

THE UNIVERSITY OF HONG KONG  
FACULTY OF ENGINEERING  
DEPARTMENT OF COMPUTER SCIENCE



FINAL YEAR PROEJCT

Year 2019 – 2020

---

An Intelligent Assistant  
for HKU Students

---

Individual Final Report

Date:	3 May 2020	Student:	Chu Chi Hang	()
Supervisor:	Dr. T.W. Chim	Groupmates:	Ma King Wai	()
			Lau Ngai Fung	()

# Abstract

Students in HKU access multiple websites, such as Portal for student account query and Moodle for learning resource. Some of these websites also not designed for mobile application, which occupied long loading time. A lot of time has spent on loading images in webpages and clicking buttons as springboards to retrieve a single piece of information like deadline of an assignment. Therefore, my project provides an intuitive solution for solving the problem: a personalized intelligent assistant in form of web application for HKU students to retrieve information from Moodle and Portal agilely and conveniently by giving voice or text commands to the Assistant directly, while at the server web scraping technique are used to capture the user needs and pack it as a response. Along the project, the web scraping part consists of highly specialized function, which imposes great effort in maintenance and development. To make it extendible to other websites and functionalities, building an organized and flexible structure is the heart of the project. Apart from that, fast scraping and parsing is crucial to the overall performance, and a set of tools and techniques are utilized. This report focuses on web scraping and explore the general solution of parsing data without direct accessing database and adoptive to visual layout.

## Acknowledgement

I would like to thank our supervisor, Dr. T.W. Chim, whose expertise was invaluable in guiding me to direction of developing new functionalities of my Final Year Project. I would also thank Eric Ma King Wai and Benjamin Lau Ngui Fung for providing a great deal of assistance and support along the project.

# Abbreviations

- HTTP

Hyper Text Transfer Protocol

- OO

Object Oriented

- HTML

Hyper Text Markup Language

- CSS

Cascading Style Script

- Web driver

A program which controls a browser to execute commands

## List of Figures

Figure 1-1	HKU Portal display in desktop .....	11
Figure 1-2	HKU Portal display in mobile device .....	11
Figure 1-3	Steps to get assignment score.....	12
Figure 3-1	PyCharm Community Edition Preview .....	16
Figure 3-2	GitHub Desktop Preview .....	17
Figure 3-3	PEP8 Coding Style Naming Convention .....	17
Figure 3-4	Layering of web scraping module .....	19
Figure 3-5	Class Diagram of web scraping module .....	20
Figure 3-6	Inspecting HTTP request headers .....	23
Figure 4-2	Initial design of NoteMaster.....	36

## List of Tables

Table 3-1	Libraries used in web scraping module .....	22
Table 3-2	Test result of Selenium Requests Integration Test .....	24
Table 3-3	Test result of Browser Speed Test.....	25
Table 3-4	Test result of web scraping approaches .....	25
Table 4-1	ScrapeError types .....	26
Table 4-2	CallError types .....	26
Table 4-3	webutil functions.....	26
Table 4-4	Testsuit development .....	28
Table 4-5	testsuit3 functions.....	28
Table 4-6	Browser attribute.....	29
Table 4-7	Browser functions .....	29
Table 4-8	global function in NativeBrowser.py .....	29
Table 4-9	WebSite attributes .....	30
Table 4-10	WebSite functions .....	30
Table 4-11	Moodle attribute .....	30
Table 4-12	Moodle functions.....	31
Table 4-13	Portal functions.....	32
Table 4-14	Global functions in HKUSites.py.....	32
Table 4-15	WebMaster attributes .....	33
Table 4-16	WebMaster functions .....	33
Table 4-17	WebMaster settings.....	34
Table 4-18	HKUNotifier functions .....	36
Table 5-1	New functions in Branch 8.....	40

## List of Codes

Code 3-1	Test program for Selenium and Requests (I).....	23
Code 3-2	Test program for Selenium and Requests (II).....	24
Code 4-1	Example use case of testsuit3.....	28
Code 4-2	Create WebMaster with additional options.....	34
Code 4-3	WebMaster use case.....	34
Code 4-4	DialogFlow integration with web scraping module .....	37

# Table of Content

Abstract .....	2
Acknowledgement .....	3
Abbreviations.....	4
List of Figures .....	5
List of Tables .....	6
List of Codes .....	7
1. Introduction .....	11
1.1. Background .....	11
1.2. Objective.....	14
2. Literature Review .....	15
2.1. Scraper .....	15
2.1.1. Requests .....	15
2.1.2. Scrapy .....	15
2.1.3. Selenium .....	15
2.2. Parser.....	15
2.2.1. Beautiful Soup .....	15
2.2.2. lxml .....	15
3. Methodology .....	16
3.1. Development Settings.....	16
3.1.1. Operating System .....	16
3.1.2. Library Documentation.....	16
3.1.3. IDE.....	16
3.1.4. Version Control .....	17
3.1.5. Code Style .....	17
3.2. Design .....	18
3.2.1. Requirements.....	18
3.2.2. Responsibility of Classes .....	19
3.2.3. Structure .....	20



3.3.	Choosing library .....	22
3.3.1.	Overview .....	22
3.3.2.	Scraper Library .....	22
3.3.3.	Parser Library .....	22
3.3.4.	Others.....	22
3.4.	Techniques .....	23
3.4.1.	Integrate Selenium and Requests .....	23
3.4.2.	Browser Speed Test .....	25
3.4.3.	Scraping Speed Test .....	25
4.	Result .....	26
4.1.	Utility .....	26
4.1.1.	webererror.py.....	26
4.1.2.	webutil.py.....	26
4.1.3.	testsuit.py.....	28
4.2.	Scraper .....	29
4.2.1.	NativeBrowser.py .....	29
4.2.2.	HKUSites.py .....	30
4.2.3.	WebMaster.py .....	33
4.3.	Web Drivers.....	35
4.4.	Notification .....	36
4.5.	Integration with DialogueFlow .....	37
5.	Conclusion .....	38
5.1.	Deliverables.....	38
5.2.	Future Work .....	39
5.2.1.	Use Scrapy.....	39
5.2.2.	Integrate Requests and Selenium .....	39
5.3.	Deployment.....	40
5.3.1.	Deploy Older Version .....	40
5.3.2.	Integrate with Django Framework .....	40

5.3.3. Additional functions in Version 8.....	40
Appendix .....	41
A. Libraries.....	41
References .....	42

# 1. Introduction

## 1.1. Background

HKU provides more than one websites for student to access. Each of them has different use. For instance, HKU Portal manages students accounts, and HKU Library is focused on teaching resource. However, there are concerns on the usability of these websites.

