Python Projects

With API

Minal Pandey

PART 1: Introducing APIs

What is an API?

If you've heard the term API before, chances are it's been used not to refer to APIs in general, but instead to a specific kind of API, the web API. A web API allows for information or functionality to be manipulated by other programs via the internet. For example, with Twitter's web API, you can write a program in a language like Python or Javascript that can perform tasks such as favoriting tweets or collecting tweet metadata.

In programming more generally, the term API, short for Application Programming Interface, refers to a part of a computer program designed to be used or manipulated by another program, as opposed to an interface designed to be used or manipulated by a human. Computer programs frequently need to communicate amongst themselves or with the underlying operating system, and APIs are one way they do it. In this tutorial, however, we'll be using the term API to refer specifically to web APIs.

When to Create an API

In general, consider an API if:

1. Your data set is large, making download via FTP unwieldy or resource-intensive.
2. Your users will need to access your data in real time, such as for display on another website or as part of an application.
3. Your data changes or is updated frequently.
4. Your users only need access to a part of the data at any one time.
5. Your users will need to perform actions other than retrieve data, such as contributing, updating, or deleting data.

If you have data you wish to share with the world, an API is one way you can get it into the hands of others. However, APIs are not always the best way of sharing data with users. If the size of the

data you are providing is relatively small, you can instead provide a "data dump" in the form of a downloadable JSON, XML, CSV, or SQLite file. Depending on your resources, this approach can be viable up to a download size of a few gigabytes.

Remember that you can provide both a data dump and an API, and individual users may find one or the other to better match their use case. Open Library, for example, provides both a data dump and an API, each of which serves different use cases for different users.

API Terminology

When using or building APIs, you will encounter these terms frequently:

- **HTTP (Hypertext Transfer Protocol)** is the primary means of communicating data on the web. HTTP implements a number of "methods," which tell which direction data is moving and what should happen to it. The two most common are GET, which pulls data from a server, and POST, which pushes new data to a server.
- **URL (Uniform Resource Locator)** - An address for a resource on the web, such as `https://programminghistorian.org/about`. A URL consists of a **protocol** (`http://`), domain (`programminghistorian.org`), and optional **path** (`/about`). A URL describes the location of a specific resource, such as a web page. When reading about APIs, you may see the terms `URL`, `request`, `URI`, or `endpoint` used to describe adjacent ideas. This tutorial will prefer the terms URL and request to avoid complication. You can follow a URL or make a GET request in your browser, so you won't need any special software to make requests in this tutorial.
- **JSON (JavaScript Object Notation)** is a text-based data storage format that is designed to be easy to read for both humans and machines. JSON is generally the most common format for returning data through an API, XML being the second most common.

- **REST (REpresentational State Transfer)** is a philosophy that describes some best practices for implementing APIs. APIs designed with some or all of these principles in mind are called REST APIs. While the API outlined in this lesson uses some REST principles, there is a great deal of disagreement around this term. For this reason, I do not describe the example APIs here as REST APIs, but instead as web or HTTP APIs.

Using APIs

Why Use APIs as a Researcher?

The primary focus of this lesson is on creating an API, not exploring or using an API that has already been implemented. However, before we start building our own API, it may be useful to discuss how APIs are useful for researchers. In this section, we'll see how APIs can be useful for approaching historical, textual, or sociological questions using a "macroscopic" or "distant reading" approach that makes use of relatively large amounts of information. In doing so, we'll familiarize ourselves with the basic elements of a good API. Considering APIs from the perspective of a user will come in useful when we begin to design our own API later in the lesson.

An API Case Study: Sensationalism and Historical Fires

Imagine that our research area is sensationalism and the press: has newspaper coverage of major events in the United States become more or less sensational over time? Narrowing the topic, we might ask whether press coverage of, for example, urban fires has increased or decreased with government reporting on fire-related relief spending.

While we won't be able to explore this question thoroughly, we can begin to approach this research space by collecting historical data

on newspaper coverage of fires using an API—in this case, the Chronicling America Historical Newspaper API. The Chronicling America API allows access to metadata and text for millions of scanned newspaper pages. In addition, unlike many other APIs, it also does not require an authentication process, allowing us to immediately explore the available data without signing up for an account.

Our initial goal in approaching this research question is to find all newspaper stories in the Chronicling America database that use the term "fire." Typically, use of an API starts with its documentation. On the Chronicling America API page, we find two pieces of information critical for getting the data we want from the API: the API's **base URL** and the **path** corresponding to the function we want to perform on the API—in this case, searching the database.

Our base URL is:

```
http://chroniclingamerica.loc.gov
```

All requests we make to the API must begin with this portion of the URL. All APIs have a base URL like this one that is the same across all requests to the API.

Our path is:

```
/search/pages/results/
```

If we combine the base URL and the path together into one URL, we'll have created a request to the Chronicling America API that returns all available data in the database:

```
http://chroniclingamerica.loc.gov/search/pages/results/
```

If you visit the link above, you'll see all items available in Chronicling America (12,243,633 at the time of writing), , not just the entries related to our search term, "fire." This request also returns a formatted HTML view, rather than the structured view we want to use to collect data.

According to the Chronicling America documentation, in order to get structured data specifically relating to fire, we need to pass one more kind of data in our request: **query parameters**.

```
http://chroniclingamerica.loc.gov/search/pages/results/?format=json&proxtext=fire
```

The query parameters follow the ? in the request, and are seperated from one another by the & symbol. The first query parameter, format=json, changes the returned data from HTML to JSON. The second, proxtext=fire, narrows the returned entries to those that include our search term.

If you follow the above link in your browser, you'll see a structured list of the items in the database related to the search term "fire." The format of the returned data is called JSON, and is a structured format that looks like this excerpt from the Chronicling America results:


```
"city": [
    "Washington"
  ],
  "date": "19220730",
  "title": "The Washington Herald.",
  "end_year": 1939,
```


By making requests to the Chronicling America API, we've accessed information on news stories that contain the search term "fire," and returned data that includes the date of publication and the page the article appears on. If we were to pursue this research question further, a next step might be finding how many stories relating to fire appear on a newspaper's front page over time, or perhaps cleaning the data to reduce the number of false positives. As we have seen, however, exploring an API can be a useful first step in gathering data to tackle a research question.

What Users Want in an API

As we've learned, documentation is a user's starting place when working with a new API, and well-designed URLs make it easier for users to intuitively find resources. Because they help users to quickly access information through your API, these elements—documentation and well-conceived URLs—are the *sine qua non* of a good API. We'll discuss these elements in greater depth later in this tutorial.

As you use other APIs in your research, you'll develop a sense of what makes a good API from the perspective of a potential user. Just as strong readers often make strong writers, using APIs created by others and critically evaluating their implementation and documentation will help you better design your own APIs.

Implementing Our API

Overview

This section will show you how to build a prototype API using Python and the Flask web framework. Our example API will take the form of a distant reading archive—a book catalog that goes beyond standard bibliographic information to include data of interest to those working on digital projects. In this case, besides title and date of publication, our API will also serve the first sentence of each book. This should be enough data to allow us to envision some potential research questions without overwhelming us as we focus on the design of our API.

We'll begin by using Flask to create a home page for our site. In this step, we'll learn the basics of how Flask works and make sure our software is configured correctly. Once we have a small Flask application working in the form of a home page, we'll iterate on this site, turning it into a functioning API.

PART 2: Python Make a Currency Converter in

Learn how to make a real-time currency converter using different ways and from various sources such as xe, yahoo finance, xrates and Fixer API in Python.

A currency converter is an app or tool that allows you to quickly convert from one currency to another. We can easily find such tools for free on the Internet. This tutorial will make a real-time currency converter using several methods utilizing web scraping techniques and APIs.

This tutorial will cover five different ways to get the most recent foreign exchange rates, some of them parse the rates from public web pages such as X-RATES and Xe, and others use official APIs for more commercial and reliable use, such as Fixer API and ExchangeRate API, feel free to use any one of these.

Feel free to jump into the method you want to use:

- Scraping X-RATES
- Scraping Xe
- Scraping Yahoo Finance
- Using ExchangeRate API
- Using Fixer API

To get started, we have to install the required libraries for all the methods below:

```
$ pip install python-dateutil requests bs4 yahoo_fin
```

Scraping X-RATES

In this section, we will extract the data from the x-rates.com website. If you go to the target web page, you'll see most of the currencies along with the most recent date and time. Let's scrape the page:

```
import requests
from bs4 import BeautifulSoup as bs
from dateutil.parser import parse
from pprint import pprint

```

The following function is responsible for making a request to that page and extracting the data from the tables:

```
def get_exchange_list_xrates(currency, amount=1):
    # make the request to x-rates.com to get current exchange rates for common currencies
    content = requests.get(f"https://www.x-rates.com/table/?from={currency}&amount={amount}").content
    # initialize beautifulsoup
    soup = bs(content, "html.parser")
    # get the last updated time
    price_datetime = parse(soup.find_all("span", attrs={"class": "ratesTimestamp"})[1].text)
    # get the exchange rates tables
    exchange_tables = soup.find_all("table")
    exchange_rates = {}
    for exchange_table in exchange_tables:
        for tr in exchange_table.find_all("tr"):
```

```
            # for each row in the table
            tds = tr.find_all("td")
            if tds:
                currency = tds[0].text
                # get the exchange rate
                exchange_rate = float(tds[1].text)
                exchange_rates[currency] = exchange_rate
    return price_datetime, exchange_rates
```

The above function takes the currency and the amount as parameters and returns the exchange rates of most currencies along with the date and time of the last update.

The time of the last update is in a `span` tag that has the class of `ratesTimestamp`. Notice we use the `parse()` function from `dateutil.parser` module to automatically parse the string into a Python DateTime object.

The exchange rates are located in two tables. We extract them using the `find_all()` method from the BeautifulSoup object and get the currency name and the exchange rate in each row in the tables, and add them to our `exchange_rates` dictionary that we will return. Let's use this function:

```
if __name__ == "__main__":
    import sys
    source_currency = sys.argv[1]
    amount = float(sys.argv[3])
    target_currency = "GBP"
    price_datetime, exchange_rates = 
get_exchange_list_xrates(source_currency, amount)
```

```
    print("Last updated:", price_datetime)
    pprint(exchange_rates)
```

Excellent, we use the built-in `sys` module to get the target currency and the amount from the command line. Let's run this:

```
$ python currency_converter_xrates.py EUR 1000
```

The above run is trying to convert 1000 Euros to all other currencies. Here is the output:

```
Last updated: 2022-02-01 12:13:00+00:00
{'Argentine Peso': 118362.205708,
 'Australian Dollar': 1586.232315,
 'Bahraini Dinar': 423.780164,
 'Botswana Pula': 13168.450636,
 'Brazilian Real': 5954.781483,
 'British Pound': 834.954104,
 'Bruneian Dollar': 1520.451015,
 'Bulgarian Lev': 1955.83,
 'Canadian Dollar': 1430.54405,
 'Chilean Peso': 898463.818465,
 'Chinese Yuan Renminbi': 7171.445692,
 'Colombian Peso': 4447741.922165,
 'Croatian Kuna': 7527.744707,
 'Czech Koruna': 24313.797041,
 'Danish Krone': 7440.613895,
 'Emirati Dirham': 4139.182587,
 'Hong Kong Dollar': 8786.255952,
```

'Hungarian Forint': 355958.035747,
'Icelandic Krona': 143603.932438,
'Indian Rupee': 84241.767127,
'Indonesian Rupiah': 16187150.010697,
'Iranian Rial': 47534006.535121,
'Israeli Shekel': 3569.191411,
'Japanese Yen': 129149.364679,
'Kazakhstani Tenge': 489292.515538,
'Kuwaiti Dinar': 340.959682,
'Libyan Dinar': 5196.539901,
'Malaysian Ringgit': 4717.485104,
'Mauritian Rupee': 49212.933037,
'Mexican Peso': 23130.471272,
'Nepalese Rupee': 134850.008728,
'New Zealand Dollar': 1703.649473,
'Norwegian Krone': 9953.078431,
'Omani Rial': 433.360301,
'Pakistani Rupee': 198900.635421,
'Philippine Peso': 57574.278782,
'Polish Zloty': 4579.273862,
'Qatari Riyal': 4102.552652,
'Romanian New Leu': 4946.638369,
'Russian Ruble': 86197.012666,
'Saudi Arabian Riyal': 4226.530892,
'Singapore Dollar': 1520.451015,
'South African Rand': 17159.831129,
'South Korean Won': 1355490.097163,
'Sri Lankan Rupee': 228245.645722,
'Swedish Krona': 10439.125427,

```
'Swiss Franc': 1037.792217,
'Taiwan New Dollar': 31334.286611,
'Thai Baht': 37436.518169,
'Trinidadian Dollar': 7636.35428,
'Turkish Lira': 15078.75981,
'US Dollar': 1127.074905,
'Venezuelan Bolivar': 511082584.868731}
```

That's about 1127.07 in USD at the time of writing this tutorial. Notice the last updated date and time; it usually updates every minute.

Scraping Xe

Xe is an online foreign exchange tools and services company. It is best known for its online currency converter. In this section, we use requests and BeautifulSoup libraries to make a currency converter based on it.

Open up a new Python file and import the necessary libraries:

```
import requests
from bs4 import BeautifulSoup as bs
import re
from dateutil.parser import parse
```

Now let's make a function that accepts the source currency, target currency, and the amount we want to convert, and then returns the converted amount along with the exchange rate date and time:

```
def convert_currency_xe(src, dst, amount):
```

```
    def get_digits(text):
        """Returns the digits and dots only from an input `text` as a float
        Args:
            text (str): Target text to parse
        """
        new_text = ""
        for c in text:
            if c.isdigit() or c == ".":
                new_text += c
        return float(new_text)

    url = f"https://www.xe.com/currencyconverter/convert/?Amount={amount}&From={src}&To={dst}"
    content = requests.get(url).content
    soup = bs(content, "html.parser")
    exchange_rate_html = soup.find_all("p")[2]
    # get the last updated datetime
    last_updated_datetime = parse(re.search(r"Last updated (.+)", exchange_rate_html.parent.parent.find_all("div")[-2].text).group()[12:])
    return last_updated_datetime, get_digits(exchange_rate_html.text)
```

At the time of writing this tutorial, the exchange rate is located in the third paragraph on the HTML page. This explains the `soup.find_all("p")[2]`. Make sure to change the extraction whenever a change is made to the HTML page. Hopefully, I'll keep an eye out whenever a change is made.

The latest date and time of the exchange rate is located at the second parent of the exchange rate paragraph in the HTML DOM.

Since the exchange rate contains string characters, I made the `get_digits()` function to extract only the digits and dots from a given string, which is helpful in our case.

Let's use the function now:

```
if __name__ == "__main__":
    import sys
    source_currency = sys.argv[1]
    destination_currency = sys.argv[2]
    amount = float(sys.argv[3])
    last_updated_datetime, exchange_rate = convert_currency_xe(source_currency, destination_currency, amount)
    print("Last updated datetime:", last_updated_datetime)
    print(f"{amount} {source_currency} = {exchange_rate} {destination_currency}")
```

This time, we get the source and target currencies as well as the amount from the command-lines, trying to convert 1000 EUR to USD:

```
$ python currency_converter_xe.py EUR USD 1000
```

Output:

```
Last updated datetime: 2022-02-01 13:04:00+00:00
1000.0 EUR = 1125.8987 USD
```

That's great! Xe usually updates every minute too, so it's real-time!

Scraping Yahoo Finance

Yahoo Finance provides financial news, currency data, stock quotes, press releases, and financial reports. This section uses the yahoo_fin library in Python to make a currency exchanger based on Yahoo Finance data.

Importing the libraries:

```
import yahoo_fin.stock_info as si from datetime import datetime, timedelta
```

yahoo_fin does an excellent job of extracting the data from the Yahoo Finance web page, and it is still maintained now; we use the get_data() method from the stock_info module and pass the currency symbol to it.

Below is the function that uses this function and returns the converted amount from one currency to another:

```
def convert_currency_yahoofin(src, dst, amount):
    # construct the currency pair symbol
    symbol = f"{src}{dst}=X"
    # extract minute data of the recent 2 days
    latest_data = si.get_data(symbol, interval="1m",
start_date=datetime.now() - timedelta(days=2))
    # get the latest datetime
    last_updated_datetime = latest_data.index[-1].to_pydatetime()
    # get the latest price
    latest_price = latest_data.iloc[-1].close
    # return the latest datetime with the converted amount
    return last_updated_datetime, latest_price * amount
```

We pass `"1m"` to the `interval` parameter in the `get_data()` method to extract minute data instead of daily data (default). We also get minute data of the previous two days, as it may cause issues on the weekends, just to be cautious.

The significant advantage of this method is you can get historical data by simply changing `start_date` and `end_date` parameters on this method. You can also change the `interval` to be `"1d"` for daily, `"1wk"` for weekly, and `"1mo"` for monthly.

