Creating a user-interface for watch selection

Changanaqui Julian, Bourleau Marc

Master in Management – Business Analytics Orientation
University of Lausanne
Lausanne, Switzerland
julian.changanaqui@unil.ch, marc.bourleau@unil.ch

Abstract **-The purpose of this project is to aid customers of Swiss luxury watchmaking brands in the selection of their desired timepiece. The main objective is to alleviate the challenges associated with making an informed purchase in the face of the overwhelming amount of information available. The driving motivation for this project stems from the inadequacy of current online watch databases, such as Watchbase, to fully address this issue. We formulated a methodology to address the problem effectively, by creating a user-friendly, efficient, and informative dashboard that empowers consumers to find their ideal watch with ease. Our approach provides a comprehensive solution to the challenge of making an informed purchase and fills a critical gap in the existing market.**

Index terms **- scraping, dashboard, luxury Swiss watches, Selenium, Tkinter**

I. Introduction

Since the inception of the first wristwatch by Patek Philippe in 1868, Swiss luxury watches have witnessed remarkable growth and global popularity. Despite the challenges posed by the recession caused by Covid-19 and inflationary pressures, the global luxury watch market demonstrated significant expansion between 2022 and 2023. The market size escalated from 30.58 billion in 2022 to 34.17 billion in 2023, indicating a compound annual growth rate (CAGR) of 11.7%. According to the Deloitte Swiss watch industry study, it is anticipated that the proportion of watches purchased online will rise to 30% by 2030, a notable increase from the current approximate level of half that. This surge in e-commerce sales can be attributed to the preference of nearly half of the Millennial and Gen Z population for online shopping over in-store experiences. Additionally, a Bain & Co report predicts that by 2030, younger generations (millennials, Z, and Alpha) will account for 80% of luxury purchases.

As the wealth of information on the internet continues to expand, the task of finding the perfect timepiece has become increasingly challenging for both new and experienced consumers. With an increasing amount of purchases made in online channels, the absence of a centralised and comprehensive database that consolidates information on all watches adds to the complexity of the process. It necessitates individuals to navigate through the websites of numerous watch brands, resulting in a time-consuming and laborious endeavour.

Therefore, as the luxury watch market continues to expand, the development of an interactive dashboard that facilitates informed purchasing decisions holds significant importance. Such a dashboard would provide consumers with a user-friendly interface, streamlining the process of finding and selecting their desired timepieces.

II. Research question

A. Problem

A current problem is that luxury watch consumers are facing difficulties to find their dream wristwatch. The luxury watch market is characterised by a wide range of brands, each offering their distinctive styles, designs, complications, and craftsmanship. The lack of a centralised resource that aggregates and organises this diverse range of watches makes it challenging for consumers to compare and evaluate different options effectively. They may need to visit multiple sources, navigate various websites, and rely on fragmented information, making the decision-making process more complex and daunting.

Currently, Watchbase stands as one of the few websites that attempts to tackle the aforementioned challenge by providing a centralised platform that consolidates various luxury watch brands. However, it is important to note that while Watchbase addresses the issue of navigating through individual brand websites, it does not fully resolve the core problem at hand. This is primarily due to the lack of a user-friendly dashboard or interface that simplifies the buying process for consumers. Thus, although Watchbase serves as a resource for accessing information on luxury watch brands in a centralised manner, it falls short in terms of offering an intuitive and streamlined solution that effectively addresses the complexities associated with purchasing luxury watches.

B. Objective

Our main goal is to fully address the problem by providing consumers with a comprehensive platform of watches. However, the development of this platform requires the construction of a comprehensive database of the watch industry. In order to do that, we have scrapped the Watchbase website using the available package on Python called Selenium. Then, in order to build the interface, we have used the package Tkinter.

C. Scope

In contrast to the Watchbase database, our approach involved selecting a specific number of brands to scrape based on their market share. We were successful in gathering 2065 images along with their accompanying text characteristics, such as price, material, type of glass, diameter, and more. However, there were two primary reasons that prevented us from scraping the entire website. The first reason relates to time constraints. The process of scraping all the watches would have necessitated an extensive loading time, potentially spanning multiple days. Given our project's limitations and time restrictions, it was not feasible to allocate such a significant amount of time to the scraping process. The second reason involves mitigating bias in our data collection. To ensure a fair and unbiased representation, we manually selected brands that had a comparable number of watches. This approach helped avoid any potential skewing of the dataset and ensured a more balanced representation across different brands.

III. Methodology

A. Database

The core of our project revolves around constructing a database. We will gather images and textual data for each of our watches using a reference website (https://watchbase.com/). To ensure variety in our data, we have decided to include watches from 20 brands, totalling 2000 watches (approximately 100 watches per brand).

B. Scraping

To retrieve the required data for our database, we utilised the Selenium library for web scraping. This library allows us to target dynamic data on websites, making automation easier. We divided the scraping process into two parts. The first program downloads all the required images, while the second program scrapes the associated textual data.

C. Data Cleaning / Mapping

Using mapping operations, we merge all our data together. Since including images directly in a CSV is not practical, we only keep their references in a new variable. The images are stored in a separate folder, and we can retrieve them directly using the references in our data frame. The end result is a data base that contains the desired variables and observations.

D. Visual Interface

Once our data is properly organised, our next task is to create a user interface that is intuitive and dynamic. To achieve this, we have decided to use the versatile Tkinter library. Tkinter allows us to personalise windows by creating buttons, widgets, and displaying images. We will directly link our database to the implemented buttons, enabling users to select various variables and customise their search. As an output, we aim to display 9 pictures that best represent the search criteria for watches. To avoid repetition, we will impose restrictions when no specific brand is selected. For instance, if the user does not specify a brand, we will use a random function which will ensure that we have diversity in our output.