The first concern is these websites are not designed for mobile devices. There are two distinct properties when mobile devices compare to others like desktop computer: (i) mobile devices has narrower monitors; and (ii) mobile devices has slower networking. For the (i), it is a common problem for mobile device users. According to the research of Alexndar Wenz [1], people took longer time to completing an online survey via mobile phone comparing to that of tablets. Besides, the research also mentioned that if the survey display is not optimized for mobile devices, the quality of survey response also degraded, indicating a poor control over devices. The display of HKU Portal is not designed for mobile devices. As reflected in Figure 1.2, the screen squeezes together without appropriate resizing, making the text unreadable and easy to wrongly clicked other buttons. This damages the user experience comparing to Figure 1.1's display on desktop.

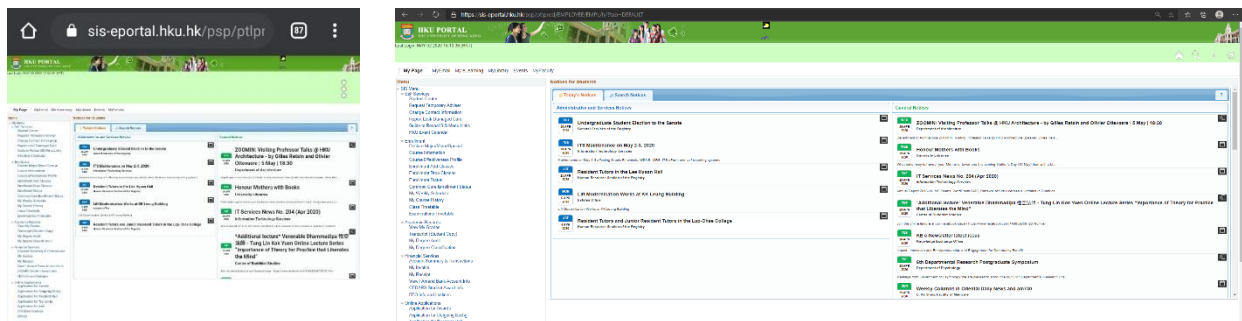


Figure 1-1 HKU Portal display in desktop



Figure 1-2 HKU Portal display in mobile device

For (ii), the loading time is slower as mobile devices. As found by Brian's research [2], mobile devices loads 87.84% longer than desktop. This may be due to the fact that mobile device uses wireless connections while desktop use wired connection. This results a significant difference when the website load background images or CSS style scripts, which further increases the loading time.

Apart from the mobile usage, multiple websites in HKU also make students difficult to manage. Sometimes student need to search across different sites to extract a piece of information when they do not know which website they should work on. It is still a problem even if one knows how to navigate the website. Illustrated by Figure 1.3, a student takes four click to access the assignment information, which means the browser need to load four pages. Besides, it consumes user's time for searching the right link and reading other unnecessary information.

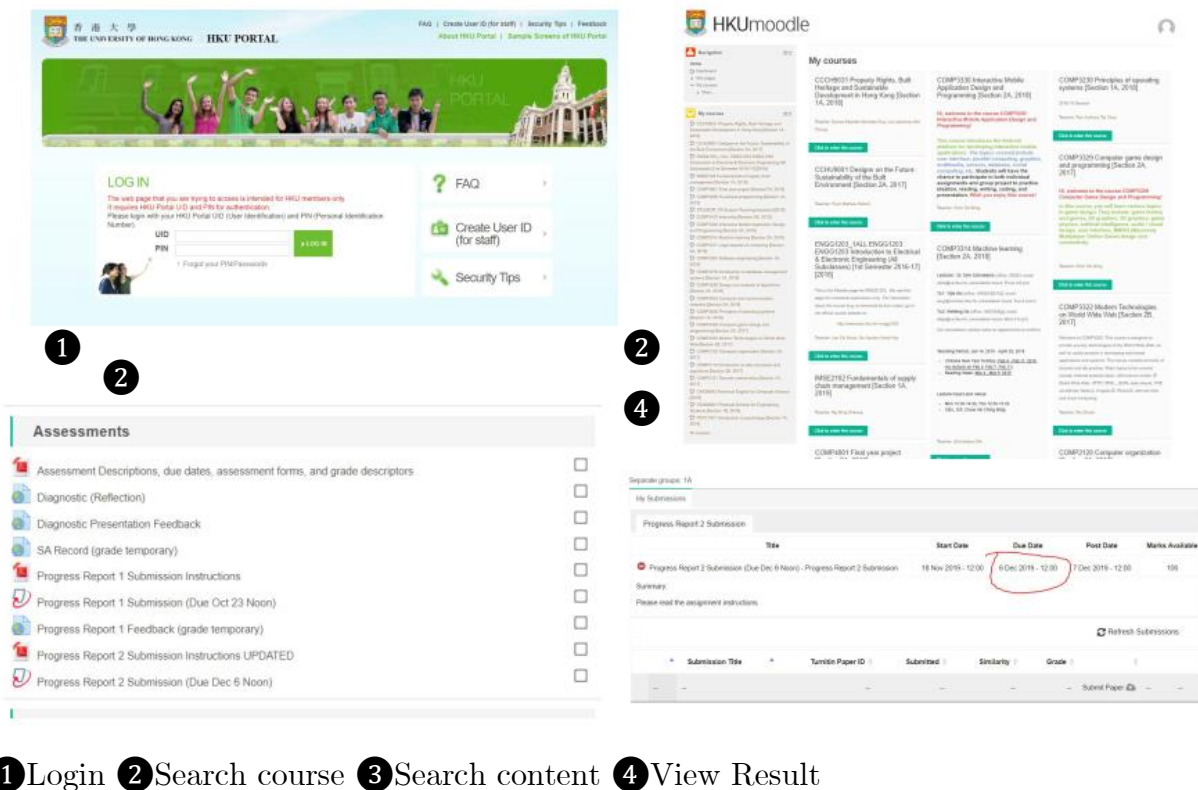


Figure 1-3 Steps to get assignment score

Therefore, to solve the above problem, a chatbot equipped with web scraping techniques is introduced for better user experience. Through using chatbot, it can simplify the long procedures and tedious management for websites. From the summary of Gil [3], about 70% of customers satisfy with a AI assistant (or chatbot) in general. More than 60% of the consumers use conventional assistant for buying products and services. This clearly

shows that the use of chatbot might improve the quality of HKU websites. For more concrete reasons, chatbot can eliminate the need of loading background images and shorten searching time, which means visiting less pages and no need to search target manually, thus contributes to a faster loading time.