Let's use the function now:

```
if __name__ == "__main__":
    import sys
    source_currency = sys.argv[1]
    destination_currency = sys.argv[2]
    amount = float(sys.argv[3])
    last_updated_datetime, exchange_rate =
convert_currency_yahoofin(source_currency, destination_currency,
amount)
    print("Last updated datetime:", last_updated_datetime)
    print(f"{amount} {source_currency} = {exchange_rate}
{destination_currency}")
```

Running the code:

```
$ python currency_converter_yahoofin.py EUR USD 1000
```

Output:

```
Last updated datetime: 2022-02-01 13:26:34
1000.0 EUR = 1126.1261701583862 USD
```

Using ExchangeRate API

As mentioned at the beginning of this tutorial, if you want a more reliable way to make a currency converter, you have to choose an API for that. There are several APIs for this purpose. However, we have picked two APIs that seem convenient and easy to get started.

ExchangeRate API supports 161 currencies and offers a free ~~monthly~~ ~~1,500 requ~~ests if you want to try it out, and there is an open API as well that offers daily updated data, and that's what we are going to use:

```
import requests from dateutil.parser import parse


def get_all_exchange_rates_erapi(src):
    url = f"https://open.er-api.com/v6/latest/{src}"
    # request the open ExchangeRate API and convert to Python dict using .json()
    data = requests.get(url).json()
    if data["result"] == "success":
        # request successful
        # get the last updated datetime
        last_updated_datetime = parse(data["time_last_update_utc"])
        # get the exchange rates
        exchange_rates = data["rates"]
    return last_updated_datetime, exchange_rates
```

The above function requests the open API and returns the exchange rates for all the currencies with the latest date and time. Let's use

this function to make a currency converter function:

```
def convert_currency_erapi(src, dst, amount):
    # get all the exchange rates
    last_updated_datetime, exchange_rates = get_all_exchange_rates_erapi(src)
    # convert by simply getting the target currency exchange rate and multiply by the amount
    return last_updated_datetime, exchange_rates[dst] * amount
```

As usual, let's make the main code:

```
if __name__ == "__main__":
    import sys
    source_currency = sys.argv[1]
    destination_currency = sys.argv[2]
    amount = float(sys.argv[3])
    last_updated_datetime, exchange_rate = convert_currency_erapi(source_currency, destination_currency, amount)
    print("Last updated datetime:", last_updated_datetime)
    print(f"{amount} {source_currency} = {exchange_rate} {destination_currency}")
```

Running it:

```
$ python currency_converter_erapi.py EUR USD 1000
```

Output:

```
Last updated datetime: 2022-02-01 00:02:31+00:00
1000.0 EUR = 1120.0 USD
```

The rates update daily, and it does not offer the exact exchange number as it's an open API; you can freely sign up for an API key to get precise exchange rates.

Using Fixer API

One of the promising alternatives is Fixer API. It is a simple and lightweight API for real-time and historical foreign exchange rates. You can easily create an account and get the API key.

After you've done that, you can use the /convert endpoint to convert from one currency to another. However, that's not included in the free plan and requires upgrading your account.

There is the /latest endpoint that does not require an upgrade and works in a free account just fine. It returns the exchange rates for the currency of your region. We can pass the source and target currencies we want to convert and calculate the exchange rate between both. Here's the function:

```
import requests
from datetime import datetime




API_KEY = "<YOUR_API_KEY_HERE>"

def convert_currency_fixerapi_free(src, dst, amount):
  """converts `amount` from the `src` currency to `dst` using the free account"""
  url = f"http://data.fixer.io/api/latest?access_key={API_KEY}&symbols={src},{dst}&format=1"
  data = requests.get(url).json()
  if data["success"]:
```

```
    # request successful        rates = data["rates"]        # since we have the
rate for our currency to src and dst, we can get
exchange rate between both
    # using below calculation
    exchange_rate = 1 / rates[src] * rates[dst]
    last_updated_datetime = datetime.fromtimestamp(data["timestamp"])
    return last_updated_datetime, exchange_rate * amount
```

Below is the function that uses the `/convert` endpoint in case you have an upgraded account:

```
def convert_currency_fixerapi(src, dst, amount):
  """converts `amount` from the `src` currency to `dst`, requires upgraded
account"""
  url = f"https://data.fixer.io/api/convert?access_key={API_KEY}&from=
{src}&to={dst}&amount={amount}"
  data = requests.get(url).json()
  if data["success"]:
    # request successful
    # get the latest datetime
    last_updated_datetime = datetime.fromtimestamp(data["info"]
["timestamp"])
    # get the result based on the latest price
    result = data["result"]
    return last_updated_datetime, result
```

Let's use either function:

```
if __name__ == "__main__":
```

```
import sys
source_currency = sys.argv[1]
destination_currency = sys.argv[2]
amount = float(sys.argv[3])
# free account
last_updated_datetime, exchange_rate = convert_currency_fixerapi_free(source_currency, destination_currency, amount)
# upgraded account, uncomment if you have one
# last_updated_datetime, exchange_rate = convert_currency_fixerapi(source_currency, destination_currency, amount)
print("Last updated datetime:", last_updated_datetime)
print(f"{amount} {source_currency} = {exchange_rate} {destination_currency}")
```

Before running the script, make sure to replace `API_KEY` with the API key you get when registering for an account.

Running the script:

```
Last updated datetime: 2022-02-01 15:54:04
1000.0 EUR = 1126.494 USD
```

You can check the documentation of Fixer API here.

Summary

There are many ways to make a currency converter, and we have covered five of them. If one method does not work for you, you can choose another one!

Fullcode:

currency_converter_xrates.py

```
import requests
from bs4 import BeautifulSoup as bs
from dateutil.parser import parse
from pprint import pprint


def get_exchange_list_xrates(currency, amount=1):
  # make the request to x-rates.com to get current exchange rates for common currencies
  content = requests.get(f"https://www.x-rates.com/table/?from={currency}&amount={amount}").content
  # initialize beautifulsoup
  soup = bs(content, "html.parser")
  # get the last updated time
  price_datetime = parse(soup.find_all("span", attrs={"class": "ratesTimestamp"})[1].text)
  # get the exchange rates tables
  exchange_tables = soup.find_all("table")
  exchange_rates = {}
  for exchange_table in exchange_tables:
    for tr in exchange_table.find_all("tr"):
      # for each row in the table
      tds = tr.find_all("td")
      if tds:
        currency = tds[0].text
        # get the exchange rate
```

```
            exchange_rate = float(tds[1].text)
            exchange_rates[currency] = exchange_rate
    return price_datetime, exchange_rates


if __name__ == "__main__":
    import sys
    source_currency = sys.argv[1]
    amount = float(sys.argv[2])
    price_datetime, exchange_rates =
get_exchange_list_xrates(source_currency, amount)
    print("Last updated:", price_datetime)
    pprint(exchange_rates)
```

currency_converter_xe.py

```
import requests
from bs4 import BeautifulSoup as bs
import re
from dateutil.parser import parse

def convert_currency_xe(src, dst, amount):
    def get_digits(text):
        """Returns the digits and dots only from an input `text` as a float
        Args:
            text (str): Target text to parse
        """
        new_text = ""
        for c in text:
```

```
        if c.isdigit() or c == ".":
            new_text += c
    return float(new_text)

  url = f"https://www.xe.com/currencyconverter/convert/?Amount={amount}&From={src}&To={dst}"
  content = requests.get(url).content
  soup = bs(content, "html.parser")
  exchange_rate_html = soup.find_all("p")[2]
  # get the last updated datetime
  last_updated_datetime = parse(re.search(r"Last updated (.+)", exchange_rate_html.parent.parent.find_all("div")[-2].text).group()[12:])
  return last_updated_datetime, get_digits(exchange_rate_html.text)


if __name__ == "__main__":
  import sys
  source_currency = sys.argv[1]
  destination_currency = sys.argv[2]
  amount = float(sys.argv[3])
  last_updated_datetime, exchange_rate = convert_currency_xe(source_currency, destination_currency, amount)
  print("Last updated datetime:", last_updated_datetime)
  print(f"{amount} {source_currency} = {exchange_rate} {destination_currency}")
```

currency_converter_yahoofin.py

```
import yahoo_fin.stock_info as si
```

```
from datetime import datetime, timedelta

def convert_currency_yahoofin(src, dst, amount):
    # construct the currency pair symbol
    symbol = f"{src}{dst}=X"
    # extract minute data of the recent 2 days
    latest_data = si.get_data(symbol, interval="1m",
start_date=datetime.now() - timedelta(days=2))
    # get the latest datetime
    last_updated_datetime = latest_data.index[-1].to_pydatetime()
    # get the latest price
    latest_price = latest_data.iloc[-1].close
    # return the latest datetime with the converted amount
    return last_updated_datetime, latest_price * amount


if __name__ == "__main__":
    import sys
    source_currency = sys.argv[1]
    destination_currency = sys.argv[2]
    amount = float(sys.argv[3])
    last_updated_datetime, exchange_rate =
convert_currency_yahoofin(source_currency, destination_currency,
amount)
    print("Last updated datetime:", last_updated_datetime)
    print(f"{amount} {source_currency} = {exchange_rate}
{destination_currency}")
```

currency_converter_erapi.py

```
import requests
from dateutil.parser import parse

def get_all_exchange_rates_erapi(src):
    url = f"https://open.er-api.com/v6/latest/{src}"
    # request the open ExchangeRate API and convert to Python dict using .json()
    data = requests.get(url).json()
    if data["result"] == "success":
        # request successful
        # get the last updated datetime
        last_updated_datetime = parse(data["time_last_update_utc"])
        # get the exchange rates
        exchange_rates = data["rates"]
    return last_updated_datetime, exchange_rates



def convert_currency_erapi(src, dst, amount):
    # get all the exchange rates
    last_updated_datetime, exchange_rates = get_all_exchange_rates_erapi(src)
    # convert by simply getting the target currency exchange rate and multiply by the amount
    return last_updated_datetime, exchange_rates[dst] * amount


if __name__ == "__main__":
    import sys
```

```
    source_currency = sys.argv[1]
    destination_currency = sys.argv[2]
    amount = float(sys.argv[3])
    last_updated_datetime, exchange_rate =
convert_currency_erapi(source_currency, destination_currency, amount)
    print("Last updated datetime:", last_updated_datetime)
    print(f"{amount} {source_currency} = {exchange_rate}
{destination_currency}")
```

currency_converter_fixerapi.py

```
import requests
from datetime import date, datetime

API_KEY = "8c3dce10dc5fdb6ec1f555a1504b1373"
# API_KEY = "<YOUR_API_KEY_HERE>"


def   convert_currency_fixerapi_free(src,   dst,   amount):            """converts
`amount` from the `src` currency to `dst` using the free
account"""
    url = f"http://data.fixer.io/api/latest?access_key={API_KEY}&symbols=
{src},{dst}&format=1"
    data = requests.get(url).json()
    if data["success"]:
        # request successful
        rates = data["rates"]
        # since we have the rate for our currency to src and dst, we can get
exchange rate between both
```

```
    # using below calculation
    exchange_rate = 1 / rates[src] * rates[dst]
    last_updated_datetime = datetime.fromtimestamp(data["timestamp"])
    return last_updated_datetime, exchange_rate * amount


def convert_currency_fixerapi(src, dst, amount):
  """converts `amount` from the `src` currency to `dst`, requires upgraded account"""
  url = f"https://data.fixer.io/api/convert?access_key={API_KEY}&from={src}&to={dst}&amount={amount}"
  data = requests.get(url).json()
  if data["success"]:
    # request successful
    # get the latest datetime
    last_updated_datetime = datetime.fromtimestamp(data["info"]["timestamp"])
    # get the result based on the latest price
    result = data["result"]
    return last_updated_datetime, result


if __name__ == "__main__":
  import sys
  source_currency = sys.argv[1]
  destination_currency = sys.argv[2]
  amount = float(sys.argv[3])
  # free account
```

```
    last_updated_datetime, exchange_rate = convert_currency_fixerapi_free(source_currency, destination_currency, amount)
    # upgraded account, uncomment if you have one
    # last_updated_datetime, exchange_rate = convert_currency_fixerapi(source_currency, destination_currency, amount)
    print("Last updated datetime:", last_updated_datetime)
    print(f"{amount} {source_currency} = {exchange_rate} {destination_currency}")

```

PART 3: Webhooks in Python with Flask

Learn how to create a streaming application with real-time charting by consuming webhooks with the help of Flask, Redis, SocketIO and other libraries in Python.

Introduction

A webhook can be thought of as a type of API that is driven by events rather than requests. Instead of one application making a request to another to receive a response, a webhook is a service that allows one program to send data to another as soon as a particular event takes place.

Webhooks are sometimes referred to as reverse APIs, because communication is initiated by the application sending the data rather than the one receiving it. With web services becoming increasingly interconnected, webhooks are seeing more action as a lightweight solution for enabling real-time notifications and data updates without the need to develop a full-scale API.

Webhooks usually act as messengers for smaller data. They help in sending messages, alerts, notifications and real-time information from the server-side application to the client-side application.

Let's say for instance, you want your application to get notified when tweets that mention a certain account and contain a specific hashtag are published. Instead of your application continuously asking Twitter for new posts meeting these criteria, it makes much

more sense for Twitter to send a notification to your application only when such event takes place.

This is the purpose of a webhook instead of having to repeatedly request the data (polling mechanism), the receiving application can sit back and get what it needs without having to send repeated requests to another system.

Webhooks can open up a lot of possibilities:

- You can use a webhook to connect a payment gateway with your email marketing software so that you send an email to the user whenever a payment bounces.
- You can use webhooks to synchronize customer data in other applications. For example, if a user changes his email address, you can ensure that the change is reflected in your CRM as well.
- You can also use webhooks to send information about events to external databases or data warehouses like Amazon's Redshift, or Google Big Query for further analysis.

Scope

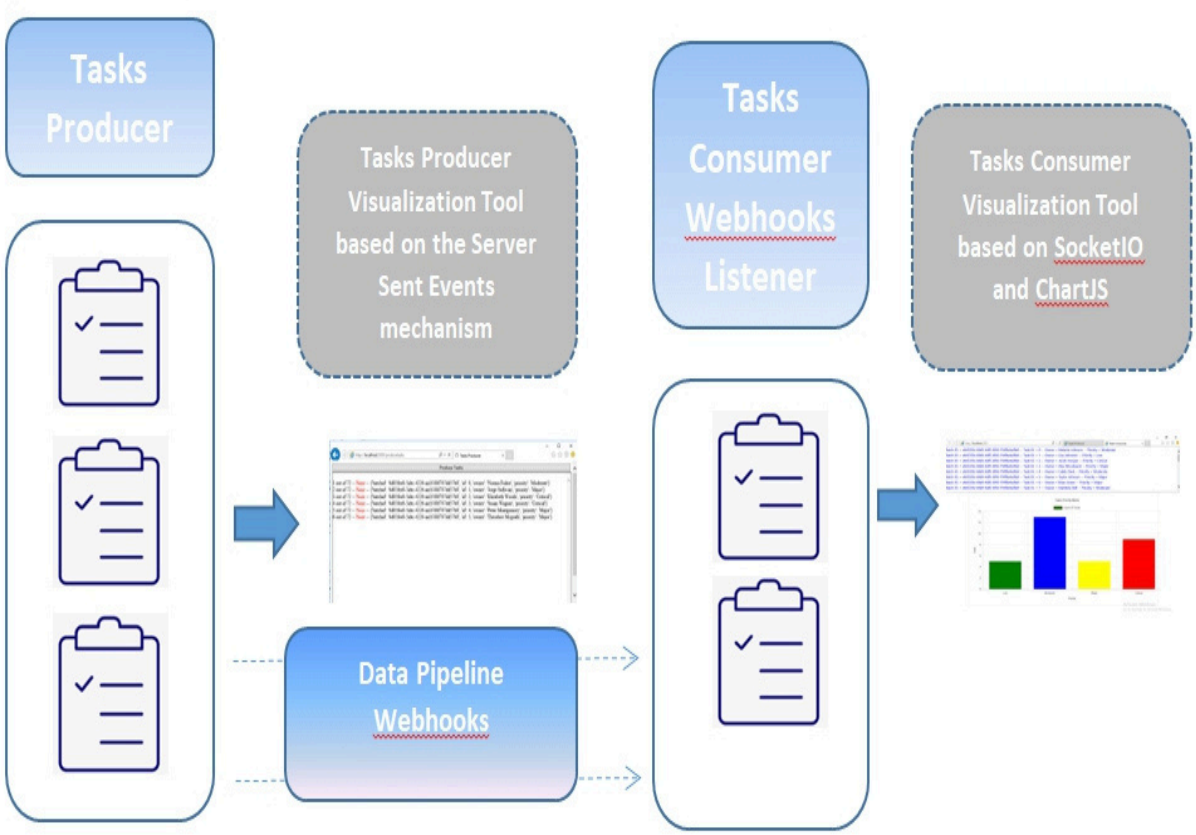
In this tutorial, we will lay the groundwork for a streaming application based on webhooks and encompassing several components:

- A webhooks generator which mimics an internal or an external service emitting tasks to a pre-configured webhook endpoint.
- A webhook listener that receives notification messages for these events/tasks. Once received, these tickets will be

rendered and converted into a bar chart that generates valuable insights. Charts reduce the complexity of the data and make it easier to understand for any user.

