

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**VoiceWeb – Browser navigator using voice
commands**

propusă de

Ștefan Ihnatiw

Sesiunea: *Iulie, 2019*

Coordonator științific

Lect. Ionuț Pistol

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ

VoiceWeb – Browser navigator using voice commands

Ștefan Ihnatiw

Sesiunea: *Iulie, 2019*

Coordonator științific

Lect. Ionuț Pistol

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemnatul(a)

domiciliul în

născut(ă) la data de, identificat prin CNP,
absolvent(a) al(a) Universității „Alexandru Ioan Cuza” din Iași, Facultatea de
..... specializarea, promoția
....., declar pe propria răspundere, cunoscând consecințele falsului în
declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr.
1/2011 art.143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul:

_____elaborată sub îndrumarea dl. / d-na
_____, pe care urmează să o susțină în fața
comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin
orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea
conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări
științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei
lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere
că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data azi,

Semnătură student

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*VoiceWeb – Browser navigator using voice commands*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, 27-06-2019

Absolvent *Ștefan Ihnatiw*

(semnătura în original)

Content

Introduction.....	6
Motivation.....	6
Similar applications.....	9
Short description.....	9
1. Speech recognition and Browser functionality.....	11
2. Technologies used.....	13
2.1 Speech Recognition module.....	13
2.2 Google Speech API.....	17
2.3 Levenshtein module.....	20
2.4 Selenium's Webdriver.....	21
2.5 Action Chains.....	26
2.6 Pynput Module's Keyboard Listener & Controller.....	27
3. Application architecture.....	29
4. Conclusions.....	52
5. References.....	53
6. Annex.....	55

Introduction

After studying the course of Artificial Intelligence and being actively engaged in developing a project that people can find useful and interesting, I've decided that I wish to implement an application with a similar idea behind, that is able to receive vocal commands given by users and interact with the web browser in order to perform certain actions that may be either difficult or inconvenient otherwise.

Motivation

The internet has been, since it publicly launched in the 1990s, one of the biggest innovations and influencers towards building the technological world that we live in today. The World Wide Web has expanded over the course of the last 30 years to reach over 5.6 billion pages [1] that are indexed through search engines, with over 1.6 billion web sites [2] being active as of today (*Figure 1*).

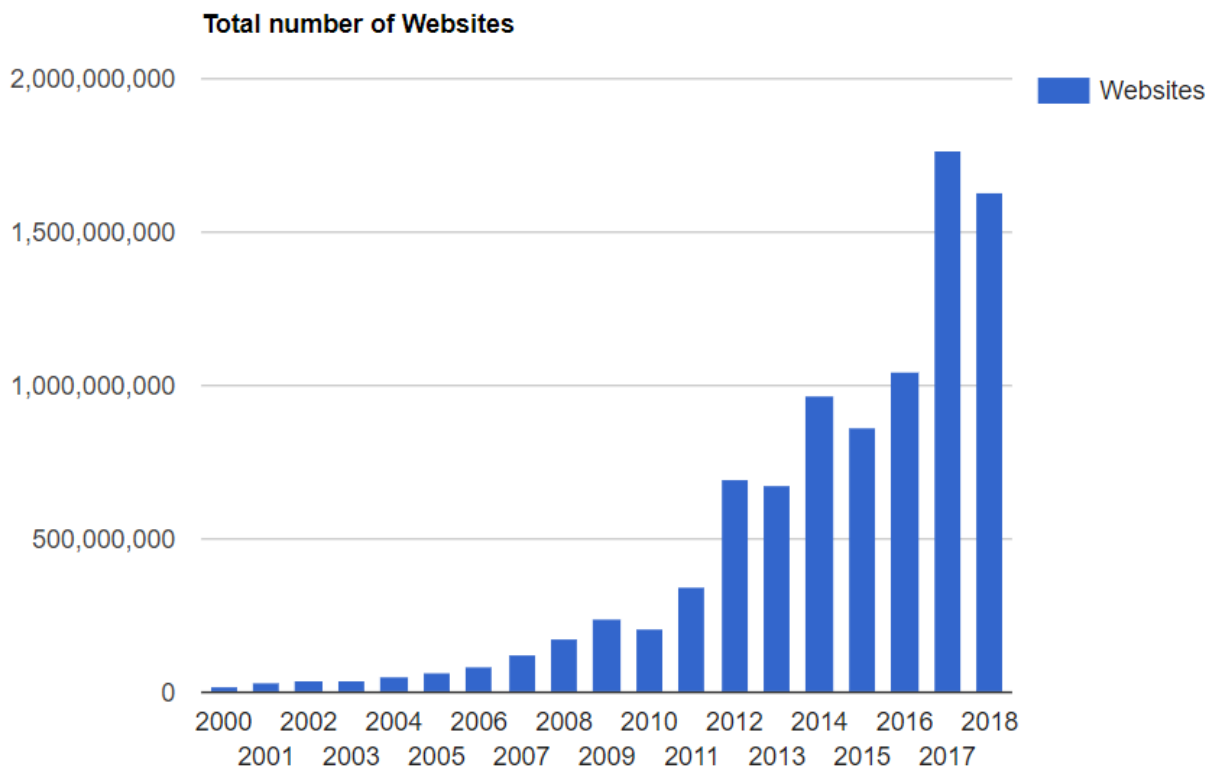


Figure 1. Total number of websites [2]

But more importantly, over 50% of the world's population is using the internet in 2019 [3], either on a computer or mobile devices. Some of the most popular activities on the internet are researching, browsing social media or online shopping, as statistics show the more popular domains being *google.com*, *facebook.com*, *wikipedia.org*, *amazon.com* [4].

There are many ways a user can access the internet, either by using a router to perform a WI-FI or Ethernet connection (indirect access), or by a direct connection which can be achieved by using a cable, or through mobile networks (*Figure 2*). Although a fixed connection through cable is likely a more reliable one, it can't be accessed remotely, which makes a mobile connection more appealing [5].

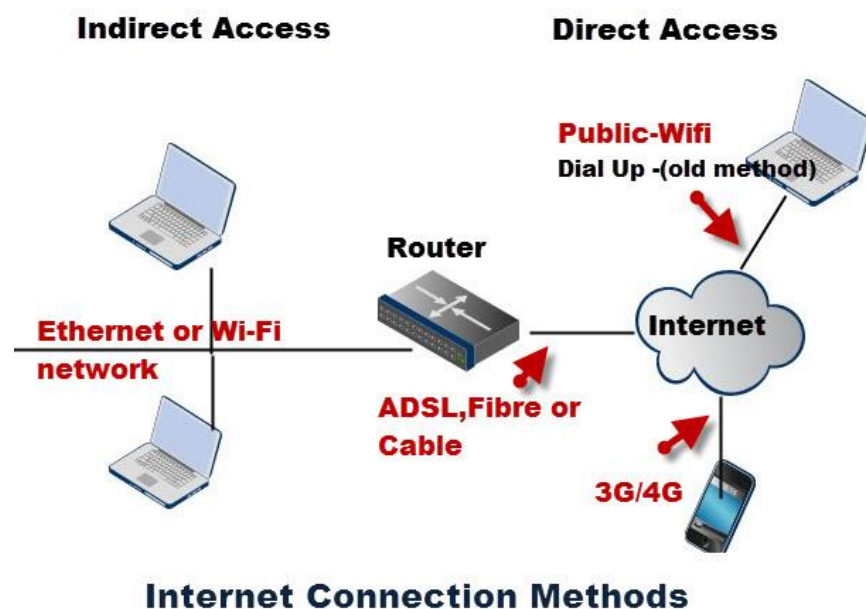


Figure 2. Internet Connection Methods [5]

There are also many types of devices nowadays that have access to internet, not only computers and mobile phones but also TVs, watches, cars, consoles or even fridges or other home devices. This makes it certain that the industry wishes to transform the day-to-day items that we use into smarter, more effective devices, and the internet plays an important role into achieving that kind of connectivity.

Another important area is the accessibility of the internet and the idea that all people, no matter their disabilities or place of living, should be able use it to their advantage and technologies

that can make that happen should be available. Some of those include voice recognition (to improve access for people with limited movement ability), scanner (that can synthesize speech from documents for blind people) or speech to text (to transcribe speeches for those who cannot hear) [6].

The high demand of internet usage makes it necessary to continue improving the browsing experience and create new technologies that can make it more efficient to accomplish certain tasks that are difficult to achieve otherwise. Using applications through voice commands has proven to be a necessary tool for the common user as most smartphones, TVs, speakers and car systems nowadays have such technologies implemented. Some of these include Google Assistant (enabled on 1 billion devices), Amazon's Alexa (100 million devices) [7], Apple's Siri (over 500 million devices). It is also estimated that the usage of voice assistants will triple by 2023 as the home devices industry will keep improving their features and become more attractive to the public [8].

Regarding web accessing on computers, for the last few years, Google Chrome has been the most used browser because of its simplicity, speed, security and large catalog of user developed extensions [9] (*Figure 3*).

Browser Market Share Worldwide
Jan 2009 - Sept 2018

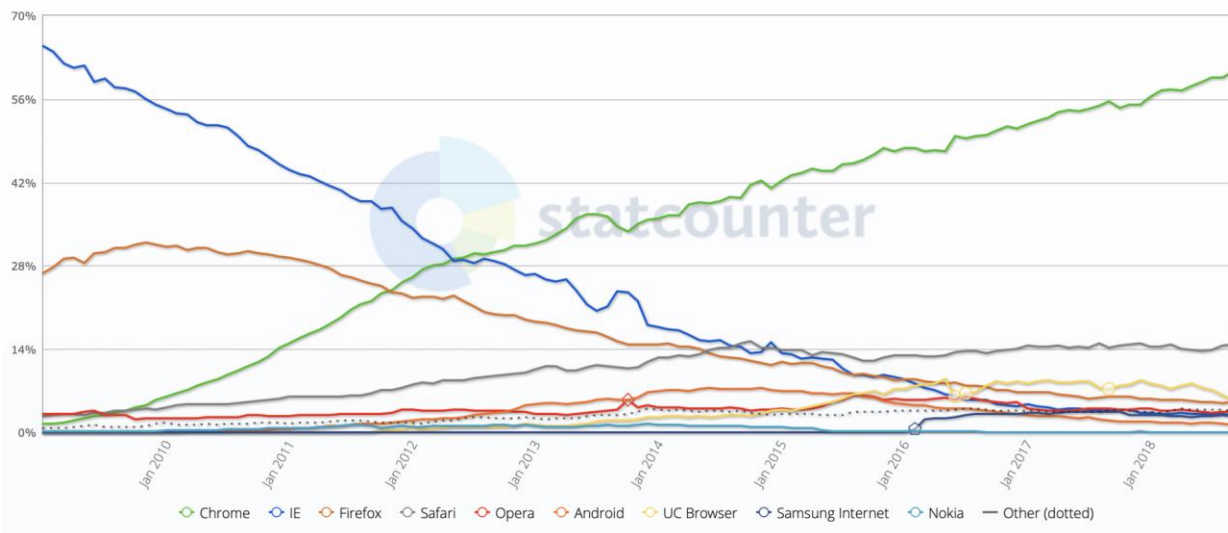


Figure 3. Browser usage share [9]

Just as mobile phones are getting more and more popular with using a voice assistant, the computer could also find it to be more convenient to be able to perform commands without having to use a mouse or keyboard. My application tackles the problem by dividing it in two parts, taking the vocal input from the user and initiating commands towards the browser based on that input.