Furthermore, the entire chatbot application is independent of the website. There is no need to changes the UI as the display would not be visible to the user. This saves cost for redesign and manage visuals. Another benefit is that the page loading runs in backend server with wired connection, which is much faster than wireless access. Lastly, Adopting the system would not incur any changes to the existing websites. As the main data extraction is web scraping, no modification is required on database or page route logics.

## 1.2. Objective

### - Centralized Query

The Assistant provides a unified platform for accessing the information in HKU Portal and HKU Moodle without direct access. Such convenient is achieved by remote controlling a backend server equipped with stable network environment.

### - Fast Loading

The Assistant can eliminate the time for accessing the website by two methods: (i) no need to load the assets in webpage such as background images, styling and unnecessary content, and (ii) compress the response of the Assistant by only providing text or image content, thus favorable to mobile devices.

### - Precise Response

The answer replied by the assistant should not be vague or general. Instead of returning a wide range of correct solution, the assistant should return only the necessary information. For instance, returning the last assignment of Algorithm Course instead of returning a list of assignment in that course.

## 2. Literature Review

### 2.1. Scraper

#### 2.1.1. Requests

Requests is a simple, clean Python library for accessing websites. One major drawback is that it only can handle static HTML webpage. However, it has a lightning speed and easy to use.

#### 2.1.2. Scrapy

Scrapy is a framework for web scraping. It provides a set of tools which user only need to extend from the spider templates.

#### 2.1.3. Selenium

Selenium is a library designed for automate test for Web Applications. In another words, it simulates a human user controlling a browser by executing pre-written commands. Although it is a bit slow, it works with different browsers and handles dynamic JavaScript.

### 2.2. Parser

#### 2.2.1. Beautiful Soup

Beautiful Soup is a famous library for parsing HTML. It enables flexible query with different HTML properties such as role and aria label. However, it requires external library for parsing.

#### 2.2.2. lxml

lxml is famous for speedy parsing. Not only HTML, it also can parses XML and support Xpath searching. However, it is less suitable for pure HTML parsing, especially for searching HTML element properties and displaying HTML code.

## 3. Methodology

### 3.1. Development Settings

#### 3.1.1. Operating System

Windows 10 is the OS for development.

#### 3.1.2. Library Documentation

In this project, Read the Docs (<https://docs.readthedocs.io/en/stable/>) serves as the standard documentation for each library. Based on its auto-update features, we believe it provides a much better display comparing to the GitHub open-source ReadMe file.

In case there is no available documentation in Read the Docs, we refer to the GitHub ReadMe file as the official documentation.

#### 3.1.3. IDE

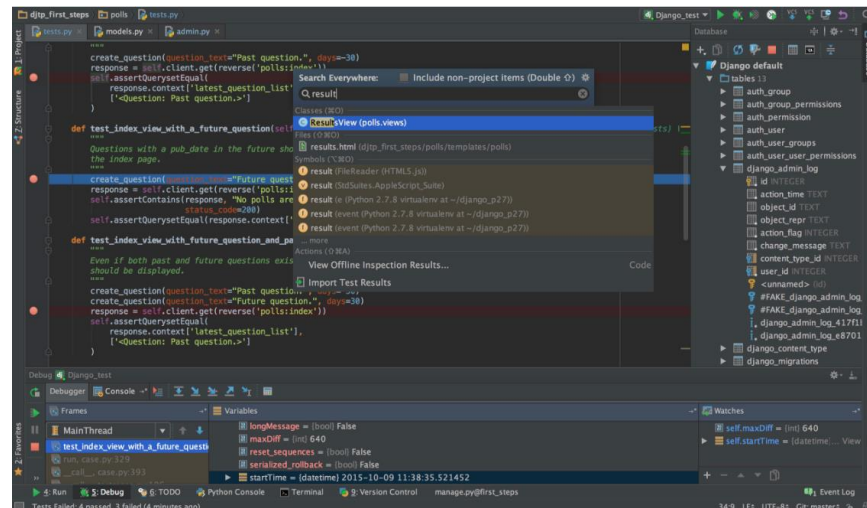


Figure 3-1 PyCharm Community Edition Preview

I use PyCharm Community Edition (<https://www.jetbrains.com/pycharm/>) as the IDE of development. It provides high level of programming-favored features like refactor, code style checking and quick search, given its auto-coding support.



### 3.1.4. Version Control

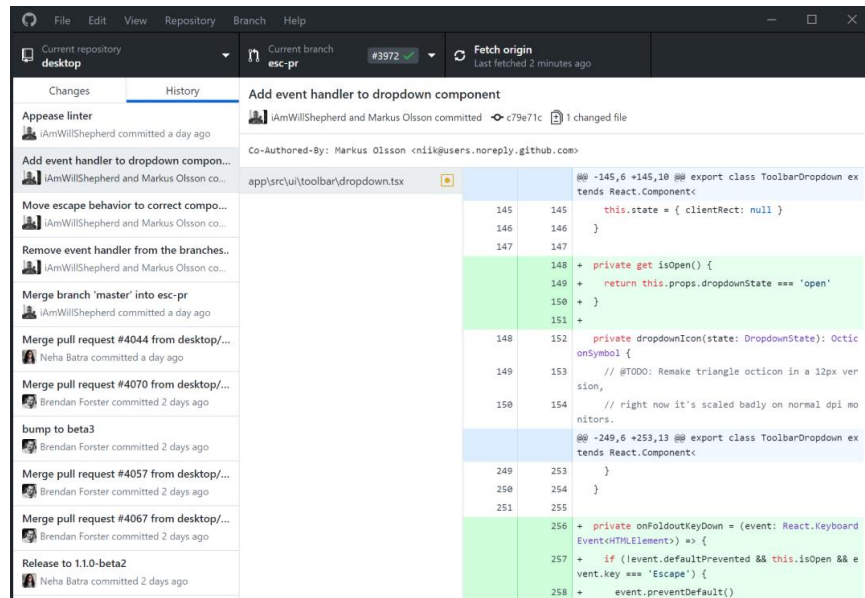


Figure 3-2 GitHub Desktop Preview

GitHub Desktop (<https://desktop.github.com/>) is the major version control tools. As fully integrated with Windows 10 environment, it provides a simple and user-friendly control for Git operations.