We will leverage several components like Redis, Flask, SocketIO, and ChartJS to develop nice-looking visualization tool for the aforementioned components.

Process Flowchart



Pre-requisites

As our requirements stand, the following components come into play:

- **Redis** is an open source, advanced key-value store and an apt solution for building high-performance, scalable web

applications. Redis has three main peculiarities that sets it apart:

    - Redis holds its database entirely in the memory, using the disk only for persistance.
    - Redis has a relatively rich set of data types when compared to many other key-value data stores.
    - Redis can replicate data to any number of slaves.Installing Redis is outside the scope of this tutorial, but you can check this tutorial for installing it on Windows.

- **Socket.IO** is a JavaScript library for real-time web applications. It enables real-time, bidirectional communication between web clients and servers. It has two parts: a client-side library that runs in the browser and a server-side library.
- **Faker** is a Python package that generates fake data for you. Whether you need to bootstrap your database, create good looking XML documents, fill-in your persistence to stress test it, or anonymize data taken from a production service, Fake is the right choice for you.
- **ChartJS** is an open source Javascript library that allows you to draw different types of charts by using the HTML5 canvas element. The HTML5 element gives an easy and powerful way to draw graphics using Javascript. This library supports 8 different types of graphs: lines, bars, doughnuts, pies, radars, polar areas, bubbles and scatters.
- **Flask** is a micro web framework written in Python.

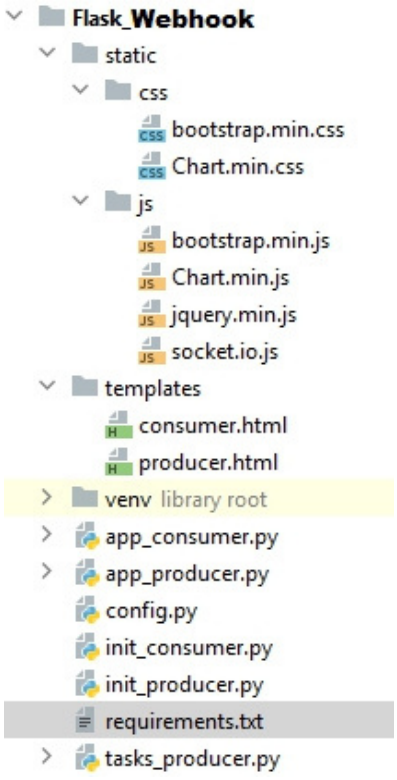
If this tutorial intrigues you and makes you want to dive into the code immediately, you can check this repository for reviewing the code used in this article.

Setup

Setting up the package is quite simple and straightforward. Of course you need Python 3 installed on your system and it is highly recommended to setup a virtual environment where we will install the needed libraries:

```
$ pip install Faker==8.2.0 Flask==1.1.2 Flask-SocketIO==5.0.1 redis==3.5.3 requests==2.25.1
```

At the end of this tutorial, our folder structure will look like the following:



Let's start writing the actual code. First, let's define the configuration parameters for our application within `config.py` :

```
#Application configuration File
```

```
##################################
#Secret key that will be used by Flask for securely signing the session cookie
# and can be used for other security related needs
SECRET_KEY = 'SECRET_KEY'
########################################
#Minimum Number Of Tasks To Generate
MIN_NBR_TASKS = 1
#Maximum Number Of Tasks To Generate
MAX_NBR_TASKS = 100
#Time to wait when producing tasks
WAIT_TIME = 1
#Webhook endpoint Mapping to the listener
WEBHOOK_RECEIVER_URL = 'http://localhost:5001/consumetasks'
########################################
#Map to the REDIS Server Port
BROKER_URL = 'redis://localhost:6379'
########################################
```

Next, creating an initialization file for our tasks and webhooks producer in `init_producer.py`:

```
# init_producer.py
from flask import Flask

#Create a Flask instance
app = Flask(__name__)

#Load Flask configurations from config.py
```

```
app.secret_key = app.config['SECRET_KEY']
app.config.from_object("config")
```

Now let's write code necessary for producing tasks using Faker module:

```
# tasks_producer.py
import random
from faker.providers import BaseProvider
from faker import Faker
import config
import time
import requests
import json
import uuid


# Define a TaskProvider
class TaskProvider(BaseProvider):
  def task_priority(self):
    severity_levels = [
      'Low', 'Moderate', 'Major', 'Critical'
    ]
    return severity_levels[random.randint(0, len(severity_levels)-1)]


# Create a Faker instance and seeding to have the same results every time we execute the script
# Return data in English
fakeTasks = Faker('en_US')
```

```
# Seed the Faker instance to have the same results every time we run the program
fakeTasks.seed_instance(0)
# Assign the TaskProvider to the Faker instance
fakeTasks.add_provider(TaskProvider)


# Generate A Fake Task
def produce_task(batchid, taskid):
    # Message composition
    message = {
        'batchid': batchid, 'id': taskid, 'owner': fakeTasks.unique.name(), 'priority': fakeTasks.task_priority()
        # ,'raised_date':fakeTasks.date_time_this_year()
        # ,'description':fakeTasks.text()
    }
    return message


def send_webhook(msg):
    """
    Send a webhook to a specified URL
    :param msg: task details
    :return:
    """
    try:
        # Post a webhook message
        # default is a function applied to objects that are not serializable = it converts them to str
```

```
        resp = requests.post(config.WEBHOOK_RECEIVER_URL,
data=json.dumps(
            msg, sort_keys=True, default=str), headers={'Content-Type':
'application/json'}, timeout=1.0)
        # Returns an HTTPError if an error has occurred during the process
(used for debugging).
        resp.raise_for_status()
    except requests.exceptions.HTTPError as err:
        #print("An HTTP Error occurred",repr(err))
        pass
    except requests.exceptions.ConnectionError as err:
        #print("An Error Connecting to the API occurred", repr(err))
        pass
    except requests.exceptions.Timeout as err:
        #print("A Timeout Error occurred", repr(err))
        pass
    except requests.exceptions.RequestException as err:
        #print("An Unknown Error occurred", repr(err))
        pass
    except:
        pass
    else:
        return resp.status_code

# Generate A Bunch Of Fake Tasks
def produce_bunch_tasks():
    """
    Generate a Bunch of Fake Tasks
    """
```

```
                n     =     random.randint(config.MIN_NBR_TASKS,
config.MAX_NBR_TASKS)
  batchid = str(uuid.uuid4())
  for i in range(n):
    msg = produce_task(batchid, i)
    resp = send_webhook(msg)
    time.sleep(config.WAIT_TIME)
    print(i, "out of ", n, " -- Status", resp, " -- Message = ", msg)
    yield resp, n, msg


if __name__ == "__main__":
  for resp, total, msg in produce_bunch_tasks():
    pass
```

The above code leverages the Faker module in order to create a stream of fictitious randomized tasks and to send for each produced task a webhook to the endpoint `WEBHOOK_RECEIVER_URL` previously defined in our configuration file `config.py`.

The number of tasks generated in each batch will be a random number controlled by the thresholds `MIN_NBR_TASKS` and `MAX_NBR_TASKS` defined in `config.py` .

The webhook JSON message is composed of the following attributes: `batchid`, `taskid`, `owner` and `priority`.

Each batch of tasks generated will be identified by a unique reference called `batchid`.

The task priority will be limited to pre-selected options: Low, Moderate, High and Critical.

The primary use of the above code

is `produce_bunch_tasks()` function, which is a generator yielding the following:

- The status of the webhook emitted.
- The total number of tasks produced.
- The webhook message generated.

Before digging further, let's test our `tasks_producer.py` program:

```
$ python tasks_producer.py
```

You should see an output similar to the following:

```
0 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Norma Fisher', 'priority': 'Moderate'}
1 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Jorge Sullivan', 'priority': 'Critical'}
2 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Elizabeth Woods', 'priority': 'Critical'}
3 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Susan Wagner', 'priority': 'Moderate'}
4 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Peter Montgomery', 'priority': 'Major'}
5 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Theodore Mcgrath', 'priority': 'Major'}
6 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Stephanie Collins', 'priority': 'Critical'}
7 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Stephanie Sutton', 'priority': 'Critical'}
8 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Brian Hamilton', 'priority': 'Critical'}
9 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571', 'owner': 'Susan Levy', 'priority': 'Low'}
10 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'Sean Green', 'priority': 'Major'}
11 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'Kimberly Smith', 'priority': 'Moderate'}
12 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'Jennifer Summers', 'priority': 'Moderate'
13 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'April Snyder', 'priority': 'Major'}
14 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'Dana Nguyen', 'priority': 'Moderate'}
15 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'Cheryl Bradley', 'priority': 'Major'}
16 out of  63  -- Status None  -- Message =  {'batchid': '79033591-317e-48c0-9c47-3002a165a571' , 'owner': 'Walter Pratt', 'priority': 'Low'}
```

Now let's build our Flask app that emulates a service producing tasks:

```
#app_producer.py
from flask import Response, render_template
from init_producer import app
import tasks_producer
```

```
def stream_template(template_name, **context):
    app.update_template_context(context)
    t = app.jinja_env.get_template(template_name)
    rv = t.stream(context)
    rv.enable_buffering(5)
    return rv

@app.route("/", methods=['GET'])
def index():
    return render_template('producer.html')

@app.route('/producetasks', methods=['POST'])
def producetasks():
    print("producetasks")
    return Response(stream_template('producer.html', data=
tasks_producer.produce_bunch_tasks() ))

if __name__ == "__main__":
  app.run(host="localhost",port=5000, debug=True)
```

Within this flask app, we defined two main routes:

- "/": Renders the template web page (producer.html)
- "/producetasks": Calls the function produce_bunch_tasks() and stream the flow of tasks generated to the Flask application.

The server sents Server-Sent Events (SSEs), which are a type of server push mechanism, where a client receives a notification

whenever a new event occurs on the server.

Next, we will define the template `producer.html` file:

```
<!doctype html>
<html>
 <head>
  <title>Tasks Producer</title>
  <style>
   .content {
    width: 100%;
   }
   .container{
    max-width: none;
   }
  </style>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
 </head>
<body class="container">
  <div class="content">
   <form method='post' id="produceTasksForm" action = "/producetasks">
    <button style="height:20%x;width:100%" type="submit" id="produceTasks">Produce Tasks</button>
   </form>
  </div>
  <div class="content">
    <div id="Messages" class="content" style="height:400px;width:100%; border:2px solid gray; overflow-y:scroll;"></div>
```

```
    {% for rsp,total, msg in data: %}
     <script>
       var rsp   = "{{ rsp }}";
       var total = "{{ total }}";
       var msg   = "{{ msg }}";
       var lineidx = "{{ loop.index }}";
       //If the webhook request succeeds color it in blue else in red.
       if (rsp == '200') {
          rsp = rsp.fontcolor("blue");
       }
       else {
          rsp = rsp.fontcolor("red");
       }
       //Add the details of the generated task to the Messages section.
       document.getElementById('Messages').innerHTML += "<br>" +
lineidx  + " out of " + total + " -- "+ rsp + " -- " + msg;
    </script>
    {% endfor %}
  </div>
</body>
</html>
```

Three variables are passed to this template file:

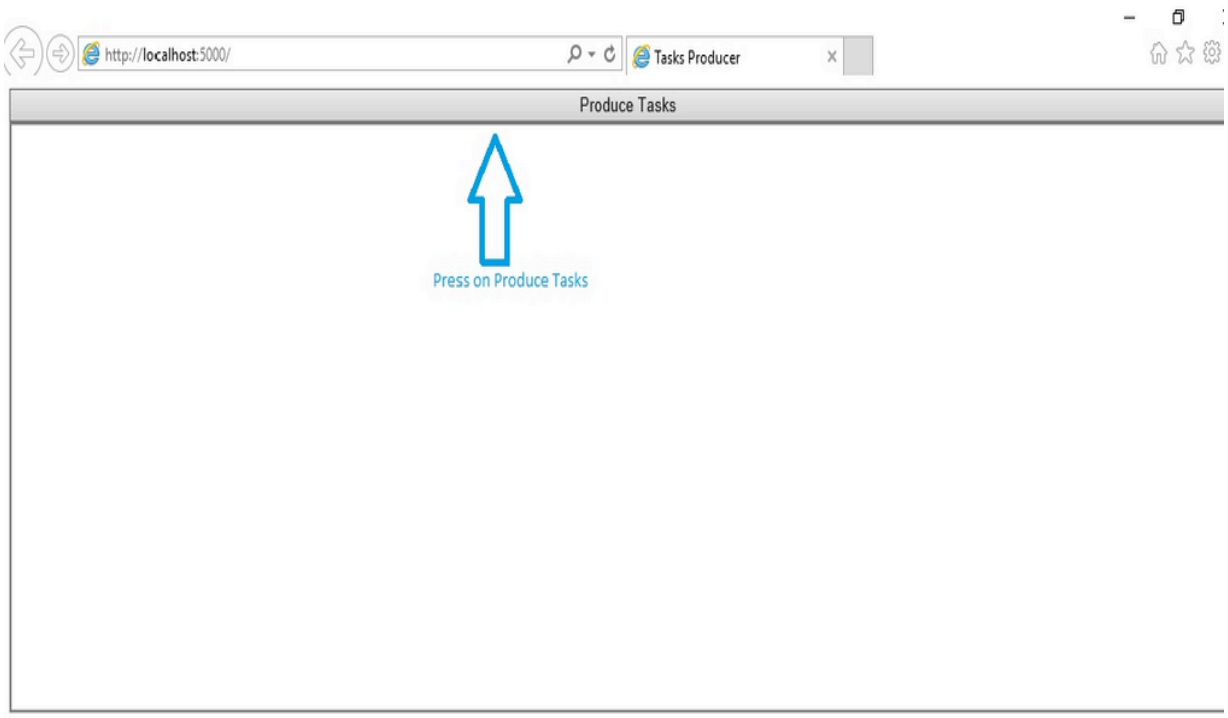
- `total`: representing the total number of tasks produced.
- `status`: representing the status of the dispatched webhook.
- `msg`: the webhook JSON message.

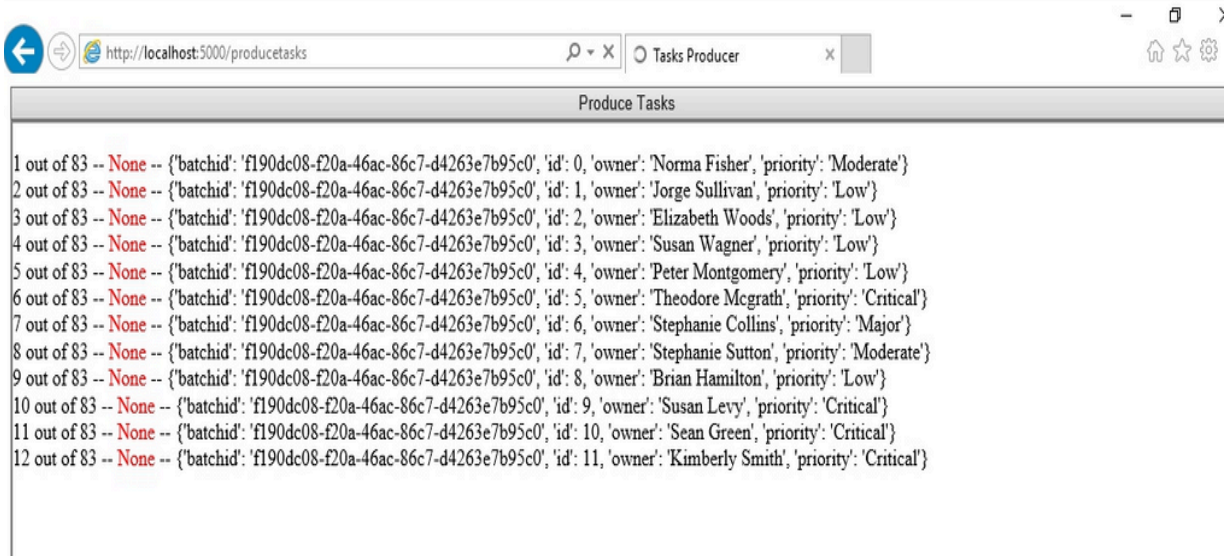
The template file contains a Javascript enabling to iterate throughout the received stream and to display the webhooks/tasks as they're received.

Now that our program is ready, let's test it out and check the output generated:

```
$ python app_producer.py
```

Access the link http://localhost:5000 where Flask instance is running, press on the button **Produce Tasks** and you will see a continuous stream of randomized tasks automatically generated as shown in the following screen:





You will notice that the response status of the dispatched webhook is equal to None, and displayed in red signaling the failure to reach its destination. Later on when we activate the tasks consumer, you will outline that the response status of the dispatched webhook is equal to 200 and displayed in blue signaling the success to reach the webhook endpoint.

