Software Testing, Automation and the Job Market - a Data Programming Project

Table of contents

Research Questions, Project Limitations and Motivation

Research Area

The goal of this project is to analyse a given dataset from a large job board in regards to software testing roles with the emphasis of the analysis placed on what the most sought for skills are in order to shed some light on the ongoing debate on whether automation testing is going to replace manual testing (e.g **Can Manual Testing Be Completely Replaced by Automation Testing?**).

Research Questions

More specifically, this project aims to answer the following questions:

- What are the most popular skills required of software testers?
- Which programming languages are the most popular for test automation?

One caveat that has to be considered; however, is that the research questions may change depending on the final data set and that the answers to these questions may be incomplete, unavailable and limited in terms of depth. Still, there is room to learn from this as discussed in the next section.

Future Work / Project Limitations

Some questions that are interesting, but out of scope for this project are listed below:

- How many job adverts mention a combination of manual testing and test automation in their descriptions?
- How many job descriptions mention training / upskilling?
- How many job listings focus on either manual testing or test automation alone?

The reason why these problems are out of scope is because the data set used in this project is limited in terms of geographical region. Additionally, to consider the questions above, a much larger dataset would be required in order to come up with a sensible analysis. Indeed -- even with a larger dataset -- the work would still be limited to the English language, excluding analysis from countries such as Japan or South Korea -- regions which rely on software that is used on a day to day basis (and where testing is, thus, of paramount importance).

Of course, one could limit the project to a specific language or region, but this kind of project still requires a large dataset that takes into consideration where the jobs are located (rural areas may have less to offer than urban ones), what time period they were posted in, what seniority level they address and which area of testing they are focused on (e.g. performance testing). The deeper one dives into the topic, the more likely one is to find irregularities and different demands, which would make it difficult to generalise-- at least without performing some thorough analysis.

Also, the time required to answer such questions would be important as well: technology is in a constant state of flux, and it would be important to analyse job boards throughout a given period of several months, with the focus placed on the areas where change is happening the most and keeping an eye on the rate at which trends are growing.

Furthermore, my experience / abilities in data acquisition are still very much at a beginner's level, which prevents me from doing the complex and thorough research this project deserves.

Motivation

The motivation behind this specific area of research stems from my job as a QA where I have worked with both automation and manual testing tools. While I personally believe that a good software tester has skills in **both white box and black box testing**, I am keen to find out what philosophy the employment market is leaning towards with data from job boards serving as a good basis for me to embrace this topic on a practical level.

Moreover, I used to work for a software house that focused on gathering data for recruitment companies. During that time, I used to dabble a bit in job board searches in order to garner what the expectations / trends for software testers were. In fact, some of the observations related to the limits of the project come from direct exposure related to analysing data of such a nature.

Previous Exploration of the Topic

While **many articles** have been and continue to be published on the subject already, I have not seen any students discussing this topic in our university slack channel or express much interest in software testing otherwise. Of interest; however, are the **annual surveys carried out by Practitest** which analyses software testing trends and provides information on them. It showcases the versatility and importance of testing, especially with the rise of AI and development teams recognising the constant need for quality assurance.

Outside of the internet, there are **also many workshops and courses available on automation testing** with their main focus placed on upskilling manual testers, which proves how important test automation is as a skill set for both budding and experienced software testers alike.

Acquisition of the Data Set and Choice of Data Source

The datasets that were considered for this project came from job boards that have public APIs available for web crawling purposes.. In particular, the site **The Programmable Web** proved to be a valuable source for finding APIs that were free and did not require any prior set up.

Due to the fact that Indeed is the biggest job site aggregator, I decided to use it for my project, even though I would have preferred using a smaller site that provided guidance on how to use its API. However, I wanted to ensure that I would have sufficient data to perform some preliminary analysis, which smaller job smaller job site aggregators would not have been able to offer.

Ethical Implications

Scraping data as *the article* to ethical web scraping indicates can have ethical implications when dealing with sensitive data -- such as someone’s health records, for example. These implications are important to consider, especially because malicious agents could use such sensitive information for criminal purposes -- such as a fraud. Therefore, it is important to respect the owners of the provided API by asking for permission to use their data. If the former is not possible, then one should at least avoid scraping their site for data too often.

While I have not asked Indeed for explicit permission, I consider my use to be ethical since Indeed is an aggregator that collects job listings from other sites -- this information is public and intended to be shared with other job seekers and recruiters alike. There is no sensitive information in the job listings, other than the job being offered, the requirements and skills being sought and some general information about the company itself.

While the data could be used to create a competitive aggregator site, the data required for such a task would have to be massive and is outside of this project's scope, both in terms of my profesional/ programming experience and budget. Additionally, it should be noted that I am not interested in using the data for commercial purposes, but to satisfy the research questions outlined in this report.

As such, while the data shall be easily available within this Notebook, there is no intention to use said data beyonds its purpose of analysing the job market trends for software testers in the UK area. In regards to the frequency of the scraping, I scraped Indeed only a couple of times to gather the data I wished to have, thus avoiding any unnecessary overloading of its services.