### 3.1.5. Code Style

Type	Public	Internal
Packages	<code>lower_with_under</code>	
Modules	<code>lower_with_under</code>	<code>_lower_with_under</code>
Classes	<code>CapWords</code>	<code>_CapWords</code>
Exceptions	<code>CapWords</code>	
Functions	<code>lower_with_under()</code>	<code>_lower_with_under()</code>
Global/Class Constants	<code>CAPS_WITH_UNDER</code>	<code>_CAPS_WITH_UNDER</code>
Global/Class Variables	<code>lower_with_under</code>	<code>_lower_with_under</code>
Instance Variables	<code>lower_with_under</code>	<code>_lower_with_under</code> (protected) or <code>__lower_with_under</code> (private)
Method Names	<code>lower_with_under()</code>	<code>_lower_with_under()</code> (protected) or <code>__lower_with_under()</code> (private)
Function/Method Parameters	<code>lower_with_under</code>	
Local Variables	<code>lower_with_under</code>	

Figure 3-3 PEP8 Coding Style Naming Convention

Following the PyCon 2016 guidelines on writing Python, PEP8 is adopted in this project. However, below are some places which fail to follow PEP8.

- Some lines are too long
- Module naming
- try-catch

## 3.2. Design

### 3.2.1. Requirements

There are a bunch of requirements the web scraping module would have to take care of. Below is a detailed list categorized by different aspects:

- Technical Requirement
  - Fast parsing and web scraping
  - Refresh webpage to prevent timeout
  - Manage browser tabs
  - Cache web scraping result
  - Check if webpage is updated
  - Record user function calls
  - Import only a single file for using web scraping module
- Structural Requirement
  - Introduce minimal changes when making changes
  - Able to work on different browsers
  - Error handling
  - Avoid duplications
- Development Requirement
  - Logging tools
  - Enable unit tests

Although the above list does not involve any use cases, it is the foundation before adding too many specific functions on scraping a particular target. To handle the above requirements, some strategies are devised to cope with:

- Create a Façade for end-user, which solve structure requirement
- Appropriate layering to separate responsibilities, thus enable robust testing
- Design templates before implementing concrete methods, thus allow pulling common methods and avoid duplications
- Indirect function calls to avoid rapid changes under development
- Extensive use of decorator pattern provided in Python to avoid duplications and enhance code readability

### 3.2.2. Responsibility of Classes

As mentioned before, appropriate layering is necessary to handle the requirements. To visualize the design, Fig. 3.4 illustrates the separation and responsibility of each layers.

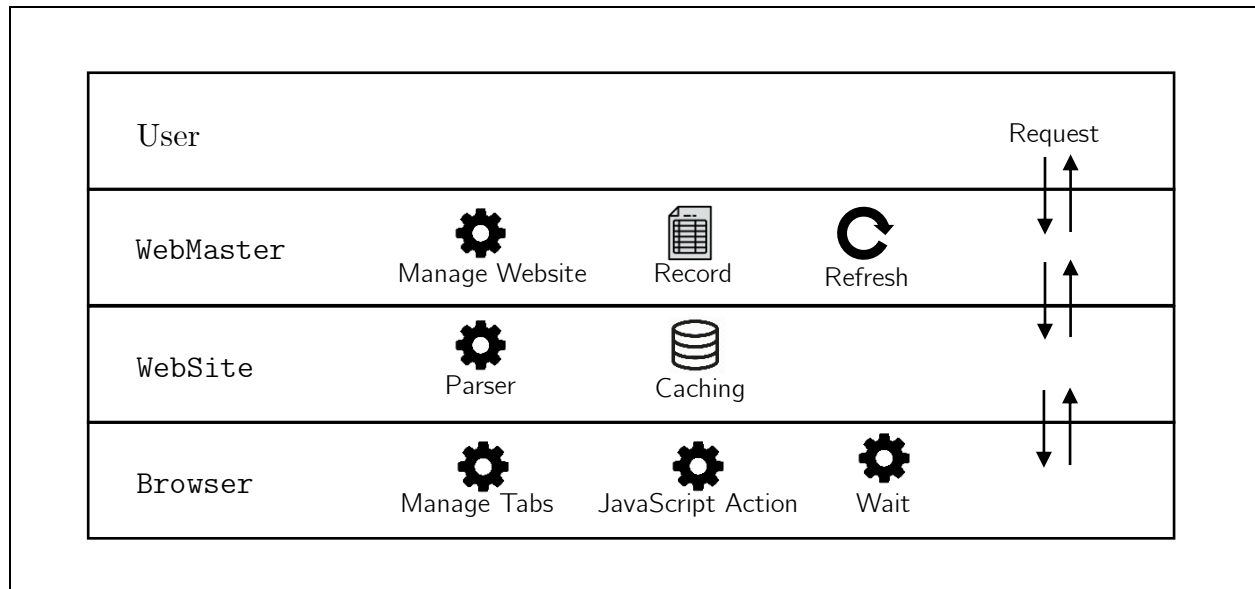


Figure 3-4 Layering of web scraping module

From the above figure, there are three layers (classes) that cover the major requirements in Section 3.2.1. In particular, below is the additional role of each class:

- WebMaster serves as the façade for user
- WebSite and Browser are templates, and open to extend
- Browser is the wrapper version of Selenium Webdriver. It provides a set of commonly used methods, such as press key, waiting and switch tab.
- Since each is a different class, programmer can test Browser, WebSite and WebMaster separately.

However, the above architecture cannot separate scraper logic and parser logic. In development version, there was a design that intended to solve this problem. A WebManager class serves as a delegate between WebSite and WebMaster, which is responsible for parser logic. But in finalization (Version 7), it was removed as the implementation of WebManager is highly dependent on WebSites, and it would be better to put parser logic with scraper logic. As a compromise, detailed documentation and comments are written to 'hold' the structure.

### 3.2.3. Structure

To realize the layered structure in Section 3.2.2, Object Oriented Design is adopted to make rather than directly building a layered structure. Figure 3.5 shows the OO design:

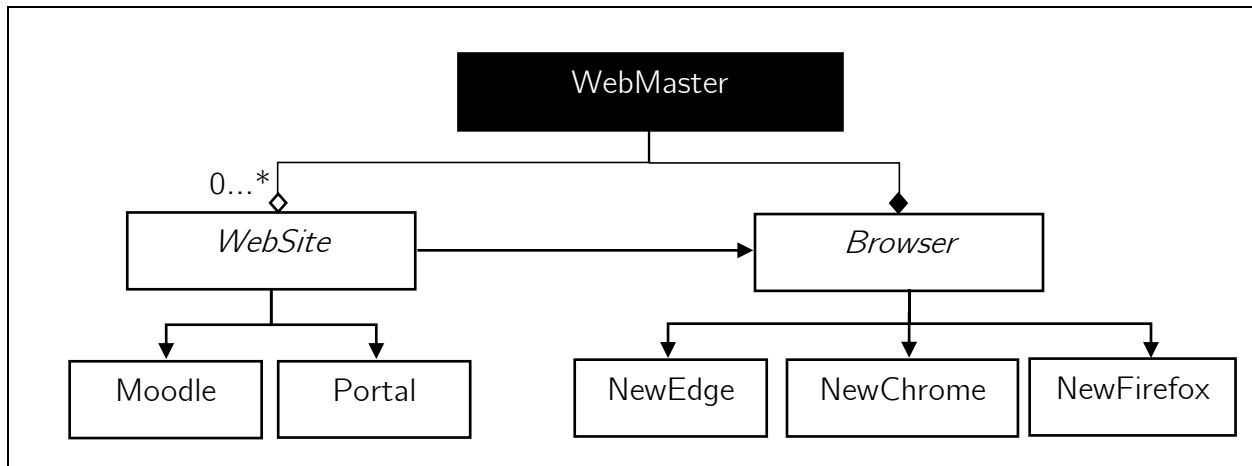


Figure 3-5 Class Diagram of web scraping module

#### WebMaster

**WebMaster** contains and manage all the components (**Website** and **Browser**). It periodically refreshes the website, capture all the user function call and handle web scraping exceptions. **Moodle** and **Portal** are classes extended from **Website** template, which has its individual implementation of different web scraping methods. **NewEdge**, **NewChrome** and **NewFirefox** are concrete implementation of **Browser** interface.

In order to refresh, **WebMaster** extends from Python Standard Library **Threading.Timer**, which is a thread designed for waiting functions to carry out. By overriding the **run** method and constructor, **webmaster** can refresh itself periodically. Before destructing **WebMaster**, one must call the **cancel** method to terminate it properly by setting its **finish** flag to **True**.

Since there might be chances that conflict occurs when the **WebMaster** is refreshing the webpages and the user also requests for information, a mutex lock is set to the browser before any method intends to use the browser.

Before the **WebMaster** terminates, it will logout from all websites and quit the browser, thus not leaving any unterminated thread or program.

#### Website

**Website** is responsible for controlling low-level scraping and parsing logics. Its main duty includes:

- Login
- Logout
- Refresh
- Get sitemap

Apart from the above functions, it also stores a set of website links, e.g. login URL.

## Browser

Browser is a wrapper of Web Driver. It mainly provides a better interface for calling function and import less libraries when using it. The function design of Browser is inspired by the jQuery API design. One of the noticeable idea is that the function can interpret the right meaning by inspecting the arguments, thus no need to split into other functions. More details will be revealed in Section 4.2.1.

### 3.3. Choosing library

#### 3.3.1. Overview

Below is the list of libraries used in the project:

Library	Usage	Installation
Selenium	Get Webpage	✓
Beautiful Soup	Parser	✓
lxml	Parser	✓
cachetools	Caching	✓
inspect	Debugging	✗
traceback	Debugging	✗
time	Timing	✗
datetime	Timing	✗
re	Regular expression	✗

Table 3-1 Libraries used in web scraping module

#### 3.3.2. Scraper Library

Given that Requests cannot handle dynamic HTML, it cannot be the main scraper library. In later part of the report, trials are introduced to integrate Selenium and Requests.

According to the comment of Michael Yin (<https://www.accordbox.com/blog/web-scraping-framework-review-scrapy-vs-selenium/>), Scrapy is suitable for fast, large scale and huge data-driven project, while Selenium works well with JavaScript and small scale websites. Given the size of my project (2 websites) and JavaScript-intensive tasks, Selenium is the best choice in this project.

#### 3.3.3. Parser Library

Both BeautifulSoup and lxml are used in this project.

#### 3.3.4. Others

Selenium-Requests and Requestium are both libraries targeted on integrating Selenium and Requests. However, the former is not working in browsers other than Firefox, and the latter is only designed for Google Chrome.

In order to speed up the web scraper and define hash keys, cachetools is included for faster and better performance.

## 3.4. Techniques

### 3.4.1. Integrate Selenium and Requests

In order to speed up the web scraping process, I have tried to combine Selenium web driver and requests program. As requests cannot run JavaScript, it cannot perform redirect and click button when login HKU Portal. Therefore, one idea is to copy all the session and cookies from the Selenium web driver to requests.

```
from myutil.testsuit import testsuit3 as testsuit

@testsuit.log('test', 6, 'destroy webmaster')
@testsuit.compileTest
def test_destroy_webmaster(webmaster):
    del webmaster

testsuit.printlog('field', 'Web Master')
testsuit.printlog('test', 0, 'create webmaster')

test_result = {}
test_result['needBrowser'] = test_destroy_webmaster(webmaster)
```

Code 3-1 Test program for Selenium and Requests (I)

Although it is workable for some trials, over 40% of the trials are invalid. Therefore, another idea is raised: supply all browser argument inside the request header, such as browser type, status code and referrer policy.

To do this, I inspected the requests by login manually and check the request headers. Programmatically, I copy the browser headers by checking the Selenium web driver and supply hard coded arguments if not provided.

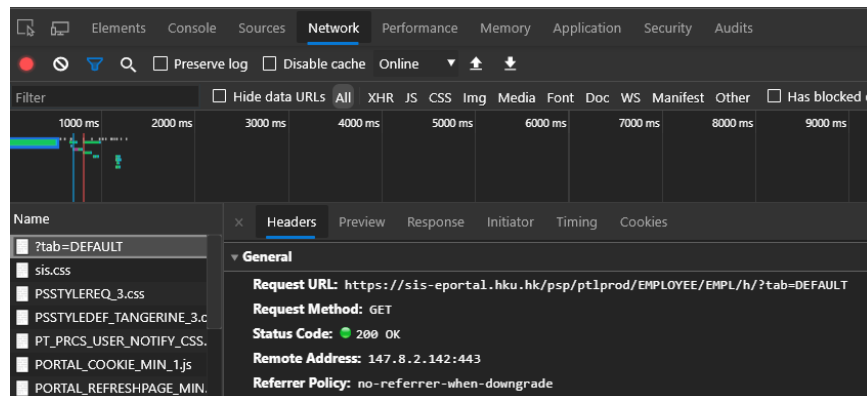


Figure 3-6 Inspecting HTTP request headers

```

from myutil.testsuit import testsuit3 as testsuit

@testsuit.log('test', 6, 'destroy webmaster')
@testsuit.compileTest
def test_destroy_webmaster(webmaster):
    del webmaster

testsuit.printlog('field', 'Web Master')
testsuit.printlog('test', 0, 'create webmaster')

test_result = {}
test_result['needBrowser'] = test_destroy_webmaster(webmaster)

```

Code 3-2 Test program for Selenium and Requests (II)

However, even supplying additional header arguments does not help to solve the problem. From Table 3.2, 40% of the trials still failed. As this is too unstable, requests is removed from the whole development.