Now, let's create the initialization file for our tasks consumer/handler:

```
# init_consumer.py
from flask import Flask


#Create a Flask instance
app = Flask(__name__)

#Load Flask configurations from config.py
app.secret_key = app.config['SECRET_KEY']
app.config.from_object("config")

#Setup the Flask SocketIO integration while mapping the Redis Server.
from flask_socketio import SocketIO
socketio =
SocketIO(app,logger=True,engineio_logger=True,message_queue=app.conf
ig['BROKER_URL'])
```

Next, let's build a Flask app for handling the dispatched webhooks/tasks. The first step to handling webhooks is to build a custom endpoint. This endpoint needs to expect data through a POST request, and confirm the successful receipt of that data:

```
#app_consumer.py
from flask import render_template, request,session
from flask_socketio import join_room
from init_consumer import app, socketio
import json
import uuid
```

```
#Render the assigned template file
@app.route("/", methods=['GET']) def
index(): return
render_template('consumer.html')

# Sending Message through the websocket
def send_message(event, namespace, room, message):
  # print("Message = ", message)
  socketio.emit(event, message, namespace=namespace, room=room)

# Registers a function to be run before the first request to this instance of
the application
# Create a unique session ID and store it within the application
configuration file
@app.before_first_request
def initialize_params():
  if not hasattr(app.config,'uid'):
    sid = str(uuid.uuid4())
    app.config['uid'] = sid
    print("initialize_params - Session ID stored =", sid)

# Receive the webhooks and emit websocket events
@app.route('/consumetasks', methods=['POST'])
def consumetasks():
  if request.method == 'POST':
    data = request.json
    if data:
      print("Received Data = ", data)
```

```
        roomid =  app.config['uid']
        var = json.dumps(data)
        send_message(event='msg', namespace='/collectHooks',
room=roomid, message=var)
    return 'OK'

#Execute on connecting
@socketio.on('connect', namespace='/collectHooks')
def socket_connect():
    # Display message upon connecting to the namespace
    print('Client Connected To NameSpace /collectHooks - ', request.sid)

#Execute on disconnecting
@socketio.on('disconnect', namespace='/collectHooks')
def socket_connect():
    # Display message upon disconnecting from the namespace
    print('Client disconnected From NameSpace /collectHooks - ',
request.sid)

#Execute upon joining a specific room
@socketio.on('join_room', namespace='/collectHooks')
def on_room():
    if app.config['uid']:
        room = str(app.config['uid'])
        # Display message upon joining a room specific to the session
previously stored.
        print(f"Socket joining room {room}")
        join_room(room)
```

```
#Execute upon encountering any error related to the websocket
@socketio.on_error_default
def error_handler(e):
    # Display message on error.
    print(f"socket error: {e}, {str(request.event)}")

#Run using port 5001
if __name__ == "__main__":
    socketio.run(app,host='localhost', port=5001,debug=True)
```

In brief, we performed the following:

- We added a function `@app.before_first_request` that ran once before the very first request to the app and is ignored on subsequent requests. Within this function we created a unique session ID and store it within the configuration file, this unique session ID will serve to allocate an exclusive room for each user when dealing with the web socket communication.
- We defined a webhook listener in `"/consumetasks"` which expects JSON data through POST requests and once received, it emits a web socket event concurrently.
- To manage effectively our connection over the web socker:
    - We will set the value `/collectHooks` for the namespace (namespaces are used to separate server logic over a single shared connection).
    - We will assign a dedicated room for each user session (rooms are subdivisions or sub-channels of namespaces).

After all this build up, let's code the frontend for our web app, create `consumer.html` in the `templates` folder and the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tasks Consumer</title>
  <link rel="stylesheet" href="
{{url_for('static',filename='css/bootstrap.min.css')}}">
  <link rel="stylesheet" href="
{{url_for('static',filename='css/Chart.min.css')}}">
</head>
<body>
  <div class="content">
    <div id="Messages" class="content" style="height:200px;width:100%;
border:1px solid gray; overflow-y:scroll;"></div>
  </div>
  <div class="container">
    <div class="row">
      <div class="col-12">
        <div class="card">
          <div class="card-body">
            <canvas id="canvas"></canvas>
          </div>
        </div>
      </div>
    </div>
  </div>
  <!-- import the jquery library -->
```

```
  <script src="{{ url_for('static',filename='js/jquery.min.js') }}"></script>
  <!-- import the socket.io library -->
  <script src="{{ url_for('static',filename='js/socket.io.js') }}"></script>
  <!-- import the bootstrap library -->
  <script src="{{ url_for('static',filename='js/bootstrap.min.js') }}">
</script>
  <!-- import the Chart library -->
  <script src="{{ url_for('static',filename='js/Chart.min.js') }}"></script>
<script>
    $(document).ready(function(){
     const config = {
        //Type of the chart - Bar Chart
        type: 'bar',
        //Data for our chart
        data: {
           labels: ['Low','Moderate','Major','Critical'],
           datasets: [{
              label: "Count Of Tasks",
              //Setting a color for each bar
              backgroundColor: ['green','blue','yellow','red'],
              borderColor: 'rgb(255, 99, 132)',
              data: [0,0,0,0],
              fill: false,
           }],
        },
        //Configuration options
        options: {
           responsive: true,
           title: {
```

```
            display: true,
            text: 'Tasks Priority Matrix'
        },
        tooltips: {
            mode: 'index',
            intersect: false,
        },
        hover: {
            mode: 'nearest',
            intersect: true
        },
        scales: {
            xAxes: [{
                display: true,
                scaleLabel: {
                    display: true,
                    labelString: 'Priority'
                }
            }],
            yAxes: [{
                display: true
             ,ticks: {
                    beginAtZero: true
                }
              ,scaleLabel: {
                    display: true,
                    labelString: 'Total'
                }
            }]
```

```
            }
        }
    };
    const context = document.getElementById('canvas').getContext('2d');
    //Creating the bar chart
    const lineChart = new Chart(context, config);
    //Reserved for websocket manipulation
    var namespace='/collectHooks';
    var url = 'http://' + document.domain + ':' + location.port + namespace;
    var socket = io.connect(url);
    //When connecting to the socket join the room
    socket.on('connect', function() {
                    socket.emit('join_room');
                });
    //When receiving a message
    socket.on('msg' , function(data) {
                var msg = JSON.parse(data);
                var newLine = $('<li>'+ 'Batch ID. = ' + msg.batchid + ' -- Task ID. = ' + msg.id + ' -- Owner = ' + msg.owner + ' -- Priority = ' + msg.priority +'</li>');
                newLine.css("color","blue");
                $("#Messages").append(newLine);
                //Retrieve the index of the priority of the received message
                var lindex = config.data.labels.indexOf(msg.priority);
                //Increment the value of the priority of the received message
                config.data.datasets[0].data[lindex] += 1;
                //Update the chart
                lineChart.update();
```

```
                });
    });
</script>
</body>
</html>
```

The above template include the following:

- The section messages for displaying the details of the received tasks or webhooks.
- A bar chart showing the total number of tasks received throughout the web sockets events per priority. The steps performed to build the chart are the following:
    - Placing a canvas element where to show the chart. Specifying the priority levels in the labels
    - property which indicates the names of the instances you want to compare.
    - Initialize a dataset property, that defines an array of objects, each of which containing the data we want to compare.
    - The bar chart will get updated synchronously whenever a new webhook is transmitted and received thoughout the web socket.

Now let's test our program, please proceed as per the following steps:

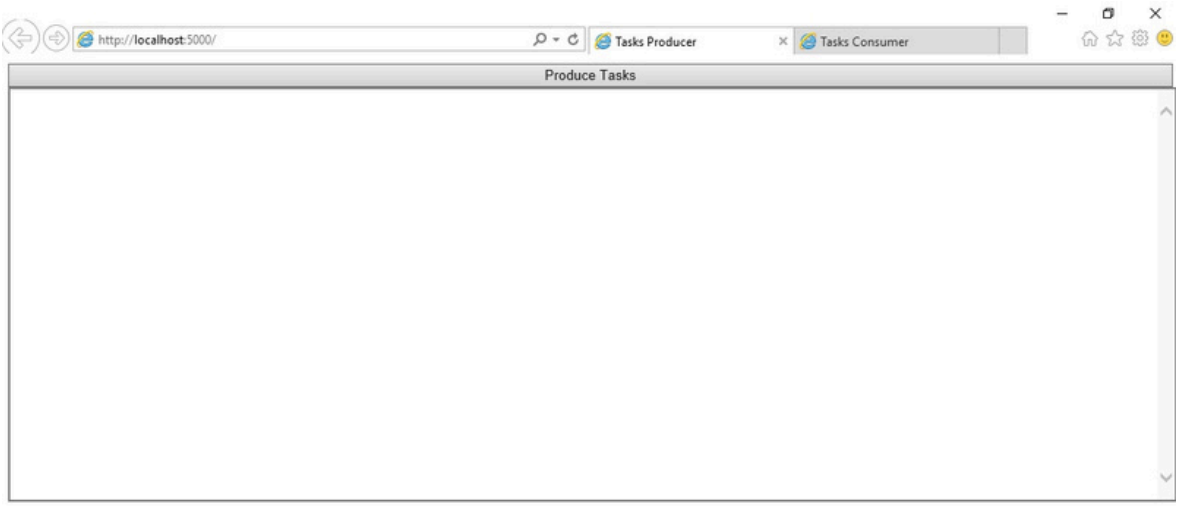
- Open up a terminal and run the app_producer.py:

```
$ python app_producer.py
```

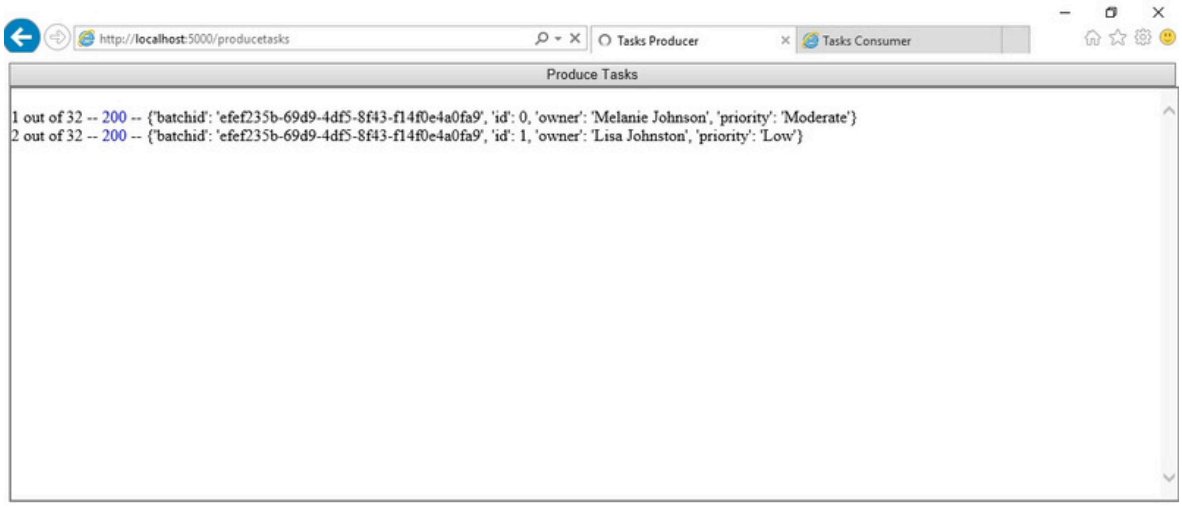
- Start the Redis server, make sure the Redis instance is running on TCP port 6479.
- Open up another terminal and run `app_consumer.py`:

```
$ python app_consumer.py
```

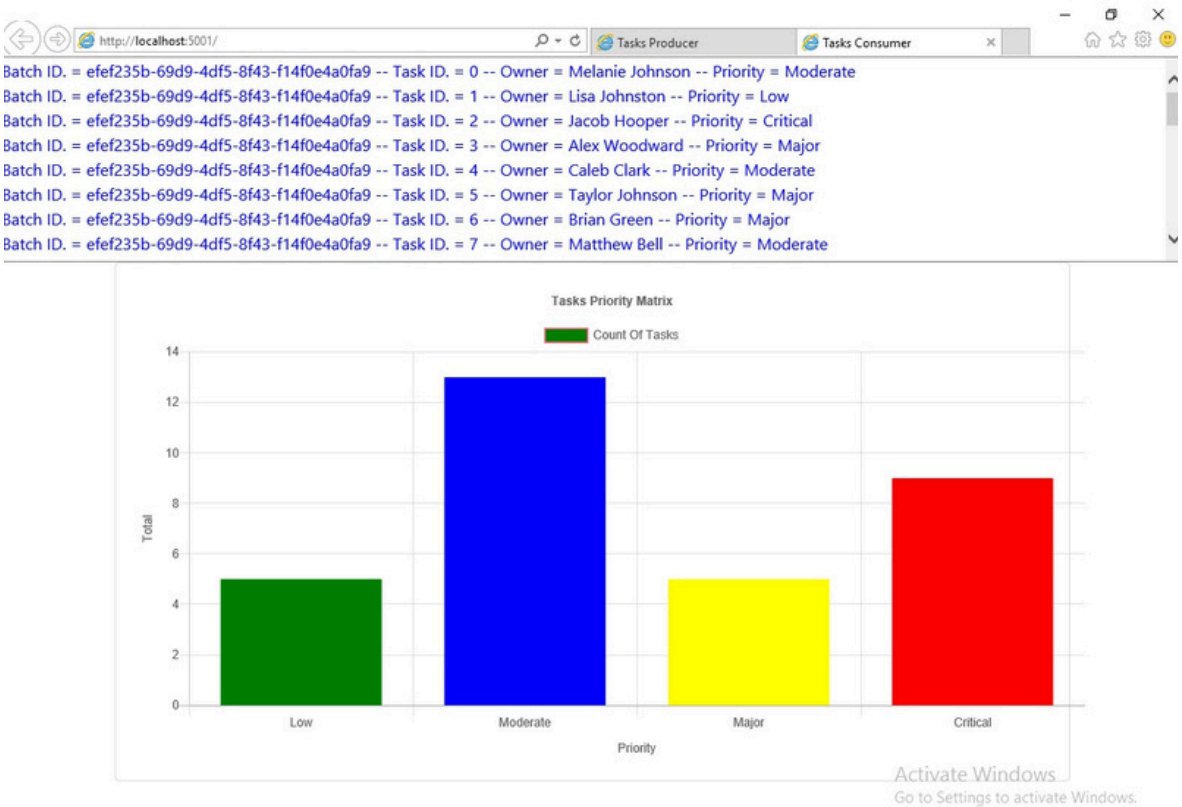
- Open your browser and access the `http://localhost:5000`link to visualize the tasks producer:



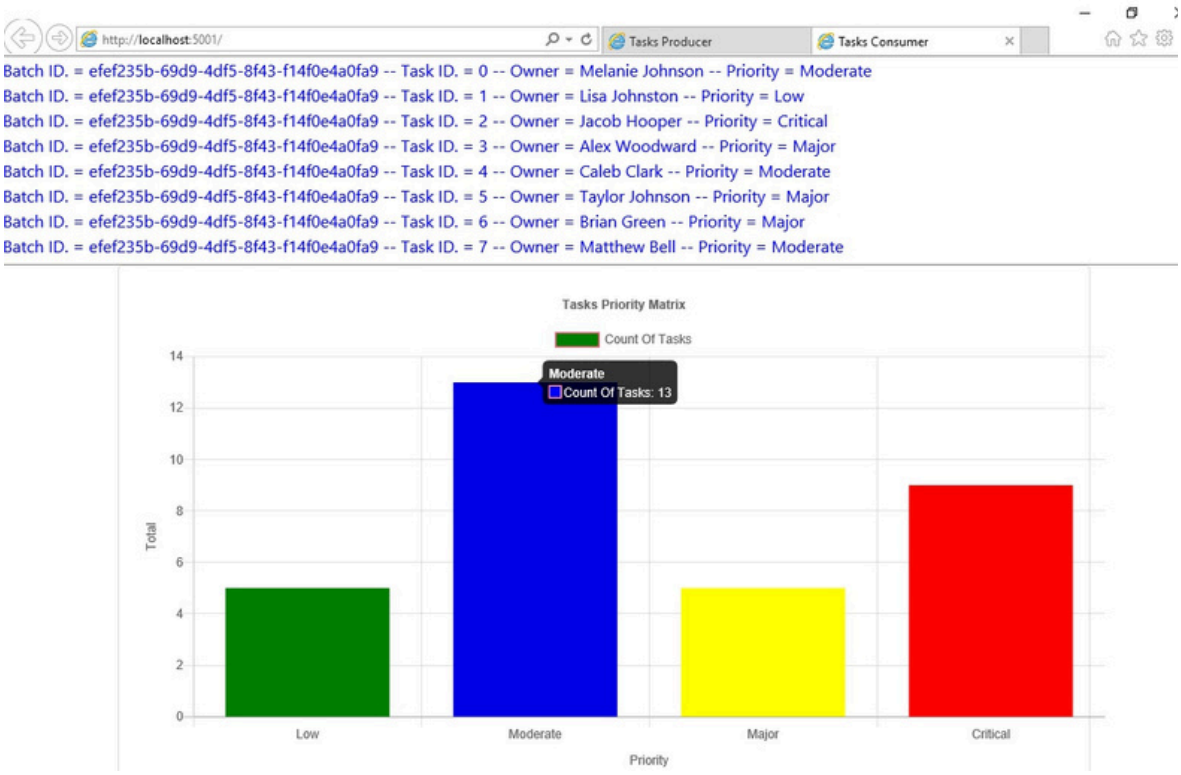
Press on the **Produce Tasks** button and a batch of tasks will be automatically generated and displayed gradually on the screen as shown below:



Now open another tab in your browser and access http://localhost:5001 in order to visualize the tasks consumer, the tasks will appear gradually in the messages section, and the bar chart will get updated automatically whenever a webhook is received:



When hovering the mouse over any of the charts bars, a tooltip showing the total number of tasks is displayed:



Summary

Webhooks are an important part of the web and they are becoming more popular. They allow your applications to exchange data instantly and seamlessly.

While webhooks are similar to APIs, they both play different roles, each with its own unique use case. Hopefully, this article has expanded your understanding, and remember that the key to getting the most out of webhooks is to know when they are the right choice for your application.

Fullcode:

config.py