Web Scraping

The purpose of this section is to highlight how the chosen data sets were gathered from the job site aggregator Indeed, with the code and any further comments documented below.

The Initial ScraperHelpers.py Script

Before setting out to do a more thorough scraping of the data, I wanted to play around with the basics of it, which is why I wrote two scripts that dealt with web crawling -- *scraperHelpers.py* and *main.py* . The former contains functions which help to scrape data from a given url, while the latter calls upon these methods in order to produce a txt file containing some relevant data.

```
from bs4 import BeautifulSoup
import requests
from typing import List
```

The script uses BeautifulSoup in order to parse web scraped data into a HTML file, which can then be further analysed and processed. Also, the request library is utilised in order to retrieve data from a given url. Both libraries were chosen because of their ease of use.

```
def retrieve_data_as_text(url: str) -> str:
    """
    Retrieves data from a url
    :param url:
    :return: data in form of a text
    """
    return requests.get(url, HEADERS).text
```

The *retrieve_data_as_text* method pulls out the text from the job advert which can be used to verify that the correct data is being collected.. The *retrieve_data_as_text* method is also reused in the *parse_data_into_html* method which is called in the main.py script.

```
def parse_data_into_html(url: str) -> BeautifulSoup:
    """
    Retrieves data in html format
    :param url: the url in string format
    :return: data parsed into HTML
```

```
    """
    data = retrieve_data_as_text(url)
    soup = BeautifulSoup(data, 'html.parser')
    return soup
```

The *parse_data_into_html* method retrieves the scraped web data as text and then – using the BeautifulSoup object – parses it into a HTML file. This file serves as the basis for all further processing and analysis.

```
def find_jobs_by_header_title(scraped_data: BeautifulSoup) -> List:
    """
    Finds jobs by header title
    :param scraped_data:
    :param scraped_data: the data to be retrieved
    :return: a list of job titles
    """
    jobs = []
    # code credit for text splitting:
    # @ https://www.geeksforgeeks.org/scraping-indeed-job-data-using-python
    for item in scraped_data.find_all("h2", class_="jobTitle"):
        data_str = "" + item.get_text()
        jobs.append(data_str.split("\n"))
    return jobs
```

The *find_jobs_by_header_title* method lists all the data that falls under the category of job titles and appends them to a list of job titles. This list is then used as input in the *save_jobs_as_txt* method below in order to create a text file.

```
def save_jobs_as_txt(jobs: List):
    """
    Saves job titles into a text file
    :param jobs: a list of jobs
    :return: returns a txt file containing job titles
    """
    with open('job_titles.txt', 'w') as f:
        f.write("\n".join(str(job) for job in jobs))
```

The methods mentioned above are called from the main script as shown below.

```
# !/usr/bin/python
# !python
from scraperHelpers import *

URL = 'https://uk.indeed.com/Remote-QA-jobs'

scraped_data = parse_data_into_html(URL)
jobs = find_jobs_by_header_title(scraped_data)
save_jobs_as_txt(jobs)
```

The Final ScraperHelpers Script

As with the previous code examples, these code samples were taken from the scripts written in PyCharm -- this was done in order to save time and focus on having the data ready to play with for future analysis. The two functions below are an extension to the ones depicted above. The full code is provided in the next section of this document.

```
#!/usr/bin/python
# !python
from bs4 import BeautifulSoup
import pandas as pd
```

```python
import requests
from typing import List

# global variables
PAGE_COUNT_ITR = 250
HEADERS = {"User-Agent": "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0)
Gecko/20100101 Firefox/88.0"}

def retrieve_pages_as_text() -> List:
    """
    Scrapes pages in the form of text
    :return: Returns a list of retrieved pages in the form of text
    """
    data_list = []
    page_count = 0
    while page_count <= PAGE_COUNT_ITR:
        url = f'https://uk.indeed.com/jobs?q=Remote%20QA&sort=date&start={page_count}'
        data_list.append(retrieve_data_as_text(url))
        page_count += 10
    return data_list
```

The *retrieve_pages_as_text* method returns a list of page texts. A while loop is used to increment the page count until a specific limit has been reached; the page count itself is added as a parameter to the URL.

```python
def extract_job_descriptions() -> List:
    """
    @Credit for this piece of code goes to:
    https://stackoverflow.com/questions/67504953/
    how-to-get-full-job-descriptions-from-indeed-using-python-and-beautifulsoup
    Any modifications are mine and mine alone

    Retrieve job summaries from multiple pages
    :return: a list of job descriptions
    """
    job_summaries = []
    api_url = "https://uk.indeed.com/viewjob?viewtype=embedded&jk={job_id}"
    url = "https://uk.indeed.com/jobs?q=Remote%20QA"
    scraped_data = BeautifulSoup(requests.get(url, headers=HEADERS).content,
"html.parser")

    for job in scraped_data.select('a[id^="job_"]'):
        job_id = job["id"].split("_")[-1]
        scraped_job_data = BeautifulSoup(requests.get(api_url.format(job_id=job_id),
                                                       headers=HEADERS).content,
"html.parser")
        job_description =
scraped_job_data.select_one("#jobDescriptionText").get_text(strip=True)
        job_summaries.append(job_description)

    return job_summaries
```