Browser	Chrome	Edge	Firefox
Version	81.0.4044.129	81.0.416.64	75.0
Headless	✓	✗	✓
Trial 1	398.989315986633	240.539052486420	198.991411209106
Trial 2	384.157512187958	243.031306028366	201.003832817078
Trial 3	383.444612503052	238.081277847290	205.299703359604
Avg.	388.864	240.551	201.765

Table 3-2 Test result of Selenium Requests Integration Test



testRequests.py



### 3.4.2. Browser Speed Test

 testBrowser.py


In order to provide the best web scraping speed, the choice of browser is critical. Therefore, I have conducted the same tasks for the three major browsers: Google Chrome, Chromium-based Edge and Firefox Quantum and repeat for 10 times. Below is the test result:

Browser	Chrome	Edge	Firefox
Version	81.0.4044.129	81.0.416.64	75.0
Headless	✓	✗	✓
Trial 1	398.989315986633	240.539052486420	198.991411209106
Trial 2	384.157512187958	243.031306028366	201.003832817078
Trial 3	383.444612503052	238.081277847290	205.299703359604
Avg.	388.864	240.551	201.765

Table 3-3 Test result of Browser Speed Test

As a final result, Firefox has the shortest loading time.

### 3.4.3. Scraping Speed Test

 testScrapeskill.py

There are two methods to parse a HTML segment: first select by selenium browser, and then parse as little as possible; or parse the whole HTML page, and select element as accurate as possible. To distinguish which approach is faster, a test is conducted by scraping an element repeatedly for 10 times. Below is the test result:


Approach	Parse All First	Select First
Trial 1	9.618555545807	9.277193546295
Trial 2	10.124194860458	9.442327976227
Trial 3	11.125134468079	9.645575761795
Avg.	10.289	9.455

Table 3-4 Test result of web scraping approaches

Note: Firefox is the browser in this experiment.

## 4. Result

### 4.1. Utility

 myutil/\*.py

#### 4.1.1. webererror.py

webererror defines the possible exceptions that might be thrown in the process.

Type	Message
0	Scrape Error: Element Not Exist
1	Scrape Error: Page Not Exist
2	Scrape Error: Invalid Result
Any	Scrape Error: Others

Table 4-1 ScrapeError types

Type	Message
0	Call Error: Function Not Exist
1	Call Error: Require Argument
2	Call Error: Ambiguous Function Call
3	Call Error: Invalid Argument
4	Call Error: Missing Argument
Any	Call Error: Others

Table 4-2 CallError types

#### 4.1.2. webutil.py

A list of functions that is useful in the project.

Function name	Input	Output	Usage
util_universal_hku_login	browser, credential	None	Login
util_soup2list	beautiful soup object	HTML text with tr and td transfere d to 2d rows	Parse HTML tables
util_list_search	search_list, quota, exact	List	Filter a list according to quota and exact

Table 4-3 webutil functions



### 4.1.3. testsuit.py

For better testing, `testsuit.py` in `myutil` is created for this purpose. Currently there are three versions of `testsuit`:

Version	Approach
1	Inherit the class to extend the test methods
2	Create a class which accept function as variables
3	Decorators and functions grouped in a dictionary

Table 4-4 Testsuit development

Only version 3 is discussed in details.

#### Functions in testsuit3

Function name	Input	Output	Usage
<code>compileTest*</code>	None	Boolean	Ensure smooth
<code>timedTest*</code>	Repeat	Duration	Time the code
<code>errorTest*</code>	None	Boolean	Show error
<code>log*</code>	Formattype, tab, msg	None	Print debug log
<code>printLog</code>	Formattype, tab, msg	None	Print debug log

\* Decorator function

Table 4-5 testsuit3 functions

```
from myutil.testsuit import testsuit3 as testsuit

@testsuit.log('test', 6, 'destroy webmaster')
@testsuit.compileTest
def test_destroy_webmaster(webmaster):
    del webmaster

testsuit.printlog('field', 'Web Master')
testsuit.printlog('test', 0, 'create webmaster')

test_result = {}
test_result['destroy_browser'] = test_destroy_webmaster(webmaster)
```

Code 4-1 Example use case of testsuit3

Instead of returning the result of the code, the decorator functions modify the return of the function and give test related data.

## 4.2. Scraper



myscraper/\*.py

### 4.2.1. NativeBrowser.py

Browser

Attribute name	Type	Usage
tabdict	Dict	Store the key-tab pairs for switching tabs

Table 4-6 Browser attribute

Function name	Input	Output	Usage
constructor	Headless(boolean), options(list), path(str)	Browser instance	Create browser
press_key	key(str)	None	Press keyboard
wait	time(int)	None	Implicit wait
wait	time(int), ec(str), by(str), elem(str)	None	Explicit wait using Selenium functions
tab	target(str)	None	Switch to tab with key=target
tab	target(str), url(str)	None	Create tab with key=target and visit page with url=url
untab	target(set)	None	Delete tab with key=target

Table 4-7 Browser functions

NewFirefox, NewEdge, NewChrome

None

Global

Function name	Input	Output	Usage
get_browser	browser(str), kwargs	Concrete Browser	Create concrete Browser specified by <i>browser</i> and <i>kwargs</i>

Table 4-8 global function in NativeBrowser.py

#### 4.2.2. HKUSites.py

##### WebSite

Attribute name	Type	Usage
sitemap	List	Sitemap of the website
site_links	Dict	Important URL of the website
site_name	Str	Name of the website
func_cache	Dict	Store the result of function call
html_cache	Dict	Store the length of HTML of URL
hash_func	Func	Hash function for caching
username	Str	Store login username
password	Str	Store login password
debug	Boolean	Print debug log

Table 4-9 WebSite attributes

Function name	Input	Output	Usage
constructor	Username(Str), password(Str), cachesize(int), verbose(int)	Website instance	Create Website
print_debug	message(str)	None	Press keyboard
refresh	browser(int)	None	Refresh stale pages
destroy	browser(Browser)	None	Logout website and delete tab
start	browser(Browser)	None	Create tab, get sitemap and Login website
get_sitemap	browser(Browser)	None	Get sitemap

Table 4-10 WebSite functions

##### Moodle

Attribute name	Type	Usage
content_type	Dict	Store the content type of link pattern