```
#Application configuration File
################################


#Secret key that will be used by Flask for securely signing the session cookie
# and can be used for other security related needs
SECRET_KEY = 'SECRET_KEY'

########################################
#Minimum Number Of Tasks To Generate
MIN_NBR_TASKS = 1

#Maximum Number Of Tasks To Generate
MAX_NBR_TASKS = 100

#Time to wait when producing tasks
WAIT_TIME = 1

#Webhook endpoint Mapping to the listener
WEBHOOK_RECEIVER_URL = 'http://localhost:5001/consumetasks'
########################################

#Map to the REDIS Server Port
BROKER_URL = 'redis://localhost:6379'
########################################
```

tasks_producer.py

```
import random
from faker.providers import BaseProvider
from faker import Faker
import config
import time
import requests
import json
import uuid


# Define a TaskProvider
class TaskProvider(BaseProvider):
  def task_priority(self):
    severity_levels = [
      'Low', 'Moderate', 'Major', 'Critical'
    ]
    return severity_levels[random.randint(0, len(severity_levels)-1)]


# Create a Faker instance and seeding to have the same results every time we execute the script
# Return data in English
fakeTasks = Faker('en_US')
# Seed the Faker instance to have the same results every time we run the program
fakeTasks.seed_instance(0)
# Assign the TaskProvider to the Faker instance
fakeTasks.add_provider(TaskProvider)
```

```
# Generate A Fake Task
def produce_task(batchid, taskid):
    # Message composition
    message = {
        'batchid': batchid, 'id': taskid, 'owner': fakeTasks.unique.name(),
'priority': fakeTasks.task_priority()
        # ,'raised_date':fakeTasks.date_time_this_year()
        # ,'description':fakeTasks.text()
    }
    return message


def send_webhook(msg):
    """
    Send a webhook to a specified URL
    :param msg: task details
    :return:
    """
    try:
        # Post a webhook message
        # default is a function applied to objects that are not serializable = it
converts them to str
        resp = requests.post(config.WEBHOOK_RECEIVER_URL,
data=json.dumps(
            msg, sort_keys=True, default=str), headers={'Content-Type':
'application/json'}, timeout=1.0)
        # Returns an HTTPError if an error has occurred during the process
(used for debugging).
        resp.raise_for_status()
```

```
  except requests.exceptions.HTTPError as err:
    #print("An HTTP Error occurred",repr(err))
    pass
  except requests.exceptions.ConnectionError as err:
    #print("An Error Connecting to the API occurred", repr(err))
    pass
  except requests.exceptions.Timeout as err:
    #print("A Timeout Error occurred", repr(err))
    pass
  except requests.exceptions.RequestException as err:
    #print("An Unknown Error occurred", repr(err))
    pass
  except:
    pass
  else:
    return resp.status_code

# Generate A Bunch Of Fake Tasks def produce_bunch_tasks(): """ Generate a Bunch of Fake Tasks """ n = random.randint(config.MIN_NBR_TASKS, config.MAX_NBR_TASKS)
  batchid = str(uuid.uuid4())
  for i in range(n):
    msg = produce_task(batchid, i)
    resp = send_webhook(msg)
    time.sleep(config.WAIT_TIME)
```

```
        print(i, "out of ", n, " -- Status", resp, " -- Message = ", msg)
        yield resp, n, msg


if __name__ == "__main__":
    for resp, total, msg in produce_bunch_tasks():
        pass
```

init_producer.py

```
from flask import Flask

#Create a Flask instance
app = Flask(__name__)

#Load Flask configurations from config.py
app.secret_key = app.config['SECRET_KEY']
app.config.from_object("config")
```

app_producer.py

```
#Flask imports
from flask import Response, render_template
from init_producer import app
import tasks_producer

def stream_template(template_name, **context):
    app.update_template_context(context)
    t = app.jinja_env.get_template(template_name)
```

```
    rv = t.stream(context)
    rv.enable_buffering(5)
    return rv

@app.route("/", methods=['GET'])
def index():
    return render_template('producer.html')

@app.route('/producetasks', methods=['POST'])
def producetasks():
    print("producetasks")
    return Response(stream_template('producer.html', data=
tasks_producer.produce_bunch_tasks() ))

if __name__ == "__main__":
  app.run(host="localhost",port=5000, debug=True)
```

init_consumer.py

```
from flask import Flask

#Create a Flask instance
app = Flask(__name__)

#Load Flask configurations from config.py
app.secret_key = app.config['SECRET_KEY']
app.config.from_object("config")

#Setup the Flask SocketIO integration while mapping the Redis Server.
```

```
from flask_socketio import SocketIO
socketio = SocketIO(app,logger=True,engineio_logger=True,message_queue=app.config['BROKER_URL'])
```

app_consumer.py

```
#Flask imports
from flask import render_template, request,session
from flask_socketio import join_room
from init_consumer import app, socketio
import json
import uuid

#Render the assigned template file
@app.route("/", methods=['GET']) def index(): return render_template('consumer.html')

# Sending Message through the websocket
def send_message(event, namespace, room, message):
    # print("Message = ", message)
    socketio.emit(event, message, namespace=namespace, room=room)

# Registers a function to be run before the first request to this instance of the application
# Create a unique session ID and store it within the application configuration file
@app.before_first_request
def initialize_params():
```

```
  if not hasattr(app.config,'uid'):
    sid = str(uuid.uuid4())
    app.config['uid'] = sid
    print("initialize_params - Session ID stored =", sid)

# Receive the webhooks and emit websocket events
@app.route('/consumetasks', methods=['POST'])
def consumetasks():
  if request.method == 'POST':
    data = request.json
    if data:
      print("Received Data = ", data)
      roomid =  app.config['uid']
      var = json.dumps(data)
      send_message(event='msg', namespace='/collectHooks',
room=roomid, message=var)
  return 'OK'

#Execute on connecting
@socketio.on('connect', namespace='/collectHooks')
def socket_connect():
  # Display message upon connecting to the namespace
  print('Client Connected To NameSpace /collectHooks - ', request.sid)

#Execute on disconnecting
@socketio.on('disconnect', namespace='/collectHooks')
def socket_connect():
  # Display message upon disconnecting from the namespace
```

```
    print('Client disconnected From NameSpace /collectHooks - ', request.sid)

#Execute upon joining a specific room
@socketio.on('join_room', namespace='/collectHooks')
def on_room():
    if app.config['uid']:
        room = str(app.config['uid'])
        # Display message upon joining a room specific to the session previously stored.
        print(f"Socket joining room {room}")
        join_room(room)

#Execute upon encountering any error related to the websocket
@socketio.on_error_default
def error_handler(e):
    # Display message on error.
    print(f"socket error: {e}, {str(request.event)}")

#Run using port 5001
if __name__ == "__main__":
    socketio.run(app,host='localhost', port=5001,debug=True)
```

templates/producer.html

```
<!doctype html>
<html>
  <head>
    <title>Tasks Producer</title>
```

```
  <style>
    .content {
      width: 100%;
    }
    .container{
      max-width: none;
    }
  </style>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
 </head>

<body class="container">
  <div class="content">
    <form method='post' id="produceTasksForm" action = "/producetasks">
      <button style="height:20%x;width:100%" type="submit" id="produceTasks">Produce Tasks</button>
    </form>
  </div>

  <div class="content">
      <div id="Messages" class="content" style="height:400px;width:100%; border:2px solid gray; overflow-y:scroll;"></div>
      {% for rsp,total, msg in data: %}
       <script>
          var rsp   = "{{ rsp }}";
          var total = "{{ total }}";
          var msg   = "{{ msg }}";
```

```
        var lineidx = "{{ loop.index }}";
        //If the webhook request succeeds color it in blue else in red.
        if (rsp == '200') {
           rsp = rsp.fontcolor("blue");
        }
        else {
           rsp = rsp.fontcolor("red");
        }
        //Add the details of the generated task to the Messages section.
        document.getElementById('Messages').innerHTML += "<br>" +
lineidx  + " out of " + total + " -- "+ rsp + " -- " + msg;
    </script>
    {% endfor %}
  </div>
</body>
</html>
```

templates/consumer.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Tasks Consumer</title>

  <link rel="stylesheet" href="
{{url_for('static',filename='css/bootstrap.min.css')}}">
  <link rel="stylesheet" href="
{{url_for('static',filename='css/Chart.min.css')}}">
</head>
```

```
<body>
  <div class="content">
    <div id="Messages" class="content" style="height:200px;width:100%;
border:1px solid gray; overflow-y:scroll;"></div>
  </div>

  <div class="container">
    <div class="row">
      <div class="col-12">
        <div class="card">
          <div class="card-body">
            <canvas id="canvas"></canvas>
          </div>
        </div>
      </div>
    </div>
  </div>

  <!-- import the jquery library -->
  <script src="{{ url_for('static',filename='js/jquery.min.js') }}"></script>
  <!-- import the socket.io library -->
  <script src="{{ url_for('static',filename='js/socket.io.js') }}"></script>

  <!-- import the bootstrap library -->
  <script src="{{ url_for('static',filename='js/bootstrap.min.js') }}">
</script>
  <!-- import the Chart library -->
  <script src="{{ url_for('static',filename='js/Chart.min.js') }}"></script>
```

```
<script>
    $(document).ready(function(){
     const config = {
        //Type of the chart - Bar Chart
        type: 'bar',
        //Data for our chart
        data: {
           labels: ['Low','Moderate','Major','Critical'],
           datasets: [{
              label: "Count Of Tasks",
              //Setting a color for each bar
              backgroundColor: ['green','blue','yellow','red'],
              borderColor: 'rgb(255, 99, 132)',
              data: [0,0,0,0],
              fill: false,
           }],
        },
        //Configuration options
        options: {
           responsive: true,
           title: {
              display: true,
              text: 'Tasks Priority Matrix'
           },
           tooltips: {
              mode: 'index',
              intersect: false,
           },
           hover: {
```

```
                mode: 'nearest',                intersect: true                },
            scales: {                xAxes: [{                display: true,
                        scaleLabel: {                display: true,
                labelString: 'Priority'                }
            }],
            yAxes: [{
              display: true
             ,ticks: {
                labelString: 'Total'
              }
             ,scaleLabel: {
                display: true,
                labelString: 'Total'
              }
            }]
          }
        }
    };
    const context = document.getElementById('canvas').getContext('2d');
    //Creating the bar chart
    const lineChart = new Chart(context, config);
```

```
    //Reserved for websocket manipulation
    var namespace='/collectHooks';
    var url = 'http://' + document.domain + ':' + location.port + namespace;
    var socket = io.connect(url);

    //When connecting to the socket join the room
    socket.on('connect', function() {
                    socket.emit('join_room');
                });

    //When receiving a message
    socket.on('msg' , function(data) {
                var msg = JSON.parse(data);
                var newLine = $('<li>'+ 'Batch ID. = ' + msg.batchid + ' -- Task ID. = ' + msg.id + ' -- Owner = ' + msg.owner + ' -- Priority = ' + msg.priority +'</li>');
                newLine.css("color","blue");
                $("#Messages").append(newLine);

                //Retrieve the index of the priority of the received message
                var lindex = config.data.labels.indexOf(msg.priority);

                //Increment the value of the priority of the received message
                config.data.datasets[0].data[lindex] += 1;

                //Update the chart
                lineChart.update();
            });
```

```
    });
</script>
</body>
</html>
```

PART 4: Extract YouTube Data using YouTube API in Python

Learn how to extract YouTube data including video and channel details, searching by keyword or channel and extracting comments with YouTube API in Python.

YouTube is no doubt the biggest video-sharing website on the Internet. It is one of the main sources of education, entertainment, advertisement, and many more fields. Since it's a data-rich website, accessing its API will enable you to get almost all of the YouTube data.

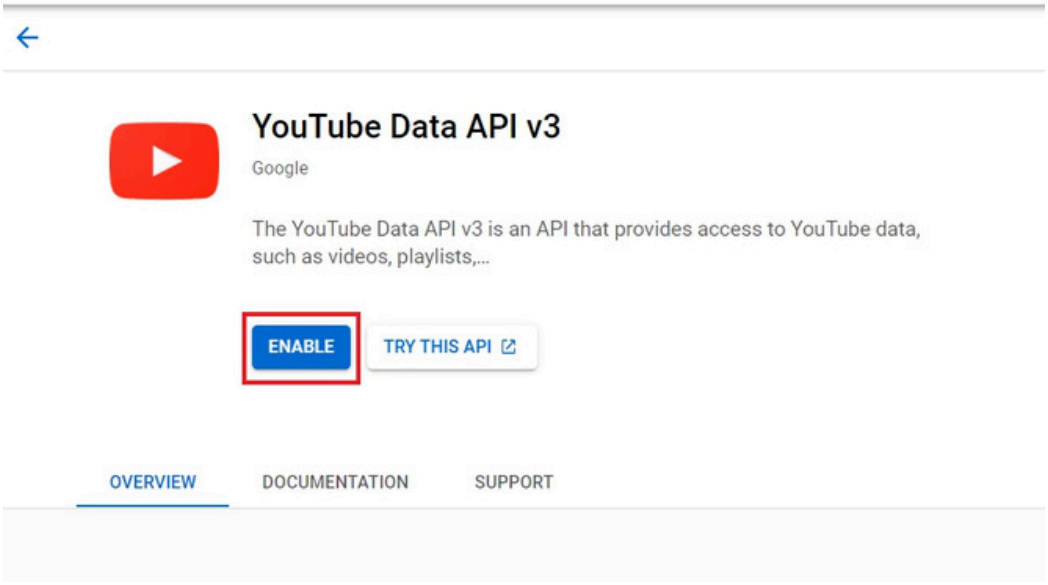
In this tutorial, we'll cover how to get YouTube video details and statistics, search by keyword, get YouTube channel information, and extract comments from both videos and channels, using YouTube API with Python.

Here is the table of contents:

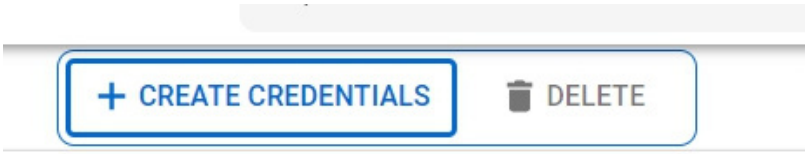
Enabling YouTube API

To enable YouTube Data API, you should follow below steps:

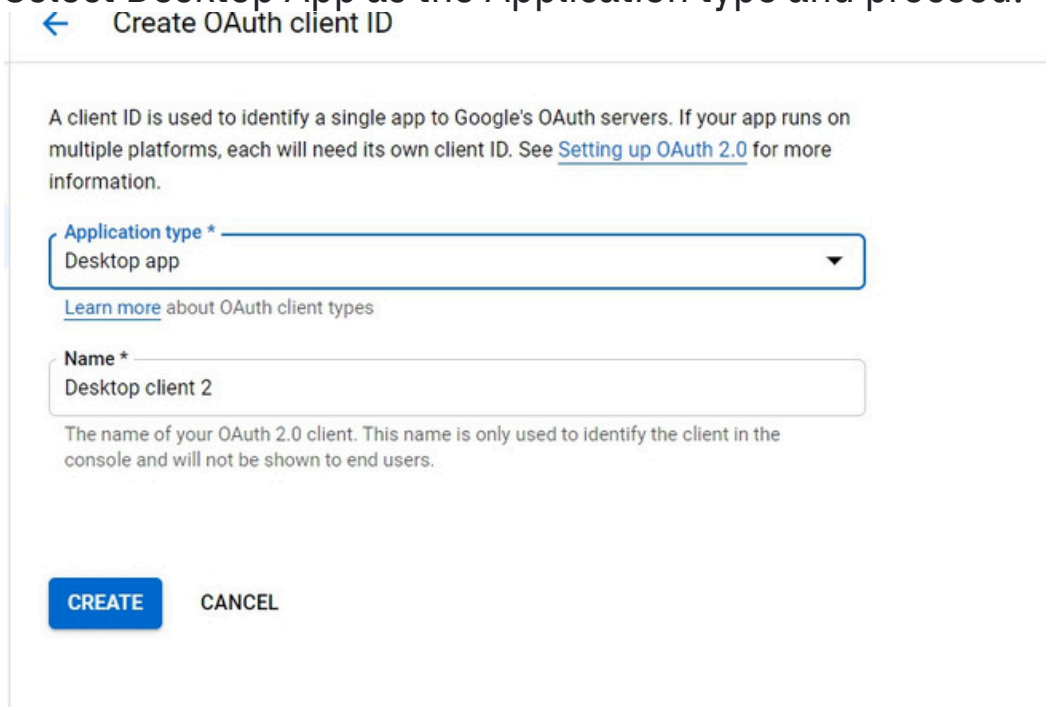
1. Go to Google's API Console and create a project, or use an existing one.
2. In the library panel, search for **YouTube Data API v3**, click on it and click **Enable**.