Similar applications

Google Assistant has launched in 2016 and it lets you interact with your Android device using voice commands. The way it works is starting up after the user says “Ok Google”, and wait for the user to specify their command. The assistant is able to find information on your device, such as contacts, messages, calendar notes or open applications, but also find information online such as the weather, bookings, points of interest, or random searching [10].

Siri is Apple’s voice assistant which launched in 2011, currently being compatible with every built-in application on Apple devices (interacting with them similarly to Google Assistant) and starts up with the command “Hey Siri”. It can only run on iOS devices while Google Assistant can run both on iOS and Android [11].

Cortana is a voice assistant launched by Microsoft in 2014 for Windows 8.1 and released also for Windows 10 and few other platforms. It is integrated in Microsoft’s Edge browser and uses Bing as a search engine, performing similar tasks to the other assistants, but on a computer [12].

Short description

In Chapter 1, I will briefly present the idea behind speech recognition technologies and web browsing functionality, in order to put into context the basics of the application’s functionality.

In Chapter 2, the technologies that were used to implement the program will be detailed. I explained the approach they take in order to achieve the needed functionality and the way they make the implementation more intuitive.

In Chapter 3, I present the architecture of the application, consisting of all the functionalities it has, the way they work and how they interact with each other. I also present the flow of the program and how the user can interact with it.

In Chapter 4, I talk about the general conclusions and achievements of the application and also what else it could've achieved.

Chapter 1.

Speech recognition and Browser functionality

Speech recognition refers to the ability of a program to transform an audio input representing a speech, into a text representing the words of what was said. It is a popular technology used in many devices nowadays, such as mobile phones, household devices, car systems or robotics.

Some of the models [13] that try to solve the problem of recognizing speech are:

- *Hidden Markov models* – “doubly stochastic process with an underlying stochastic process that is not observable (it is hidden), but can only be observed through another set of stochastic processes that produce the sequence of observed symbols” [14]
- *Dynamic time warping* – algorithm that analyzes words based on the speed of the speaker
- *Neural networks* – network trained to map vocal inputs to text outputs in order to be able to determine the output for new inputs

These technologies are used in implementing the speech recognizers available today, such as Google API, Microsoft API And CMU Sphinx.

Another important aspect in developing the application is to understand the **browser’s functionality**. The World Wide Web [15] is a collection of web sites linked through URIs, that host resources which can be accessed through HTTP requests. In order to view the information listen on a web page, a user has to type an URL in the address bar of a browser and submit it, which will send a request to the host of the web site, returning the resources necessary to build the page in the browser.

Out of all the resources of a web site, the HTML file [16] is the most important in determining the structure and the elements belonging to that specific page. The basic structure of an HTML file looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

Figure 4. Simple HTML file [16]

Finding the elements of a web page can be realized by searching through different tags or values inside the file, such as `<p>` or `<h1>` to determine paragraphs, `<button>` to find buttons, `<a>` for hyperlinks, or `<input>` for input fields. Then, the program would have to read or modify the information of that field.

Chapter 2.

Technologies used

Speech Recognition Module

Python's *SpeechRecognition* module [17] is a library that supports multiple speech recognition APIs, and has the ability to listen and record a microphone in the background, convert audio input to text and save audio data to files. Some of the APIs supported are: CMU Sphinx, Google Speech API, Wit.ai, Microsoft API, Houndify or IBM.

To use the **Microphone**, the user has first to install the additional *PyAudio* module, which is used to record audio input. On instantiation, the constructor of the *Microphone* can be given the following optional parameters [18]:

Microphone(device_index: Union[int, None] = None, sample_rate: int = 16000, chunk_size: int = 1024) -> Microphone

1. *device_index*: an integer between 0 and the number of audio devices on the system, representing the index of the device that will be used to obtain audio input from; if unspecified, the default microphone will be selected
2. *sample_rate*: the rate of samples per second at which audio chunks are recorded; a higher value can result in a better audio quality, but slower recognition
3. *chunk_size*: the number of chunks in which the audio input will be separated

On instantiation, the *Microphone* object will check whether the user has assigned any optional parameters, and if so, whether they are the correct type (throws exception otherwise). Then it tries to import the *PyAudio* module and instantiate an object of it to use for the recording, raising *AttributeError* if the library hasn't been installed. If no error occurs, all the components needed for the *Microphone* object will be saved as class variables.

The **Recognizer** is a class used for setting up different functionalities of speech recognition, starting and stopping the recording process and selecting the API that will perform the transcription. It has no additional parameters and on initialization, it will set some class variables to default values [18]:

- *recognizer_instance.energy_threshold = 300 # type: float*

A variable that represents the perceived loudness of the sound, the value 300 being considered the threshold between silence and speech. But since each microphone has a different sensitivity that requires a different threshold, this value will be considered only as the initial (minimum) value and will need to be dynamically adjusted based on the hardware used and the loudness of the environment. This initial value can be modified to a lower value in the case that the microphone is not able to detect audio input, or to a higher value if the background noise makes it hard to understand and transcribe what the speaker is saying.

- *recognizer_instance.dynamic_energy_threshold = True # type: bool*

In most cases, the energy threshold needs to be automatically adjusted based on the used microphone device and the background noise in the room. This value should be changed only if the ambient of the room remains constant and the initial energy threshold is set to the appropriate value.

- *recognizer_instance.dynamic_energy_adjustment_damping = 0.15 # type: float*

Approximates the speed at which the energy threshold is dynamically adjusted in one second (if the *dynamic_energy_threshold* property is enabled). A lower value means a faster adjustment, but it could miss certain parts of the audio input, while a higher value will decrease the rate of modification.

- *recognizer_instance.dynamic_energy_adjustment_ratio = 1.5 # type: float*

Represents the minimum ratio at which the speaker's voice is louder than the background noise (if the *dynamic_energy_threshold* property is enabled). A smaller value would result in a harder differentiation between the two, while a larger value requires a quieter ambient noise.

- `recognizer_instance.pause_threshold = 0.8 # type: float`

The minimum number of seconds that represents silence between phrases. Lowering the value could generate a negative output for slower speakers, while increasing it could damage it for faster ones.

- `recognizer_instance.operation_timeout = None # type: Union[float, None]`

The number of seconds after which an internal operation (such as an API request) will timeout. The *None* default value means there is no timeout and the program will wait for the action to be performed.

In order to initiate the recording of the microphone, the following function will be used [18]:

`recognizer_instance.listen_in_background(source:AudioSource,callback:Callable[[Recognizer, AudioData], Any]) -> Callable[bool, None]`

Given an *AudioSource* instance as input (in this case, the *Microphone* object), it will transform it into an *AudioData* instance and send it as a parameter to the given callback function. It returns a function that, when called, will stop the background listening. The process will execute on a separate thread, so that the main thread will be able to perform other actions before it chooses to stop it.

When called, the function will check whether the given input is of the correct type (*AudioSource*), then it will initialize and start the thread that performs the listening. The thread is also marked as a daemon thread, which means that it will terminate automatically if the main thread completes (as long as there's no other non-daemon thread present), and it can also be shut down at any time during its execution. The recording thread will continuously call a separate *listen* function that will handle the conversion of *AudioSource* to *AudioData* for each phrase, and, after it, the callback function which will do the recognizing. The thread will check every second if the stop function has been called.

The stop function is the function returned by the *listen_in_background* method, that, when called, will stop the recording thread from sending sound data to the converting function and thus

finishing the thread. It has one optional parameter, *wait_for_stop*, that is set to a default *True* value, which will make the function wait for the thread to finish before returning. If set to *False*, the method will return right away, while the thread will still be on its way to end.

While *AudioSource* is an abstract class, implemented by the *Microphone* class, an *AudioData* object holds in the *frame_data* variable a sequence of bytes that represent audio samples. This raw data can be then converted into different audio formats, if needed. The object also contains a *sample_width*, representing the width of each sample in bytes, and a *sample_rate* for the data [18].

AudioData(frame_data: bytes, sample_rate: int, sample_width: int) -> AudioData

The *listen* function that performs the conversion from the *Microphone* to *AudioData* has the following structure [18]:

recognizer_instance.listen(source: AudioSource, timeout: Union[float, None] = None, phrase_time_limit: Union[float, None] = None, snowboy_configuration: Union[Tuple[str, Iterable[str]], None] = None) -> AudioData

The recording takes place when the energy detected is above the *recognizer_instance.energy_threshold* value, and ends after few seconds of silence, given by the value of *recognizer_instance.pause_threshold*. It can stop also when there's no more audio input sent to the method (when the stop function of *listen_in_background* is called). It has the following parameters [18]:

1. *source*: the input *AudioSource* object
2. *timeout*: the number of seconds the function waits for an input phrase before exiting; if *None*, it will wait indefinitely
3. *phrase_time_limit*: the maximum number of seconds allowed for recording one phrase, before stopping and processing what was recorded in that timeframe; if *None*, a phrase will have no time limit

4. *snowboy_configuration*: a configuration for a third party recognizer that can train a neural network based on the given inputs; if *None*, it will not be used

The way it works is verifying, first, that the source is a valid *AudioSource* object, then recording the audio input until there is a silence of *timeout* length (if not *None*), or until a phrase is long enough. If *dynamic_energy_threshold* is enabled, the *energy_threshold* will be updated based on the current threshold, the *dynamic_energy_adjustment_damping* and *dynamic_energy_adjustment_ratio* (Figure 5).

```
damping = self.dynamic_energy_adjustment_damping **  
seconds_per_buffer  
target_energy = energy * self.dynamic_energy_ratio  
self.energy_threshold = self.energy_threshold * damping +  
target_energy * (1 - damping)
```

Figure 5. Updating the energy threshold

The audio input of the microphone will be turned into a sequence of bytes that, concatenated, will represent the *frame_data* element of an *AudioData* object. The transformation is done using *PyAudio*'s *read* function, that takes as a parameter the number of chunks to be read. After finishing, the function will return the newly created *AudioData* object.

