, it doesn’t always appear, as it only asks on older cars, that might not have one, as newer car all do.
* This can present a problem, as when we search for a tag that is not present, an error occurs, thus an exception is raised
  + either ElementNotFoundError: for driver.find\_element()
  + or TimeoutError: for Wait.until(…)
* This is solved by using error handling, as we attempt to find this intermittent html tag within the try block.
  + Thus, if the tag happens not to be present on this occasion, then we can just go to the except block to handle the issue

### Error Code system

***No Error***: If the code ran on that website for the given example and scraped out the premium

***Several Car Variant Options Warning***: A message that warns that there were several options when selecting the car variant. This means that it is possible that the option selected is not precisely the same as the option presented in the dataset

***Website Does Not Quote For This Car Variant/ Person***: Occurs when the website in question does not automatically quote for the given car. To get quotes for these cars the websites will prompt the user to call into their help centre to get a quote. Sometimes it takes issue with the person who is insuring the car, for example on some cars tower refuses to insure under 25 drivers.

***Unable to find car variant***: Occurs when there are no options for car variant that align with the data in the spreadsheet

***Excess cannot be changed from …***: An error code for examples where the websites do no allow the excess value to be changed at all from what they set by default (…)

***Unknown Error***: A general message for any other error encountered while scraping. Often these errors are caused by slowness on the browser preventing the scraping from executing correctly. The code attempts to remedy this by going back over all the unknown error once, at the end of the code, to attempt to ‘fix’ the situation scraping error. However, some of these errors are unavoidable, however.

## Overall Code description

### Insurance-premium\_web-scraping\_v3 Overview

This code is set up to facilitate the scraping of insurance premiums from AA, AMI and Tower

1. First the code calls the *main()* function

* Calls the function *auto\_scrape\_all()*

1. The function *auto\_scrape\_all()*

* Prompts the user to select which company websites to scrape from
  + This is done in the command line by entering either 1 (include) or 0 (not include) when prompted to regarding each company
  + Noting that to ‘enter’ 1 or zero, you type the number in the command line, then press enter
* Starts all the processes. Each ‘processes’ runs another separate python file. It runs all of the individual company website scrapers, to scrape from all the websites simultaneously. This is done so that each website can do its scraping independently of the others, thus speeding up the web scraping a lot.
* Next the code attempts to fix all the potentially fixable errors from the first pass of the scraper by calling the function *redo\_website\_scrape\_errors()*
* Then attempt to ensure all scraped values are using a consistent agreed value, by calling the function *redo\_changed\_agreed\_value()*
* Calls the function *export\_auto\_data()* to export the scraped premiums.
* Finally calls the function *delete\_intermediary\_csvs()* to delete all the intermediary csv files, to ensure that there are no unexpected results from the scraping

1. The function *export\_auto\_data()*

* Exports the data that has been scraped, along with the dummy data to the file ‘scraped\_auto\_premiums.csv’

### Individual Insurance Company python files overview

The scrapers for each individual company are as follows,

* Insurance\_premium\_web\_scraping\_AA.py
* Insurance\_premium\_web\_scraping\_AMI.py
* Insurance\_premium\_web\_scraping\_Tower.py

Each file performs a similar purpose, to scrape insurance premiums for the given website, then to export the scraped premiums into a csv called ’company’\_scraped\_auto\_premiums.csv. The main code file (Insurance-premium\_web-scraping\_v3) then reads in those files, combines them with the dummy data files and exports the final csv.

In Selecting the correct car, there are some nuances as there can often be many options to select

* Insurance\_premium\_web\_scraping\_AA.py, uses a for loop with various car properties from the data, to iterate through the options until one is selected
* Insurance\_premium\_web\_scraping\_AMI.py and Insurance\_premium\_web\_scraping\_Tower.py use the fuzzywuzzy package to perform ‘fuzzy matching’
  + The function fuzz.partial\_ratio() gives a score (0-100) based on how similar two strings are
  + The code uses this to compare the information from the spreadsheet to each of the potential car model options
  + It then chooses the option with the highest ‘similarity score’, implying that it is the most correct option, as it is most similar to the spreadsheet info.
* NOTE: I did not use Fuzzy matching for AA because it already had functional code, and I did not want to disrupt that
  + Additionally, the format of the options for AA is inconsistent, as they adapt how much is put in each option, depending on how much they already know about the car. This makes fuzzy matching much less reliable, as there is no basic ‘string format’ that the information from the spreadsheet can be put into, to compare well with every possible option