Selenium Advanced

Ilya Antonov

September 6, 2019

Contents

SECTION 1

MOTIVATION

The **selenium bot** automates the extraction of content (scraping) from emails and skype conversation, by recreating the process of scrolling-clicking-copying-pasting conversations - something that would otherwise have to be done by an obedient employee.

Messages from **Outlook** and **Skype** are formatted into a tabular form, as shown in Fig. 1. This document will cover the essentials of building a python program to accomplish this task.

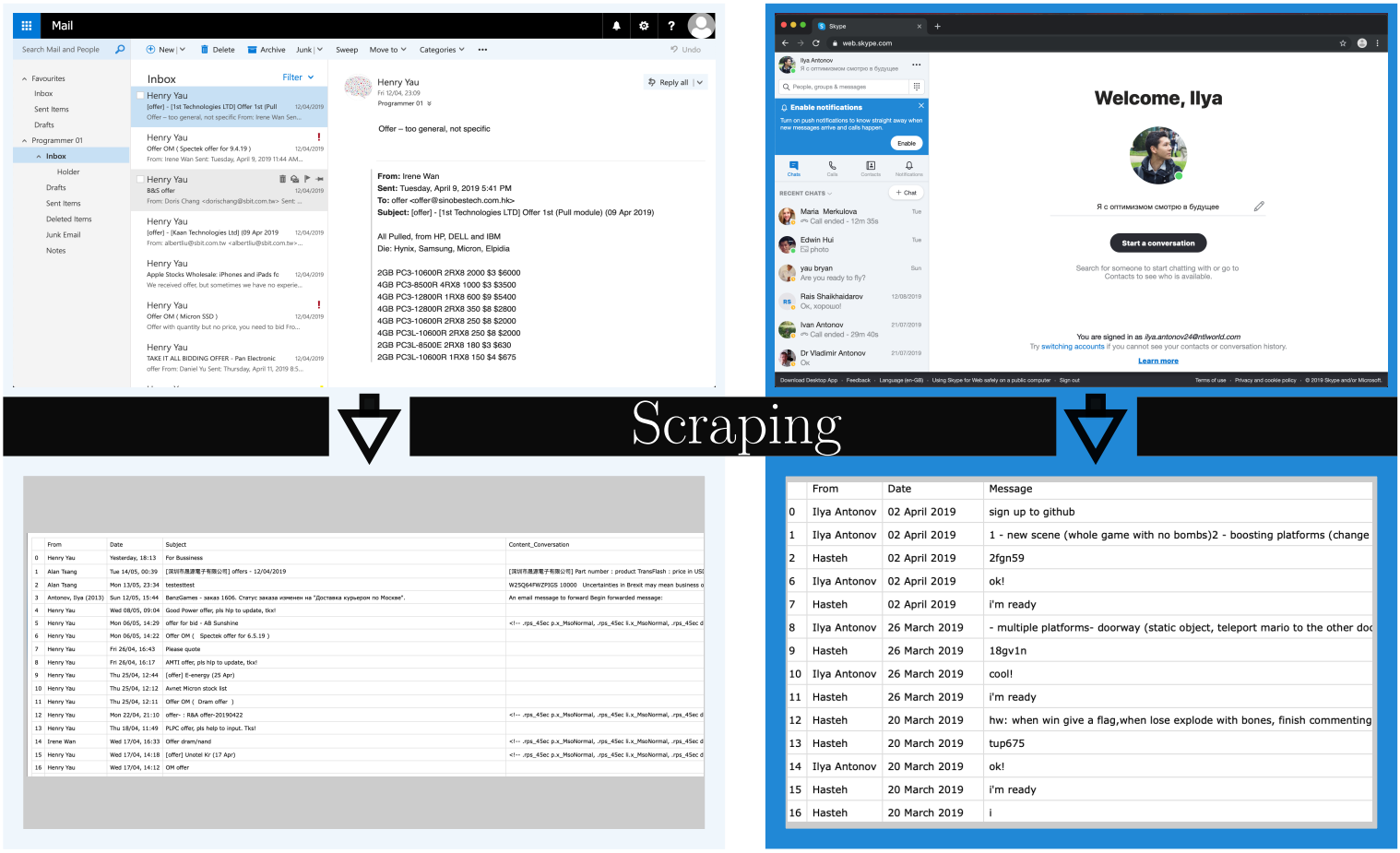


Figure 1: Extraction of messages and conversations into tables by the **selenium** bot.

Before starting, visit Appendix B to fulfill all of the dependencies of such a project.

SECTION 2

CLASS STRUCTURE

The class structure of the program is summarized in Fig.2. All common methods, such a loading a webpage, and finding "blocky" elements resides in the **Selenium class**, leaving the elements unique for different websites (buttons, specific forms), to be handled by the **Skype** and **Outlook** classes.

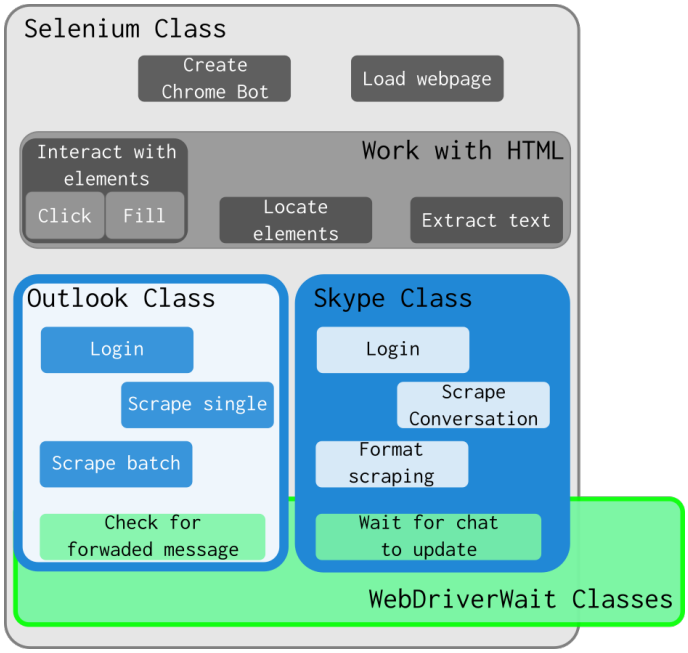


Figure 2: The Skype and Outlook bots are based off a generic **Selenium** class, that handles common htlm-parsing routines.

The basic steps for scraping a webpage are:

1. Launch bot and load webpage **Selenium Class**;
2. Login **Skype/Outlook Class**;
   - ↻ Find elements on webpage to interact/extract **Skype/Outlook Class**;
   - ↻ Perform interaction/extraction **Selenium Class**.

The upcoming sections will identify the key methods of each class.

SECTION 3

SELENIUM CLASS

Generic operations, irrespective if the bot is running on **Skype** of **Outlook**, are defined in the **Selenium** class.

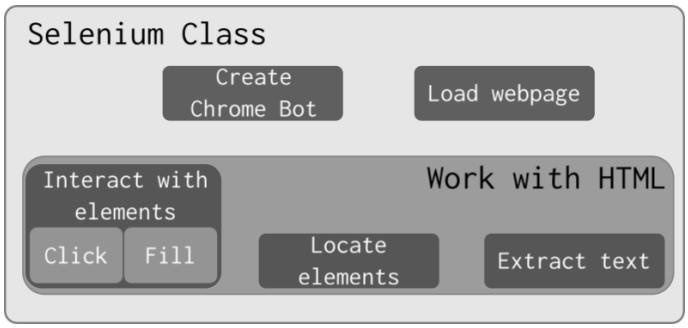


Figure 3: Class that handles generic functionality.