IV. Implementation

The project is hosted in a Github repository, which contains all the required files and instructions on a step by step basis. Please refer to the README for the adequate installation of the environment and packages required for the proper replication of the platform. The program runs in Python, version 3.9. and is structured using the following packages:

- Tkinter - version 8.5.9
- Numpy - version 1.24.3
- Pandas - version 2.0.1
- Matplotlib - version 3.7.1
- Selenium - version 4.9.1
- Tqdm - version 4.65.0
- Pillow - version 9.5.0

A. Program implementation

The platform code has a modular structure presented as follows (we only display the most important executable files for diagramming reasons):

```
pro_project/
├── README.md
├── mapping.py
├── data_with_images.csv
├── watch_text.csv
├── watch_text.py
├── images_scraping.py
├── Watches_Images/
│   ├── (images.png ...)
└── tkinter.py
```

1) mapping.py: cleaning and pre-processing
2) data_with_images.csv: final data base
3) watch_text.csv: watches attributes
4) watch_text.py: code for scraping watches attributes
5) Images_scraping.py: code for scraping watches images
6) Watches_Images: folder including all the watches images (.png)
7) Tkinter.py: code for creating the interface

B. Scraping

For our scraping purposes, we use the Selenium library, which allows us to scrape dynamic websites. We start by defining a vector with the names of the 20 brands we are incorporating into our database. The number of images we are planning to download in each category is stored in the variable *images_per_category*, which we have set to 55. After trying various values for this variable, we decided to use 55 because it allows us to have around 2000 images in the end. We've experienced a significant number of image losses, either because the image didn't download correctly or because the program didn't go through every watch.

Using the WebDriver from Selenium, we are able to connect to the website.

By examining its structure (cf: figure 1), we see that categories are divided into subparts on each brand's page. Using CSS selectors, we are able to select the common HTML identifier for each category (.family-box.row .col-md-6 h2.title a). We chose to go through 5 categories for each watch brand, which will provide more variety in our database.

We create a for loop that navigates to each category page. We artificially scroll down on each page to trigger image loading and add a wait statement that waits for each image to load properly. We then save the image URLs and store them in a variable called *image_urls*. We replace the prefix 'md' with 'lg' in each link to have higher quality images. Within each category, we search for the <img> tag.

Once we have all the URLs of all the watches, we create a new for loop to save our observations. The end result is a folder named 'Watches_Images' that contains subfolders for each brand and the corresponding reference name for all watches. We have a total of 2065 images. Unfortunately, not all brands have the same number of watches. For example, Rolex has 114 observations, whereas Audemars Piguet has 68. Still overall, we were able to obtain approximately 100 images for most of the watches. Please note that these values may vary depending on different factors when running our code, such as the website structure (which can change), image formats, page loading speed, and others.

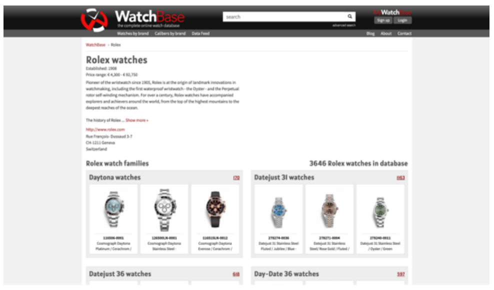
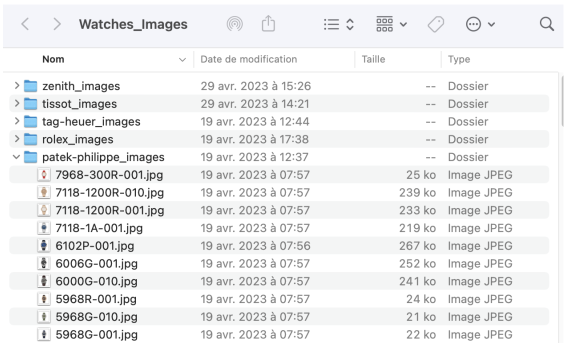


Figure 1: The Rolex page. Sub categories are contained inside separate blocks (Source: https://watchbase.com/rolex)



Watches_Images

Nom	Date de modification	Taille	Type
zenith_images	29 avr. 2023 à 15:26	--	Dossier
tissot_images	29 avr. 2023 à 14:21	--	Dossier
tag-heuer_images	19 avr. 2023 à 12:44	--	Dossier
rolex_images	19 avr. 2023 à 17:38	--	Dossier
patek-philippe_images	19 avr. 2023 à 12:37	--	Dossier
7968-300R-001.jpg	19 avr. 2023 à 07:57	25 ko	Image JPEG
7118-1200R-010.jpg	19 avr. 2023 à 07:57	239 ko	Image JPEG
7118-1200R-001.jpg	19 avr. 2023 à 07:57	233 ko	Image JPEG
7118-1A-001.jpg	19 avr. 2023 à 07:57	219 ko	Image JPEG
6102P-001.jpg	19 avr. 2023 à 07:56	267 ko	Image JPEG
6006G-001.jpg	19 avr. 2023 à 07:57	252 ko	Image JPEG
6000G-010.jpg	19 avr. 2023 à 07:57	241 ko	Image JPEG
5968R-001.jpg	19 avr. 2023 à 07:57	24 ko	Image JPEG
5968G-010.jpg	19 avr. 2023 à 07:57	21 ko	Image JPEG
5968G-001.jpg	19 avr. 2023 à 07:57	22 ko	Image JPEG

Figure 2: A view of the resulting folder Watches_Images

C. Watch characteristics

To extract the text characteristics of each watch, we employed a systematic scraping process similar to the one used for image scraping. We initiated by creating an empty dataframe called "watch_data" that encompasses columns representing the desired text characteristics, such as materials, price, diameter, and others. Subsequently, we iterated through each brand's webpage, extracting the links for the first five categories on the page. Within this iteration, we nested another loop that iterated through each category and extracted the corresponding text information. To achieve this, we identified three distinct CSS selectors (labeled as "1", "2", and "3" in Figure 2) to locate and retrieve the desired text data. The first selector, "info-table", was used to identify the relevant table containing the name, category and reference of the watch. The second selector, "col-xs-6", enabled the extraction of specific attributes within the table. Lastly, the selector "#pricechart" was employed to locate the URL associated with the price attributes.