The *extract_job_descriptions* method returns a list of job summaries extracted from a single page on Indeed. Headers are used in order to retrieve more than one job id.

```python
def parse_page_data_into_html() -> List:
    """
    Retrieves page data in html format
    :return: parsed page data as HTML supplied into a list
    """
```

```
    data_list = retrieve_pages_as_text()
    soup = []
    for data in data_list:
        soup.append(BeautifulSoup(data, 'html.parser'))
    return soup
```

The *parse_page_data_into_html* method above returns a list of parsed HTML data which is further used to extract job titles from a given number of pages.

```
# noinspection PyTypeChecker
def save_summaries_as_csv(extracted_job_titles: List):
    """
    Save the job summaries into a csv file
    :param extracted_job_titles:
    :return: a csv file containing job summaries
    """
    data_frame = pd.DataFrame(extracted_job_titles)
    data_frame.to_csv("job_descriptions.csv")


# noinspection PyTypeChecker
def save_titles_as_csv(extracted_job_titles: List):
    """
    Saves the extracted job titles into a csv file
    :param extracted_job_titles:
    :return: a csv file containing job titles
    """
    data_frame = pd.DataFrame(extracted_job_titles)
    data_frame.to_csv("job_titles.csv")

def find_jobs_by_summary(scraped_data: BeautifulSoup) -> List:
    """
    Finds jobs by the summary
    :param scraped_data:
    :param scraped_data: the data to be retrieved
    :return: a list of job summaries
    """
    jobs = []
    # code credit for text splitting:
    # @ https://www.geeksforgeeks.org/scraping-indeed-job-data-using-python
    for item in scraped_data.find_all("div", class_="job-snippet"):
        data_str = "" + item.get_text()
        jobs.append(data_str.split("\n"))
    return jobs
```

Full Code

The next cell provides the full scripts, commented out where necessary to prevent any errors while running the Juptyer notebook.

In [20]:

```
# #!/usr/bin/python
# # !python
from bs4 import BeautifulSoup
import pandas as pd
import requests
from typing import List

# global variables
PAGE_COUNT_ITR = 250
```

```
HEADERS = {"User-Agent": "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0) Gecko/20100101


def retrieve_pages_as_text() -> List:
    """
    Scrapes pages in the form of text
    :return: Returns a list of retrieved pages in the form of text
    """
    data_list = []
    page_count = 0
    while page_count <= PAGE_COUNT_ITR:
        url = f'https://uk.indeed.com/jobs?q=Remote%20QA&sort=date&start={page_count}'
        data_list.append(retrieve_data_as_text(url))
        page_count += 10
    return data_list


def extract_job_descriptions() -> List:
    """
    @Credit for this piece of code goes to:
    https://stackoverflow.com/questions/67504953/how-to-get-full-job-descriptions-from-inc
    Any modifications are mine and mine alone

    Retrieve job summaries from multiple pages
    :return: a list of job descriptions
    """
    job_summaries = []
    api_url = "https://uk.indeed.com/viewjob?viewtype=embedded&jk={job_id}"
    url = "https://uk.indeed.com/jobs?q=Remote%20QA"
    scraped_data = BeautifulSoup(requests.get(url, headers=HEADERS).content, "html.parser'

    for job in scraped_data.select('a[id^="job_"]'):
        job_id = job["id"].split("_")[-1]
        scraped_job_data = BeautifulSoup(requests.get(api_url.format(job_id=job_id),
                                                      headers=HEADERS).content, "html.pars
        job_description = scraped_job_data.select_one("#jobDescriptionText").get_text(stri
        job_summaries.append(job_description)

    return job_summaries


def find_jobs_by_summary(scraped_data: BeautifulSoup) -> List:
    """
    Finds jobs by the summary
    :param scraped_data:
    :param scraped_data: the data to be retrieved
    :return: a list of job summaries
    """
    jobs = []
    # code credit for text splitting:
    # @ https://www.geeksforgeeks.org/scraping-indeed-job-data-using-python
    for item in scraped_data.find_all("div", class_="job-snippet"):
        data_str = "" + item.get_text()
        jobs.append(data_str.split("\n"))
    return jobs


def retrieve_data_as_text(url: str) -> str:
    """
    Retrieves data from a url
    :param url:
    :return: data in form of a text
    """
    return requests.get(url, HEADERS).text
```

```python
def parse_data_into_html(url: str) -> BeautifulSoup:
    """
    Retrieves data in html format
    :param url: the url in string format
    :return: data parsed into HTML
    """
    data = retrieve_data_as_text(url)
    soup = BeautifulSoup(data, 'html.parser')
    return soup


def parse_page_data_into_html() -> List:
    """
    Retrieves page data in html format
    :return: parsed page data as HTML supplied into a list
    """
    data_list = retrieve_pages_as_text()
    soup = []
    for data in data_list:
        soup.append(BeautifulSoup(data, 'html.parser'))
    return soup


def find_jobs_by_header_title(scraped_data: BeautifulSoup) -> List:
    """
    Finds jobs by header title
    :param scraped_data:
    :param scraped_data: the data to be retrieved
    :return: a list of job titles
    """
    jobs = []
    # code credit for text splitting:
    # @ https://www.geeksforgeeks.org/scraping-indeed-job-data-using-python
    for item in scraped_data.find_all("h2", class_="jobTitle"):
        data_str = "" + item.get_text()
        jobs.append(data_str.split("\n"))
    return jobs


def save_jobs_as_txt(jobs: List):
    """
    Saves job titles into a text file
    :param jobs: a list of jobs
    :return: returns a txt file containing job titles
    """
    with open('job_titles.txt', 'w') as f:
        f.write("\n".join(str(job) for job in jobs))


# noinspection PyTypeChecker
def save_summaries_as_csv(extracted_job_titles: List):
    """
    Save the job summaries into a csv file
    :param extracted_job_titles:
    :return: a csv file containing job summaries
    """
    data_frame = pd.DataFrame(extracted_job_titles)
    data_frame.to_csv("job_descriptions.csv")


# noinspection PyTypeChecker
def save_titles_as_csv(extracted_job_titles: List):
    """
    Saves the extracted job titles into a csv file
    :param extracted_job_titles:
    :return: a csv file containing job titles
```

```
    """
    data_frame = pd.DataFrame(extracted_job_titles)
    data_frame.to_csv("job_titles.csv")

# # !/usr/bin/python
# # !python
# from scraperHelpers import *

parsed_pages = parse_page_data_into_html()

extracted_job_titles = []
extracted_job_descriptions = []
extracted_job_snippets =[]

## The below is commented out

# for page in parsed_pages:
    # extracted_job_titles.append(find_jobs_by_header_title(page))
    # extracted_job_snippets.append(find_jobs_by_summary(page))

# save_jobs_as_txt(extracted_job_titles)
# job_summaries = extract_job_descriptions()
# save_summaries_as_csv(extracted_job_snippets)
# save_titles_as_csv(extracted_job_titles)
```