3.1 Creating a running bot

The bot should always be created with the following call, which will open up a Chrome **driver**.

```
# 1 - set options and capabilities for Chrome
capabilities = {'chromeOptions':
                        {
                            'useAutomationExtension': False,
                            'args': ['--disable-extensions']}
                        }

chrome_options = Options()
chrome_options.add_experimental_option("prefs", {
    "download.prompt_for_download":   False,
    "download.directory_upgrade":     True,
    "safebrowsing.enabled":           True
})

# 2 - create a driver instance with defined options
driver = webdriver.Chrome(executable_path="./chromedriver",
                          desired_capabilities=capabilities,
                          options=chrome_options)
```

It is important that the downloaded `chromedriver` (see Appendix B), is placed into the folder with the current python script. The `'./'` in `'./chromedriver'` is the file path to the current folder.

3.2 Loading webpage

With a Chrome driver running, a specific webpage is loaded with

```
driver.maximize_window()
driver.get(‘‘https://suckless.org/rocks/’’)
```

3.3 Work with HTML

The HTML of a webpage can be viewed following the instructions of Fig. 4.

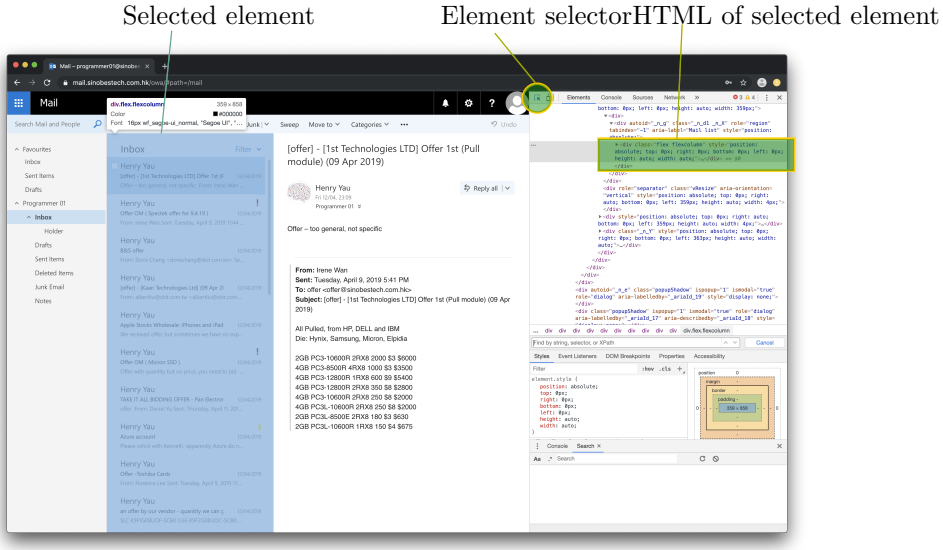


Figure 4: Use the **element selector** to find the html structure of a particular object. `Right Click` → `Inspect` → `Element Selector`. This is a convenient way of probing html on the page.

The typical html content of a webpage, would be a nesting of structures within each other, where each ‘<...>’ encapsulates a further layer.

```
<div class="_lvv_K _lvv_Q">
  <span class="_lvv_T">
    <span autoid="_lvv_6" class="lvHighlightAllClass lvHighlightSubjectClass"
       >[offer]  - [1st Technologies LTD] Offer 1st (Pull module) (09 Apr 2019)
    </span>
  </span>
</div>
```

It is pictorially shown in Fig. **??**. Even 3 layers looks moderately ugly. Supporting document shows was a shitshow 50 layers results in.

To access a particular html element, one would needs to define a unique path through these layers, and there are two approaches at hand:

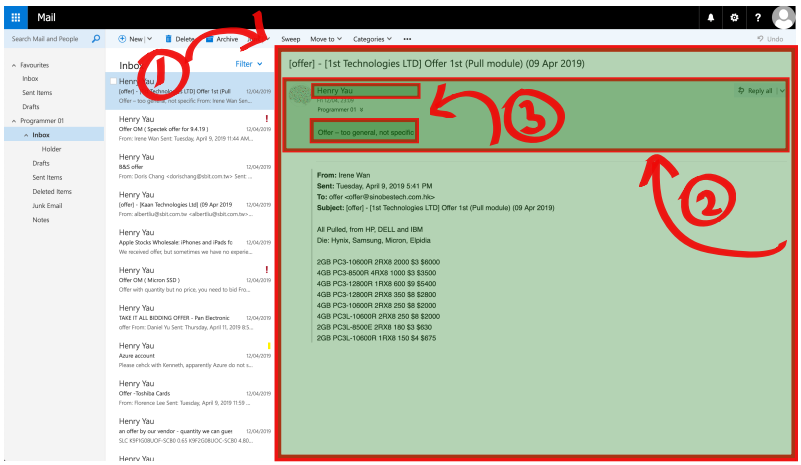


Figure 5: Locating big blocks, and searching within them, quickly converges on the desired element.

1. Brute-force construction of an `xPath` that is used to search the html code.

```
xpath = "//div[@style='position: relative;]/div[5]/div[2]/div[6]/div[2]/div[3]"
user_structure = driver.find_element_by_xpath(element_xpath)
```

★ **Use when elements need to be interacted with (clicked, filled)**

2. Loading the whole page into `BeautifulSoup` and use it's internal search functions:

```
html = driver.page_source
soup = BeautifulSoup(html, 'html.parser')
user_structure = soup.find('div', {'role': 'option',
                                   'aria-label': 'Reading Pane'})
user_structures = soup.findall('div', {'role': 'option',
                                   'aria-label': 'Reading Pane'})
```

★ **Use when text need to be extracted from the webpage.**

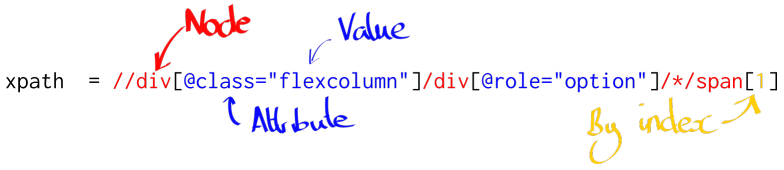
3.4 xPaths

★**Use when elements need to be interacted with or fully loaded**

`xPaths` finds elements on a webpage that can be subsequently **clicked, filled out or queried**. The most important step is the building of the `xPath`. Follow these links for `xPath` cheatsheets: **xpath basics**, **xpath cheatsheet**

3.4.1 Building xPaths

Using the anatomy of an xPath and example HTML code below, we cover some important concepts.




```
<div class="flexcolumn">
   <div role="option">
          <button type="button" title="Select all items in view" role="checkbox">
               <span class="_fc_3"> </span>
               <span class="_fc_4" id="_ariaId_22" style="display: none;"></span>
          </button>
   <div class="_n_21" role="marquee" aria-hidden="true" style="display: none;"></div>
</div>
```

- Navigation is performed through **nodes**.
  - Any node name that starts off html environments <node ...>:

    /div /span /button .
  - To not start from the root node, use double slashes "//div" (else, the full node path would need to be specified):

```
/div[@class='flexcolumn']/div[@role='option']//span[2]
```

- In a given layer, a specific node is selected with an **@ttribute** and corresponding **value**.
  - Multiple attributes are separated by and or or:

```
//button[@type='button' and @role='checkbox']
```

- In a given layer, it is also possible to access a node by index must start from 1 :

```
/div[@role='option']/span[2]
/div[@role='option']/span[last()-1]
```

- A *, @* wildcard matches any node or attribute do not overdo ;

3.4.2 xPath functions

With an xPath built according to the rules of the previous section, the following generic functions can be used to locate and interact with **webelements**:

- **Locating elements** to use later on

```
xpath = "/div[@class='flexcolumn']/div[@role='option']//span[2]"
webelement_example = driver.find_element_by_xpath(xpath)
```

The 's' that costs 2 hours

The following functions return different values. In one case, only the first match to the xPath is returned. In the second all matches are returned in a list.