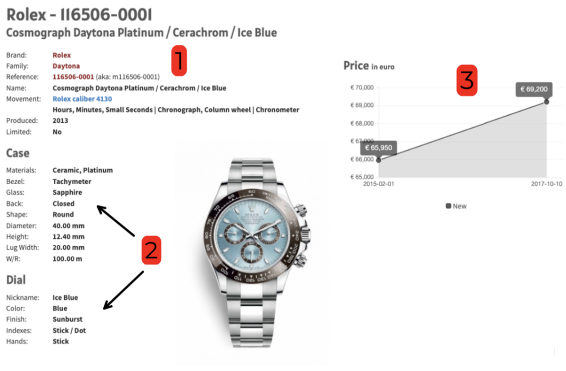
To successfully scrape the price, we employed the URL obtained from "#pricechart" and opened it in a new window. Through inspection of the HTML source, we identified the CSS selector "#\/datasets\/0\/data\/1 > td:nth-child(2) > span:nth-child(1) > span:nth-child(1)" to target the price element.

Additionally, we incorporated an exception handling mechanism, specifically a NoSuchElementException, to prevent the loop from stopping when a watch did not have a price. Any NaN values in the Price column were subsequently replaced with "NA" to denote missing data.

Throughout the scraping process, we encountered two inconsistencies that required our attention. Firstly, the watch references provided on the website occasionally included an additional AKA ("Also Known As") following the main reference number.

Secondly, the variable representing the material of the watches exhibited inconsistency in its naming convention on the website. In some instances, the material was listed in plural form, while in others, it was singular. To ensure consistency and avoid data duplication, we incorporated both variations in the dataframe and combined them into a unified column named "Materials."

Finally, we saved the resulting database as "watch_text_.csv," ensuring that the extracted text characteristics were organised and readily accessible for further analysis and utilisation.



Rolex - 116506-0001
Cosmograph Daytona Platinum / Cerachrom / Ice Blue

Brand:	Rolex
Family:	Daytona
Reference:	116506-0001 (aka: m116506-0001)
Name:	Cosmograph Daytona Platinum / Cerachrom / Ice Blue
Movement:	Rolex caliber 4130
	Hours, Minutes, Small Seconds | Chronograph, Column wheel | Chronometer
Produced:	2013
Limited:	No

Case

Materials:	Ceramic, Platinum
Bezel:	Tachymeter
Glass:	Sapphire
Back:	Closed
Shape:	Round
Diameter:	40.00 mm
Height:	12.40 mm
Lug Width:	20.00 mm
W/R:	100.00 m

Dial

Nickname:	Ice Blue
Color:	Blue
Finish:	Sunburst
Indexes:	Stick / Dot
Hands:	Stick

Figure 3: Cosmograph Daytona Platinum watch. (Source: https://watchbase.com/rolexdaytona/116506-0001)

D. Data base

In our final database, "data_with_images," we have a total of 2062 observations. Initially, we collected the text characteristics on 3937 watches. However, during the image scraping process, we encountered an issue where some of the images were found to be black or empty. We used a mapping dictionary, linking the watch reference to its corresponding characteristics, to associate the characteristics with the watches.

This mapping automatically dropped the watches from our database for which we had the text characteristics but did not have the respective image. As a result, the final "data_with_images" database contains only the watches for which we successfully obtained both the text characteristics and the corresponding image. During the mapping process, we encountered a problem where the inclusion of the "price" variable in our final database resulted in a significant reduction in the number of observations from 2062 to 1449. It was observed that at least 20% of the watches did not have a price value available. In order to prioritize a larger number of observations, it was decided to exclude the "price" variable from the "data_with_images" dataset.

Furthermore, during the mapping process, we made some modifications to the "Reference" variable. Specifically, we removed any additional alternative names (AKA) and selected only the variables that would be useful for constructing the interface. These selected variables include: "Reference", "Limited", "Glass", "Shape", "Diameter", and most importantly, "image_file". The "image_file" variable will serve as a crucial link between the text characteristic filters and the corresponding image to be displayed as the output.

E. Tkinter Interface

The final stage of our report involves creating the user interface. We used Tkinter, which is a powerful library in python to build GUI's. It offers a great deal of flexibility and a wide range of widgets, buttons, and visual elements.

Before diving into coding, it's important to define the basic layout we want to adopt. We plan to have the variables, which the user can adjust, on the left side. Then, we want to showcase the recommendations on the right side. Additionally, we will include an image and a title. Below is the first draft of our intended tkinter interface:

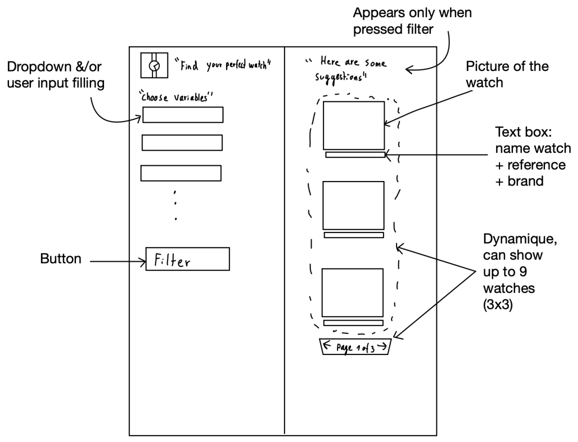

Appears only when pressed filter
Dropdown &/or user input filling
"Find your perfect watch"
"Choose variables"
"Here are some suggestions"
Picture of the watch
Text box: name watch + reference + brand
Button
Filter
Dynamique, can show up to 9 watches (3x3)
Page 1 of 3