Table 4-11 Moodle attribute

Public Function	Input	Output	Usage
find_course_by_keywords	browser(Browser) , keywords	(name, url)	Get the matching course name and link from sitemap
find_all_courses_by_keywords	browser(Browser) , keywords	(name, url)	Get all the matching course names and links from sitemap
find_deadlines	browser(Browser)	List of dict	Get the deadlines of Moodle submissions
find_page_preview	browser(Browser) , URL	Filename (str)	Get the screenshot of the URL page
find_course_contents	browser(Browser) , keywords (string), type(list of string), search (2D list of string), quota (int), exact (Boolean)	List of dict / dict	Get the course contents by (i) keywords (search course), (ii) type of resource, search string (file name & section name), quota (position of contents), exact (return one or a list)

Table 4-12 Moodle functions

## Portal

Public Function	Input	Output	Usage
find_transcript	browser(Browser), keywords	Dict with lists	Extract transcript information
find_receipt	browser(Browser), url(str)	Dict with lists	Extract receipt information
find_account_activity	browser(Browser), url(str)	Dict with lists	Extract account activities
display_weekly_sch	browser(Browser), target_date(str), start_time(str), end_time(str)	HTML string	Extract and modify schedule HTML
find_weekly_sch	browser(Browser), target_date(str), start_time(str), end_time(str)	List of dicts	Extract schedule

Table 4-13 Portal functions

## Global functions

Function name	Input	Output	Usage
get_website	website(str), username(str), password(str), kwargs	Concrete Website instance	Create concrete WebSite specified by <i>website</i> and <i>kwargs</i>
switch_tab*	None	None	Switch to the tab of website page
browser_cached*	None	HTML	Check if the webpage is changed; if not, return cached output; else execute input function and caches it

\* Decorator function

Table 4-14 Global functions in HKUSites.py



### 4.2.3. WebMaster.py

#### WebMaster

Attribute name	Type	Usage
websites	Dict	Store website name - Concrete WebSite pairs
record	List	Record user function call
browser	Browser	Store Concrete Browser instance
mutex	Lock	Lock for using <i>browser</i>
username	Str	Login
password	Str	Login
debug	Boolean	Determine to print debug log

Table 4-15 WebMaster attributes

Some attributs in WebMaster are just for creating website or browser instance, which are not shown in the above table.

Function name	Input	Output	Usage
constructor	username, password	Website	Create webaster instance
need_browser*	None	None	Infernal functions for acquiring lock for browser
refresh	None	None	Refresh websites in webmaster
cancel	None	None	Decstructor
create_website	website_name(str)	None	Create website instance with key= <i>website_name</i>
delete_website	website_name(str)	None	Remove website instance with key= <i>website_name</i>
format_record	website_name, func_name, args	Formatted String	Internal functions for format records
add_record	website_name, func_name, *args	None	Add record
get_record	None	List of records	Get the user function calls
print_debug	Message(str)	None	Print debug log
query	website_name, func_name, *args	Result of calling <i>func_name</i>	Call function as <i>function_name</i> of website
scrape	website_name, func_name, *args	Result query method	Wrapper for calling query method

\* Decorator function

Table 4-16 WebMaster functions

## WebMaster settings

As for the purpose of development, you can enable a bunch of settings for webmaster:

Settings	Value Type	Options	Effect / Use
headless	Boolean	True	No browser display
		False	Open Browser Display
browser	String	Edge	NewBrowser is Edge
		Chrome	NewBrowser is Chrome
		Firefox	NewBrowser is Firefox
verbose	Integer	0	No debug log
		1	Print debug log of WebMaster
		2	Print debug log of WebMaster and Website
cache	Integer		Cache size of Website
Init_setting	String	All	Login all websites at creation
		Only Portal	Only login Portal at creation
		On demand	Only login when needed

Table 4-17 WebMaster settings

The method to include the setting is exemplified as below:

```
from myscraper.webmaster import WebMaster

webmaster = WebMaster(credential['username'], credential['password'], verbose=2,
browser_name='Firefox')
```

Code 4-2 Create WebMaster with additional options

## WebMaster use case

```
from myscraper.webmaster import WebMaster

# create WebMaster
webmaster = WebMaster('username', 'password')

# use WebMaster
result = webmaster.query('Moodle', 'scrape_deadlines')

# get WebMaster history
records = webmaster.get_record()

# not in use
webmaster.cancel()
```

Code 4-3 WebMaster use case

### 4.3. Web Drivers




myengine/\*.exe

This folder stores the list of web drivers used in the web scraping module.

- `chromedriver.exe`: web driver for Google Chrome
- `geckodriver.exe`: web driver for Quantum Firefox
- `MicrosoftWebDriver.exe`: web driver for Microsoft Edge

## 4.4. Notification

 notification/\*.py

Notification module is a module build on top of the web scraping module. The key idea of the notification module is to actively alert user important information. Therefore, it serves as the user of WebMaster, parse information and record the time to alert the notification. The idea of the notification module is similar to web scraping module which can be explained by Figure 4.2.

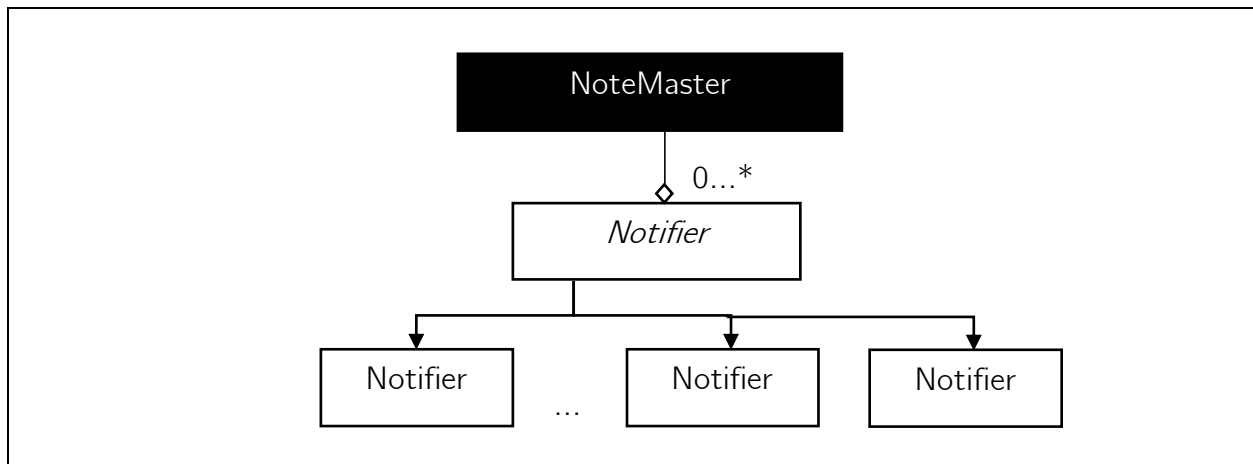


Figure 4-1 Initial design of NoteMaster

In late development, the notifier class even disintegrated into functions only, as the notifiers only return a structured dictionary.