- `find_element_by_xpath(xpath)` ⇒ `WebElement`;
- `find_elements_by_xpath(xpath)` ⇒ `[WebElement1, WebElement2, ...]`.

Be aware of this, since it will affect how to treat the returned result:

```
webelement_example = driver.find_element_by_xpath(xpath)
webelement_example = driver.find_elements_by_xpath(xpath)[6767]
```

- **Waiting on elements** is a very important procedure, since certain elements of the page may not be loaded in time for clicking or reading operations.

1. First a `WebDriverWaiter` is launched.

```
from selenium.webdriver.support.ui import WebDriverWaiter
from selenium.webdriver.support import expected_conditions as EC

timeout = 50            # timeout given in seconds
WebDriverWaiter = WebDriverWait(driver, timeout)
```

2. An `until` functions calls on another functions and waits for it to return True:

```
WebDriverWaiter.until(EC.presence_of_element_located((By.XPATH, xpath)),
    message=f"Did not find {xpath} within the {timeout}s timeout time")
)
```

Examples of functions/classes to pass to `until`:

`EC.presence_of_element_located((By.CLASS_NAME, "SOME_CLASS_NAME"))`
`EC.presence_of_element_located((By.XPATH, "SOME_XPATH"))`
User-defined `WebDriverWait` class with specific wait conditions

3. After a successful return, the object on the given `xpath` can be used, as shown in the examples below.

◉ **Writing to elements**:

```
xpath = "/div[@class='flexcolumn']/div[@role='option']//span[2]"

# 1 - wait for element to load onto page
WebDriverWaiter.until(EC.presence_of_element_located((By.XPATH, xpath)),
    message=f"Did not find {xpath} within the {timeout}s timeout time")
)

# 2 - gain access to element
webelement_example = driver.find_element_by_xpath(xpath)

# 3 - fill out some kind of value
webelement.send_keys("FORM_VALUE"))
```

◉ **Clicking on elements**:

```
xpath = "/div[@class='flexcolumn']/div[@role='option']//span[2]"
driver.find_element_by_xpath(xpath).click()
```

◉ **Reloading page** by waiting on a specific element to load:

```
page_loaded_xpath = "//div[@class='site-description']/h2[@id='site-description']
    "
driver.get("https://radishmag.wordpress.com/")
WebDriverWaiter.until(EC.presence_of_element_located((By.XPATH, xpath)),
        message=f"Did not load page within the timeout time")
)
self.supp_wait_for_xpath(self.page_loaded_xpath, "main page")
```

- More functions such as `is_displayed` or `is_enabled` can be found on this page.

3.5 BeautifulSoup

★Use when text need to be extracted from the webpage

There is a single preparations stage with `BeautifulSoup` - convert the HTML into an internal format, after which functions can be used to "efficiently" find specific elements to extract:

```
html = driver.page_source
soup = BeautifulSoup(html, 'html.parser')
```

3.5.1 BeautifulSoup searching and extracting

```
<div class="flexcolumn">
   <div role="option">
          <button type="button" title="Select all items in view" role="checkbox">
               <span class="_fc_3"> </span>
               <span class="_fc_4" id="_ariaId_22" style="display: none;"></span>
          </button>
   <div class="_n_21" role="marquee" aria-hidden="true" style="display: none;"></div>
</div>
```

- With a loaded `soup` object, you can search a html structure, by specifying the `node` and `attributes`:

```
single_structure = soup.find("div", attrs={"role": "option"})
multiple_structures = soup.find_all("span",
                                    attrs={"role": "option",
                                           "style": "aria-label": "Reading Pane"
                                               })
```

- Each return is another `BeautifulSoup` object that can be searched, and thus a recursive call can be setup to navigate down the HTML layers, as in Fig 5.

```
search =  [{"node": "div", "attr": {"class": "flexcolumn"}},
           {"node": "button", "attr": {'role': "checkbox", "type": "button"}}]
for i in search:
        unpacked_elements = soup.find_all(i['node'], attrs=i['attr'])
```

would return a `soup` object

```
<button type="button" title="Select all items in view" role="checkbox"> <span class="
    _fc_3"> </span>
<span class="_fc_4" id="_ariaId_22" style="display: none;"></span>
</button>
```

- Using `BeautifulSoup` objects extracted from previous function, text can be accessed with:

```
unpacked_elements = #Block from above example
unpacked_text = [i.get_text().strip() for i in unpacked_elements]
```

- Using `BeautifulSoup` objects specific tags can be accessed

```
unpacked_element = # from above example
unpacked_tag = unpacked_element.get("title")
```

would return “Select all items in view”.

SECTION 4

WebDriverWait Class

Recall, to ensure that a page element is fully loaded, the following wait procedure is called:

```
timeout = 50            # timeout given in seconds
WebDriverWaiter = WebDriverWait(driver, timeout)

WebDriverWaiter.until(EC.presence_of_element_located((By.XPATH, xpath_to_load)))
```

The `WebDriverWaiter` waits for `EC.presence_of_element_located` to return `True` (which will occur once `xpath_to_load` is fully loaded).

If the condition is more complex, define a `__call__(driver)` function:

- First parameter **must** be `driver`;
- Returns `True` once a user-defined criterion is satisfied;
- Executes repeatedly.

Below are 2 examples of such classes created for **Outlook** and **Skype**.

Figure 6: `WebDriverWait` can be used to call very specific wait functions.

- **Outlook** - Forwarded messages are last to load in the email, potentially being scraped before being fully loaded. This class checks that **if** a the email does have a forwarded message, it is given time to fully load.

```
class wait_for_content_forwarded():
  """returns True if either:
        - there is no forwarded message
        - forwarded message has been loaded
  """

  def __call__(self, driver):
```

```
    """
    __ Parameters __
    [selenium.Driver] driver:           that is being run
    __ Description __
    ensure that any forwarded email is fully loaded

    a forwarded email has a non empty <div of forwarded email>:

    <div aria-label='Reading-Pane> .....
        <div>....</div>
        <div>        <---------- div[2]
            <div>...</div>
            etc. etc.
            <div of forwarded email> <---------- NON empty when there is forwarding
            <div>...</div>        <---------- div[last()]
        </div>
    </div>

    __ Return __
    True: if forwarded email loaded
    False: if forwaded email has NOT loaded
    """

    # 1 - test if there is a forwarded section, by checking that the <div of forwarded
        email> is not empty
    try:
        driver.find_element_by_xpath("//div[@aria-label='Reading Pane']/div[2]/div[last
            ()-1]/*")
    except NoSuchElementException:
        #  if no email is being forwarded then we don't have to wait
        return True

    # 2 - IF there is a forwarded email, wait for full email to load
    try:
        driver.find_element_by_xpath("//div[@id='Conversation.FossilizedTextBody']/div
            [1]")
        return True
    except NoSuchElementException:
        #  treurn flase if the email has not loaded yet
        return False
```