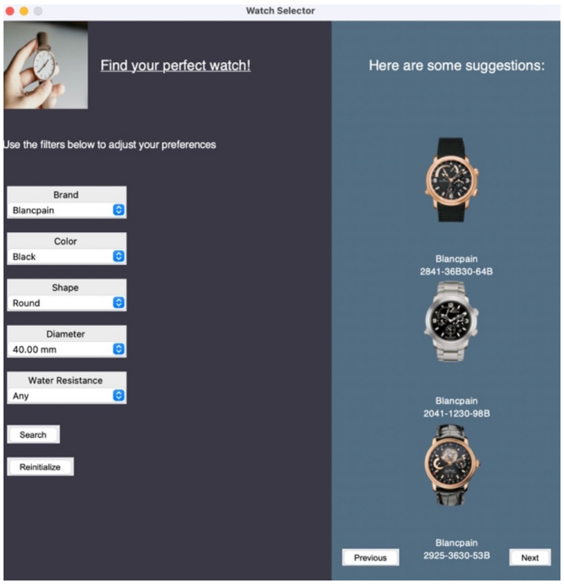
Figure 4: The draft used as a reference

There are two main tools used to create a UI with Tkinter: widgets and frames. Widgets can be either dynamic (i.e., the user can interact with them) or static (such as a text title). On the other hand, frames are used to create spaces to implement those widgets. Therefore, we start by delimiting our windows into two separate frames: the left frame and the right frame. We use what is called « gridded » frames. They work by having geometrical locations on a row/column basis, allowing for better flexibility when adding widgets later on.

We begin with our left frame. We add some texts, an image along with our title, and create dropdown buttons. These buttons will contain the following five variables: brand, color, shape, diameter, and water resistance. The user will be able to choose from all the desired combinations. In order to extract these variables, we define a function called 'extract_unique_values' that will be applicable to the selected columns of our CSV file. Since diameter and water resistance represent numerical values, we apply an additional lambda function to sort their values by converting them first to float. After extracting all that we need, we add our data to the five buttons.

Next, we create the right frame. We define three placeholder spots where our watch images will be shown. This allows us to visualize and adjust the layout of the images. We initially provide plain white images. Under each image, we want to display the brand name and the reference number. To do so, we create lists and store the values that will be retrieved based on the filtering. In order to map the variables, our images, and our text output, we create a class called 'Watch'. It takes all the necessary values, from variables to image paths. Afterwards, we create a function called 'search_button_click'. With this function, we are able to connect the watches with the user's filters and output the images to the placeholder spots. Inside the function, we decide to use a random function to select the watches that match with the criterias. It allows us to avoid having too many of the same brands if no brands are specified. We also define a default error image if no matches are found. Finally, we just need to create our buttons. We create two separate functions, 'next_button_click' and 'previous_button_click'. We initialize the current page to 0 and set a maximum of three pages. We then add the two widgets at the bottom of the right frame.

The end result is a functioning interface where users can filter watches based on their preferences. Figure 5 shows how our finale interface looks like.


Watch Selector
Find your perfect watch!
Use the filters below to adjust your preferences
Search
Reinitialize
Here are some suggestions:
Previous
Next

Brand: Blancpain
Color: Black
Shape: Round
Diameter: 40.00 mm
Water Resistance: Any

Blancpain 2841-36B30-64B
Blancpain 2041-1230-98B
Blancpain 2925-3630-53B

Figure 5: The end result interface

IV. Maintenance and update

For the maintenance, continuously scraping the Watchbase dataset is of utmost importance due to the frequent emergence of new watches and the rapid fluctuations in prices.

While our existing code does not possess the capability for self-maintenance, it offers a high level of customization. The programmer has the flexibility to specify the number of watches to scrape as well as the desired categories to extract from the dataset. By allowing customization, the code can be tailored to meet specific scraping requirements, enabling the collection of relevant data based on the programmer's preferences and needs. This adaptability ensures that the scraping process remains adaptable to changes in the Watchbase dataset.

As a further improvement, we strongly suggest applying a Naive Bayes algorithm. The algorithm would learn from the preferences of the user based on the filters it applies and use this information to recommend watches. By applying the Naive Bayes algorithm, the system can analyse the patterns and relationships between the user's selected filters and the corresponding watches. It can learn which combinations of filters tend to be preferred by the user and use this knowledge to provide personalised recommendations.
Users can benefit from a more personalised and relevant watch browsing experience, enhancing their overall satisfaction and engagement with the dashboard.

V. Results

Through the process of web scraping, we have successfully obtained a collection of 2062 images along with their associated text characteristics. To improve the user experience and facilitate the search for specific timepieces, we have developed a Tkinter-based graphical user interface (GUI) that serves as a user-friendly dashboard. This GUI allows users to interactively filter and explore watches based on their desired characteristics. By providing manual control over the filtering process, users can customize their search criteria and find watches that align with their preferences.

However, there is room for improvement. Despite the significance of price as a key attribute when purchasing a watch, we were unable to incorporate it into our final database due to a substantial number of missing values.

Furthermore, it is recommended to expand the dataset, ensuring a balanced representation across different categories, and mitigating the introduction of any bias. By increasing the data volume and maintaining balance, the model's performance can be improved, resulting in more accurate and reliable predictions.

VI. References

1) Report: “The Deloitte Swiss Watch Industry Study 2022”
https://www2.deloitte.com/content/dam/Deloitte/ch/Documents/consumer-business/deloitte-ch-en-swiss-watch-industry-study-2022.pdf
2) Report: “THE MILLENNIAL STATE OFMIND”
https://www.bain.com/contentassets/0b0b0e19099a448e83af2fb53a5630aa/bain20media20pack_the_millennial_state_of_mind.pdf
3) Report: “Luxury Goods in Switzerland”
https://www.statista.com/outlook/cmo/luxury-goods/switzerland

4) Report: “Watch Industry statistics”
https://www.fhs.swiss/eng/statistics.html

5) Study: “State of the industry - Swiss watchmaking in 2022”
https://watchesbysjx.com/2022/03/morgan-stanley-watch-industry-report-2022.html