Google Speech API

“Google has improved its speech recognition by using a new technology in many applications with the Google App such as Goog411, Voice Search on mobile, Voice Actions, Voice Input (spoken input to keypad), Android Developer APIs, Voice Search on desktop, YouTube transcription and Translate, Navigate, TTS. After Google, has used the new technology that is the deep learning neural networks, Google achieved an 8 percent error rate in 2015 that is reduction of more than 23 percent from year 2013. According to Pichai, senior vice president of Android, Chrome, and Apps at Google, “We have the best investments in machine learning over the past many years. Indeed, Google has acquired several deep learning companies over the years, including DeepMind, DNNresearch, and Jetpac”.”[19]

The way Google Speech API is integrated in the *SpeechRecognition* module is through the method *recognize_google* of the *Recognizer* object [18].

```
recognizer_instance.recognize_google(audio_data: AudioData, key: Union[str, None] =  
None, language: str = "en-US", , pfilter: Union[0, 1], show_all: bool = False) -> Union[str,  
Dict[str, Any]]
```

The function would be called after obtaining an *AudioData* object from the *listen* function and has the purpose to obtain the transcription of audio data (represented as a sequence of bytes that build a specific format file) by sending a request to the API with the given input. It will return an error if the speech cannot be recognized or if the API cannot be accessed. As parameters, it takes:

1. *audio_data*: the *AudioData* object that will be transcribed
2. *key*: a key required to access the API; if *None*, will use a default key
3. *language*: the language of the spoken text
4. *pfilter*: profanity filter
5. *show_all*: if *False*, will return the response as a string, else, as a json dictionary

On implementation, the function will check that the given *audio_data* is of the correct type, and then will convert the bit string of the *frame_data* element in a byte representation of a FLAC file, using a third party executable. Then, it will send the request to the API and determine the result with the highest confidence rate from within the response dictionary (*Figure 6*).

```

url = "http://www.google.com/speech-
api/v2/recognize?{}".format(urllib.parse.urlencode({
    "client": "chromium",
    "lang": language,
    "key": key,
}))
request = Request(url, data=flac_data, headers={"Content-Type":
"audio/x-flac; rate={}".format(audio_data.sample_rate)})

# obtain audio transcription results
try:
    response = urlopen(request, timeout=self.operation_timeout)
except HTTPError as e:
    raise RequestError("recognition request failed:
{}".format(e.reason))
except URLError as e:
    raise RequestError("recognition connection failed:
{}".format(e.reason))
response_text = response.read().decode("utf-8")

```

Figure 6. Call to the Google Speech API

Possible outcome:

```

{
  "result": [
    {
      "alternative": [
        {
          "transcript": "hello world",
          "confidence": 0.80891722
        },
        {
          "transcript": "hello-world"
        },
        {
          "transcript": "hello word"
        }
      ]
    }
  ]
}

```

```

{
    "transcript":"hello work"
},
{
    "transcript":"hallo world"
}
],
"final":true
}
],
"result_index":0
}

```

Levenshtein Module

Python's *Levenshtein* module has implemented the ability to calculate the Levenshtein distance between two strings, and return a ratio between 0 and 1 of their similarity.

The Levenshtein distance [20] between two strings represents a way to tell how similar the strings are, by calculating the minimum number of single characters edits it takes to transform one into the other. The possible edits to a character can be substitution, insertion or deletion.

$$\text{lev}_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1, j) + 1 \\ \text{lev}_{a,b}(i, j-1) + 1 \\ \text{lev}_{a,b}(i-1, j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure 7. Calculating Levenshtein distance [20]

The algorithm (*Figure 7*) takes a recursive approach, by calculating the minimum number of edits between deleting a character from the first string, the second, or both, and checking whether that character from the same position matches in the two.

Selenium's Webdriver

Selenium is a Python module that provides an API which can automate the interaction with the web browser [21]. It can be installed using the following command:

pip install selenium

It also needs a web driver for a browser in order to be able to interact with it, which can be found on the developer's web site [22]:

Chrome:	https://sites.google.com/a/chromium.org/chromedriver/downloads
Edge:	https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
Firefox:	https://github.com/mozilla/geckodriver/releases
Safari:	https://webkit.org/blog/6900/webdriver-support-in-safari-10/

In order to launch the browser, the following lines need to be executed:

```
from selenium import webdriver  
driver = webdriver.Chrome()
```

Figure 8. Launching the Chrome browser

The first line imports the *selenium.webdriver* module and the second one instantiates a Chrome driver, which will launch the browser on a blank page (*Figure 9*). The module can also interact with other browsers, such as Firefox, Edge, Internet Explorer, Safari, Opera.

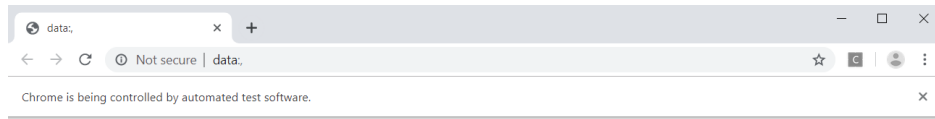


Figure 9. Browser opening on blank page

To navigate to a page on the browser, the *get* function is used:

```
driver.get("https://www.python.org/")
```

Figure 10. Usage of driver.get function

Then, to find an element on the page, multiple functions can be used, that parse the HTML file and look for certain identifiers [22]:

- *find_element_by_id*: searches based on the id attribute of an element (*Figure 11*)

```
<form id="loginForm">
```

```
<input name="username" type="text" />
```

```
<input name="password" type="password" />
```

```
<input name="continue" type="submit" value="Login" />
```

```
</form>
```

```
driver.find_element_by_id('loginForm')
```

Figure 11. Usage of driver.find_element_by_id function

- *find_element_by_name*: searches based on the name attribute of an element (*Figure 12*)

```
<input name="username" type="text" />
```

```
driver.find_element_by_name('username')
```

Figure 12. Usage of driver.find_element_by_name function

- *find_element_by_xpath*: performs a search similar to one in an XML file, being able to determine a relative path from one element to the other, or an absolute one from the root of the document (*html*); it is also able to find items of same rank based on their order and select only that with specific attributes (*Figure 13*)

```
<html>
```

```
<body>
```

```
<form id="loginForm"> ... </form>
```

```
</body>
```

```
</html>
```

```
driver.find_element_by_xpath("/html/body/form[1]")  
driver.find_element_by_xpath("//form[1]")  
driver.find_element_by_xpath("//form[@id='loginForm']")
```

Figure 13. Usage of driver.find_element_by_xpath function

- *find_element_by_link_text*: can locate a hyperlink (element with the tag *a*) that has a certain value given as a parameter (*Figure 14*)

- *find_element_by_partial_link_text*: locates the hyperlink whose value has a substring equal with the value given as parameter (*Figure 14*)

`Continue`

```
driver.find_element_by_link_text('Continue')
driver.find_element_by_partial_link_text('Conti')
```

Figure 14. Usage of driver.find_element_by_link_text function

- *find_element_by_tag_name*: determines the first element with a certain tag (*Figure 15*)

`<h1>Welcome</h1>`

```
driver.find_element_by_tag_name('h1')
```

Figure 15. Usage of driver.find_element_by_tag_name function

- *find_element_by_class_name*: determines the first element with a certain value for the class attribute (*Figure 16*)
- *find_element_by_css_selector*: searches based on the syntax of the CSS selector given as parameter (*Figure 16*)

`<p class="content">Site content goes here.</p>`

```
driver.find_element_by_class_name('content')
driver.find_element_by_css_selector('p.content')
```

Figure 16. Usage of driver.find_element_by_class_name/css_selector function

It is important to note that these functions will only return the first element found that respects the given characteristics (or nothing if there's no such element). In order to find all of the elements, these functions will return a list containing them:

- *find_elements_by_name*
- *find_elements_by_xpath*

- *find_elements_by_link_text*
- *find_elements_by_partial_link_text*
- *find_elements_by_tag_name*
- *find_elements_by_class_name*
- *find_elements_by_css_selector*

(The *id* attribute is unique for each element in the HTML document)

Another useful feature of the library is the ability to execute javascript commands, using the method *driver.execute_script*. This enables the possibility of doing tasks that are not possible with the methods available in *Selenium*, such as scrolling, opening tabs or editing the HTML document.

All of the functionalities are applicable only to the current selected tab of the browser, that is indexed as an integer in the tab list, represented by *driver.window_handles*. The values stored in the array are the names of the active tabs, which can be switched to by using the available function *driver.switch_to_window("windowName")*.

The implementation of these commands is based on sending a request to the browser, given the type of the request, the data added by the user and a javascript path identifier recognized by the server (*Figure 17*).

```

commands = {
    Command.QUIT: ('DELETE', '/session/$sessionId'),
    Command.GET_WINDOW_HANDLES: ('GET',
'/session/$sessionId/window_handles'),
    Command.GET: ('POST', '/session/$sessionId/url'),
    Command.GO_BACK: ('POST', '/session/$sessionId/back'),
    Command.EXECUTE_SCRIPT: ('POST',
'/session/$sessionId/execute'),
    Command.FIND_ELEMENTS: ('POST',
'/session/$sessionId/elements'),
    Command.SEND_KEYS_TO_ELEMENT: ('POST',
'/session/$sessionId/element/$id/value'),
    Command.GET_ELEMENT_VALUE: ('GET',
'/session/$sessionId/element/$id/value'),
    Command.GET_ELEMENT_TAG_NAME: ('GET',
'/session/$sessionId/element/$id/name')
}

```

Figure 17. Browser path identifiers

Action Chains

A useful library added to *Selenium* is the implementation of the *ActionChains* class [22]. *ActionChains* allows for the automation of mouse movements, button presses and interaction with the context menus. It can be used for tasks such as hover on buttons to reveal context menus and interact with them, or drag and drop operations.

The implementation allows for storing multiple actions inside a queue in the *ActionChains* object, that can be performed together, in the order of storing. After adding the actions, the *perform* function will execute them in order using the same approach as the methods in the base *selenium.webdriver* module (Figure 18).

```

from selenium.webdriver.common.action_chains import ActionChains

actions = ActionChains(driver)
actions.move_to_element(menu)
actions.click(hidden_submenu)
actions.perform()

```

Figure 18. Usage of ActionChains

Pynput Module's Keyboard Listener & Controller

The module allows for listening the keyboard and mouse for when an action takes place, and detect which key has been pressed, but also to control them and automate the pressing of buttons.