Sampler Code to Demonstrate the Web Scraper Code

The code in the next cell demonstrates some of the methods from the sections above to showcase how data can be scraped from websites. Comment out the lines underneath the function to see a sample HTML page in a prettified form.

In [11]:
```
from bs4 import BeautifulSoup
import requests

HEADERS = {"User-Agent": "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0) Gecko/20100101

def retrieve_data_as_text(url: str) -> str:
    """
    Retrieves data from a url
    :param url:
    :return: data in form of a text
    """
    return requests.get(url, HEADERS).text

def parse_data_into_html(url: str) -> BeautifulSoup:
    """
    Retrieves data in html format
    :param url: the url in string format
    :return: data parsed into HTML
    """
    data = retrieve_data_as_text(url)
    soup = BeautifulSoup(data, 'html.parser')
    return soup

url = "https://uk.indeed.com/jobs?q=Remote%20QA"
# scraped_data = parse_data_into_html(url)
# print(scraped_data.prettify())
```

Lessons Learnt from Web Scraping

Although web scraping was a successful venture for the most part, there were a few Eureka moments that have made me draw the following conclusions:

- Descriptions are hard to scrape, with the quality of the data that was retrieved being something of an issue. High expectations are to be avoided.
- Tutorials online are only useful as long as they do not advertise some service that has to be paid for.
- Moreover, it is easier to come across tutorials that teach sheer basics rather than a solution to a more sophisticated problem.
- Cleaning up the data before storing it in its final format is important as it saves a lot of hassle during the later cleaning process.

Cleaning Up Data

The following section is concerned with the cleaning up of data, with the code executed on the notebook itself. The assumption is made that the data is stored in the same directory. If the code is not working, please consider replacing the relative file paths with their absolute ones.

Cleaning Up Data Code

The section below displays the code used for cleaning up the data sets. It is very much hardcoded and a bit brittle, due to unexpected complexities arising from the expectations of what the final data should look like.