6) Data base: “Watch base”
https://watchbase.com/

7) Watch picture used for interface:
https://unsplash.com/fr/photos/xfNeB1stZ_0
7) Watch picture used for interface:

images_scraping.py

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
import time
import os
import urllib.request
import selenium.common.exceptions
from selenium.webdriver.support.ui import
WebDriverWait
from selenium.webdriver.support import
expected_conditions as EC
import ssl

ssl._create_default_https_context =
ssl._create_unverified_context

# Brands and their corresponding URLs
brands = ["Rolex", "Omega", "Patek-philippe",
"Audemars-piguet", "Cartier", "Breitling",
"Tag-Heuer", "Iwc", "Chopard",
          "Jaeger-lecoultre", "Blancpain",
"Hublot", "Zenith", "Certina", "Panerai",
"Girard-perregaux", "Breguet",
          "Montblanc", "Tissot", "Bulgari"]

urls = ["https://watchbase.com/" +
brand.lower() for brand in brands]

# Number of images to download per category
per brand
images_per_category = 55
timeToSleep = 2
```

```python
# Loop through each brand's page
for url in urls:
    driver.get(url)

    # Find the first 5 categories of the brand and get their
links
    categories = driver.find_elements(By.CSS_SELECTOR, ".family-
box.row .col-md-6 h2.title a")[:5]
    category_links = [category.get_attribute("href") for category
in categories]

    # Loop through each category and extract the image URLs
    total_count = 0
    for link in category_links:
        print("--------------------" + link)
        try:
            driver.get(link)
        except selenium.common.exceptions.NoSuchWindowException:
            # Reopen the window or tab
            driver.execute_script("window.open('');")
            driver.switch_to.window(driver.window_handles[-1])
            # Navigate to the URL again
            driver.get(link)

        # Wait for page to load
        time.sleep(timeToSleep)

        # Scroll down the page several times to trigger image
loading
        for i in range(5):
            driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
            time.sleep(timeToSleep)

        # Wait for all images to load
        WebDriverWait(driver,
10).until(EC.visibility_of_all_elements_located((By.TAG_NAME,
'img')))
        # Find all the image tags on the page
        images = driver.find_elements(By.TAG_NAME,
'img')[:images_per_category]
        # Collect the URLs of the images that match the expected
prefix
        vector = ["speedmaster-automatic-and-other", "datejust"]
        brand_prefix =
f"https://cdn.watchbase.com/watch/md/{link.split('/')[-2] + '/' +
link.split('/')[-1]}"
        image_urls = [image.get_attribute('src') for image in
images if
image.get_attribute('src').startswith(brand_prefix) or
                      (image.get_attribute('src').startswith(v)
for v in vector)]
        # Replace "md" with "lg" in each URL to get higher
quality images
        modified_urls = [url.replace('/md/', '/lg/') for url in
image_urls]
        # Create a folder on your desktop to save the images
        folder_path =
os.path.expanduser(f"~/Desktop/{link.split('/')[-2]}_images")
        if not os.path.exists(folder_path):
            os.makedirs(folder_path)

        # Download the images and save them to the folder
        count = 0
        for i, url in enumerate(modified_urls):
            if count == images_per_category:
                break

            file_name = os.path.join(folder_path, f"image_{i +
1}.jpg")
            urllib.request.urlretrieve(url, file_name)
            print(f"Downloaded image {i + 1}: {file_name}")
            count += 1
            total_count += 1

        if total_count >= images_per_category *
len(category_links):
            break

# Close the browser
driver.quit()
```

watch_text.py

```
import csv
import os
import time
import pandas as pd
from selenium import webdriver
from selenium.webdriver.common.by import By
from tqdm import tqdm
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import
expected_conditions as EC
from selenium.common.exceptions import
NoSuchElementException
from urllib3.util.url import get_host
import ssl

ssl._create_default_https_context =
ssl._create_unverified_context

# Create an empty pandas DataFrame to store the watch
data: we will then add all the values that are stored in
the watch_data_dict
watch_data = pd.DataFrame(
    columns=["Brand", "Family", "Reference", "Name",
"Movement", "Produced", "Limited", "Materials",
"Material",
             "Bezel", "Glass", "Back", "Shape",
"Diameter", "Height", "Lug Width", "W/R", "Nickname",
"Color", "Finish",
             "Indexes", "Hands", "Price"])

# Brands and their corresponding URLs
brands = ["Rolex", "Omega", "Patek-philippe", "Audemars-
piguet", "Cartier", "Breitling", "Tag-Heuer", "Iwc",
"Chopard",
         "Jaeger-lecoultre", "Blancpain", "Hublot",
"Zenith", "Certina", "Panerai","Girard-perregaux",
"Breguet",
          "Montblanc", "Tissot", "Bulgari"]
urls = ["https://watchbase.com/" + brand.lower() for
brand in brands]

# Cleaning the data: sometimes the variable Material is
saved in plural. We added both in the dataframe watch
data and we merged them into "Materials".
watch_data['merged_column'] =
watch_data['Materials'].fillna(watch_data['Material'])
watch_data.drop(['Materials', 'Material'], axis=1,
inplace=True)
watch_data.rename(columns={'merged_column': 'Materials'},
inplace=True)

# Open the browser
driver = webdriver.Firefox()

# Loop through each brand's page
for url in urls:
    driver.get(url)

    # Find the first 5 categories on the page and get
their links
    categories = driver.find_elements(By.CSS_SELECTOR,
".family-box.row .col-md-6 h2.title a")[:5]
    category_links = [category.get_attribute("href") for
category in categories]

    # Loop through each category and extract the text
    for link in tqdm(category_links):
        driver.get(link)

        # Wait for page to load
        time.sleep(2)

        # Scroll down the page several times to trigger
image loading
        for i in range(5):
            driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
            time.sleep(2)
```

```
        # Now we landed on the category website but we have
to be inside each watch website
        watches = driver.find_elements(By.CSS_SELECTOR,
".watch-block-container a.item-block")[:50]  # we seek to
take 50 images
        watches_link = [watch.get_attribute("href") for watch
in watches]

        for each_watch in watches_link:
            driver.get(each_watch)