In order to control or listen to the keyboard, the following packages need to be imported [23]:

- *pynput.keyboard.Key*
- *pynput.keyboard.Controller*
- *pynput.keyboard.Listener*

The *Key* package has the necessary mappings for all the keys on the keyboard. The *Controller* can be used by simply instantiating it and using the *press*, *release* and *pressed* functions to handle pressing, releasing and holding down keys (*Figure 19*).

```
from pynput.keyboard import Key, Controller

keyboard = Controller()

keyboard.press('A')
keyboard.release('A')
with keyboard.pressed(Key.shift):
    keyboard.press('a')
    keyboard.release('a')
```

Figure 19. Usage of the keyboard controller

The *Listener* is a thread that will record actions until it is released, by either calling the *listener.stop* function, or returning *False* from one of its callbacks. The callbacks are set through the *on_press* and *on_release* parameters of the listener. Both of the callback functions take as parameter an object which represents the key that has been pressed or released by the user (*Figure 20*).

```
from pynput.keyboard import Key, Listener

def on_press(key):
    print('alphanumeric key {0} pressed'.format(key.char))

def on_release(key):
    print('{0} released'.format(key))
    if key == Key.esc:
        return False # Stop listener

listener = Listener(on_press=on_press, on_release=on_release)
listener.start()
listener.join()
```

Figure 20. Usage of the keyboard listener

Chapter 3.

Application architecture

My application is focused towards handling the recurrent actions that a user performs on a specific web page, such a scrolling, clicking on buttons, selecting text, filling input fields or downloading content. To interact with the program the user needs to hold down the *RCtrl* key in order to input their command and then release it so the voice input can be recognized and transformed into text. Then, the text is processed by the application which will determine based on the similarity of it with the supported commands, what did the user want to happen, and then the function which launches the interaction with the browser on that specific command will be called and executed. If the command cannot be identified by the speech recognizer it will not be forwarded to the text analyzer and, similarly, if the text produced by the user's input doesn't resemble any of the supported commands, the user will be informed that they should try to repeat their request. Each request from the user is handled at a separate time and they will not be able to insert a new command until the previous one has finished its execution.

The complexity of the application depends on the function that the user requests at a certain time, many being executed in constant time such as pressing some button combinations on the keyboard or performing a javascript call, but others such as clicking on a button or scrolling the page require a linear time depending on the number of buttons that need to be searched on that page or the height that needs to be scrolled.

The purpose of my project is to create a program that can interact with the browser in a different way, that has not been popularized as of today and that, in my opinion, has the potential to be a more appealing way of using internet services to a certain group of users.

The application's functionality allows the user to input audio speech to the microphone, which is captured and converted to text using Google's Speech API, then handed over to the browser interactor which will send a request to the web based on the command that should be performed. The structure is divided in 2 principal items, the *Voice recognizer*, that handles the recording and recognition of the speech input, and the *Browser interactor*, that performs the interaction with the web driver.

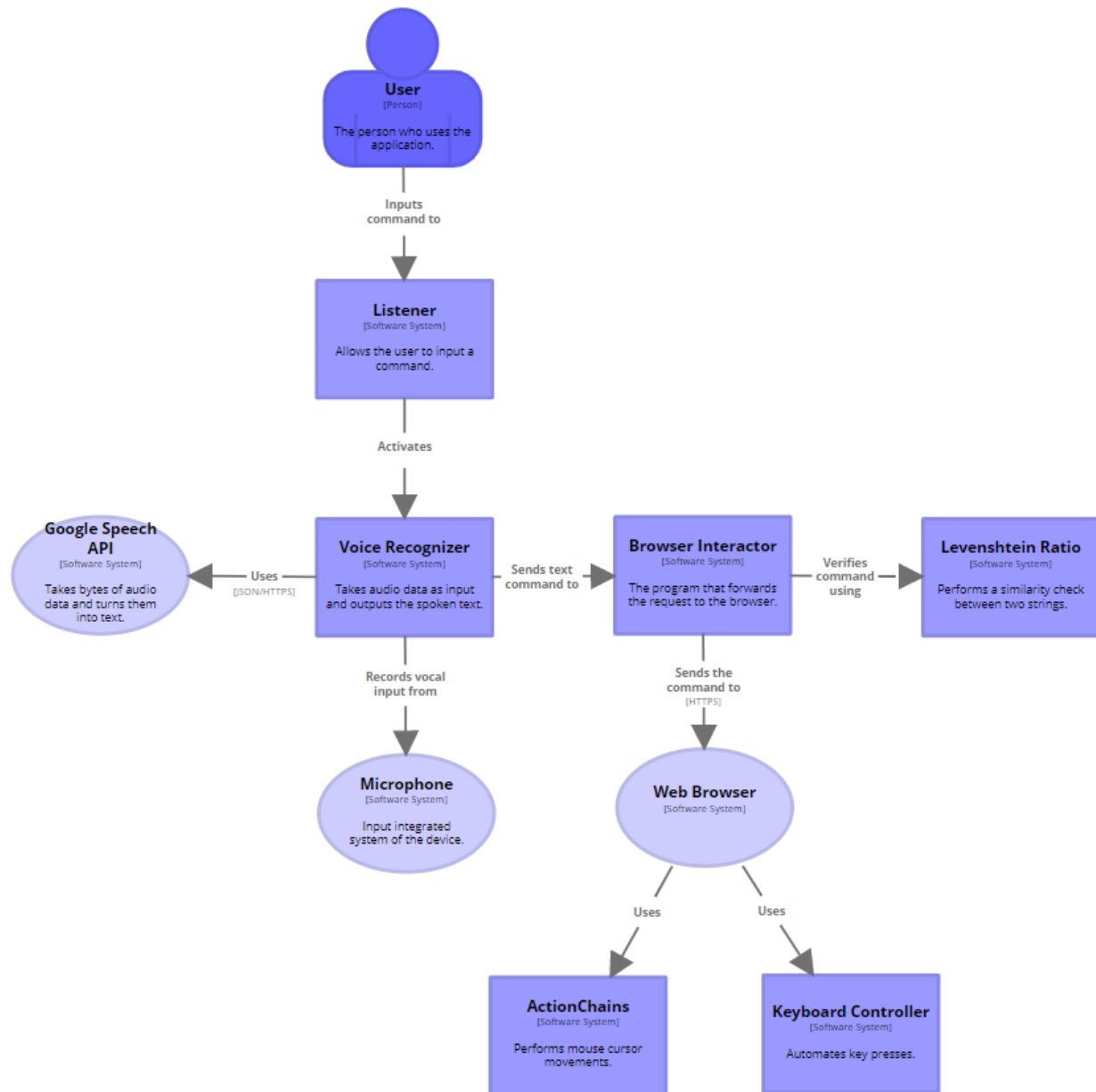


Figure 21. Architecture of the application

Application flow

On startup, the application will initialize a *Browser*, *Voice* and *Controller* objects, and start a loop which will call for a *Listener* object at each iteration in order to wait for the user to input a command. After the user does so, the program will determine that the command just given by the user has been recognized, started its execution and should not be executed again until the user inputs another command. Then, the recognized command will be compared with the commands supported by the application using Levenshtein's ratio, in order to bypass a certain degree of error between what the user meant to say and what the speech recognizer has guessed, and to determine which command should be performed, if any. If a command has been identified, it will be separated in two parts, one that represents the identifiers for that command and another that is the user's input to that command (if the command requires any). The *Browser* created object will then call the function which handles the command with the additional input, if necessary, and after the command has been performed, the application will continue looping and request a new command from the user. The program will stop when the user inputs the *exit* command, which will finish the interaction by exiting the main loop.

The list of functionalities supported by the application:

- Receive and handle voice input from the user
- Interact with the browser by sending requests
- Wait for the user to input a command through background listening
- Open a certain web page
- Use search input fields
- Click on buttons based on their text value
- Hover on buttons with a certain text value
- Press keys that perform certain actions (Page down, Page up, Enter, Escape, Caps Lock)
- Do smooth scrolling on the page at different speeds
- Handle opening, closing and switching of the tabs
- Refresh the page
- Go back or forward in history
- Add pages to bookmarks

- Save a page or an image
- View page source
- Find and select buttons on a page
- Find and complete input fields on a page
- Find and view images on a page
- Fill in and clear input fields
- Translate a page
- Find certain text on a page
- Perform undo and redo operations
- Select and paste text from the page
- View bookmarks and access their domain
- Stop the interaction

Voice recognizer

The purpose of this functionality is to initiate the recording of the microphone, to stop the recording, and to implement the callback which calls the speech recognizer on the obtained audio data.

```
class Voice:
    def __init__(self):
        self.command = ""
        self.executed = False
        self.stop_listening = None
```

Figure 22. Initializing the voice recognizer

On initialization (*Figure 22*), the object will store the text returned by the speech recognizer, an identifier that tells whether a command that has been computed has started execution or not, and a variable that will store the stop function of the background listener.


```

def listen(self):
    r = sr.Recognizer()
    m = sr.Microphone()
    print("Say something!")
    self.stop_listening = r.listen_in_background(m,
self.recognize)

def stop(self):
    self.stop_listening(wait_for_stop=False)

```

Figure 23. Recognizer's listen and stop functions

In order to easily call for the program to start or to stop recording audio input, the commands that perform these actions have been added to a *listen* and *stop* function (Figure 23).

The *listen* function will use the *Recognizer* and *Listener* from the *SpeechRecognition* module, with default settings, to continuously record audio input from the microphone and obtain the sequence of bytes that makes it up. The returned stop function that, when called, will stop recording the microphone and prompt the callback function to execute, will be stored by the object for a later use. The user is informed that the microphone is active with a message.

The *stop* function will call on the background listener's stop function by using the variable in which the method was stored previously. The *wait_for_stop* parameter is set to *False*, which means that the program will not wait for the listener thread to finish and execute the callback right away, in order to avoid unnecessary delays.

```

def recognize(self, r, audio):
    try:
        self.command = r.recognize_google(audio)
        self.executed = False
    except sr.UnknownValueError:
        self.command = "?!?"
    except sr.RequestError as e:
        self.command = "Could not request results from Google
Speech Recognition service; {0}".format(e)

```

Figure 24. The recognize method

The *recognize* function (Figure 24) is the callback set for when the background listener stops recording input from the microphone, and the information found so far should be transformed into text. The method takes as parameter the *Recognizer* object initialized before and the audio

data computed. The data is sent to the Google Speech API, which will attempt to output a JSON transcription with multiple possible outcomes and confidence levels and select the best possible option. The output will either be set to this “best guess” answer, or to an unidentified message if the API couldn’t understand what has been said. An intuitive message is also displayed in case the application cannot perform a connection with the API due to communication errors.

From the order of execution perspective, an object initialized from the *Voice* class will call on the *listen* function in order for the user to be able to input their command, and then on the *stop* method that will shut down the recording process and call on the recognizer. The program will then be able to read the identified text output from the *command* variable.