In [21]:

```
# please ensure that all the files are located within the same directory

# import os in order to get the current directory
import os
cwd = os.getcwd()

JOB_TITLES_PATH = os.path.join(cwd, "job_titles.csv") # if necessary, use a string with th
JOB_SNIPPETS_PATH = os.path.join(cwd, "job_descriptions.csv") # if necessary, use a string

# import the necessary libraries for data cleaning
import io
import pandas as pd
import re
from typing import List

# Read the data set in their raw and unedited format
job_titles_data = pd.read_csv(JOB_TITLES_PATH)
job_snippets_data = pd.read_csv(JOB_SNIPPETS_PATH)

# The function carries out some actions that clean up the csv file
def clean_data(data_frame : pd.DataFrame):
    # remove square brackets
    data_frame_modified = job_titles_data.replace(to_replace ="[\([{})\]]", value = ",", r

    # remove all digits
    pattern_with_numbers_removed = ''.join([i for i in data_frame_modified.to_string() if

    # remove unnecessary space and commas
    pattern_with_all_numbers_removed = re.sub(' +', '', pattern_with_numbers_removed)
    pattern_with_commas_removed = re.sub(',,', ',', pattern_with_all_numbers_removed)

    # remove special characters
    pattern_with_ampersand_removed = re.sub('&', ',', pattern_with_commas_removed)
    pattern_with_special_char_removed = re.sub('#', ',', pattern_with_ampersand_removed)

    # remove unnamed
    final_data_str = re.sub('Unnamed:', 'JOB TITLES', pattern_with_special_char_removed)
```

```
    return final_data_str

# We can use the list below to perform some interesting analysis
modified_job_titles_data = clean_data(job_titles_data)

# print(modified_job_titles_data)

# Unfortunately, the clean up function when too far for the job snippets, leaving it
# to look identical to the job titles. Comment out the code below for 'proof'

# print(clean_data(job_snippets_data))
```

Job Snippets Data: Data Cleaning Problem

Unfortunately, the job snippets data cannot be further cleaned. In fact, cleaning led to it being identical to the job titles data, which is why it has been disregarded from any further analysis as the results would be the same / not provide any further valuable information.

Lessons Learnt from Data Cleaning

The takeaway lessons from the data cleaning attempts are:

- Cleaning up data is a far more involved process than expected, with the coding being very much hands on / learning while doing.
- In the same vein, garbage data will be collected which can be seen as creating redundancy, thus creating an impact on any future analysis.
- There is no guarantee that data will not get lost when cleaning up the csv file / removing white spaces.
- A larger project would require far more planning in terms of cleaning up data algorithms. A naive attempt should be avoided.

Natural Language Processing Analysis Methods: Frequency of Job Titles

This section performs some analysis using word clouds and other NLP-related methods, listing advantages and drawbacks for each respective approach. The reason why NLP methods were used is due to the fact that the frequency of skills within job titles can be displayed using such techniques. Moreover, the visual representation offered by word clouds and other graphs provides a more tangible approach to the problem.

In [14]:

```
### Natural Language Processing  - installing the necessary dependencies

import sys
!{sys.executable} -m pip install nltk
!{sys.executable} -m pip install wordcloud
```

```
Requirement already satisfied: nltk in c:\users\anita.pal\anaconda3\lib\site-packages (3.
6.5)
Requirement already satisfied: click in c:\users\anita.pal\anaconda3\lib\site-packages (fr
om nltk) (8.0.3)
Requirement already satisfied: joblib in c:\users\anita.pal\anaconda3\lib\site-packages (f
rom nltk) (1.1.0)
Requirement already satisfied: regex>=2021.8.3 in c:\users\anita.pal\anaconda3\lib\site-pa
ckages (from nltk) (2021.8.3)
Requirement already satisfied: tqdm in c:\users\anita.pal\anaconda3\lib\site-packages (fro
m nltk) (4.62.3)
Requirement already satisfied: colorama in c:\users\anita.pal\anaconda3\lib\site-packages
(from click->nltk) (0.4.4)
```

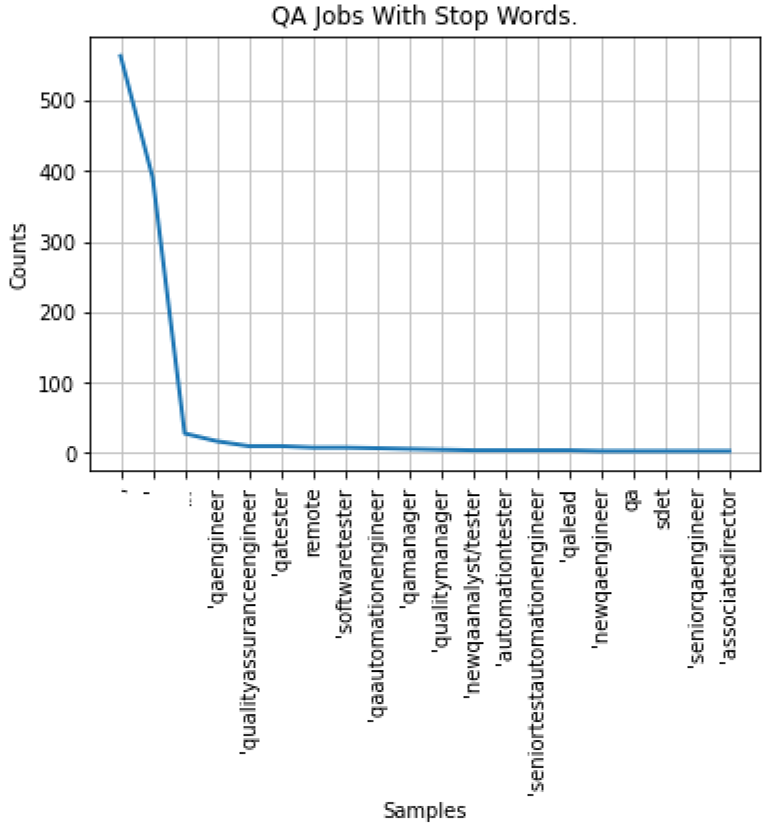
```
Requirement already satisfied: wordcloud in c:\users\anita.pal\anaconda3\lib\site-packages
(1.8.1)
Requirement already satisfied: numpy>=1.6.1 in c:\users\anita.pal\anaconda3\lib\site-packa
ges (from wordcloud) (1.20.3)
Requirement already satisfied: pillow in c:\users\anita.pal\anaconda3\lib\site-packages (f
rom wordcloud) (8.4.0)
Requirement already satisfied: matplotlib in c:\users\anita.pal\anaconda3\lib\site-package
s (from wordcloud) (3.4.3)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\anita.pal\anaconda3\lib\site-
packages (from matplotlib->wordcloud) (1.3.1)
Requirement already satisfied: pyparsing>=2.2.1 in c:\users\anita.pal\anaconda3\lib\site-p
ackages (from matplotlib->wordcloud) (3.0.4)
Requirement already satisfied: cycler>=0.10 in c:\users\anita.pal\anaconda3\lib\site-packa
ges (from matplotlib->wordcloud) (0.10.0)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\anita.pal\anaconda3\lib\si
te-packages (from matplotlib->wordcloud) (2.8.2)
Requirement already satisfied: six in c:\users\anita.pal\anaconda3\lib\site-packages (from
cycler>=0.10->matplotlib->wordcloud) (1.16.0)
```