            # Wait for page to load
            time.sleep(2)

            # Scroll down the page several times to trigger
image loading
            for i in range(5):
                driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
                time.sleep(2)

            watch_data_dict = {}

            # Extract watch details from the first table
            first_table = driver.find_element(By.CLASS_NAME,
"info-table")
            rows_first_table =
first_table.find_elements(By.TAG_NAME, "tr")

            for row in rows_first_table:
                row_data = row.text.split(":", 1)  # split on
first occurrence of ":"

                if len(row_data) == 2:
                    key = row_data[0].strip()
                    value = row_data[1].strip()
                    watch_data_dict[key] = value
                    print(watch_data_dict)

            # Extract watch details from the second table
            second_table = driver.find_element(By.CLASS_NAME,
"col-xs-6")
            row_second_table =
second_table.find_elements(By.TAG_NAME, "tr")

            for row in row_second_table:
                row_data = row.text.split(":", 1)  # because
reference sometimes has (aka number)

                if len(row_data) == 2:
                    key = row_data[0].strip()
                    value = row_data[1].strip()
                    watch_data_dict[key] = value
                    print(watch_data_dict)

            try:
                # Find the element with the "data-url"
attribute
                element =
driver.find_element(By.CSS_SELECTOR, "#pricechart")
                data_url = element.get_attribute("data-url")

                # Open the "data-url" in a new tab

driver.execute_script(f"window.open('{data_url}')")

                # Switch to the new tab
                # Add an explicit wait to ensure the new tab
is fully loaded before switching
                WebDriverWait(driver,
10).until(EC.number_of_windows_to_be(2))

driver.switch_to.window(driver.window_handles[1])

                # Wait for the new tab to load
                # Add an explicit wait to ensure the content
of the new tab is loaded
                WebDriverWait(driver,
10).until(EC.presence_of_element_located((By.CSS_SELECTOR,"#\
\/datasets\\/0\\/data\\/1 > td:nth-child(2) > span:nth-
child(1) > span:nth-child(1)")))
```

```
Find the price
                price_element =
driver.find_element(By.CSS_SELECTOR,
"#\\/datasets\\/0\\/data\\/1 > td:nth-child(2) >
span:nth-child(1) > span:nth-child(1)")
                price_value = price_element.text if
price_element.text != "null" else "NA"

                # Add the price value to the
watch_data_dict
                watch_data_dict["Price"] = price_value

                # Close the new tab
                driver.close()

                # Switch back to the main tab

driver.switch_to.window(driver.window_handles[0])

            except NoSuchElementException:
                print("No price section found.
Skipping watch.")
            # Add the watch data to the pandas
DataFrame
            watch_data.loc[len(watch_data)] =
watch_data_dict
            print(watch_data)

# Replace "NaN" values in the "Price" column with "NA"
watch_data["Price"].fillna("NA", inplace=True)
# Count the number of missing values
for col in watch_data.columns:
    missing_values = watch_data[col].isna().sum()
    print(f"Number of missing {col} values:
{missing_values}")

# Save the pandas DataFrame to a CSV file
watch_data.to_csv("cleaned_watch_text_def.csv",
index=False)
```

Mapping.py

```
import os
import pandas as pd
import re

# Load the characteristics dataset
char_df = pd.read_csv('cleaned_watch_text_def.csv')
pattern = r' \(aka.*\)'
char_df['Reference'] =
char_df['Reference'].apply(lambda x: re.sub(pattern,
'', x))
char_df['Reference'] =
char_df['Reference'].str.replace('/', '-')

# Create a new column with the image file names
char_df['image_file'] =
char_df['Reference'].apply(lambda x: f"{x}.jpg")
char_df['image_file_processed'] = 'processed_' +
char_df['image_file']

# Define the directory where the image files are
stored
image_dir =
'/Users/lawrencejesudasan/Downloads/Watches_Images'

# Create a dictionary to map the characteristics to
the images
char_to_image = {}
for subdir, _, files in os.walk(image_dir):
    for file in files:
        if file.endswith('.jpg'):
            image_path = os.path.join(subdir, file)
            reference = file[:-4]  # Remove the file
extension
            if reference in
char_df['Reference'].values:
                char_to_image[reference] =
char_df.loc[char_df['Reference'] ==
reference].iloc[0].to_dict()

mapped_df = pd.DataFrame.from_dict(char_to_image,
orient='index')

mapped_df.to_csv("data_with_images.csv", index=False)

# Print the dictionary to verify the mapping
print(char_to_image)
```

Tkinter.py

```
import tkinter as tk
from PIL import Image, ImageTk
import csv
import os
import random

# Create the main window
window = tk.Tk()
window.title("Watch Selector")
window.geometry('800x800')

# Calculate the screen width and height
screen_width = window.winfo_screenwidth()
screen_height = window.winfo_screenheight()

# Calculate the x and y coordinates for centering the window
x = (screen_width // 2) - (600 // 2)
y = (screen_height // 2) - (600 // 2)

# Set the window position
window.geometry(f'800x800+{x}+{y}')

# Make the window static and non-resizable
window.resizable(False, False)

# Define the placeholder image
placeholder_image_path = "placeholder.png"
placeholder_original = Image.open(placeholder_image_path)
placeholder_resized = placeholder_original.resize((130, 130))
placeholder_image = ImageTk.PhotoImage(placeholder_resized)

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Create the
left frame
left_frame = tk.Frame(window, bg='#3A3845')
left_frame.grid(row=0, column=0, sticky='nsew')

left_frame.grid_rowconfigure(1, minsize=100)

# Open and resize the image
image_path_title = "image.jpg"
original_image = Image.open(image_path_title)
resized_image = original_image.resize((125, 125))
logo_image = ImageTk.PhotoImage(resized_image)