Example call:

```
    WebDriverWaiter.until(wait_for_content_forwarded())
```

Skype

```
class wait_for_chat_update():
  """Checking that Skype chat has updated after scrolling has been performed

  To be used in the following way:
  WebDriverWait.until(wait_for_chat_update(old_top_message))
  """

  def __init__(self, top_message_text_old):
      self.top_message_text_old = top_message_text_old

  def __call__(self, driver):
    """
    __ Description __
    compares the id of the top message after scrolling.
    if scrolling has stopped (end of conversation) the id will remain the same
    __ Return __
    True: if text stayed the same - need to perform a click action
    False: if text has changed - can continue scrolling
    """

    ########################################xpaths
    chatBox_xpath = "//div[@style='position: relative; display: flex; flex-direction:
        row; flex-grow: 1; flex-shrink: 1; overflow: hidden; align-items: stretch;
        background-color: rgb(255, 255, 255);']"
    ######################################

    # 1 - locate the Skype conversation box
    chatBox = driver.find_element_by_xpath(chatBox_xpath).find_elements_by_xpath("//div[
        @role='region']")

    # 2 - get top message id and compare it with old id
    top_message_text_new = chatBox[0].id

    if(top_message_text_new == self.top_message_text_old):
        return False
    else:
        return True
```

Example call:

```
        WebDriverWaiter.until(wait_for_chat_update(top_id))
```

SECTION 5

Outlook SPECIFICS

In **Outlook** the process being automated is:

- Perform login;
- Scrape single email, storing From, Date, Subject, Conversation and Forwarded Email in a `Pandas DataFrame`:

```
pandas_columns = [ "From", "Date", "Subject", "Content_Conversation", "
    Content_Forwarded"]
pandas_scraped = pd.DataFrame(columns=pandas_columns)
```

- Perform the scraping for all emails fullfiling certain criteria:

```
only_unread = True
# Date format [Year, Month, Day]
date_min = [2018, 1, 2]
date_max = [2019, 12, 31]
# Range of emails to scrape
scan_min = 0
scan_max = 100
```

using the criteria check

```
def criteria_check(self, metadata, criteria):
    # 1 - extract criteria info
    date_min = criteria['date_min']
    data_max = criteria['data_max']
    date_min = datetime.datetime(date_min[0], date_min[1], date_min[2])
    date_max = datetime.datetime(date_max[0], date_max[1], date_max[2])

    # 2 - extract metadata
    unread = metadata['unread']
    email_no = metadata['email_no']
    date = metadata['date']
    date = datetime.datetime(date[0],date[1],date[2])

    return_val = False

    ########################################
    # Perform check
```

```
        ######################################
        # 1 - check that email is within indicies
        if((criteria['email_min'] <= email_no) and (email_no <= criteria['email_max'])):
            # 2 - check date
            if ((date_min <= date) and (date <= date_max)):
                # 3 - if scraping only unread, check unread status
                if(criteria['only_unread']):
                    if(unread):
                        return_val = True
                    else:
                        return_val = False
                else:
                    return_val = True

        return return_val
```

Some functions that are used, are wrapped versions of commands seen earlier in the document, and given in fully in Appendix C:

Function	Brief Summary
`supp_wait_for_xpath(xpath)`	Waits for a given `xPath` to load on the page
`supp_write_to_element(xpath, text_to_write)`	Writes given content to the supplied element using `xPaths`
`self.supp_extract_html(soup, search)`	Extract html content using `BeautifulSoup`

5.1 Perform Login

With the login page ready, this series of steps ensures safe login:

[1] Wait for full login page to load → [2] Fill out login details and submit → [3] Wait for Outlook to load

```
        driver = #LOADED DRIVER
        password = #PASSWORD
        outlook_id = #OUTLOOKID
        succesful_login_xpath = "//div[@class = 'flex flexcolumn']"

        # 1 - wait for outlook page to fully load
        supp_wait_for_xpath("//input[@id='username']", "user_name_input_field")

        # 2 - locate credential fields and fill them in
        supp_write_to_element("//input[@id='username']", outlook_id)
        supp_write_to_element("//input[@id='password']", password)
```

```
        driver.find_element_by_xpath(
            "//div[@onclick='clkLgn()']").click()

        # 3 - ensure that login is succesfull and wait for emails to load
        supp_wait_for_xpath(succesful_login_xpath, "main_page")
```

5.2 Scrape single email

Given the `webelement` of an email (found using the `xPath` functions from Sec. 3.4.2), the email is scraped for content.

[1] Click on the `webelement` → [2] Convert whole page into `BeautifulSoup` → [3] Extract text using `soup`

```
def outlook_scrape_email(self, email_webelement):
        """
        __ Parameters __
        [web_element] email_webelement: found with xPath  "self.driver.find_element_by_xpath
            (...)"
        __ Description __
        extacts data from the given email (passed as a web_element)
        __ Return __
        [dict]          {"From": e_from,
                        "Date": e_date,
                        "Subject": e_subject,
                        "Content_Conversation": e_content_conversation,
                        "Content_Forwarded": e_content_forwarded}
        """
        # 1 - load up the email_webelement and wait for for load
        try:
            email_webelement.click()
            self.supp_wait_for_xpath("//div[@id = 'Item.MessageUniqueBody']", "NA")
            self.WebDriverWaiter.until(wait_for_content_forwarded())

        except TimeoutException:
            print(
                "**> Email failed to load. Increase timeout (currently %.1fs)" % (self.
                    timeout))
            return

        # 2 - extract html on the page. soup is the chad way to search this html
        soup = self.supp_load_soup()
```

Once a `BeautifulSoup` object is loaded, text is extracted by searching specific parts of the webpage:

```
        # 3 - extract text using soup function
        # a - subject
        e_subject = "".join(self.supp_extract_text(soup,
                                    [["div", {"aria-label": "Reading Pane"} ],
                                     ["div", {'role': "heading", "aria-level": "2"}]]))
        # b - from
        e_from = "".join(self.supp_extract_text(soup,
                                       [["div", {"aria-label": "Persona card"} ]]))
        match_groups = re.search("([^<]*)(.*)?", e_from) # remove useremail <ilya.antonv....>
        e_from = match_groups.group(1).strip()
        # c - date and time
        e_date = "".join(self.supp_extract_text(soup,
                                      [["div", {"class": "_rp_f8"}],
                                       ["span", {"class": "allowTextSelection"} ]]))
        # d - email_webelement content
        e_content_conversation = "".join(self.supp_extract_text(soup,
                                         [["div", {"aria-label": "Reading Pane"}],
                                          ["div", {"role": "document"}]]))
        e_content_forwarded = "".join(self.supp_extract_text(soup,
                                      [["div", {"aria-label": "Reading Pane"}],
                                       ["div", {"id": "Conversation.FossilizedTextBody"
                                           }]]))

        # 4 - structure building and return
        email_entry = {"From": e_from,
                       "Date": e_date,
                       "Subject": e_subject,
                       "Content_Conversation": e_content_conversation,
                       "Content_Forwarded": e_content_forwarded}

        return email_entry
```