Plotting the Twenty Most Frequent Job Titles

The focal point of this project is to analyse the frequency of job titles within the data set. One approach is to use the nlkt library to plot all tokens in the modified job titles list to showcase the frequency of words found within it. Unfortunately, the nature of the dataset is such that there is no difference between using stopwords or not.

In [22]:

```python
import nltk
from nltk.corpus import stopwords
from nltk.probability import FreqDist

sentence = modified_job_titles_data.lower()
tokens = nltk.tokenize.word_tokenize(sentence)

# creating a list without stop words
filtered = [word for word in tokens if not word in stopwords.words()]
# print(filtered)

# Plotting data, with stop words
import matplotlib.pyplot as plt

fd_stopwords = nltk.FreqDist(tokens)
fd_stopwords.plot(20, cumulative=False, title="QA Jobs With Stop Words.")

# Plotting data without stop words
# The two graphs are commented out as -- spoiler alert -- they are identical to the ones a

# fd_no_stopwords = nltk.FreqDist(filtered)
# fd_no_stopwords.plot(20, cumulative=False, title="QA Jobs With No Stop Words.")
```



Out[22]:
```
<AxesSubplot:title={'center':'QA Jobs With Stop Words.'}, xlabel='Samples', ylabel='Count
s'>
```

Advantages and Drawbacks to Plotting

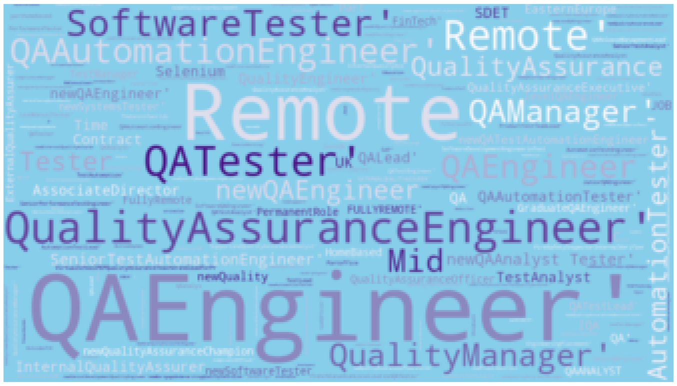
The advantage of the plotting method is that the data is presented in a user-friendly format, showing the frequency of job titles in a way that is easy to comprehend. The fact that the graph contains a numerical representation of how often certain words appear is useful for further analysis.

Despite that; however, some garbage words have made it into the graph. While this is due to the data set and its imperfect state, this does showcase how important data cleaning is in order to avoid problems of such a nature in the future.

Job Titles as a WordCloud

The cell below displays job titles as a word cloud, which was generated using the WordCloud library.