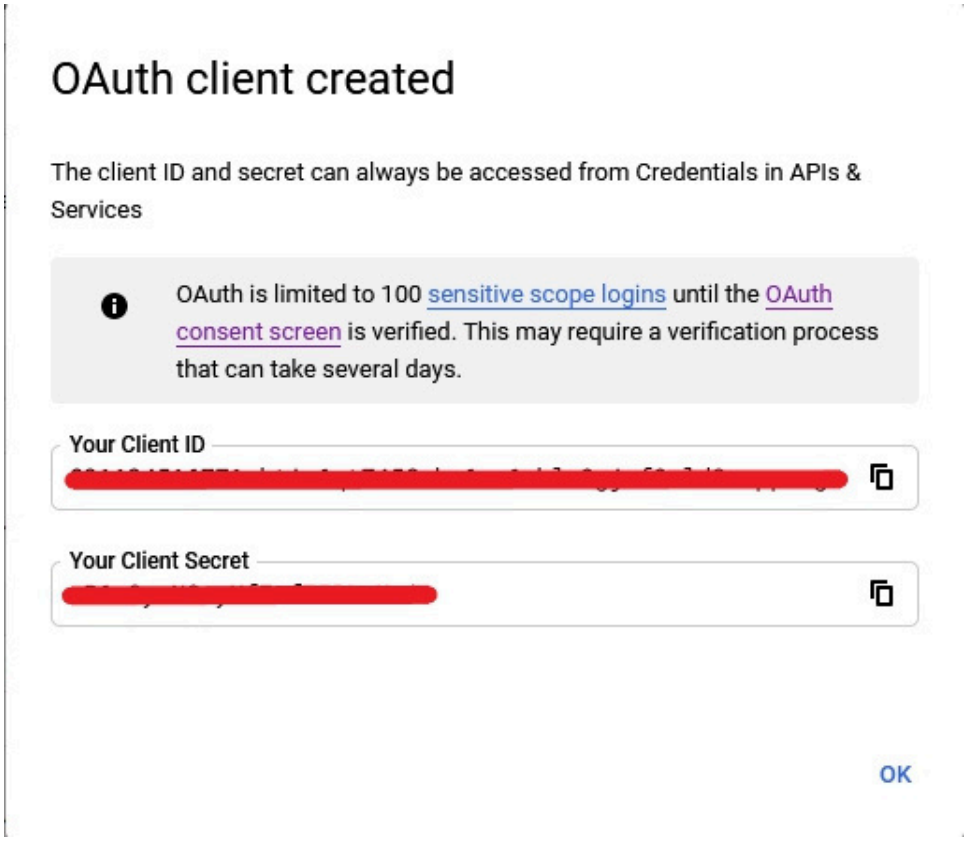
3. In the credentials panel, click on **Create Credentials**, and choose **OAuth client ID**.



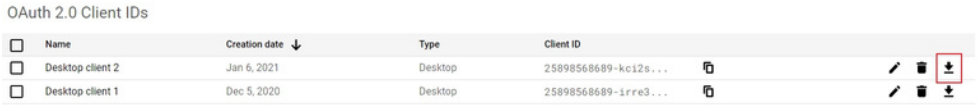
4. Select Desktop App as the Application type and proceed.



5. You'll see a window like this:



6. Click **OK** and download the credentials file and rename it to `credentials.json` :



Note: *If this is the first time you use Google APIs, you may need to simply create an OAuth Consent screen and add your email as a testing user.*

Now that you have set up YouTube API, get your `credentials.json` in the current directory of your notebook/Python file, and let's get started.

First, install required libraries:

```
$ pip3 install --upgrade google-api-python-client google-auth-httplib2 google-auth-oauthlib
```

Now let's import the necessary modules we gonna need:

```
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request

import urllib.parse as p
import re
import os
import pickle

SCOPES = ["https://www.googleapis.com/auth/youtube.force-ssl"]
```

`SCOPES` is a list of scopes of using YouTube API; we're using this one to view all YouTube data without any problems.

Now let's make the function that authenticates with YouTube API:

```
def youtube_authenticate():
    os.environ["OAUTHLIB_INSECURE_TRANSPORT"] = "1"
    api_service_name = "youtube"
    api_version = "v3"
    client_secrets_file = "credentials.json"
    creds = None
    # the file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first time
    if os.path.exists("token.pickle"):
        with open("token.pickle", "rb") as token:
            creds = pickle.load(token)
    # if there are no (valid) credentials availablle, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow =
InstalledAppFlow.from_client_secrets_file(client_secrets_file, SCOPES)
            creds = flow.run_local_server(port=0)
        # save the credentials for the next run
        with open("token.pickle", "wb") as token:
            pickle.dump(creds, token)

    return build(api_service_name, api_version, credentials=creds)
```

```
# authenticate to YouTube API
youtube = youtube_authenticate()
```

`youtube_authenticate()` looks for the `credentials.json` file that we downloaded earlier, and try to authenticate using that file, this will open your default browser the first time you run it, so you accept the permissions. After that, it'll save a new file `token.pickle` that contains the authorized credentials.

It should look familiar if you used a Google API before, such as Gmail API, Google Drive API, or something else. The prompt in your default browser is to accept permissions required for the app. If you see a window that indicates the app isn't verified, you may just want to head to **Advanced** and click on your App name.

Getting Video Details

Now that you have everything set up, let's begin with extracting YouTube video details, such as title, description, upload time, and even statistics such as view count, and like count.

The following function will help us extract the video ID (that we'll need in the API) from a video URL:

```
def get_video_id_by_url(url):
    """
    Return the Video ID from the video `url`
    """
    # split URL parts
    parsed_url = p.urlparse(url)
```

```
    # get the video ID by parsing the query of the URL
    video_id = p.parse_qs(parsed_url.query).get("v")
    if video_id:
        return video_id[0]
    else:
        raise Exception(f"Wasn't able to parse video URL: {url}")
```

We simply used the urllib.parse module to get the video ID from a URL.

The below function gets a YouTube service object (returned from `youtube_authenticate()` function), as well as any keyword argument accepted by the API, and returns the API response for a specific video:

```
def get_video_details(youtube, **kwargs):
    return youtube.videos().list(
        part="snippet,contentDetails,statistics",
        **kwargs
    ).execute()
```

Notice we specified part of `snippet`, `contentDetails` and `statistics`, as these are the most important parts of the response in the API.

We also pass `kwargs` to the API directly. Next, let's define a function that takes a response returned from the above `get_video_details()` function, and prints the most useful information from a video:

```
def print_video_infos(video_response):
    items = video_response.get("items")[0]
```

```
    # get the snippet, statistics & content details from the video response
    snippet         = items["snippet"]
    statistics      = items["statistics"]
    content_details = items["contentDetails"]
    # get infos from the snippet
    channel_title = snippet["channelTitle"]
    title         = snippet["title"]
    description   = snippet["description"]
    publish_time  = snippet["publishedAt"]
    # get stats infos
    comment_count = statistics["commentCount"]
    like_count    = statistics["likeCount"]
    view_count    = statistics["viewCount"]
    # get duration from content details
    duration = content_details["duration"]
    # duration in the form of something like 'PT5H50M15S'
    # parsing it to be something like '5:50:15'
    parsed_duration = re.search(f"PT(\d+H)?(\d+M)?(\d+S)",
duration).groups()
    duration_str = ""
    for d in parsed_duration:
        if d:
            duration_str += f"{d[:-1]}:"
    duration_str = duration_str.strip(":")
    print(f"""\
    Title: {title}
    Description: {description}
    Channel Title: {channel_title}
    Publish time: {publish_time}
```

```
    Duration: {duration_str}
    Number of comments: {comment_count}
    Number of likes: {like_count}
    Number of views: {view_count}
    """)
```

Finally, let's use these functions to extract information from a demo video:

```
video_url = "https://www.youtube.com/watch?v=jNQXAC9IVRw&ab_channel=jawed"
# parse video ID from URL
video_id = get_video_id_by_url(video_url)
# make API call to get video info
response = get_video_details(youtube, id=video_id)
# print extracted video infos
print_video_infos(response)
```

We first get the video ID from the URL, and then we get the response from the API call and finally print the data. Here is the output:

```
    Title: Me at the zoo
    Description: The first video on YouTube. Maybe it's time to go back to the zoo?
    Channel Title: jawed
    Publish time: 2005-04-24T03:31:52Z
    Duration: 19
    Number of comments: 11018071
    Number of likes: 5962957
```

```
    Number of views: 138108884
```

You see, we used the `id` parameter to get the details of a specific video, you can also set multiple video IDs separated by commas, so you make a single API call to get details about multiple videos, check the documentation for more detailed information.

Searching By Keyword

Searching using YouTube API is straightforward; we simply pass `q` parameter for query, the same query we use in the YouTube search bar:

```
def search(youtube, **kwargs):
    return youtube.search().list(
        part="snippet",
        **kwargs
    ).execute()
```

This time we care about the snippet, and we use `search()` instead of `videos()` like in the previously defined `get_video_details()` function.

Let's, for example, search for `"python"` and limit the results to only 2:

```
# search for the query 'python' and retrieve 2 items only
response = search(youtube, q="python", maxResults=2)
items = response.get("items")
for item in items:
    # get the video ID
    video_id = item["id"]["videoId"]
```

```
    # get the video details
    video_response = get_video_details(youtube, id=video_id)
    # print the video details
    print_video_infos(video_response)
    print("="*50)
```

We set `maxResults` to 2 so we retrieve the first two items, here is a part of the output:

```
Title: Learn Python - Full Course for Beginners [Tutorial]
    Description: This course will give you a full introduction into all of the core concepts in python...<SNIPPED>
    Channel Title: freeCodeCamp.org
    Publish time: 2018-07-11T18:00:42Z
    Duration: 4:26:52
    Number of comments: 30307
    Number of likes: 520260
    Number of views: 21032973
==================================================
    Title: Python Tutorial - Python for Beginners [Full Course]
    Description: Python tutorial - Python for beginners
  Learn Python programming for a career in machine learning, data science & web development...<SNIPPED>
    Channel Title: Programming with Mosh
    Publish time: 2019-02-18T15:00:08Z
    Duration: 6:14:7
    Number of comments: 38019
    Number of likes: 479749
    Number of views: 15575418
```

You can also specify the `order` parameter in `search()` function to order search results, which can be `'date'`, `'rating'`, `'viewCount'`, `'relevance'` (default), `'title'`, and `'videoCount'`.

Another useful parameter is the `type`, which can be `'channel'`, `'playlist'` or `'video'`, default is all of them. Please check this page for more information about the `search().list()` method.

Getting YouTube Channel Details

This section will take a channel URL and extract channel information using YouTube API.

First, we need helper functions to parse the channel URL. The below functions will help us to do that:

```
def parse_channel_url(url):
    """
    This function takes channel `url` to check whether it includes a
    channel ID, user ID or channel name
    """
    path = p.urlparse(url).path
    id = path.split("/")[-1]
    if "/c/" in path:
        return "c", id
    elif "/channel/" in path:
        return "channel", id
    elif "/user/" in path:
        return "user", id
```

```python
def get_channel_id_by_url(youtube, url):
    """
    Returns channel ID of a given `id` and `method`
    - `method` (str): can be 'c', 'channel', 'user'
    - `id` (str): if method is 'c', then `id` is display name
        if method is 'channel', then it's channel id
        if method is 'user', then it's username
    """
    # parse the channel URL
    method, id = parse_channel_url(url)
    if method == "channel":
        # if it's a channel ID, then just return it
        return id
    elif method == "user":
        # if it's a user ID, make a request to get the channel ID
        response = get_channel_details(youtube, forUsername=id)
        items = response.get("items")
        if items:
            channel_id = items[0].get("id")
            return channel_id
    elif method == "c":
        # if it's a channel name, search for the channel using the name
        # may be inaccurate
        response = search(youtube, q=id, maxResults=1)
        items = response.get("items")
        if items:
            channel_id = items[0]["snippet"]["channelId"]
            return channel_id
```

```
    raise Exception(f"Cannot find ID:{id} with {method} method")
```

Now we can parse the channel URL. Let's define our functions to call the YouTube API:

```
def get_channel_videos(youtube, **kwargs):
    return youtube.search().list(
        **kwargs
    ).execute()


def get_channel_details(youtube, **kwargs):
    return youtube.channels().list(
        part="statistics,snippet,contentDetails",
        **kwargs
    ).execute()
```

We'll be using `get_channel_videos()` to get the videos of a specific channel, and `get_channel_details()` will allow us to extract information about a specific youtube channel.

Now that we have everything, let's make a concrete example:

```
channel_url = "https://www.youtube.com/channel/UC8butISFwT-Wl7EV0hUK0BQ"
# get the channel ID from the URL
channel_id = get_channel_id_by_url(youtube, channel_url)
# get the channel details
response = get_channel_details(youtube, id=channel_id)
# extract channel infos
```

```
snippet = response["items"][0]["snippet"]
statistics = response["items"][0]["statistics"]
channel_country = snippet["country"]
channel_description = snippet["description"]
channel_creation_date = snippet["publishedAt"]
channel_title = snippet["title"]
channel_subscriber_count = statistics["subscriberCount"]
channel_video_count = statistics["videoCount"]
channel_view_count  = statistics["viewCount"]
print(f"""
Title: {channel_title}
Published At: {channel_creation_date}
Description: {channel_description}
Country: {channel_country}
Number of videos: {channel_video_count}
Number of subscribers: {channel_subscriber_count}
Total views: {channel_view_count}
""")
# the following is grabbing channel videos
# number of pages you want to get
n_pages = 2
# counting number of videos grabbed
n_videos = 0
next_page_token = None
for i in range(n_pages):
    params = {
        'part': 'snippet',
        'q': '',
        'channelId': channel_id,
```

```
            'type': 'video',
        }
        if next_page_token:
            params['pageToken'] = next_page_token
        res = get_channel_videos(youtube, **params)
        channel_videos = res.get("items")
        for video in channel_videos:
            n_videos += 1
            video_id = video["id"]["videoId"]
            # easily construct video URL by its ID
            video_url = f"https://www.youtube.com/watch?v={video_id}"
            video_response = get_video_details(youtube, id=video_id)
            print(f"================Video #
{n_videos}================")
            # print the video details
            print_video_infos(video_response)
            print(f"Video URL: {video_url}")
            print("="*40)
        print("*"*100)
        # if there is a next page, then add it to our parameters
        # to proceed to the next page
        if "nextPageToken" in res:
            next_page_token = res["nextPageToken"]
```

We first get the channel ID from the URL, and then we make an API call to get channel details and print them.

After that, we specify the number of pages of videos we want to extract. The default is ten videos per page, and we can also change

that by passing the `maxResults` parameter. We iterate on each video and make an API call to get various information about the video, and we use our predefined `print_video_infos()` to print the video information.

Here is a part of the output:

```
==========Video #1==========
Title: Async + Await in JavaScript, talk from Wes Bos
Description: Flow Control in JavaScript is hard! ...
Channel Title: freeCodeCamp.org
Publish time: 2018-04-16T16:58:08Z
Duration: 15:52
Number of comments: 52
Number of likes: 2353
Number of views: 74562
Video URL: https://www.youtube.com/watch?v=DwQJ_NPQWWo
==========Video #2==========
Title: Protected Routes in React using React Router
Description: In this video, we will create a protected route using...
Channel Title: freeCodeCamp.org
Publish time: 2018-10-16T16:00:05Z
Duration: 15:40
Number of comments: 158
Number of likes: 3331
Number of views: 173927
Video URL: https://www.youtube.com/watch?v=Y0-qdp-XBJg
...<SNIPPED>
```

You can get other information; you can print the `response` dictionary for further information or check the documentation for this endpoint.

Extracting YouTube Comments

YouTube API allows us to extract comments; this is useful if you want to get comments for your text classification project or something similar.