Browser interactor

This functionality is used to initiate the web driver and to implement the execution of the commands supported by the application. Each command is implemented in a different method and can access the application’s web driver that sends requests to the browser.

```
class Browser:
    def __init__(self):
        self.driver = webdriver.Chrome()
        self.current_tab = 0
        self.scroll_base_speed = 5
        self.scroll_update_speed = 5
        self.scroll_speed = self.scroll_base_speed
        self.button_map = dict()
        self.input_map = dict()
        self.selected_field = None
        self.caps_lock = False
```

Figure 25. Initialization of the browser object

Certain fields that need to be accessed by multiple methods are allocated memory by the program when the browser object is created. These are

- the web driver that interacts with the browser
- the index of the current selected tab
- the base speed of scrolling

- the value that the scrolling speed is updated with
- the current scrolling speed
- a dictionary that stores the buttons found on a page
- a dictionary that stores the input fields found on a page
- the currently selected input field
- the value of the Caps Lock key

The main functionality of the object is to be able to easily perform browser related actions without rewriting large portions of code. Each command given by the user is executed in a separate function, that the program will wait for to finish before recording a new command.

Keyboard listener

The *Listener* will be called when the user is supposed to enter a new command, that is, when the previous command has finished executing, or a command that is executing can be updated. It will handle the events in a separate thread and pause the execution until a command has been given. The program will start listening to the microphone the *on_press* method is called and stop when the *on_release* method executes. The audio data captured during that interval will be sent for recognition.

Following up, I will detail the actions that can be performed by the browser.

Open a page

```
def open(self, url):
    try:
        self.driver.get("http://" + url)
    except:
        print("Couldn't open " + url)
```

Figure 26. Browser's open method

The browser is prompted to go to a certain url given by its domain. If the domain is invalid, an error message will be displayed.

Exit the browser

```
def quit(self):  
    self.driver.quit()
```

Figure 27. Browser's quit method

The interaction between the user and the browser will stop. The user is not able to send any other command.

Search text by input field

```
def search(self, text):  
    try:  
        elem = self.driver.find_element_by_name("q")  
        elem.clear()  
        elem.send_keys(text)  
        elem.send_keys(Keys.RETURN)  
    except:  
        print("Couldn't search for " + text)
```

Figure 28. Browser's search method

It will find a search input field on the current page (usually mapped with the attribute `name="q"` in the HTML file), clear the text inside it (if there's any), write the given `text` parameter by pressing each key in order of the characters and then press the *Enter* key to submit.

Click a button with a certain value

```
def click(self, value):
    try:
        nr_tabs = len(self.driver.window_handles)
        buttons =
self.driver.find_elements_by_css_selector("button") + \
        self.driver.find_elements_by_css_selector("a")
        avg_value = None
        for button in buttons:
            if similar(value, button.text) is True:
                avg_value = button.text
                break
        if avg_value is None:
            for button in buttons:
                if value.lower() in button.text.lower():
                    avg_value = button.text
                    break
        if avg_value is None:
            raise Exception
        self.driver.find_element_by_link_text(avg_value).click()
        if len(self.driver.window_handles) != nr_tabs:
            self.switch_tab()
    except:
        print("Couldn't find button " + value)
```

Figure 29. Browser's click method

A function to click on the first button found that has its value similar with the given parameter. If no such button is found, the method will try to find a button that at least has the given parameter as a substring to its value. If no button is found, an error message is displayed.

The function will identify all of the clickable items on the page (marked with the tag *button* or *a* for hyperlinks), and then, checks for similarity between the value of each button and the given parameter. If none of the values are similar, it searches again for a button that only contains the given parameter in its name. An exception printing an error message is raised, if no button contains the parameter value, else the method will click on the button that has been identified. The method also checks whether clicking on the button opens a new tab, by counting the number of tabs before performing the click operation and then recounting it after. If a new tab has been opened, it will be marked as the current active tab.

```
def similar(s1, s2):
    s1 = re.sub("[^A-Za-z]", "", s1).lower()
    s2 = re.sub("[^A-Za-z]", "", s2).lower()
    if s1 == s2 or Levenshtein.ratio(s1, s2) > 0.9:
        return True
    return False
```

Figure 30. Determining similarity between two strings

The similarity between the value of a button and the given parameter is computed as follows: removes all the nonalphabetic characters from the two strings (using regular expressions) and makes all of their characters lowercase. They are compared based on the computed Levenshtein ratio between them, to determine if they are close to each other or not (*Figure 30*).

Hover on a button with a certain value

```
elem = self.driver.find_element_by_link_text(avg_value)
action = ActionChains(self.driver).move_to_element(elem)
action.perform()
```

Figure 31. Performing a mouse hover using ActionChains

This method will move the mouse cursor to the first button found that has the value similar or contains the given parameter. The implementation is identical to that of the *click* function, except instead of clicking on the found button, it will use the *ActionChains* library to initiate an action which will move the cursor to an HTML element. One possible use for this function is opening a submenu of a button, which is done by hovering on it.

Page down, Page up

```
def page_down(self):  
    keyboard.press(Key.page_down)  
    keyboard.release(Key.page_down)  
  
def page_up(self):  
    keyboard.press(Key.page_up)  
    keyboard.release(Key.page_up)
```

Figure 32. Browser's page down/up implementation

Using a keyboard controller, it simulates the pressing of the two keys.

Smooth scrolling

This method will perform a smooth scrolling of the current page, by using javascript to scroll to a given height.

```
last_height = self.driver.execute_script("return  
document.body.scrollHeight")  
sc_height = self.driver.execute_script("return  
window.pageYOffset")
```

Figure 33. Computing the total and current page height

The total height of the page and the current height at which the display is on are necessary in order to decide when the scrolling automatically stops (*Figure 33*).

```
while True:  
    while sc_height < last_height:  
        with Listener(on_press=on_press, on_release=on_release)  
as listener:  
        time.sleep(0.03)  
        if not listener.running:  
            break
```

Figure 34. Adding a listener to support commands while scrolling

The current scrolled height is increased until it reaches the maximum of the page. In order for the user to be able to stop the scrolling before it reaches the end, they are able to input a

command that does just that (Figure 34, 35). If no command has been detected, the loop will continue to scroll.

```
if similar("stop", voice.command) and voice.executed is False:
    voice.executed = True
    self.scroll_speed = self.scroll_base_speed
    return
elif similar("go faster", voice.command) and voice.executed is False:
    voice.executed = True
    self.scroll_speed += self.scroll_update_speed
elif similar("slow down", voice.command) and voice.executed is False:
    voice.executed = True
    if self.scroll_speed > self.scroll_base_speed:
        self.scroll_speed -= self.scroll_update_speed
```

Figure 35. Commands supported while scrolling

Else, if the user wishes for a faster or a slower scrolling, the speed can be increased through commands and will be reset to its initial value when the scrolling stops.

```
self.driver.execute_script("window.scrollTo(0, %s)" % sc_height)
sc_height += self.scroll_speed
last_height = self.driver.execute_script("return
document.body.scrollHeight")
```

Figure 36. Performing the scroll operation

Javascript is be used to request the scrolling to be performed by the browser, then the new current height will be updated and also the total height (Figure 36).

```
while True:
    while sc_height < last_height:
        # ...
        new_height = self.driver.execute_script("return
document.body.scrollHeight")
        if new_height == last_height:
            self.scroll_speed = self.scroll_base_speed
            return
        last_height = new_height
```

Figure 37. Updating the total height of the page

When the current height reaches the height of the page, the program will recompute the total height of the page. If the new height is equal with the last height, it means that the scrolling is complete, the speed should reset to the initial value and the method should return. Otherwise, it means that the page has dynamically generated new content as the user was scrolling it (or when the scrolling was finished), and the method should continue to scroll the page by setting the new value for the new total height, until the user chooses to stop (*Figure 37*).

One possible use case for this feature is to automate scrolling for a social media web site, that generates new content continuously as the user reaches the end of the current content.

Scrolling up on the page is implemented similarly to scrolling down. The user is still able to stop the scrolling or change its speed and there is no additional checking for whether new content can be generated since it's not the case. The scrolling will stop when it reaches height 0, if the user doesn't end it before.

Open a new tab

```
def new_tab(self):  
    self.driver.execute_script("window.open('');")  
    self.current_tab = len(self.driver.window_handles) - 1  
    self.driver.switch_to.window(self.driver.window_handles[  
self.current_tab])
```

Figure 38. Implementation of the new tab method

The function will open a new tab by using a javascript command, will save the index to the newly opened tab and will mark the tab as the current one.

Switch tab

```
def switch_tab(self):  
    self.current_tab = (self.current_tab + 1) %  
len(self.driver.window_handles)  
    self.driver.switch_to.window(self.driver.window_handles[  
self.current_tab])
```

Figure 39. Implementation of the switch tab method

To move between the opened tabs, this function will switch to the next one in the list. If the last tab is the current one, the method will perform a modulo operation to return to the first in list.

Close tab

```
def close_tab(self):  
    self.driver.close()  
    try:  
        self.switch_tab()  
    except:  
        global browser  
        browser.quit()
```

Figure 40. Implementation of the close tab method

Closes the current opened tab and switch to the next one. If only one tab is active, the browser will close.

Refresh

```
def refresh(self):  
    self.driver.refresh()
```

Figure 41. Implementation of the refresh method

Will refresh the current opened page.

Go back, Go forward

```
def back(self):  
    self.driver.execute_script("window.history.go(-1)")  
  
def forward(self):  
    self.driver.execute_script("window.history.go(1)")
```

Figure 42. Going back or forward in browser history methods

Will go either back or forward in the browser's history, using javascript. If there's no previous or next page, nothing will happen.

Bookmark

```
def bookmark(self):  
    with keyboard.pressed(Key.ctrl):  
        keyboard.press('d')  
        keyboard.release('d')  
    time.sleep(0.01)  
    keyboard.press(Key.esc)  
    keyboard.release(Key.esc)
```

Figure 43. Method to bookmark the current page

Will add the current page to bookmarks, with the shortcut *Ctrl+D*, using the keyboard controller. The *Escape* key is then needed to exit the Bookmarks dialog.

Remove bookmark

```
def remove_bookmark(self):  
    with keyboard.pressed(Key.ctrl):  
        keyboard.press('d')  
        keyboard.release('d')  
        time.sleep(0.01)  
    for index in range(0, 4):  
        keyboard.press(Key.tab)  
        keyboard.release(Key.tab)  
    keyboard.press(Key.enter)  
    keyboard.release(Key.enter)
```

Figure 44. Removing the bookmark of the current page

Will remove the current page from the bookmarks, if it has been added there previously. Using *Ctrl+D*, the Bookmarks dialog (*Figure 45*) is open, then pressing *Tab* 4 times will select the remove button which is selected.