# Add image logo and text to the left frame
logo_label = tk.Label(left_frame, image=logo_image,
bg='#3A3845', borderwidth=0, highlightthickness=0)
logo_label.grid(row=0, column=0, sticky='w')

text_label = tk.Label(left_frame, text="Find your perfect
watch!", font=('Helvetica 20 underline'), bg=('#3A3845'),
                      fg='white')
text_label.grid(row=0, column=0, columnspan=2, sticky='e',
padx=140)

# Add instruction text for user on the left frame
text_instruction = tk.Label(left_frame, text="Use the filters
below to adjust your preferences", font=('Helvetica', 15),
                            bg=('#3A3845'), fg='white')
text_instruction.grid(row=1, column=0, sticky='w',
columnspan=2)

# Read the CSV file
csv_file_path = "AP/data_with_images.csv"

# Read the CSV file and extract the image file names
image_file_names = []
with open(csv_file_path, 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        image_file_names.append(str(row['image_file']))

# Construct the image paths
image_folder_path = "Watches_Images"
image_paths = [os.path.join(image_folder_path, image_file)
for image_file in image_file_names]
```

```
# Function to extract unique values from a column in the
CSV file
def extract_unique_values(column_name):
    unique_values = set()
    with open(csv_file_path, "r") as file:
        reader = csv.DictReader(file)
        for row in reader:
            unique_values.add(row[column_name])
    return unique_values


# Extract unique brand values
brand_values = extract_unique_values("Brand")
brand_values = sorted(list(brand_values))
brand_values.insert(0, "Any")

# Extract unique color values
color_values = extract_unique_values("Color")
color_values = sorted(list(color_values))
color_values.insert(0, "Any")

# Extract unique shape values
shape_values = extract_unique_values("Shape")
shape_values = sorted(list(shape_values))
shape_values.insert(0, "Any")

# Diameter
diameter_values = extract_unique_values("Diameter")
sorted_values_diameter = sorted(diameter_values,
key=lambda x: float(x.split()[0]) if x.split() else 0)
sorted_values_diameter.insert(0, "Any")

# Water Resistance
wr_values = extract_unique_values("W/R")
sorted_values_wr = sorted(diameter_values, key=lambda x:
float(x.split()[0]) if x.split() else 0)
sorted_values_wr.insert(0, "Any")

# Create the dropdown buttons
dropdown_frame1 = tk.Frame(left_frame, relief='flat')
dropdown_frame1.grid(row=3, column=0, padx=10, pady=10,
sticky='w')

title_label1 = tk.Label(dropdown_frame1, text="Brand")
title_label1.configure(width=18)
title_label1.pack(anchor='w')

dropdown_var1 = tk.StringVar()
dropdown_var1.set(brand_values[0])  # Set the default
value to "Any"
dropdown_menu1 = tk.OptionMenu(dropdown_frame1,
dropdown_var1, *brand_values)
dropdown_menu1.configure(width=14)
dropdown_menu1.pack(anchor='w')

dropdown_frame2 = tk.Frame(left_frame, relief='flat')
dropdown_frame2.grid(row=4, column=0, padx=10, pady=10,
sticky='w')

title_label2 = tk.Label(dropdown_frame2, text="Color")
title_label2.configure(width=18)
title_label2.pack(anchor='w')

dropdown_var2 = tk.StringVar()
dropdown_var2.set(color_values[0])  # Set the default
value to "Any"

# Update the dropdown menu options with the sorted values
dropdown_menu2 = tk.OptionMenu(dropdown_frame2,
dropdown_var2, *color_values)

dropdown_menu2.configure(width=14)
dropdown_menu2.pack(anchor='w')

dropdown_frame3 = tk.Frame(left_frame, relief='flat')
dropdown_frame3.grid(row=5, column=0, padx=10, pady=10,
sticky='w')

title_label3 = tk.Label(dropdown_frame3, text="Shape")
title_label3.configure(width=18)
title_label3.pack(anchor='w')

dropdown_var3 = tk.StringVar()
dropdown_var3.set(shape_values[0])  # Set the default
value to "Any"
dropdown_menu3 = tk.OptionMenu(dropdown_frame3,
dropdown_var3, *shape_values)
dropdown_menu3.configure(width=14)
dropdown_menu3.pack(anchor='w')

dropdown_frame4 = tk.Frame(left_frame, relief='flat')
dropdown_frame4.grid(row=6, column=0, padx=10, pady=10,
sticky='w')
```

```
title_label4 = tk.Label(dropdown_frame4, text="Diameter")
title_label4.configure(width=18)
title_label4.pack(anchor='w')

dropdown_var4 = tk.StringVar()
dropdown_var4.set(sorted_values_diameter[0])
dropdown_menu4 = tk.OptionMenu(dropdown_frame4,
dropdown_var4, *sorted_values_diameter)
dropdown_menu4.configure(width=14)
dropdown_menu4.pack(anchor='w')

dropdown_frame5 = tk.Frame(left_frame, relief='flat')
dropdown_frame5.grid(row=7, column=0, padx=10, pady=10,
sticky='w')

title_label5 = tk.Label(dropdown_frame5, text="Water
Resistance")
title_label5.configure(width=18)
title_label5.pack(anchor='w')

dropdown_var5 = tk.StringVar()
dropdown_var5.set(sorted_values_wr[0])
dropdown_menu5 = tk.OptionMenu(dropdown_frame5,
dropdown_var5, *sorted_values_wr)
dropdown_menu5.configure(width=14)
dropdown_menu5.pack(anchor='w')

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Create the
right frame
right_frame = tk.Frame(window, bg='#526D82')
right_frame.grid(row=0, column=1, sticky='nsew')

# Configure grid weights to make frames expand equally
window.grid_columnconfigure(1, weight=1)
window.grid_columnconfigure(0, weight=1)
window.grid_rowconfigure(0, weight=1)

# Add text label to the right frame
suggestions_label = tk.Label(right_frame, text="Here are
some suggestions:", font=('Helvetica', 20), bg=('#526D82'),
                             fg=('white'))
suggestions_label.grid(row=0, column=1, sticky='w',
padx=50, pady=48)