Function name	Input	Output	Usage
portal_next_lesson_strategy	Webmaster	Dict	Alert if next lesson is near
portal_get_invoice_strategy	(WebMaster)		Alert if unpaid invoice
moodle_deadline_strategy			Alert if deadline is near

Table 4-18 HKUNotifier functions

For NoteMaster, it remains as a class extended from Threading.Timer, and acts as a registry to check whether the alerts are ready to signal to user.

## 4.5. Integration with DialogueFlow

To integrate web scraping module with DialogueFlow, additional logics are included in `views.py`.

```
from myscraper.webmaster import WebMaster
from .detect_intent_texts import detect_intent_texts

PROJECT_ID = 'hku-intelligent-assistant-bojt'
LANGUAGE_CODE = 'en-US'

def chat_view(request):
    session_id = request.session.session_key
    query_text = request.POST.get('msg')
    detect_intent_response_data = detect_intent_texts(PROJECT_ID, session_id,
    query_text, LANGUAGE_CODE)
    detect_intent_response_data["intent_display_name"] == "invoice-info-intent":
        findInvoice = getWebMaster(request.session.session_key).query('Portal',
        'findInvoice')
        response_data['content'] = findInvoice
        response_data['detected_intent'] = 'findInvoice'
    return JsonResponse(response_data)
```

*Code 4-4 DialogueFlow integration with web scraping module*

The major function is to use `detect_intent_texts` to bind between the project and the application. It passes the user input to the model and receives the matched intent (`detect_intent_response_data`). After that, from the matched intent, supply the argument and call corresponding function of webmaster for scraping (discussed in Section 4.3).

## 5. Conclusion

### 5.1. Deliverables

All the files can be access in GitHub page:

<https://github.com/Vini3x3/HKU-Website-Scraping/>

Version	Description
0	A set of scraper functions
1	Start using OOP concepts
5	Stable release before HKU Portal Upgrade, Provided infrastructure for development, Does not contain workable scraping functions.
6	Stable release before HKU Portal Upgrade, Provided infrastructure for development, Contain workable scraping functions.
7	Stable release before HKU Portal Upgrade Eliminate development tools
8	Stable release after HKU Portal Upgrade
9	Stable release after HKU Portal Upgrade, Rewritten with PEP-8 standard and redesigned

*Table 5.1 Versions in web scraping module*

Branch	Description
8_1	Additional extension for deployment Add notification module
8_2	Finalized version for deployment
9_1	Add notification module Additional extension for deployment More structural upgrade
9_2	Finalized Version Semantically use Python OOP

*Table 5.2 Branches in web scraping module*

Branch X\_Y means that the branch is fork of version X and locates at subversion Y. Due to the situation in deployment explained in Section 5.3.1, Branch 8\_2 is used in the group project deliverables.

## 5.2. Future Work

### 5.2.1. Use Scrapy

Although Scrapy is not included in this project, it is interesting to test will Scrapy scrapes faster than raw code. Besides, using Scrapy will only changes the middle layer of the structure (`WebSite`) in Section 3.2.2, making the adaptation easier.

### 5.2.2. Integrate Requests and Selenium

As mentioned in Section 3.4.1, the integration has failed. However, it is worthy to keep exploring the method of simulating browser requests by supplying the arguments in HTTP requests. In many cases, the web scraping does not need much JavaScript action, thus fast scraping static webpage can boost the overall speed.

## 5.3. Deployment

### 5.3.1. Deploy Older Version

Since Version 8 of the web scraping module appears earlier than Version 9, it has already integrated with the Django and DialogueFlow. Therefore, Version 8 is the main base for development and continues to add new functions upon request. Branch 8\_1 and Branch 8\_2 are created for this purposes.

Although Version 8 is operational, some of its code remain unrefactored, such as the caching of scraping functions in `HKUSites.py`. In order to make it better, part of the code is refactored, such as exception handling and caching.

### 5.3.2. Integrate with Django Framework

As mentioned in Section 4.4, `NoteMaster` refreshes itself periodically to check for alerts and send alert to front-end. To complete the request-response cycle rather than an abrupt backend signal to front-end, a thread is written in front-end to refresh every 10 minutes and check for any alerts. Therefore, `NoteMaster` does not need to extend from a Timer thread; it only need to passively wait for call. This simplifies the design logic and make less error by managing threads.

### 5.3.3. Additional functions in Version 8

At the late stage of development, new scraping functions are built in Branch 8\_1 and Branch 8\_2 at `HKUSites.py`. The summary of these functions are listed as below:

Function	Description
<code>findTable</code>	Find tables with table id in HKU Portal, reusing <code>findInvoice</code> , <code>findReceipt</code> , <code>findAccountActivity</code>
<code>scrapeFolder</code>	Scraper function which scrape folder page in HKU Moodle
<code>findCourseContents</code>	Scraper function which scrape course content with section name (e.g. add “lecture” label to file under section “lecture”)
<code>listSearch</code>	Handle dynamic list search, such as “second last item in list”
<code>listMatch</code>	Check if the target string matches any of the keywords in list
<code>superSearch</code>	Reusing <code>listSearch</code> , <code>scrapeFolder</code> and <code>findCourseContents</code> to combine a super search function

*Table 5-1 New functions in Branch 8*



# Appendix

## A. Libraries

Cachetools

<https://cachetools.readthedocs.io/en/stable/>

Selenium

<https://selenium-python.readthedocs.io/>

lxml

<https://lxml.readthedocs.io/en/latest/>

Beautiful Soup

<https://beautiful-soup-4.readthedocs.io/en/latest/>

Requests

<https://requests.readthedocs.io/en/master/>

Scrapy

<https://docs.scrapy.org/en/latest/>

Requestium

<https://github.com/tryolabs/requestium>

Selenium-Requests

<https://github.com/cryzed/Selenium-Requests>

PEP8

<https://pep8.readthedocs.io/en/release-1.7.x/>

## References

- [1] A. Wenz, "Completing Web Surveys on Mobile Devices:," Institute for Social and Economic Research, 2017.
- [2] B. Dean, "WE ANALYZED 5.2 MILLION DESKTOP AND MOBILE PAGES Here's What We Learned About Page Speed," backlinko, 2019.
- [3] G. Press, "AI Stats News: Chatbots Lead To 80% Sales Decline, Satisfied Customers And Fewer Employees," *Forbes*, 25 September 2019.