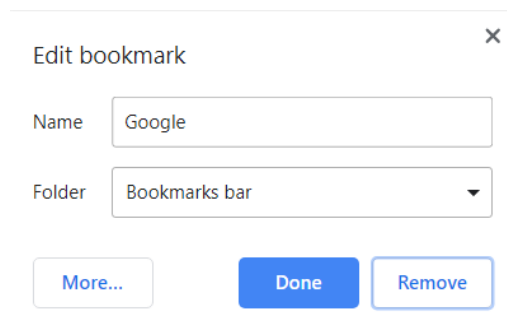


Figure 45. Bookmarks editor

Save page/image

```
def save(self):  
    with keyboard.pressed(Key.ctrl):  
        keyboard.press('s')  
        keyboard.release('s')
```

Figure 46. Browser's save method

Will perform the save operation, using the *Ctrl+S* shortcut.

View page source

```
def source(self):  
    with keyboard.pressed(Key.ctrl):  
        keyboard.press('u')  
        keyboard.release('u')  
    time.sleep(0.01)  
    self.switch_tab()
```

Figure 47. Browser's view source method

Views the source code for the current page, using the *Ctrl+U* shortcut. Since the source is opened in a new tab, it must also be marked as the current tab.

Submit, Cancel

```
def submit(self):  
    keyboard.press(Key.enter)  
    keyboard.release(Key.enter)  
  
def cancel(self):  
    keyboard.press(Key.esc)  
    keyboard.release(Key.esc)
```

Figure 48. Browser's submit and cancel methods

Simulate the pressing of the *Enter* or *Escape* keys.

Find buttons on page

This functionality will map all the buttons found on a page (or those with the value containing the given parameter) to a number and store them in a dictionary. That way, the user will be able to choose which button he wishes to click by specifying the index at which that button is mapped.

```
self.button_map = dict()
buttons = self.driver.find_elements_by_css_selector("button") +
          self.driver.find_elements_by_css_selector("a")
```

Figure 49. Finding the clickable elements on page

All the buttons on the page are identified by searching for the *button* and *a* tagged elements (Figure 49).

```
for button in buttons:
    if value.lower() in button.text.lower():
        self.button_map[str(len(self.button_map.keys()) + 1)] =
button
```

Figure 50. Mapping buttons to a dictionary

The dictionary (Figure 50) contains as keys the indexes at which the buttons are stored, and as values the objects that represent the clickable elements on page.

```
for index in self.button_map:
    self.driver.execute_script('''
        var node = document.createElement("div");
        var textnode = document.createTextNode("{}");
        node.appendChild(textnode);
        node.setAttribute('style',
'position:relative;left:10px;display:table-cell;border-
radius:20px;font-size:1.5em;background-color:red;color:white');
        arguments[0].appendChild(node);
        '''.format(index), self.button_map[index])
```

Figure 51. Adding visible identifiers to the buttons

The HTML file of the page is edited (Figure 51) so that each of the selected buttons can be visible, by creating a new node under each element that contains the index of it.

Choose button

```
if value in self.button_map:
    nr_tabs = len(self.driver.window_handles)
    self.button_map[value].click()
    self.button_map = dict()
    if len(self.driver.window_handles) != nr_tabs:
        self.switch_tab()
```

Figure 52. Choose a button from the mapped dictionary

Will choose the button from the list mapped previously (taking the index as a parameter) and click it. If a new tab opens, it will be selected. Also, if the index is not valid or the dictionary is empty, an error message will be shown. The dictionary is emptied if the operation is successful so that the program will not try to click on the same button from another page.

Finding and selecting images

It is done similarly to determining the buttons present on the page, except that the mapped fields are tagged with the identifier *img*, and selecting them will open the source of the image.

Find input fields

The function will map all the input fields on a page in a dictionary, similarly to the buttons, in order for the user to be able to select a field which they wish to complete.

```
self.input_map = dict()
fields = self.driver.find_elements_by_xpath('//input[not(@type="submit") ]')
for field in fields:
    self.input_map[str(len(self.input_map.keys()) + 1)] = field
```

Figure 53. Map input fields to a dictionary

The input fields added to the map (Figure 53) are only those that do not have a *type="submit"* attribute, which have the purpose to submit the completed forms.

```

for index in self.input_map:
    self.driver.execute_script("arguments[0].setAttribute(
'type', 'text')", self.input_map[index])
    self.driver.execute_script("arguments[0].setAttribute(
'style','color:white;background-color:red')",
                                self.input_map[index])
    self.driver.execute_script("arguments[0].setAttribute(
'value','{}')".format(index), self.input_map[index])

```

Figure 54. Visually identifying input fields

Apart from changing the background color, the text of the field is marked with the index at which it is mapped, and the type attribute is set to text so the user is able to see possible errors in writing passwords.

Select input field

```

if value in self.input_map:
    self.selected_field = self.input_map[value]

```

Figure 55. Selecting an input field from the dictionary

Selects the input field at the index given as a parameter by saving it in the object, so it can be identified when the user wishes to type or clear text.

Type text

```
def type_text(self, text):
    if self.caps_lock is True:
        text = text.upper()
    else:
        text = text.lower()
        fields = self.driver.find_elements_by_xpath('//input[not(
@type="submit") ]')
        if self.selected_field in fields:
            if text.lower() in keys:
                self.selected_field.send_keys(keys[text.lower()])
            else:
                self.selected_field.send_keys(text.replace(' ', ''))
```

Figure 56. Browser's type text method

Will type the given text parameter to the selected input field. It checks whether *Caps Lock* is enabled, then verifies that the current page is the same one where the input field was selected. If the given text is a key defined word, such as *space*, *backslash* or *comma*, it will write the mapped character. In order to write the words that make up the defined names, the user would have to spell them by characters.

Clear text

```
def clear_text(self, positions=None):
    fields = self.driver.find_elements_by_xpath('//input[not(
@type="submit") ]')
    if self.selected_field in fields:
        if positions is None:
            self.selected_field.clear()
        else:
            while positions != 0:
                positions -= 1
                keyboard.press(Key.backspace)
                keyboard.release(Key.backspace)
```

Figure 57. Browser's clear text method

Will clear all the characters from a selected input field, or only the last few characters. Nothing will happen if the selected field is not on the current page.

Change caps

Verifies the value of the setter, that should either be *on* or *off*, and activates or deactivates the *Caps Lock* key accordingly.

Undo, Redo

```
def undo(self):
    with keyboard.pressed(Key.ctrl):
        keyboard.press('z')
        keyboard.release('z')

def redo(self):
    with keyboard.pressed(Key.ctrl):
        with keyboard.pressed(Key.shift):
            keyboard.press('z')
            keyboard.release('z')
```

Figure 58. Browser's undo and redo methods

Uses the keyboard shortcuts *Ctrl+Z* and *Ctrl+Shift+Z* to perform the operations.

Find text in page

```
def look_for(self, text):
    with keyboard.pressed(Key.ctrl):
        keyboard.press('f')
        keyboard.release('f')
    time.sleep(0.01)
    for ch in text.lower():
        keyboard.press(ch)
        keyboard.release(ch)
```

Figure 59. Method to search for text on a page

Uses the *Ctrl+F* shortcut to open the page finder and inputs the given parameter text.

Translate

Translates the current page in English by using Google's Translate API and the extension that enables it. The extension is accessible via *Ctrl+Q* shortcut and the translation can be performed with keyboard commands.

Copy text from the page

```
def select_text(self, text):
    elems = self.driver.find_elements_by_css_selector("*")
    for elem in elems:
        if text.lower() in elem.text.lower() and
len(elem.find_elements_by_css_selector(":not(a):not(b):not(i)"))
== 0:
            self.copied_text = elem.text
            self.driver.execute_script("arguments[0].
setAttribute('style','color:white;background-color:red')",elem)
            return
```

Figure 60. Selecting text from the current page

Will browse all of the elements on the HTML document until it reaches the one that contains the given parameter and doesn't contain any other element except for hyperlinks, bold or italic text. The paragraph will be saved in the browser object and highlighted.

Paste copied text

A method to paste the text copied previously to a selected input field. The keyboard will input each of the characters in the text. If no input field is selected, nothing will happen.

View and select bookmarks

Will use the shortcut *Ctrl+Shift+O* to access the saved bookmarks page. To select one, the HTML document will be parsed in order to find the requested title and its domain.

Chapter 4.

Conclusions

The application has, in my opinion, a great usability for people that find it easier to perform recurrent tasks in browsing the web, using voice commands, than typing or scrolling throughout a page. It also has a great potential to perform automation of different browser commands through recording audio commands and using them as input to the program.

Although it manages to contain some of the very used functionalities that users need when browsing the web, there are many other possible ideas that can be achieved in the future using voice commands to interact with the browser. For example, the ability to easily perform mouse movements throughout the display would make not only the browser, but many other applications on a computer, be very accessible towards vocal input. Also, the ability to perform real time speech recognition very fast would make browsing the web probably faster than using peripherals. More interestingly, would be the ability to completely customize your browsing experience and interaction, through defining your own commands and the functionalities that the command should perform. This could lead to easily achieve custom tasks that a user performs on a regular basis.

References

- [1] The size of the World Wide Web, <https://www.worldwidewebsize.com/>, as of June 2019
- [2] Internet Live Stats - Total number of Websites, <https://www.internetlivestats.com/total-number-of-websites/>, as of June 2019
- [3] Internet World Stats, <https://www.internetworldstats.com/stats.htm>, as of June 2019
- [4] Lifewire – “The Top 10 Most Popular Sites of 2019”, <https://www.lifewire.com/most-popular-sites-3483140>, as of June 2019
- [5] Steve's Smart Home Guide – “Internet Connection and Access Methods”, <https://steve-smarthomeguide.com/connect-methods/>, as of June 2019
- [6] Internet Society – “Internet Accessibility: Internet use by persons with disabilities: Moving Forward “, <https://www.internetsociety.org/resources/doc/2012/internet-accessibility-internet-use-by-persons-with-disabilities-moving-forward/>, as of June 2019
- [7] Retail Dive – “Google Assistant to be available on 1B devices”, <https://www.retaildive.com/news/google-assistant-to-be-available-on-1b-devices/545604/>, as of June 2019
- [8] Juniper Research – “Digital voice assistants in use to triple to 8 billion by 2023, driven by smart home devices”, <https://www.juniperresearch.com/press/press-releases/digital-voice-assistants-in-use-to-triple>, as of June 2019
- [9] Wikipedia - Usage share of web browsers, https://en.wikipedia.org/wiki/Usage_share_of_web_browsers, as of June 2019
- [10] Pocket-lint – “What is Google Assistant and what can it do?”, <https://www.pocket-lint.com/apps/news/google/137722-what-is-google-assistant-how-does-it-work-and-which-devices-offer-it>, as of June 2019
- [11] Pocket-lint – “What is Siri and how does Siri work?”, <https://www.pocket-lint.com/apps/news/apple/112346-what-is-siri-apple-s-personal-voice-assistant-explained>, as of June 2019