# Create a list to store all the brand labels
brand_labels = []

# Create a list to store all the reference labels
reference_labels = []

# Create a list to store all the placeholder labels
placeholder_labels = []

# Create placeholder labels (White images where output
images will be shown)
for i in range(3):
    placeholder_label = tk.Label(right_frame,
image=placeholder_image, bg='#526D82', borderwidth=0,
                                 highlightthickness=0)
    placeholder_label.grid(row=i + 1, column=1, sticky='w',
padx=105, pady=36)
    placeholder_labels.append(placeholder_label)

    brand_label = tk.Label(right_frame, text="",
font=('Helvetica', 13), bg='#526D82', fg='white')
    brand_label.grid(row=i + 2, column=1, sticky='n',
padx=10, pady=0)
    brand_labels.append(brand_label)

    reference_label = tk.Label(right_frame, text="",
font=('Helvetica', 13), bg='#526D82', fg='white')
    reference_label.grid(row=i + 2, column=1, sticky='n',
padx=0, pady=18)
    reference_labels.append(reference_label)


# Create class
class Watch:
    def __init__(self, brand, color, shape, diameter, wr,
reference, image_path):
        self.brand = brand
        self.color = color
        self.shape = shape
        self.diameter = diameter
        self.wr = wr
        self.reference = reference
        self.image_path = image_path
```

```python
# We add a "no results" image when the filter doesn't
match specific watches
no_results_path = "no_results.png"
no_results_original = Image.open(no_results_path)
no_results_resized = no_results_original.resize((130,
130))
no_results_image = ImageTk.PhotoImage(no_results_resized)

# Function to handle image filtering output
def search_button_click():
    selected_brand = dropdown_var1.get()
    selected_color = dropdown_var2.get()
    selected_shape = dropdown_var3.get()
    selected_diameter = dropdown_var4.get()
    selected_wr = dropdown_var5.get()

    # Filter watches based on selected options
    filtered_watches = []
    for watch in watches_list:
        if (selected_brand == "Any" or watch.brand ==
selected_brand) and \
                (selected_color == "Any" or watch.color
== selected_color) and \
                (selected_shape == "Any" or watch.shape
== selected_shape) and \
                (selected_diameter == "Any" or
watch.diameter == selected_diameter) and \
                (selected_wr == "Any" or watch.wr ==
selected_wr):
            filtered_watches.append(watch)

    # Calculate the start and end indices for the current
page
    start_index = current_page * 3
    end_index = start_index + 3

    # Take the watches for the current page
    selected_watches =
filtered_watches[start_index:end_index]

    # Clear the image labels and display placeholder
images
    for label in placeholder_labels:
        label.configure(image=placeholder_image)
        label.image = placeholder_image  # Store a
reference image

    # Update the placeholder labels with the images and
text of the watches
    for i, watch in enumerate(selected_watches):
        image_path = watch.image_path
        try:
            watch_image = Image.open(image_path)
            resized_image = watch_image.resize((130,
130))
            watch_photo =
ImageTk.PhotoImage(resized_image)

placeholder_labels[i].configure(image=watch_photo)
            placeholder_labels[i].image = watch_photo  #
Store a reference to prevent image garbage collection
        except FileNotFoundError as e:
            # Handle the exception if the image file is
not found

placeholder_labels[i].configure(image=no_results_image)
            placeholder_labels[i].image =
no_results_image  # Store a reference image
        # Update brand label
        brand_labels[i].configure(text=watch.brand)

        # Update reference label

reference_labels[i].configure(text=watch.reference)

    # Display "no results" image if no watches match the
filter
    if not selected_watches:
        for label in placeholder_labels:
            label.configure(image=no_results_image)
            label.image = no_results_image  # Store a
reference image

        suggestions_label.configure(text="No results
found")
```

```python
# Read the CSV file and create watch objects
watches_list = []
with open(csv_file_path, "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        brand = row["Brand"]
        color = row["Color"]
        shape = row["Shape"]
        diameter = row["Diameter"]
        wr = row["W/R"]
        reference = row["Reference"]
        image_file = row["image_file"]

        image_path = os.path.join(image_folder_path,
image_file)

        watch = Watch(brand, color, shape, diameter, wr,
reference, image_path)
        watches_list.append(watch)

# Shuffle the watches
random.shuffle(watches_list)

# Function to handle the next button click

# Define page start and finish for navigation
current_page = 0
max_pages = 3
def next_button_click():
    global current_page, max_pages
    if current_page < max_pages - 1:
        current_page += 1
        search_button_click()

# Function to handle the previous button click
def previous_button_click():
    global current_page
    if current_page > 0:
        current_page -= 1
        search_button_click()

# Create the search button
search_button = tk.Button(left_frame, text="Search",
font=('Arial', 12), bg='#526D82',
command=search_button_click)
search_button.grid(row=8, column=0, sticky='w', padx=10,
pady=20)

# Create the next and previous buttons
next_button = tk.Button(right_frame, text="Next",
font=('Arial', 12), bg='#526D82',
command=next_button_click, highlightthickness=0)
next_button.grid(row=4, column=1, sticky='e', padx=45,
pady=20)

previous_button = tk.Button(right_frame, text="Previous",
font=('Arial', 12), bg='#526D82',
command=previous_button_click, highlightthickness=0)
previous_button.grid(row=4, column=1, sticky='w', padx=15,
pady=15)

# Create the reinitialize button
def reinitialize_button_click():
    for label in placeholder_labels:
        label.configure(image=placeholder_image)
        label.image = placeholder_image  # Store a
reference image

    for brand_label in brand_labels:
        brand_label.configure(text="")

    for reference_label in reference_labels:
        reference_label.configure(text="")

    suggestions_label.configure(text="Here are some
suggestions:")

reinitialize_button = tk.Button(left_frame,
text="Reinitialize", font=('Arial', 12), bg='#526D82',
command=reinitialize_button_click)
reinitialize_button.grid(row=9, column=0, sticky='w',
padx=10, pady=0)

# Run the main window loop
window.mainloop()
```