In [23]:
```python
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# generating the word cloud, with some specific properties
wordcloud = WordCloud(width=850, height=480, background_color="skyblue", colormap="Purples

# Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```



Advantages and Drawbacks to the Word Cloud

One of the advantages to the word cloud is that it presents information in a manner that is very aesthetic. Furthemore, the word cloud can be customised to one's taste.

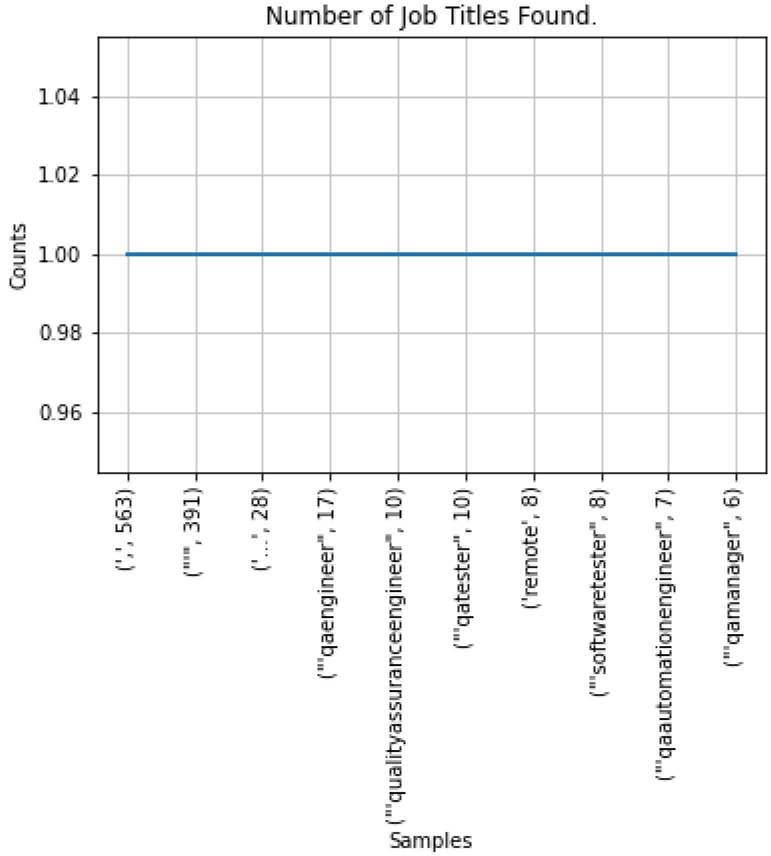
However, the disadvantage of the word cloud is that it showcases the frequency of words by depicting its size in a visual fashion and not via a numerical count. This makes the word cloud more useful for illustrative purposes rather than an in-depth analysis.

Most Frequent Job Titles Calculated Using the Counter Class

The method below showcases how to get the most frequent job titles using the Counter library.

In [24]:

```
from collections import Counter

# Count the frequencies using the Counter method
frequencies = Counter(tokens)
ten_most_frequent = frequencies.most_common(10)

# print(ten_most_frequent)

# plot a frequency graph
job_titles_freq = nltk.FreqDist(ten_most_frequent)
job_titles_freq.plot(len(ten_most_frequent), cumulative=False, title="Number of Job Titles
```



Out[24]:
```
<AxesSubplot:title={'center':'Number of Job Titles Found.'}, xlabel='Samples', ylabel='Cou
nts'>
```

Advantages and Drawbacks to the Counter Library

The advantage of using the Counter library is that it comes with a count value that states how often a certain word occurs. This is more precise than the previous methods, because it gives a good starting point for further analysis. Moreover, it can be combined with the plotting method used before, which provides visual as well as numerical information.

One of the obvious drawbacks is that rubbish data has been collected again due to the flawed way the data was cleaned.

Conclusions Drawn from the NLP Analysis

Some of the tentative conclusions from the NLP analysis – despite the existence of garbage words - are:

- The fact that the words ‘automation’ and ‘sdet’ (Software Development Engineer in Test) are among the most frequent suggests that test automation is popular.
- The occurrence of words such as ‘manager’, ‘lead’ and ‘senior’ suggests that software testers with a lot of experience are valuable to employers / recruiters. It can be assumed that testers with experience are preferable to ones with none.
- Quality assurance is associated with engineering, suggesting that software testing is something that is created – just like programming, which strengthens its link to automation (a skill that requires creating tests).

Plotting the Frequency of Skills and Programming Languages

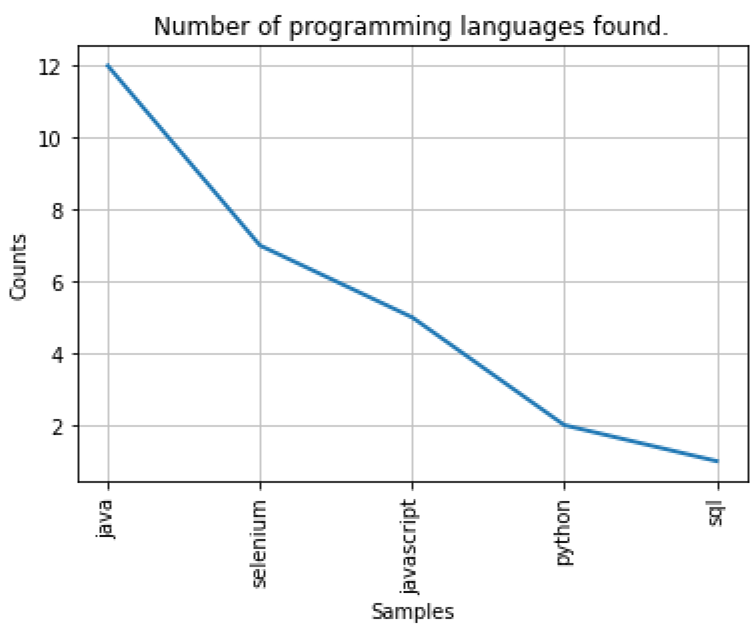
This section deals with analysing the frequency of certain skills and programming languages found within the existing job titles data set.

Plotting the Frequency of Programming Languages

The cell belows provides a method that iterates through a given list of programming languages.

In [25]:

```
# function that checks whether a token contains a given programming language
def find_programming_languages(tokens: List) -> List:

    # list containing programming languages
    programming_languages = ['selenium', 'java', 'c#', 'python',
                             'c++', 'ruby', 'javascript', 'sql']

    # empty found languages list
    found_languages = []

    # check whether token is equal to or contains one
    # of the programming languages above
    for token in tokens:
        for language in programming_languages:
            if token.lower() == language or language in token.lower():

                # if token contains or is equal to a programming language, append it to tl
                # empty found languages list
                found_languages.append(language)

    return found_languages

languages_contained = find_programming_languages(tokens)

# plot a frequency graph
number_of_programming_languages = nltk.FreqDist(languages_contained)
number_of_programming_languages.plot(len(number_of_programming_languages), cumulative=Fals
```



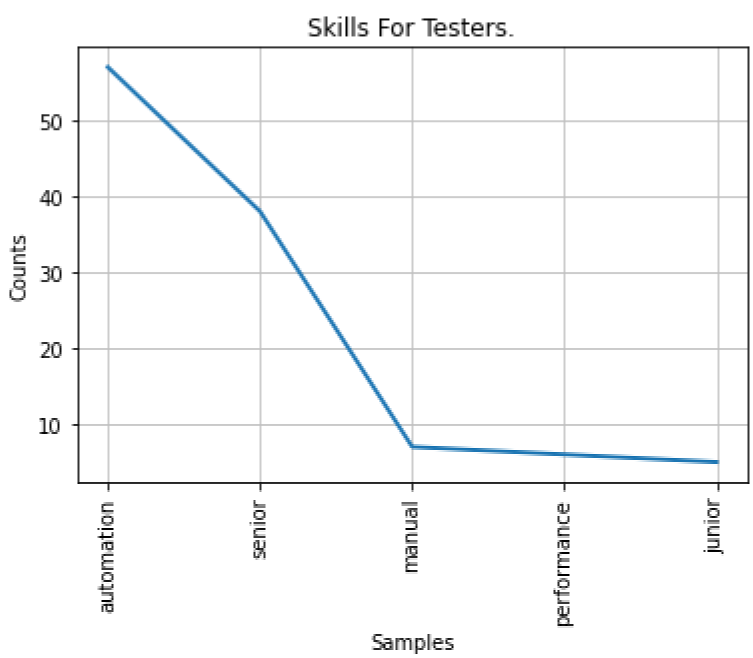
Out[25]:

```
<AxesSubplot:title={'center':'Number of programming languages found.'}, xlabel='Samples', ylabel='Counts'>
```

Plotting the Frequency of Skills

The method from above has been modified to run through an array of skills rather than programming languages.

In [10]:
```
# function that checks whether a token contains a certain skill
def find_freq_of_skills(tokens: List) -> List:

    # list containing QA skills
    skills = ['automation', 'manual', 'regression', 'code', 'performance', 'senior', 'juni

    # empty found skills list
    found_skills = []

    # check whether token is equal to or contains one
    # of the programming languages above
    for token in tokens:
        for skill in skills:
            if token.lower() == skill or skill in token.lower():

                # if token contains or is equal to a programming language, append it to th
                # empty found languages list
                found_skills.append(skill)

    return found_skills

skills_within_titles = find_freq_of_skills(tokens)

# plot a frequency graph
number_of_skills_in_titles = nltk.FreqDist(skills_within_titles)
number_of_skills_in_titles.plot(len(number_of_skills_in_titles), cumulative=False, title='
```



Out[10]: `<AxesSubplot:title={'center':'Skills For Testers.'}, xlabel='Samples', ylabel='Counts'>`

Conclusions Drawn from the NLP Analysis in Regards to Skills

Considering the limited range, the following conclusions should be viewed as speculative. Some of these observations have been noted in previous sections:

- The popularity of Java and Selenium implies that test automation is an important part of the software testing process.
- The existence of SQL as an important programming language suggests that software testing is also concerned with how data is stored and / or processed within a database.

- The fact that Python and Java are used for automation showcases that open source is preferred over using tools that have to be paid for.
- The fact that one of the most sought after skills is automation suggests that software testing is becoming intertwined with test automation.
- However, the fact that manual testing is within the medium range of skills within the skills list suggests that it is integral to the overall process as well.
- The word senior occurs more often than junior, suggesting that experience is preferred over lack thereof.
- The fact that the word performance shows up implies that specialist knowledge in this area is also considered important by the job market.

Final Conclusions

Overall, this project was rewarding in the sense that some answers could be found re: the most popular languages and skills required of software testers. While these answers need to be expanded upon, this project does offer some groundwork for the future.

In [111…

```
```

In [ ]:

```
```

In [ ]:

```
```