The below function takes care of making an API call to `commentThreads()`:

```
def get_comments(youtube, **kwargs):
  return youtube.commentThreads().list(
    part="snippet",
    **kwargs
  ).execute()
```

The below code extracts comments from a YouTube video:

```
# URL can be a channel or a video, to extract comments
url = "https://www.youtube.com/watch?v=jNQXAC9IVRw&ab_channel=jawed"
if "watch" in url:
  # that's a video
  video_id = get_video_id_by_url(url)
  params = {
    'videoId': video_id,
    'maxResults': 2,
    'order': 'relevance', # default is 'time' (newest)
```

```
    }
else:
    # should be a channel
    channel_id = get_channel_id_by_url(url)
    params = {
        'allThreadsRelatedToChannelId': channel_id,
        'maxResults': 2,
        'order': 'relevance', # default is 'time' (newest)
    }
# get the first 2 pages (2 API requests)
n_pages = 2
for i in range(n_pages):
    # make API call to get all comments from the channel (including posts & videos)
    response = get_comments(youtube, **params)
    items = response.get("items")
    # if items is empty, breakout of the loop
    if not items:
        break
    for item in items:
        comment = item["snippet"]["topLevelComment"]["snippet"]["textDisplay"]
        updated_at = item["snippet"]["topLevelComment"]["snippet"]["updatedAt"]
        like_count = item["snippet"]["topLevelComment"]["snippet"]["likeCount"]
        comment_id = item["snippet"]["topLevelComment"]["id"]
        print(f"""\
        Comment: {comment}
```

```
        Likes: {like_count}
        Updated At: {updated_at}
        ==================================\
        """)
    if "nextPageToken" in response:
        # if there is a next page
        # add next page token to the params we pass to the function
        params["pageToken"] =  response["nextPageToken"]
    else:
        # must be end of comments!!!!
        break
    print("*"*70)
```

You can also change `url` variable to be a YouTube channel URL so that it will pass `allThreadsRelatedToChannelId` instead of `videoId` as a parameter to `commentThreads()` API.

We're extracting two comments per page and two pages, so four comments in total. Here is the output:

```
        Comment: We&#39;re so honored that the first ever YouTube video was filmed here!
        Likes: 877965
        Updated At: 2020-02-17T18:58:15Z
            ==================================
        Comment: Wow, still in your recommended in 2021? Nice! Yay
        Likes: 10951
        Updated At: 2021-01-04T15:32:38Z
            ==================================
```

```
**********************************************************
*********
    Comment: How many are seeing this video now
    Likes: 7134
    Updated At: 2021-01-03T19:47:25Z
        ==================================
    Comment: The first youtube video EVER. Wow.
    Likes: 865
    Updated At: 2021-01-05T00:55:35Z
        ==================================
**********************************************************
*********
```

We're extracting the comment itself, the number of likes, and the last updated date; you can explore the response dictionary to get various other useful information.

You're free to edit the parameters we passed, such as increasing the `maxResults`, or changing the `order`. Please check the page for this API endpoint.

Summary

YouTube Data API provides a lot more than what we covered here. If you have a YouTube channel, you can upload, update and delete videos, and much more.

Fullcode:

utils.py

```
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request

import urllib.parse as p
import re
import os
import pickle

SCOPES = ["https://www.googleapis.com/auth/youtube.force-ssl"]

def youtube_authenticate():
  os.environ["OAUTHLIB_INSECURE_TRANSPORT"] = "1"
  api_service_name = "youtube"
  api_version = "v3"
  client_secrets_file = "credentials.json"
  creds = None
  # the file token.pickle stores the user's access and refresh tokens, and is
  # created automatically when the authorization flow completes for the
first time
  if os.path.exists("token.pickle"):
    with open("token.pickle", "rb") as token:
      creds = pickle.load(token)
  # if there are no (valid) credentials availablle, let the user log in.
  if not creds or not creds.valid:
    if creds and creds.expired and creds.refresh_token:
      creds.refresh(Request())
```

```
    else:
        flow = InstalledAppFlow.from_client_secrets_file(client_secrets_file, SCOPES)
        creds = flow.run_local_server(port=0)
    # save the credentials for the next run
    with open("token.pickle", "wb") as token:
        pickle.dump(creds, token)

  return build(api_service_name, api_version, credentials=creds)


def get_channel_details(youtube, **kwargs):
  return youtube.channels().list(
    part="statistics,snippet,contentDetails",
    **kwargs
  ).execute()


def search(youtube, **kwargs):
  return youtube.search().list(
    part="snippet",
    **kwargs
  ).execute()


def get_video_details(youtube, **kwargs):
  return youtube.videos().list(
    part="snippet,contentDetails,statistics",
    **kwargs
```

```
    ).execute()


def print_video_infos(video_response):
    items = video_response.get("items")[0]
    # get the snippet, statistics & content details from the video response
    snippet         = items["snippet"]
    statistics      = items["statistics"]
    content_details = items["contentDetails"]
    # get infos from the snippet
    channel_title = snippet["channelTitle"]
    title         = snippet["title"]
    description   = snippet["description"]
    publish_time  = snippet["publishedAt"]
    # get stats infos
    comment_count = statistics["commentCount"]
    like_count    = statistics["likeCount"]
    view_count    = statistics["viewCount"]
    # get duration from content details
    duration = content_details["duration"]
    # duration in the form of something like 'PT5H50M15S'
    # parsing it to be something like '5:50:15'
    parsed_duration = re.search(f"PT(\d+H)?(\d+M)?(\d+S)",
duration).groups()
    duration_str = ""
    for d in parsed_duration:
        if d:
            duration_str += f"{d[:-1]}:"
    duration_str = duration_str.strip(":")
```

```
    print(f"""
    Title: {title}
    Description: {description}
    Channel Title: {channel_title}
    Publish time: {publish_time}
    Duration: {duration_str}
    Number of comments: {comment_count}
    Number of likes: {like_count}
    Number of views: {view_count}
    """)


def parse_channel_url(url):
    """
    This function takes channel `url` to check whether it includes a
    channel ID, user ID or channel name
    """
    path = p.urlparse(url).path
    id = path.split("/")[-1]
    if "/c/" in path:
        return "c", id
    elif "/channel/" in path:
        return "channel", id
    elif "/user/" in path:
        return "user", id


def get_channel_id_by_url(youtube, url):
    """
```

```
    Returns channel ID of a given `id` and `method`
    - `method` (str): can be 'c', 'channel', 'user'
    - `id` (str): if method is 'c', then `id` is display name
        if method is 'channel', then it's channel id
        if method is 'user', then it's username
    """
    # parse the channel URL
    method, id = parse_channel_url(url)
    if method == "channel":
        # if it's a channel ID, then just return it
        return id
    elif method == "user":
        # if it's a user ID, make a request to get the channel ID
        response = get_channel_details(youtube, forUsername=id)
        items = response.get("items")
        if items:
            channel_id = items[0].get("id")
            return channel_id
    elif method == "c":
        # if it's a channel name, search for the channel using the name
        # may be inaccurate
        response = search(youtube, q=id, maxResults=1)
        items = response.get("items")
        if items:
            channel_id = items[0]["snippet"]["channelId"]
            return channel_id
    raise Exception(f"Cannot find ID:{id} with {method} method")
```

```
def get_video_id_by_url(url):
    """
    Return the Video ID from the video `url`
    """
    # split URL parts
    parsed_url = p.urlparse(url)
    # get the video ID by parsing the query of the URL
    video_id = p.parse_qs(parsed_url.query).get("v")
    if video_id:
        return video_id[0]
    else:
        raise Exception(f"Wasn't able to parse video URL: {url}")
```

video_details.py

```
from utils import (
    youtube_authenticate,
    get_video_id_by_url,
    get_video_details,
    print_video_infos
)


if __name__ == "__main__":
    # authenticate to YouTube API
    youtube = youtube_authenticate()
    video_url = "https://www.youtube.com/watch?
v=jNQXAC9IVRw&ab_channel=jawed"
    # parse video ID from URL
```

```
    video_id = get_video_id_by_url(video_url)    # make
API   call   to   get   video   info              response   =
get_video_details(youtube,  id=video_id)         #  print
extracted video infos    print_video_infos(response)
```

search_by_keyword.py

```
from utils import (
    youtube_authenticate,
    get_video_details,
    print_video_infos,
    search
)


if __name__ == "__main__":
    # authenticate to YouTube API
    youtube = youtube_authenticate()
    # search for the query 'python' and retrieve 2 items only
    response = search(youtube, q="python", maxResults=2)
    items = response.get("items")
    for item in items:
        # get the video ID
        video_id = item["id"]["videoId"]
        # get the video details
        video_response = get_video_details(youtube, id=video_id)
        # print the video details
        print_video_infos(video_response)
```

```
    print("="*50)
```

channel_details.py from

```
utils import (
  youtube_authenticate,
  get_channel_id_by_url,
  get_channel_details,
  get_video_details,
  print_video_infos
)


def get_channel_videos(youtube, **kwargs):
  return youtube.search().list(
    **kwargs
  ).execute()


if __name__ == "__main__":
  # authenticate to YouTube API
  youtube = youtube_authenticate()
  channel_url = "https://www.youtube.com/channel/UC8butISFwT-Wl7EV0hUK0BQ"
  # get the channel ID from the URL
  channel_id = get_channel_id_by_url(youtube, channel_url)
  # get the channel details
  response = get_channel_details(youtube, id=channel_id)
  # extract channel infos
```

```
snippet = response["items"][0]["snippet"]
statistics = response["items"][0]["statistics"]
channel_country = snippet["country"]
channel_description = snippet["description"]
channel_creation_date = snippet["publishedAt"]
channel_title = snippet["title"]
channel_subscriber_count = statistics["subscriberCount"]
channel_video_count = statistics["videoCount"]
channel_view_count  = statistics["viewCount"]
print(f"""
Title: {channel_title}
Published At: {channel_creation_date}
Description: {channel_description}
Country: {channel_country}
Number of videos: {channel_video_count}
Number of subscribers: {channel_subscriber_count}
Total views: {channel_view_count}
""")
# the following is grabbing channel videos
# number of pages you want to get
n_pages = 2
# counting number of videos grabbed
n_videos = 0
next_page_token = None
for i in range(n_pages):
    params = {
        'part': 'snippet',
        'q': '',
        'channelId': channel_id,
```

```
        'type': 'video',
    }
    if next_page_token:
        params['pageToken'] = next_page_token
    res = get_channel_videos(youtube, **params)
    channel_videos = res.get("items")
    for video in channel_videos:
        n_videos += 1
        video_id = video["id"]["videoId"]
        # easily construct video URL by its ID
        video_url = f"https://www.youtube.com/watch?v={video_id}"
        video_response = get_video_details(youtube, id=video_id)
        print(f"================Video #
{n_videos}================")
        # print the video details
        print_video_infos(video_response)
        print(f"Video URL: {video_url}")
        print("="*40)
    # if there is a next page, then add it to our parameters
    # to proceed to the next page
    if "nextPageToken" in res:
        next_page_token = res["nextPageToken"]
```

comments.py

```
from utils import youtube_authenticate, get_video_id_by_url,
get_channel_id_by_url
```

```
def get_comments(youtube, **kwargs):
    return youtube.commentThreads().list(
        part="snippet",
        **kwargs
    ).execute()



if __name__ == "__main__":
    # authenticate to YouTube API
    youtube = youtube_authenticate()
    # URL can be a channel or a video, to extract comments
    url = "https://www.youtube.com/watch?
v=jNQXAC9IVRw&ab_channel=jawed"
    if "watch" in url:
        # that's a video
        video_id = get_video_id_by_url(url)
        params = {
            'videoId': video_id,
            'maxResults': 2,
            'order': 'relevance', # default is 'time' (newest)
        }
    else:
        # should be a channel
        channel_id = get_channel_id_by_url(url)
        params = {
            'allThreadsRelatedToChannelId': channel_id,
            'maxResults': 2,
            'order': 'relevance', # default is 'time' (newest)
```

```
    }
  # get the first 2 pages (2 API requests)
  n_pages = 2
  for i in range(n_pages):
    # make API call to get all comments from the channel (including posts & videos)
    response = get_comments(youtube, **params)
    items = response.get("items")
    # if items is empty, breakout of the loop
    if not items:
      break
    for item in items:
      comment = item["snippet"]["topLevelComment"]["snippet"]["textDisplay"]
      updated_at = item["snippet"]["topLevelComment"]["snippet"]["updatedAt"]
      like_count = item["snippet"]["topLevelComment"]["snippet"]["likeCount"]
      comment_id = item["snippet"]["topLevelComment"]["id"]
      print(f"""\
      Comment: {comment}
      Likes: {like_count}
      Updated At: {updated_at}
      ==================================\
      """)
    if "nextPageToken" in response:
      # if there is a next page
      # add next page token to the params we pass to the function
      params["pageToken"] =  response["nextPageToken"]
```

```
    else:
        # must be end of comments!!!!
        break
    print("*"*70)
```

PART 5: Use Gmail API in Python

Learn how to use Gmail API to send emails, search for emails by query, delete emails, mark emails as read or unread in Python.

Gmail is by far the most popular mail service nowadays, it's used by individuals and organizations. Many of its features are enhanced with AI, including its security (and detection of fraudulent emails) and its suggestions when writing emails.

In the previous tutorials, we explained how you can send emails as well as reading emails with Python, if you didn't read them yet, I highly recommend you check them out.

While the previous tutorials were on using the IMAP/SMTP protocols directly, in this one, we will be using Google's API to send and read emails, by doing so, we can use features that are specific to Google Mail, for example; add labels to some emails, mark emails as unread/read and so on.

For this guide, we will explore some of the main features of the Gmail API, we will write several Python scripts that have the ability to send emails, search for emails, deletes, and mark as read or unread, they'll be used as follows:

```
$ python send_emails.py destination@gmail.com "Subject" "Message body" --files file1.txt file2.pdf file3.png
$ python read_emails.py "search query"
$ python delete_emails.py "search query"
$ python mark_emails.py --read "search query"
$ python mark_emails.py --unread "search query"
```

Here is the table of contents:

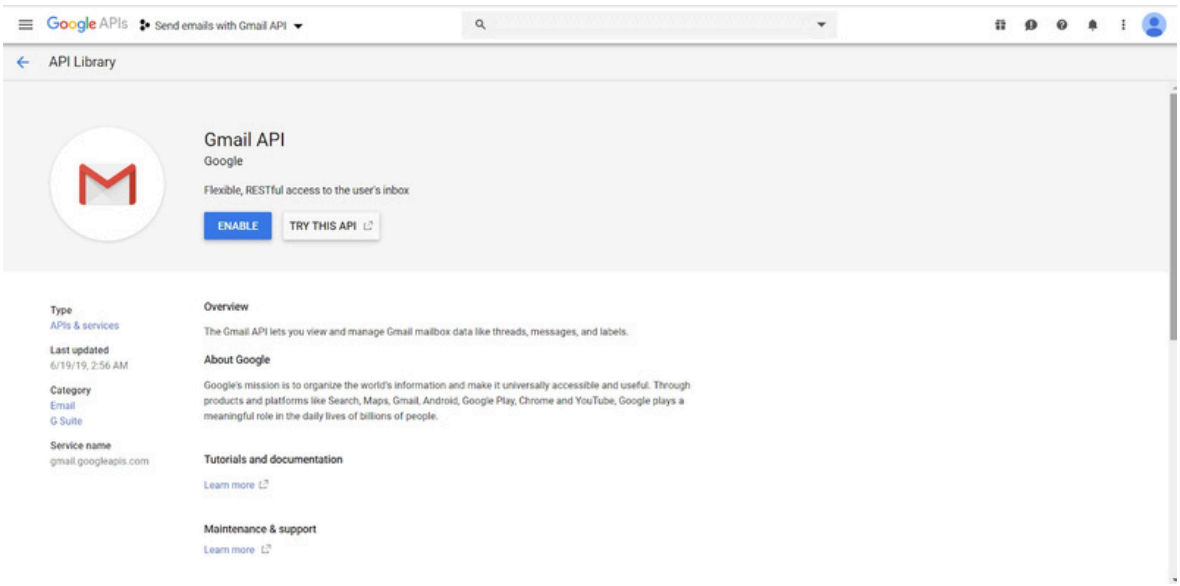
To get started, let's install the necessary dependencies:

```
$ pip3 install --upgrade google-api-python-client google-auth-httplib2 google-auth-oauthlib
```

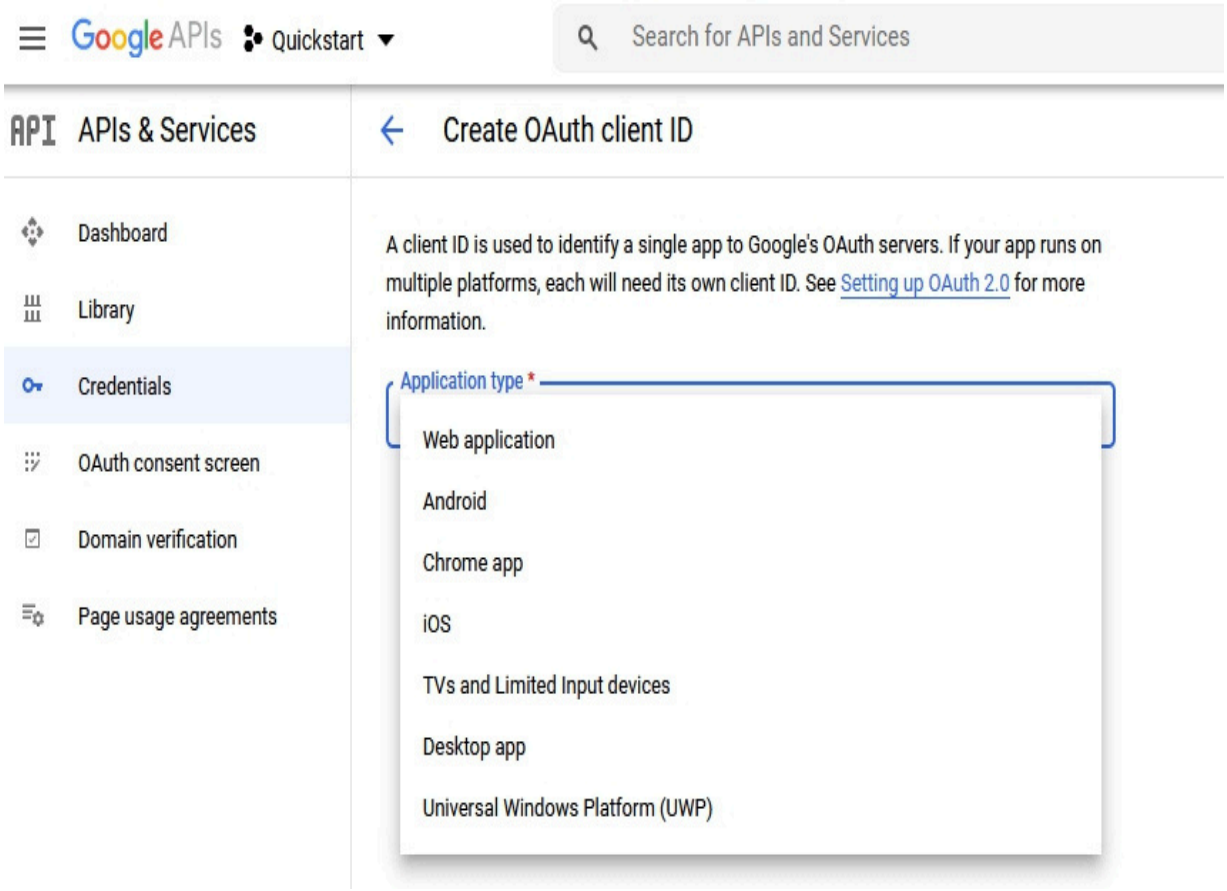
Enabling Gmail API

To use the Gmail API, we need a token to connect to Gmail's API, we can get one from the Google APIs' dashboard.