5.3 Scrape batch emails

[1] Enumerate visible mail → [2] Collect metadata of visible mail → [3] Create list of unscraped emails → [4] Scrape selected emails → [5] Scroll and check for end

```
    def outlook_scrape(self, file_name="pandas_out", ext="csv"):
        ########################################xpaths
        inbox_mail_L1_xp = "//div[@class = 'flex flexcolumn']"
        inbox_mail_L2_xp = "//div[@role ='option']"
```

```
        inbox_mail_soup = [["div", {"class": "flex flexcolumn"}],
                           ["div", {"role": "option"}]]
        ########################################

        inbox_cycle = True
        email_loop_no = 0
        email_base_index = 0            # required to stitch across different loops
        email_already_scraped = set()   # scraped mails store their unique id here
```

1. First a unique ID is generated for visible mail in inbox, using `xPaths` to search through 2 layers of HTML.

```
        while(inbox_cycle):

            # 1 - [XPATH] extract unique tag and webelement of each email
            visible_webElements = self.driver.find_element_by_xpath(inbox_mail_L1_xp).
                find_elements_by_xpath(inbox_mail_L2_xp)
            visible_uniqueID = [i.text for i in visible_webElements]
```

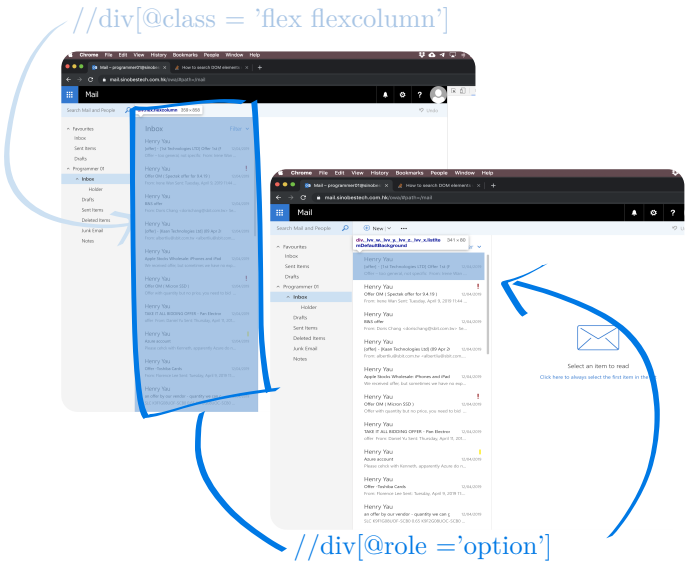


Figure 7: Accessing individual emails through 2 layers of HTML.

2. Almost the same process is repeated for the metadata of each email, but this time extraction occurs with `BeauitfulSoup`. Whereas in previous case it was about extracting as much unqiue text as possible (where the brutal `xPath` extraction that kept noise characters was ideal), searching for specific tags is more efficient with `BeutifulSoup`.

(the `outlook_inbox_date` and `outlook_inbox_unread` functions are defined in Appendix D)

```
        # 2 - [SOUP] extract metadata of visible mail
        soup = self.supp_load_soup()
        visible_metadataRaw = self.supp_extract_html(soup, inbox_mail_soup)
        visible_metadata = []
        for i in visible_metadataRaw:
            visible_metadata.append({"date": self.outlook_inbox_date(i),
                                     "unread": self.outlook_inbox_unread(i)})
```

Thus at this stage, there is a set parameters for every email visible on the screen:

`webElement`	handle to click on that email
`uniqueID`	bundle of characters unique to each email
`metadata`	metadata on each email

3. Due to the retarded nature of scrolling, it is entirely possible to come across emails that have already been scraped, but still appear in view. By storing the `uniqueID` of each scraped email, it can be ensured that only unscraped emails are analyzed:

```
        # 3 - only iterate through unscraped mail i.e. uniqueID is not in "
            uniqueID_already_scraped" set
        emails_to_scrape = []
        i = 0
        for email_webElement, email_uniqueID, email_metadata in zip(
            visible_webElements, visible_uniqueID, visible_metadata):
            if(email_uniqueID not in uniqueID_already_scraped):

                # 3a - add the email number
                email_metadata["email_no"] = email_base_index + i
                i += 1
                # 3b - store email id and email_metadata for further extraction
                emails_to_scrape.append({"email_webElement": email_webElement,
                                         "email_metadata": email_metadata})
                # 3c - store the unqiue tag to prevent scraping it in the future
                uniqueID_already_scraped.add(email_uniqueID)
```

4. A for loop goes through the list of emails to check, performs a criteria check on their metadata and uses `outlook_scrape_email` (Sec. 5.2) to scrape the email, which is written to a `Pandas DataFrame`.

```
        # 4 - iterate through only the new emails
        for i, email in enumerate(emails_to_scrape):

            # a - check that email_metadata fulfills criteria
```

```
                criteria_satisfied = self.criteria_check(email['email_metadata'],
                                                         self.criteria)

                if(criteria_satisfied):
                    print(self.outlook_scrape_print_progress(email['email_metadata']))
                    email_content = self.outlook_scrape_email(email['email_webElement'])
                    self.pandas_scraped = self.pandas_scraped.append(email_content,
                        ignore_index=True)
```

5. Once all visible emails are scraped, click on the last email to load older mail.

```
            # 5 - click on last mail (to scroll down)
            self.outlook_scrape_email(emails_to_scrape[-1]['email_webElement'])

            # a - set variables for next loop
            email_loop_no += 1
            email_base_index = len(uniqueID_already_scraped)

            # b - check against max emails scraped
            if(email_base_index > self.criteria['scan_max']):
                # if we have scraped all the emails, stop
                inbox_cycle = False

            # c - check against no scrolling (same emails displayed)
            visible_idx_new = self.driver.find_element_by_xpath(inbox_mail_L1_xp).
                find_elements_by_xpath(inbox_mail_L2_xp)
            if (visible_webElements == visible_idx_new):
                inbox_cycle = False
```

SECTION 6

Skype SPECIFICS

The core mechanics for scraping **Skype** are the same as for **Outlook** (Sec. 5), and so repetitive details will not be restated here. Emphasis is placed on resolving issues to do with the continuous scrolling problem, see Fig. 8.

- Perform login → closing any pop-up boxes;
- Scrape each conversation into a `Pandas DataFrame`:

```
self.pandas_columns = [ "From", "Date", "Message"]
self.pandas_scraped = pd.DataFrame(columns=self.pandas_columns)
```

  Check continuously against certain criteria with `skype_scrollCheck_date`. .

```
# Date format [Year, Month, Day]
date_min = [2018, 1, 2]
date_max = [2019, 12, 31]
# Maximum number of scrolls that don't move the chat
max_number_of_stalls = 20
```

- Eliminate redundancy during scrolling:

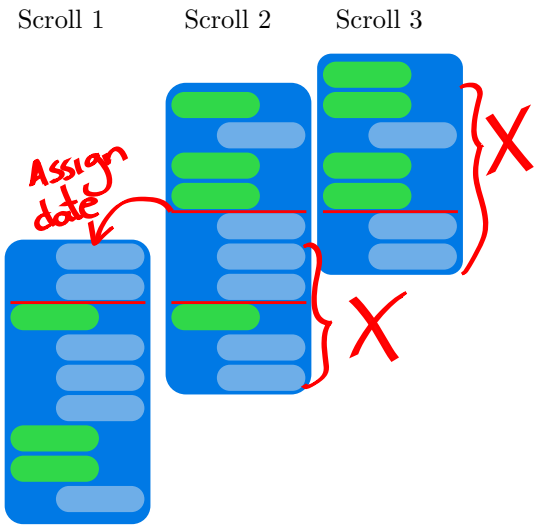


Figure 8: Scrolling and extracting messages on visible screen can cause overlaps. Dates for top messages will also need to be assigned.