- [12] Wikipedia – Cortana, <https://en.wikipedia.org/wiki/Cortana>, as of June 2019
- [13] Wikipedia – Speech recognition, https://en.wikipedia.org/wiki/Speech_recognition, as of June 2019
- [14] Rabiner, Lawrence R., and Biing-Hwang Juang. "An introduction to hidden Markov models." *ieee assp magazine* 3.1 (1986): 4-16.
- [15] Wikipedia – World Wide Web, https://en.wikipedia.org/wiki/World_Wide_Web, as of June 2019
- [16] Wikipedia – HTML, <https://en.wikipedia.org/wiki/HTML>, as of June 2019
- [17] *SpeechRecognition* Python module, <https://pypi.org/project/SpeechRecognition/>, as of June 2019
- [18] *SpeechRecognition* library reference, https://github.com/Uberi/speech_recognition/blob/master/reference/library-reference.rst, as of June 2019
- [19] Kėpuska, Veton, and Gamal Bohouta. "Comparing speech recognition systems (Microsoft API, Google API and CMU Sphinx)." *Int. J. Eng. Res. Appl* 7.03 (2017): 20-24.
- [20] Wikipedia – Levenshtein distance, https://en.wikipedia.org/wiki/Levenshtein_distance, as of June 2019
- [21] Lawson, Richard. *Web scraping with Python*. Packt Publishing Ltd, 2015.
- [22] *Selenium* Python module, <https://selenium-python.readthedocs.io/>, as of June 2019
- [23] *Pynput* Python module, <https://pynput.readthedocs.io/en/latest/>, as of June 2019

Annex

Guide to using the application:

Hold down the *RCtrl* key and input any of the following commands:

- *Go to <Url>*
- *Search <Text>*
- *Click on <ButtonValue>*
- *Hover on <ButtonValue>*
- *Page down*
- *Page up*
- *Scroll down*
- *Scroll up*
- *Open new tab*
- *Switch tab*
- *Close tab*
- *Refresh*
- *Go back*
- *Go forward*
- *Bookmark*
- *Remove bookmark*
- *Save*
- *View page source*
- *Submit*
- *Cancel*
- *Find the button <ButtonValue>*
- *Find all buttons*
- *Choose button number <Value>*
- *Find input fields*
- *Select input field <Value>*

- *Find images*
- *Select image* <Value>
- *Type text* <Text>
- *Clear* (<Positions>)
- *Caps* <Setter>
- *Translate*
- *Look for* <Text>
- *Undo*
- *Redo*
- *Select text* <Text>
- *Paste text*
- *View bookmarks*
- *Select bookmark* <Value>
- *Exit*

<Url> - the name of the domain to visit

<Text> - the text that needs to be written or found on a page

<ButtonValue> - the name of a button as seen on page

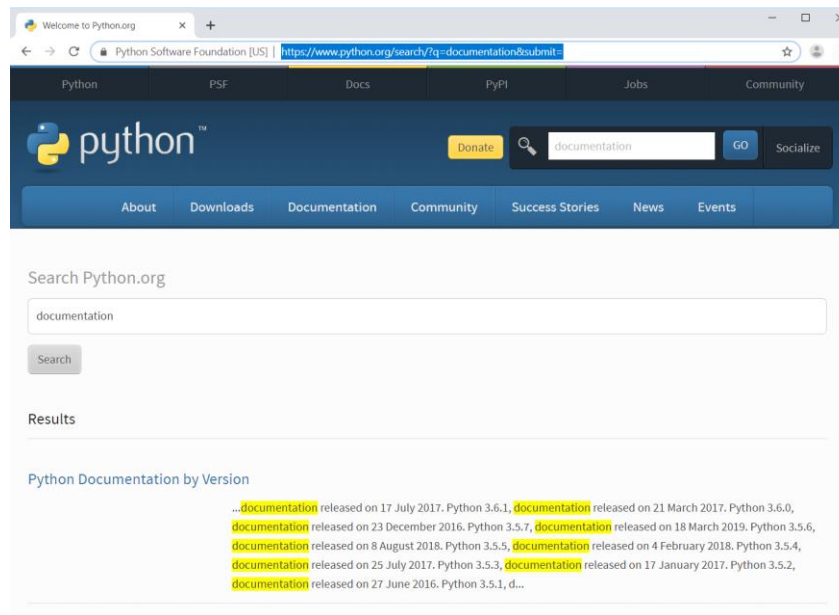
<Value> - a number that represents a mapping of an element

<Positions> - a number that represents the number of positions to be cleared

Possible use cases:

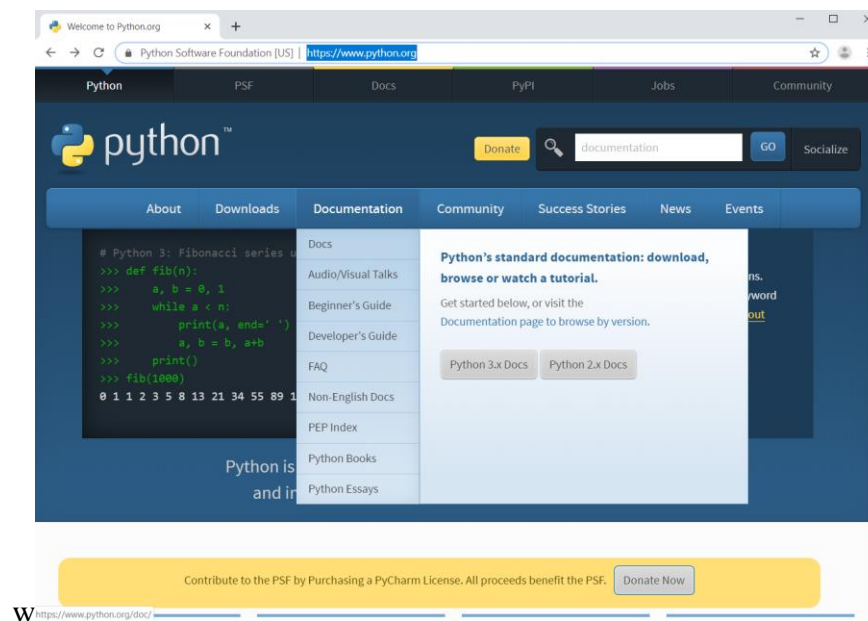
- Use the search input field to browse the web site for data

Command: *search documentation*



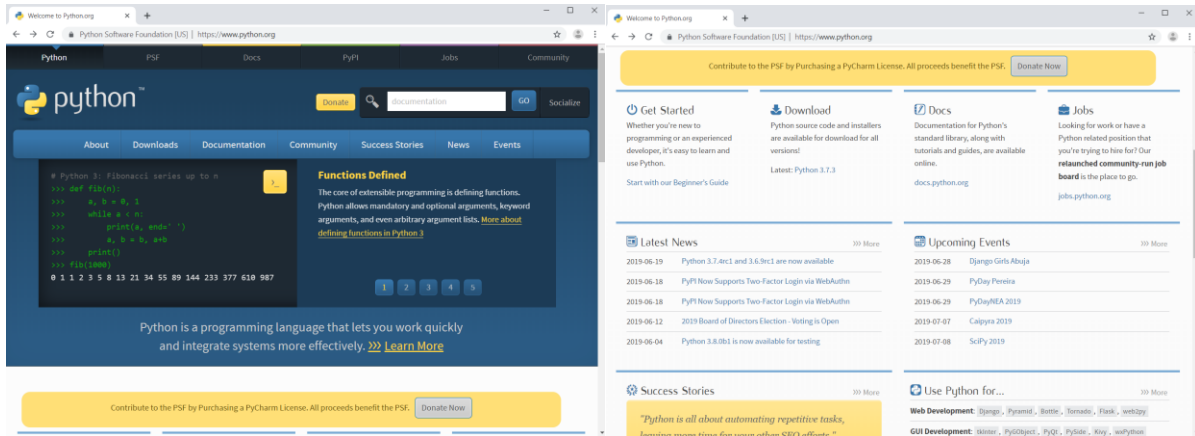
- Hover on a button identified by its text value

Command: *hover on documentation*



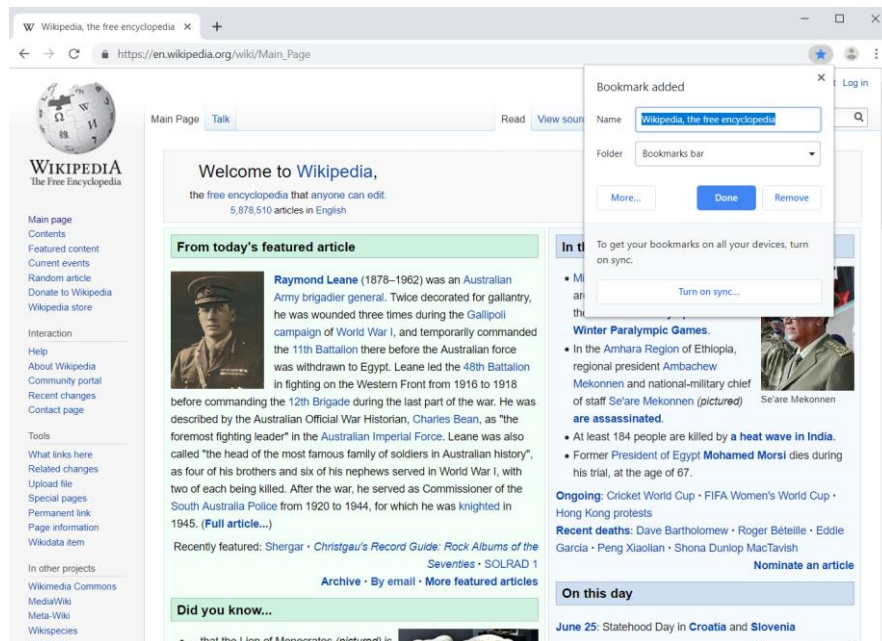
- Simulate the pressing of *Page Down* button

Command: *page down*



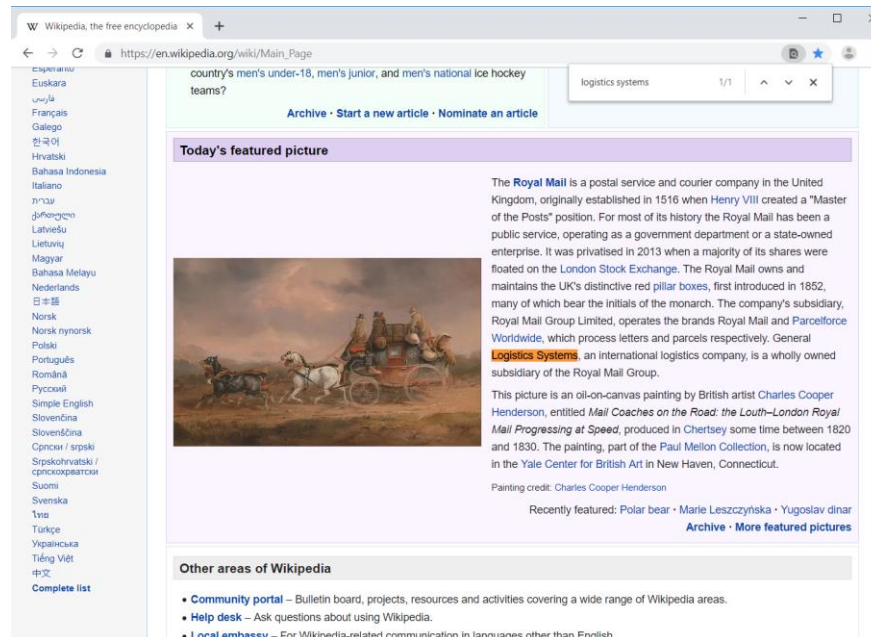
- Add the current page to bookmarks

Command: *bookmark*



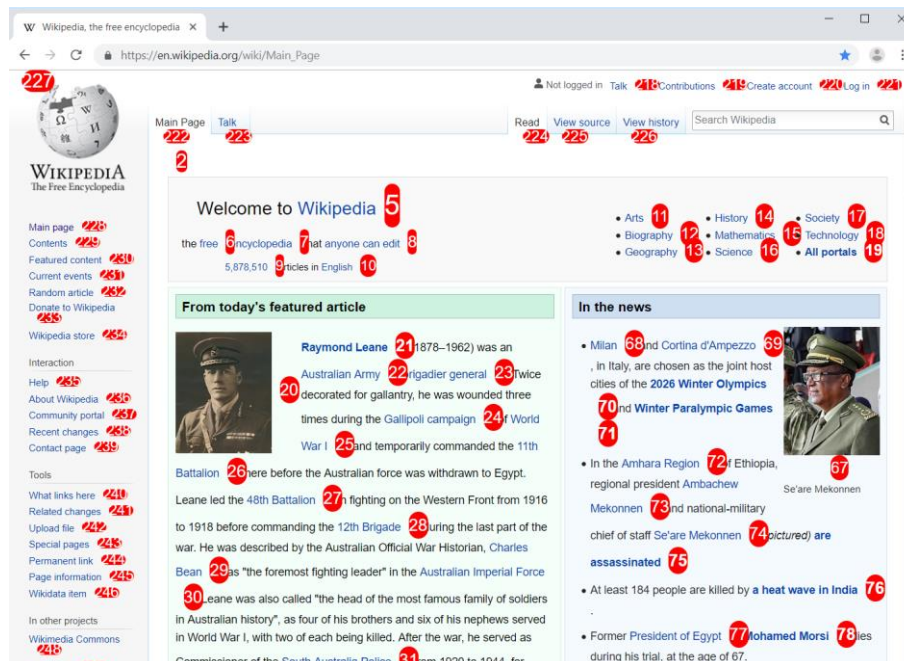
- Find text on the current page

Command: *look for logistics systems*



- Click on a certain button on the current page

Command: *find all buttons*

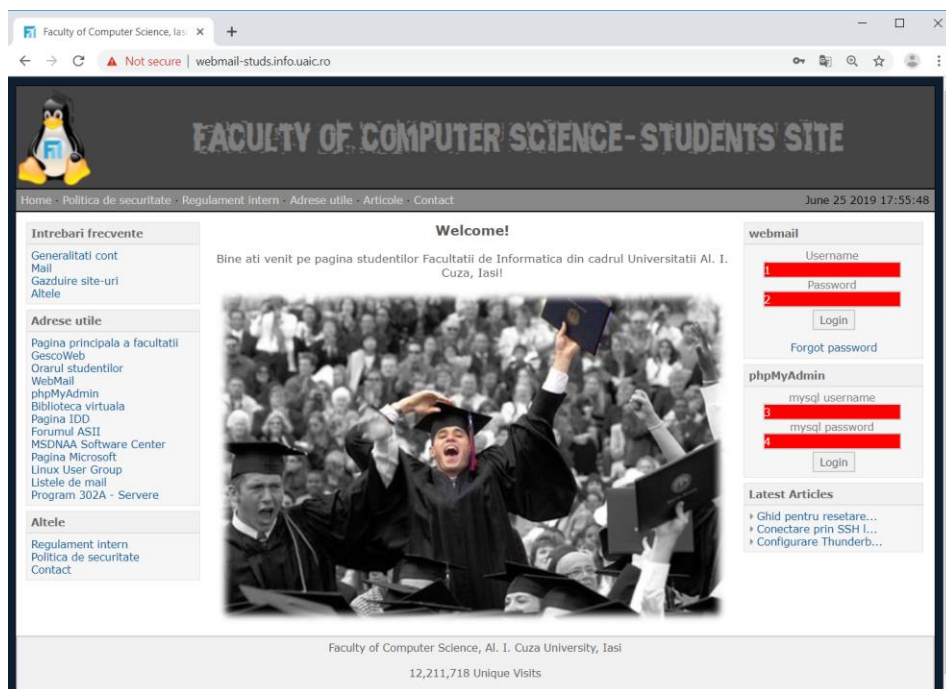


Command: *choose button number 21*

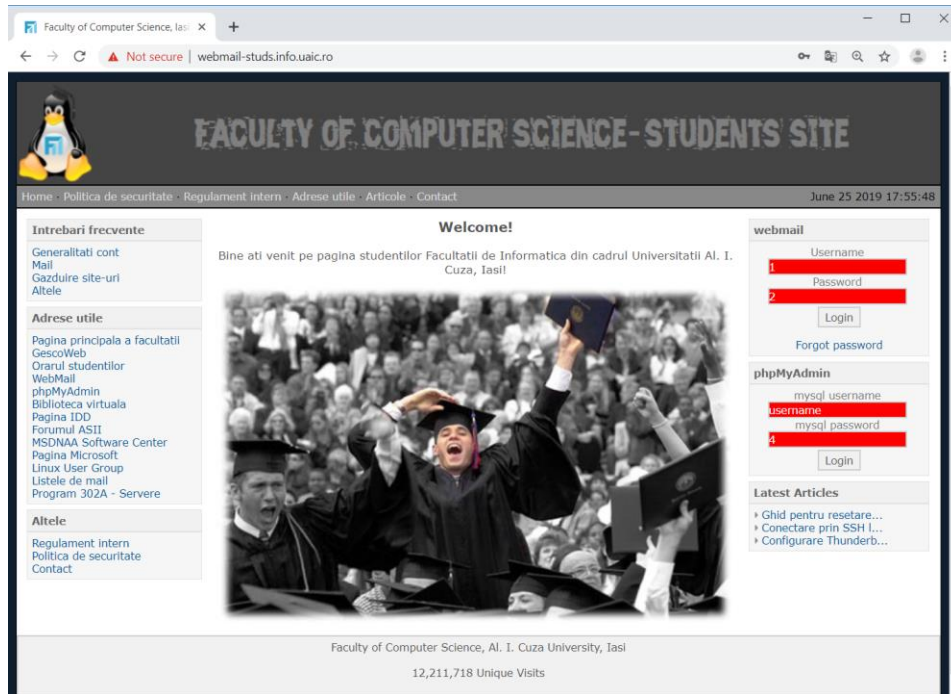


- Complete and submit input fields

Command: *find input fields*

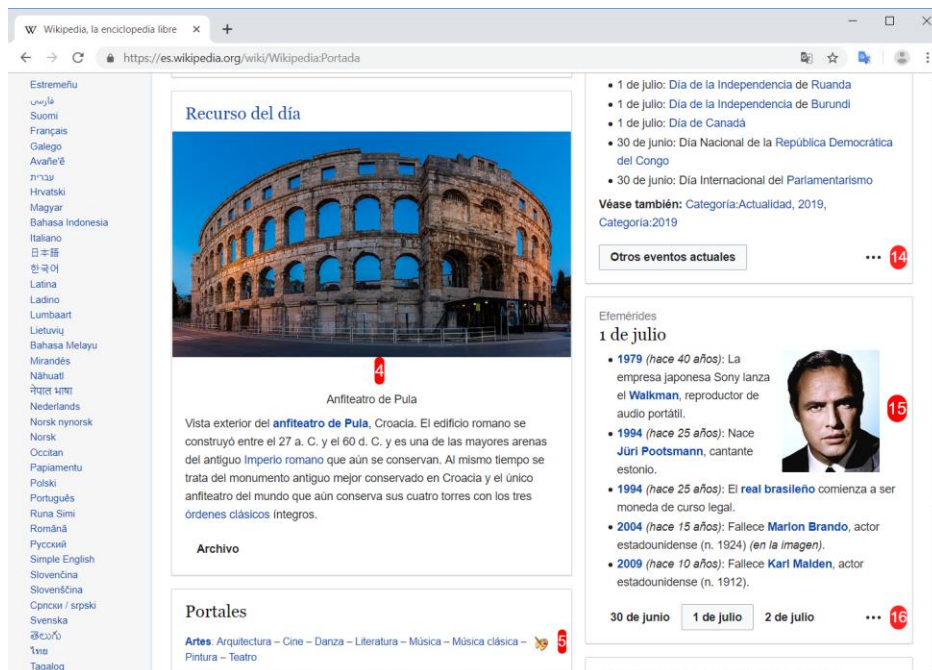


Commands: *select input field 3 / type username*



- Find all the images on the current page

Command: *find images*



- Translate the current page

Command: *translate*


Wikipedia, the free encyclopedia

https://es.wikipedia.org/wiki/Wikipedia:Portada

Translated to: English Show original

Options

Resource of the day



Pula Amphitheater

Exterior view of the **amphitheater of Pula**, Croatia. The Roman building was built between 27 a. C. and 60 d. C. and is one of the largest arenas of the ancient Roman Empire that are still preserved. At the same time it is the best preserved ancient monument in Croatia and the only amphitheater in the world that still retains its four towers with the three classical orders intact.

Archive

Portales

Arts - Architecture - Cinema - Dance - Literature - Music - Classical music - Painting - Theater

Commemorations and parties

- 1 July: Independence Day of Rwanda
- 1 July: Independence Day in Burundi
- July 1: Canada Day
- June 30: National Day of the Democratic Republic of the Congo
- June 30: International Day of Parliamentarism

See also: Category: Actualidad , 2019 , Category: 2019

Other current events

Ephemerides

1st of July

- 1979 (40 years ago) : The Japanese company Sony launches the **Walkman** , portable audio player.
- 1994 (25 years ago) : **Jüri Pootsmann** , Estonian singer born.
- 1994 (25 years ago) : The **Brazilian real** begins to be legal tender.
- 2004 (15 years ago) : **Marlon Brando** , American actor (born 1924) (*In the image*) dies.
- 2009 (10 years ago) : **Karl Malden** , American actor (born 1912) dies.

June 30th 1st of July July 2nd