6.1 Perform Login

[1] Fill out SkypeID → [2] Fill out password and submit → [3] Close popups

```
        ########################################xpaths
        skype_login_box_xp = "//input[@type='email']"
        skype_password_box_xp = "//input[@type='password']"
        skype_submit_button_xp = "//input[@id='idSIButton9']"
        skype_got_it_xp = "//div[@data-text-as-pseudo-element='Got it!']"
        ########################################

        # 1 - wait for email to load
        self.supp_write_to_element(skype_login_box_xp, skype_id)
        self.driver.find_element_by_xpath(skype_submit_button_xp).click()
        time.sleep(3)           # <---------------------- need to wait for password box to come
            up

        # 2 - wait for password
        self.supp_write_to_element(skype_password_box_xp, password)
        self.driver.find_element_by_xpath(skype_submit_button_xp).submit()

        # 3 - remove popups after page has loaded
        self.supp_wait_for_xpath(self.succesful_login_xpath, "page")
        time.sleep(2)           # <---------------------- wait for the popup box to come up
        try:
            self.driver.find_element_by_xpath(skype_got_it_xp).click()
        except:
            pass
```

6.2 Scraping single chat

[1] Click on bottom message → [2] Scroll the chat → [3] Scrape visible messages → [4] Check scrolling criteria and repeat

```
    def skype_scrape_chat(self):
        ################################# XPATH of chat
        messages_xp = "//div[@style='position: relative; display: flex; flex-direction: row;
            flex-grow: 1; flex-shrink: 1; overflow: hidden; align-items: stretch; background-
            color: rgb(255, 255, 255);']/div/div[2]/div/div/div/div/div/div/div/div/div/div/
            div/div/div/div[2]/div[@role='region']"
        ######################################
```

```
        scraped_messages = []           # cumulative array of all the scraped_messages
        oldest_date =  "Undefined"
        continue_scroll = True
        current_number_of_scrolls = 0
        scroll_stall = 0            # counter to check how long scrolling has been stalled
            for
```

1. Scan all the messages currently visible in the chat, storing the `id` of the top message, to use as a scrolling anchor later on.

```
        # 1 - click on the bottom of the chat after it has loaded
        self.supp_wait_for_xpath(messages_xp, "at_least_one_message_in_chat")
        all_messages = self.driver.find_elements_by_xpath(messages_xp)
        topMessage_ID_old = all_messages[0].id
        ActionChains(self.driver).move_to_element(all_messages[-1]).click().perform()
```

2. Scroll the chat to reveal new content (established by probing the `id`). A simple `PAGE_UP` scroll is not always successful, in which case some "jittering" is required to force content to load.

```
    while (continue_scroll):
        # 2 - scroll the chat and rebase
        ActionChains(self.driver).send_keys(Keys.PAGE_UP).perform()
        all_messages = self.driver.find_elements_by_xpath(messages_xp)
        topMessage_new = all_messages[0]
        topMessage_ID_new = topMessage_new.id

        if(topMessage_ID_new == topMessage_ID_old):
            # 2a - if the top of the chat has not updated, jitter the chat by
                clicking and scrolling up and down
            ActionChains(self.driver).move_to_element(topMessage_new).click().
                perform()
            scroll_stall += 1
            ActionChains(self.driver).send_keys(Keys.PAGE_DOWN).perform()
        else:
            # 2a - otherwise continue scrolling
            scroll_stall = 0
```

3. Extract messages in view with `skype_scrape_chat_visible`. Since they will be extracted top → bottom (old → new), the order needs to be reversed, so that it is easier to stack chronologically later on.

```
        # 3 - extract all scraped_messages and reverse the order so thatthey go NEW ->
            OLD
        # (instead of the OLD -> NEW that top down scraping gives)
```

```
            messages_to_add = self.skype_scrape_chat_visible()
            messages_to_add = messages_to_add[::-1]
```

4. Finalize the current scroll, by storing processed messages → checking scroll criteria → resetting for the next round.

```
            # 4a - store all scraped_messages with defined dates
            for i in messages_to_add:
                if(i[1] != "Undefined"):
                    scraped_messages.append(i)
                    oldest_date = i[1]

            # 4b - check if scroll conditions are still satisfied and repeat loop
            continue_scroll = self.skype_scrollCheck_date(self.criteria,
                                                          {"current_date": oldest_date,
                                                           "current_number_of_stalls":
                                                               scroll_stall})

            # 4c - reset variables next loop
            topMessage_ID_old = topMessage_ID_new
            current_number_of_scrolls +=1
```

6.3 Formatting scrapped messages

```
    def skype_format_messages(self, messages_to_format, critetia):
        """
        __ Parameters __
        [1D-(sender, dateString, message)] messages_to_format:
        __ Description __
        the list of messages is scanned and:
        - messages with an unassigned dates ("Undefined") are given a date
        - duplicate messages are removed
        - messages newer than a certain values are removed
        """

        message_to_format = messages_to_format[::-1] # revese the message order, so that
             oldest (with defined date) are on top
        self.reset()

        # setup dates
        date_min = criteria['min_date']
        date_min = datetime.datetime(date_min[0], date_min[1], date_min[2])
```

```
        date_max = criteria['max_date']
        date_max = datetime.datetime(date_max[0], date_max[1], date_max[2])
        running_date = "Undefined" # date that keeps track of where we are in the
            mssages
```

Once messages have been scraped, overlaps need to be removed, and messages with unassigned dates need to be resolved:

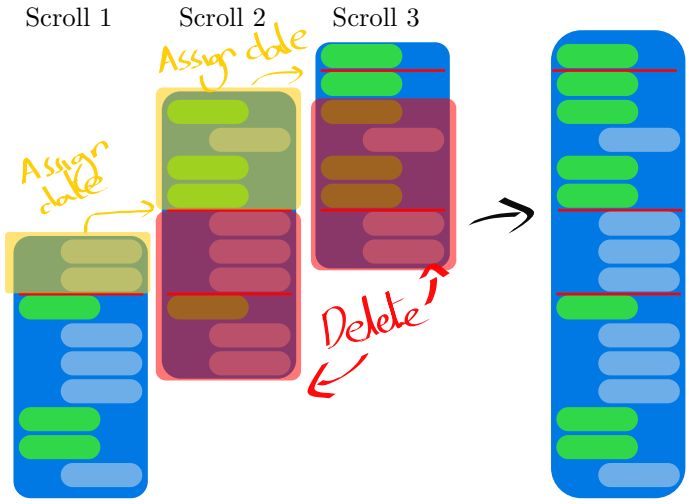


Figure 9: Scrolling leads to overlap of data - duplicated messages need to be deleted, and unassigned dates resolved.

```
        # 1 - iterate the messages, resolving any dates that were not extracted during
            scrolling
        for i in messages_to_format:
            date = i[1]
            if(date == "Undefined"):
                # a - if a date was not defined, look at the previously defined date
                date = running_date
            else:
                # b - convert date to [year, month, day]
                date = self.date_from_string(date)
                running_date = date # store the running date, so that further dates can
                    be infered from it

            # 2 - write messages with defined dates that fall within the max/min dates
            if(date != "Undefined"):
                date = self.datetime_from_date(date)
                if((date >= date_min) and (date <= date_max)):
                    date_string = date.strftime("%d %B %Y")
                    # 2 - store the message in a dataframe
                    message_to_store = {"From": i[0],
```

```
                                        "Date": date_string,
                                        "Message": i[2]}

                    self.pandas_scraped = self.pandas_scraped.append(message_to_store,
                        ignore_index=True)

        # 3 - remove duplicate entries due to scrolling overlap
        self.pandas_scraped = self.pandas_scraped.drop_duplicates()
```

SECTION A

Extra Material

1. **xpath basics**
2. **xpath cheatsheet**
3. `BeautifulSoup`

SECTION B

Requirements

1. Install Google Chrome;
2. Download the Chrome Driver and place it in the directory with the python script;
3. Install the *selenium* and `BeautifulSoup` package for python

```
python -m pip install selenium bs4
```

4. Use the following imports at the start of the python file

```
import time

# create a browser instance
from selenium import webdriver

# emulate keyboard inputs
from selenium.webdriver.common.keys import Keys

# creatinga single browser instance
import selenium.webdriver.firefox.service as service
from selenium.webdriver.chrome.options import Options
from selenium.common.exceptions import TimeoutException
from selenium.common.exceptions import ElementNotInteractableException

# WebDriverWait and EC to allow waiting for element to load on page
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# module to search for elements using xpaths
from selenium.webdriver.common.by import By
```

```
from selenium.webdriver.support import expected_conditions

# exception handling
from selenium.common.exceptions import NoSuchElementException
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

# quick clicking and scrolling
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

# searching of html with "find()"
from bs4 import BeautifulSoup
import pandas as pd
import sys
import math
import os                       # file saving
import re
```

SECTION C

Selenium Functions

These functions are wrapped versions of those presented in Sec. 3.4 and Sec. 3.5.

```
    def supp_extract_html(self, soup, html_tags_array):
        """
        __ Parameters __
        [soup] soup: html to extract from formatted with BeautifulSoup
        [arr] html_tags_array: array of the form

                                    [["div", {"role": "option"}],
                                    ["div", {"aria-label": "Reading Pane"}],
                                    ...]

        which specifies the name ("div", "span") and attributes ({"id": ["test1", "test2"], "
            aria-label": "pane"})
        from outer to inner tags, iteratively going down specificity levels

        __ Description __
        iterates through the supplied "soup" html looking for tags whose parrents match all
            the supplied "html_tags"

        __ Return __
        [htmltag1, htmltag2, htmltag3]: array of html tags that fit the search requirement
        """

        structure_depth  = len(html_tags_array)
        debug_counter = 0

        try:
            if(structure_depth != 1):
                # 1 - unpack the first structure
                current_structure = soup.find(
                    html_tags_array[0][0], attrs=html_tags_array[0][1])

                # 2 - unpack further structures until we get to the last one
                for i in range(1, structure_depth - 1):
                    debug_counter += 1
                    name = html_tags_array[i][0]
                    attrs = html_tags_array[i][1]
                    current_structure = current_structure.find(names, attrs=attrs)
```

```
                # 3 - extract all matches from the lowest structure
                current_structure = current_structure.find_all(
                    html_tags_array[-1][0], attrs=html_tags_array[-1][1])
            else:
                # 1 - in the special case that only one structure is specified
                current_structure = soup.find_all(
                    html_tags_array[0][0], attrs=html_tags_array[0][1])

            return current_structure

        except AttributeError:
            # Error when an entry is missing
            print("The page does not have the html element:\n\t[%s, %s]"
                  % (html_tags_array[debug_counter], html_tags_array[debug_counter]))

            return ""
```

```
    def supp_extract_text(self, soup, html_tags_array):
        """
        __ Parameters __
        [soup] soup: html to extract from formatted with BeautifulSoup
        html_tags_array: array of the form

        [["div", {"role": "option"}],
        ["div", {"aria-label": "Reading Pane"}],
        ...]

        which specifies the name ("div", "span") and attributes ({"id": ["test1", "test2"], "
            aria-label": "pane"})
        from outer to inner tags, iteratively going down specificity levels

        __ Description __
        iterates through the supplied "soup" html looking for tags whose parrents match all
            the supplied "html_tags"
        then a text array is extracted from this tag

        __ Return __
        [array] matching text in the innter structure
        """

        html_structure = self.supp_extract_html(soup, html_tags_array)

        # 1 - take all of the tags found and extract text
        array_to_return = [i.get_text().strip() for i in html_structure]
```

```
        return array_to_return
```

```
    def supp_wait_for_xpath(self, xpath, description):
        """
        __ Parameters __
        [str] xpath: xpath to wait for
        [str] description: the object that is trying to be located. will be printed to
            console.
                        "NA" to skip

        __ Description __
        pauses the browser until "xpath" is loaded on the page
        """

        if(description != "NA"):
            print("  > Waiting for \"%s\" to load" %(description))

        self.WebDriverWaiter.until(
            EC.presence_of_element_located(
                (By.XPATH, xpath)),
            message="Did not find %s within the timeout time you set of %i"%(xpath, self.
                timeout)
        )
```

SECTION D

Outlook Functions

Functions which extract metadata on an email

```
    def outlook_inbox_date(self, inbox_tag):
        """
        __ Parameters __
        [soup] inbox_tag: a html tag of an particular email in the inbox
        __ Description __
        extracts a date of the email in the inbox column by searching the "inbox_tag"
        __ Returns __
        [day, month, year]
        """
        ########################################
        date_attr = {"class": ["_lvv_M"]}
        ########################################

        # 1 - extract date tag
        date_tag = inbox_tag.find(attrs=date_attr)
        date_inbox = date_tag.get_text()

        # 2 - split date put by slashes. this will work for old entries
        date_return = date_inbox.split("/")
        date_return = date_return[::-1] # reverse order so that [year, month, day]

        if(len(date_return) != 3):
            # 3 - for email sent this week, the first string is the day of the week, which is
                converted to [year, month, day]
            weekday = date_inbox.split(" ")[0]
            date_return = self.date_from_string(weekday)


        # 3 - convert to int
        date_return = [int(i) for i in date_return]

        return date_return
```

```
    def outlook_inbox_unread(self, inbox_tag):
        """
        __ Parameters __
        [soup] inbox_tag: a html tag of an particular email in the inbox
        __ Description __
```

```
        checks if email unread or not
        __ Returns __
        True if unread. False otherwise
        """

        # 1 - read email have a "_lvv_y_" tag
        mg = re.search("\s_lvv_y\s", str(inbox_tag))

        # 2 - check if match was found, indicating that email has been read
        if(mg):
            return False
        else:
            return True
```

D.1 Skype Functions

D.1.1 Checking criteria

During each scroll cycle, a criteria check is run to see if more messages need to be scraped.

```
    def skype_scrollCheck_date(self, critetia, current_values):
        """
        __ Parameters __
        [dict] critetia:       {"chats_to_scrape":      [1D-int] starting from 0,
                                "date_min":              [year,month,day]
                                "date_max":              [year,month,day]
                                "max_number_of_stalls": [int] before continuing}
        [dict] current_values  {"current_date":         [year,month,dayy],
                                "current_number_of_stalls": [int]}

        __ Description __
        checks if the filter-defined date has been reached, to determine if scrolling should
            continue

        __ Return __
        True if scrolling should continue
        False if it should be stopped
        """

        criteria_min_date = criteria['min_date']
        current_date = current_values['current_date']

        return_val = False
```

```
        if(current_values['current_number_of_stalls'] < criteria['max_number_of_stalls']):
            # 1 - continue scrolling if date is not defined
            if(current_date == "Undefined"):
                return True

            # 2 - compare the date reached so far in the chat with the filter date
            current_date = self.date_from_string(current_date) # convert the date from string
                 to array
            current_date = datetime.datetime(current_date[0], current_date[1], current_date
                [2]) # initialie a datetime object
            criteria_min_date = datetime.datetime(criteria_min_date[0], criteria_min_date[1],
                 criteria_min_date[2])

            if (criteria_min_date <= current_date):
                return_val = True

            else:
                print("\n  > Chat scrolled past the user-defined date: %s [now at %s]"
                      %(criteria_min_date.strftime("%d %B %Y"), current_date.strftime("%d %B
                          %Y")))
                print("  > Stopping scrolling of chat")
        else:
            print("\n  > Chat stalled for the maximal user-defined number of scrolls: %i"
                  %(criteria['max_number_of_stalls']))
            print("  > Stopping scrolling of chat")

        return return_val
```

D.1.2 Scraping messages in window

`BeautifulSoup` is used to extract the date, sender and message content for each message visible on the screen. Encountered dates are stored and assigned to the subsequent messages.

```
    def skype_scrape_chat_visible(self):
        """
        __ Return __
        [1D-(sender, date, message)]  array of tuples holding info on each message.
        """
        ######################################## Tags to identify messages
        soup_chat_message = [["div", {"role": ["region", "heading"], "tabindex": re.compile("
            (-1|0)"), "aria-label": re.compile(".")}]]
        ########################################
        messages = []
```

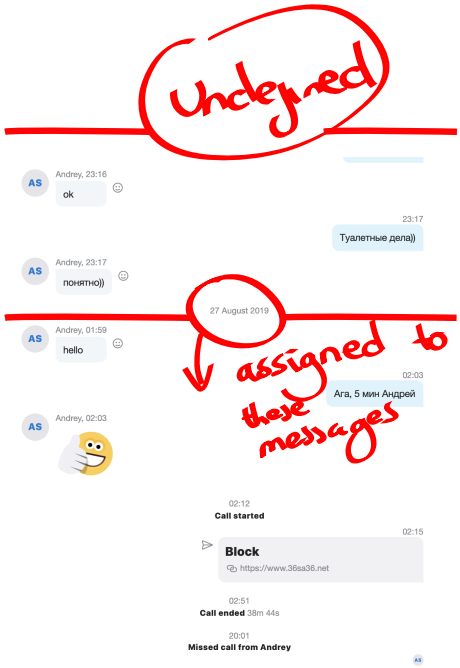


Figure 10: The latest date in the conversation is stored and assigned to following block of messages.

```
        date_current = "Undefined"

        # 1 - get html of the page and look for messages with beautiful soup
        soup = self.supp_load_soup()
        chatContent = self.supp_extract_html(soup, soup_chat_message)
        for i in chatContent:
            # 2 - extract contents of the messages
            message_content = i["aria-label"]

            # a - date extraction
            date = re.match(
                re.compile("((\d{2}\s(January|February|March|April|May|June|July|August|
                    September|October|November|December)\s\d{4})|(Monday|Tuesday|Wednesday|
                    Thursday|Friday|Saturday|Sunday))"),
                           message_content)
            if(date):
                date = date.group(1)
                date_current = date

            # b - sender extraction
            #     sender comes before the main part of the message e.g. "YAU, shall we go
                ...."
            sender = re.search(re.compile("(^\w+(\s\w+)?)(,)"), message_content)
```

```
            if(sender):
                sender = sender.group(1)
            else:
                sender = None

            # c - message: comes after the sender with a comma and before the time sent e.g.
                "yau, SHALL WE GO..., sent at 18:00"
            message = re.search(re.compile("(^\w+(\s\w+)?,)(.*)(, sent at \d{2}:\d{2})"),
                message_content)
            if(message):
                message = message.group(3)
            else:
                message = None

            # 2 - store the message if it was NOT a date (e.g. 09 March 2019)
            if((not date) and message):
                messages.append((sender, date_current, message))

        return messages
```

SECTION E

SUPPORTING METHODS

The following methods are not related to **Selenium**, but support common operations.

```
    def reset(self):
        """
        __ Description __
        clears the pandas_out array to the initial value
        """

        self.pandas_out = pd.DataFrame(columns=self.entry_list)

    def save_data(self, file_name="pandas_out", ext="csv"):
        """
        __ Parameters __
        [str] file_name: the file to save to. provide .pkl or .csv extension

        __ Description __
        Saves data accumulated in "pandas_out" to output file
        """

        # 1 - create output directory
        if not os.path.exists("./output"):
            os.mkdir("output")

        # 2 - cut any extensions that were given by accident
        file_name = file_name.split(".")[0]
        file_name = "./output/%s" % (file_name)

        if(ext == "pkl"):
            self.pandas_out.to_pickle("%s.pkl" % file_name)
        else:
            self.pandas_out.to_csv("%s.csv" % file_name)
```

E.1 Datetime conversions

```
    import datetime
```

The date of a message or email is often given in the form “Tue” or “18th May 2019”. The following set of

functions facilitates conversion between

$$\begin{cases} \text{'Tue'} \\ \text{'18th May 2019'} \end{cases} \quad \Leftrightarrow \quad [2019, 02, 18]$$

according to what form is required.

```
def date_from_string(self, date_string):
    """
    __ Parameters __
    [str] date_string: either day of week or "18 May 2019"
    __ Description __
    convert to an array numerical date values. if a weekday was supplied, find
    the most recent date for that weekday
    __ Return __
    [year, month, day] date: array of the date
    """
    weekday_list = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                    "Saturday", "Sunday",
                    "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]


    if (date_string in weekday_list):
        # 1 - set loop parameters
        date = datetime.date.today()
        date_shift = datetime.timedelta(days = 1)
        date_found = False

        # 2 - decrease date, until the weekday_list match
        while(not date_found):
            date = date - date_shift
            day_of_the_week_long = weekday_list[date.weekday()]
            day_of_the_week_short = weekday_list[date.weekday() + 7]
            if((day_of_the_week_long == date_string) or (day_of_the_week_short ==
                date_string)):
                date_found = True
    else:
        date = datetime.datetime.strptime(date_string, '%d %B %Y')

    date_array = [date.year, date.month, date.day]
    return date_array
```

```
def datetime_from_date(self, date_array):
    """
```

```
        __ Parameters __
        [year, month, day] date: array of the date
        __ Description __
        converts the array to a datetime object
        __ Return __
        [datetime] datetimeObject
        """
        return datetime.datetime(date_array[0], date_array[1], date_array[2])
```

```
    def string_from_date(self, date_array):
        """
        __ Parameters __
        [year, month, day] date: array of the date
        __ Description __
        converts the array to string representation "18 May 2019"
        __ Return __
        [str] date_string
        """

        date = datetime.datetime(date_array[0], date_array[1], date_array[2])
        return date.strftime("%d %B %Y")
```