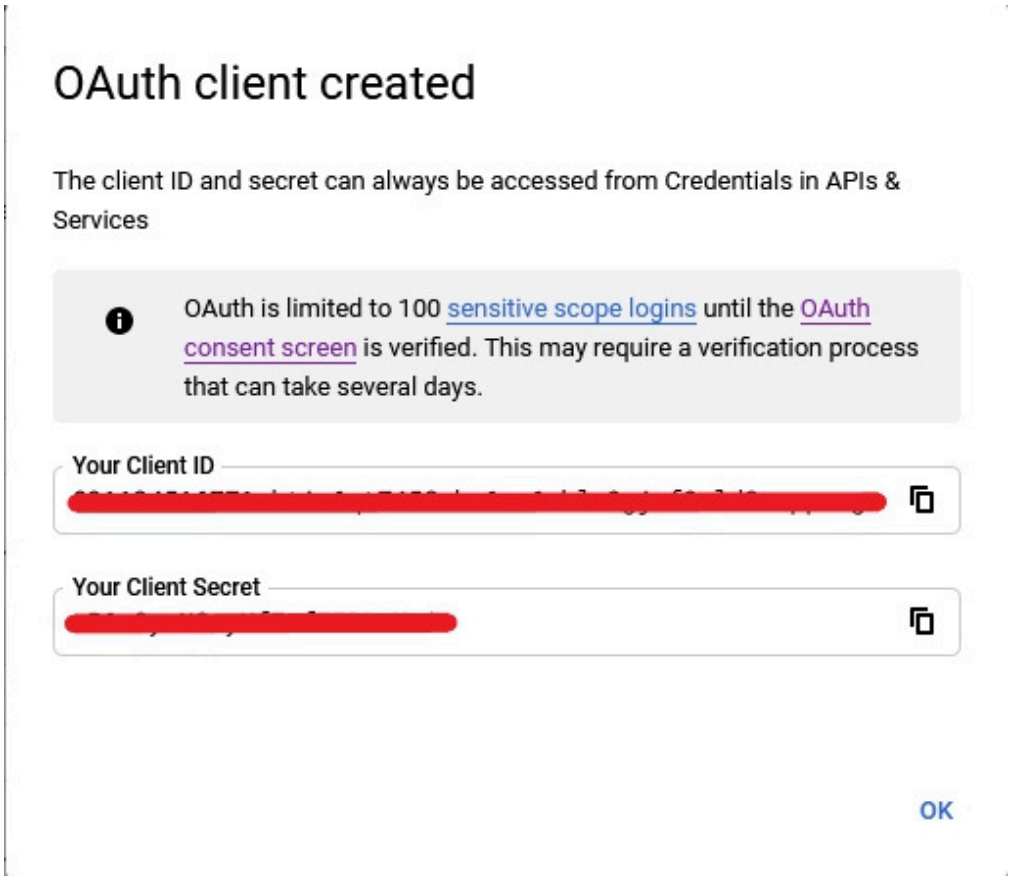
We first enable the Google mail API, head to the dashboard, and use the search bar to search for Gmail API, click on it, and then enable:



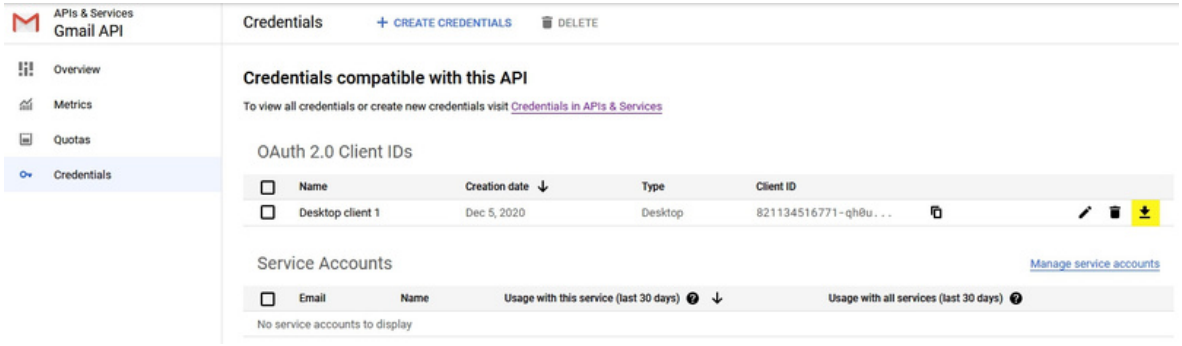
We then create an OAuth 2.0 client ID by creating credentials (by heading to the **Create Credentials** button):



Select **Desktop App** as the Application type and proceed, you'll see a window like this:



We download our credentials file and save it as `credentials.json` in the current directory:



Note: *If this is the first time you use Google APIs, you may need to simply create an OAuth Consent screen and add your email as a testing user.*

Now we're done with setting up the API, let's start by importing the necessary modules:

```
import os
import pickle
# Gmail API utils
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
# for encoding/decoding messages in base64
from base64 import urlsafe_b64decode, urlsafe_b64encode
# for dealing with attachement MIME types
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from mimetypes import guess_type as guess_mime_type


# Request all access (permission to read/send/receive emails, manage the inbox, and more)
SCOPES = ['https://mail.google.com/']
our_email = 'your_gmail@gmail.com'
```

Obviously, you need to change `our_email` to your address, make sure you use the email you created the API auth with.

First of all, let's make a function that loads the `credentials.json`,

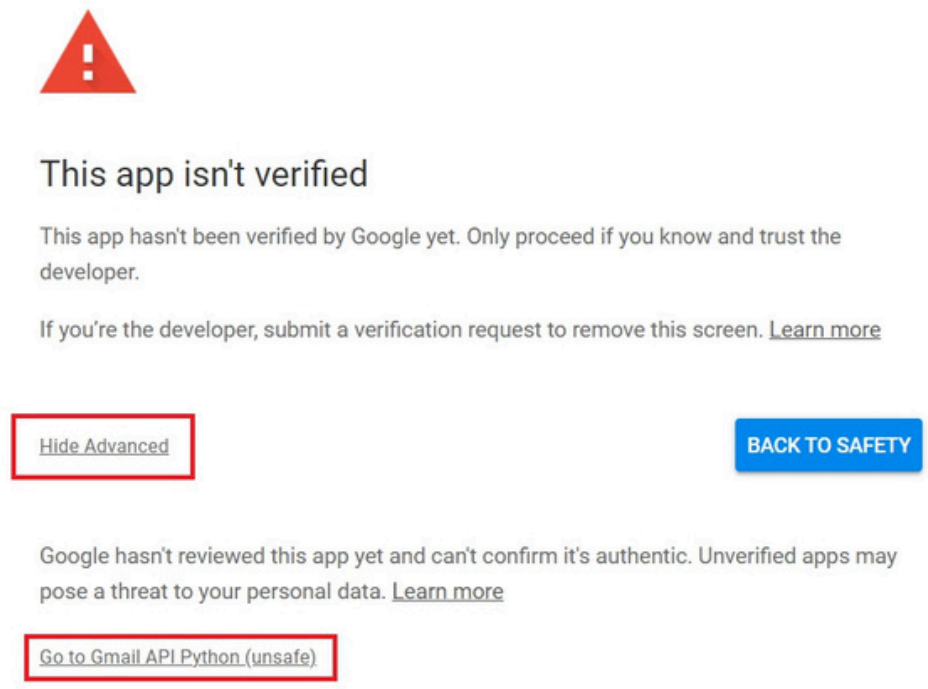
does

the authentication with Gmail API and returns a service object that can be used later in all in our upcoming functions:

```
def gmail_authenticate():
    creds = None
    # the file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first time
    if os.path.exists("token.pickle"):
        with open("token.pickle", "rb") as token:
            creds = pickle.load(token)
    # if there are no (valid) credentials availablle, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file('credentials.json',
SCOPES)
            creds = flow.run_local_server(port=0)
        # save the credentials for the next run
        with open("token.pickle", "wb") as token:
            pickle.dump(creds, token)
    return build('gmail', 'v1', credentials=creds)

# get the Gmail API service
service = gmail_authenticate()
```

You should see this familiar if you already used a Google API before, such as Google drive API, it is basically reading the `credentials.json` and saving it to `token.pickle` file after authenticating with Google in your browser, we save the token so the second time we run the code we shouldn't authenticate again.

This will prompt you in your default browser to accept the permissions required for this app, if you see a window that indicates the app isn't verified, you may just want to head to **Advanced** and click on go to **Gmail API Python (unsafe)**:



Sending Emails

First, let's start with the function that sends emails, we know that emails can contain attachments, so we will define a function that adds an attachment to a message, a message is an instance of `MIMEMultipart` (or `MIMEText`, if it doesn't contain attachments):

```
# Adds the attachment with the given filename to the given message
def add_attachment(message, filename):
    content_type, encoding = guess_mime_type(filename)
    if content_type is None or encoding is not None:
```

```
        content_type = 'application/octet-stream'
    main_type, sub_type = content_type.split('/', 1)
    if main_type == 'text':
        fp = open(filename, 'rb')
        msg = MIMEText(fp.read().decode(), _subtype=sub_type)
        fp.close()
    elif main_type == 'image':
        fp = open(filename, 'rb')
        msg = MIMEImage(fp.read(), _subtype=sub_type)
        fp.close()
    elif main_type == 'audio':
        fp = open(filename, 'rb')
        msg = MIMEAudio(fp.read(), _subtype=sub_type)
        fp.close()
    else:
        fp = open(filename, 'rb')
        msg = MIMEBase(main_type, sub_type)
        msg.set_payload(fp.read())
        fp.close()
    filename = os.path.basename(filename)
    msg.add_header('Content-Disposition', 'attachment', filename=filename)
    message.attach(msg)
```

Second, we write a function that takes some message parameters, builds, and returns an email message:

```
def build_message(destination, obj, body, attachments=[]):
    if not attachments: # no attachments given
        message = MIMEText(body)
```

```
    message['to'] = destination
    message['from'] = our_email
    message['subject'] = obj
  else:
    message = MIMEMultipart()
    message['to'] = destination
    message['from'] = our_email
    message['subject'] = obj
    message.attach(MIMEText(body))
    for filename in attachments:
      add_attachment(message, filename)
  return {'raw': urlsafe_b64encode(message.as_bytes()).decode()}
```

And finally, we make a function that takes message parameters, uses the Google mail API to send a message constructed with the `build_message()` we previously defined:

```
def send_message(service, destination, obj, body, attachments=[]):
  return service.users().messages().send(
    userId="me",
    body=build_message(destination, obj, body, attachments)
  ).execute()
```

That's it for sending messages. Let's use the function to send an example email:

```
# test send email
send_message(service, "destination@domain.com", "This is a subject",
      "This is the body of the email", ["test.txt", "anyfile.png"])
```

Put your email as the destination address, and real paths to files, and you'll see that the message is indeed sent!

Searching for Emails

```
def search_messages(service, query):
  result = service.users().messages().list(userId='me',q=query).execute()
  messages = [ ]
  if 'messages' in result:
    messages.extend(result['messages'])
  while 'nextPageToken' in result:
    page_token = result['nextPageToken']
    result = service.users().messages().list(userId='me',q=query,
pageToken=page_token).execute()
    if 'messages' in result:
      messages.extend(result['messages'])
  return messages
```

We had to retrieve the messages page by page because they're paginated. This function would return the IDs of the emails that match the query, we will use it for the delete, mark as read, mark as unread, and search features.

Reading Emails

In this section, we'll make Python code that takes a search query as input and reads all the matched emails; printing email basic information (`To`, `From` addresses, `Subject` and `Date`) and `plain/text` parts.

We'll also create a folder for each email based on the subject and download `text/html` content as well as any file that is attached to the

email and saves it in the folder created.

Before we dive into the function that reads emails given a search query, we gonna define two utility functions that we'll use:

```
# utility functions
def get_size_format(b, factor=1024, suffix="B"):
    """
    Scale bytes to its proper byte format
    e.g:
        1253656 => '1.20MB'
        1253656678 => '1.17GB'
    """
    for unit in ["", "K", "M", "G", "T", "P", "E", "Z"]:
        if b < factor:
            return f"{b:.2f}{unit}{suffix}"
        b /= factor
    return f"{b:.2f}Y{suffix}"


def clean(text):
    # clean text for creating a folder
    return "".join(c if c.isalnum() else "_" for c in text)
```

The `get_size_format()` function will just print bytes in a nice format (grabbed from this tutorial), and we gonna need the `clean()` function to make a folder name that doesn't contain spaces and special characters.

Next, let's define a function that parses the content of an email partition:

```
def parse_parts(service, parts, folder_name, message):
    """
    Utility function that parses the content of an email partition
    """
    if parts:
        for part in parts:
            filename = part.get("filename")
            mimeType = part.get("mimeType")
            body = part.get("body")
            data = body.get("data")
            file_size = body.get("size")
            part_headers = part.get("headers")
            if part.get("parts"):
                # recursively call this function when we see that a part
                # has parts inside
                parse_parts(service, part.get("parts"), folder_name, message)
            if mimeType == "text/plain":
                # if the email part is text plain
                if data:
                    text = urlsafe_b64decode(data).decode()
                    print(text)
            elif mimeType == "text/html":
                # if the email part is an HTML content
                # save the HTML file and optionally open it in the browser
                if not filename:
                    filename = "index.html"
```

```
            filepath = os.path.join(folder_name, filename)            print("Saving
HTML  to",  filepath)                              with  open(filepath,  "wb")  as  f:
                    f.write(urlsafe_b64decode(data))              else:                  #
attachment other than a plain text or HTML                  for part_header in
part_headers:                      part_header_name = part_header.get("name")
                part_header_value = part_header.get("value")                  if
part_header_name == "Content-Disposition":
                if "attachment" in part_header_value:
                    # we get the attachment ID
                    # and make another request to get the attachment itself
                    print("Saving the file:", filename, "size:",
get_size_format(file_size))
                    attachment_id = body.get("attachmentId")
                    attachment = service.users().messages() \
                            .attachments().get(id=attachment_id, userId='me',
messageId=message['id']).execute()
                    data = attachment.get("data")
                    filepath = os.path.join(folder_name, filename)
                    if data:
                        with open(filepath, "wb") as f:
                            f.write(urlsafe_b64decode(data))
```

Now, let's write our main function for reading an email:

```
def read_message(service, message):
    """
```

```
    This function takes Gmail API `service` and the given `message_id` and does the following:
        - Downloads the content of the email
            - Prints email basic information (To, From, Subject & Date) and plain/text parts
        - Creates a folder for each email based on the subject
            - Downloads text/html content (if available) and saves it under the folder created as index.html
        - Downloads any file that is attached to the email and saves it in the folder created
    """
    msg = service.users().messages().get(userId='me', id=message['id'], format='full').execute()
    # parts can be the message body, or attachments
    payload = msg['payload']
    headers = payload.get("headers")
    parts = payload.get("parts")
    folder_name = "email"
    has_subject = False
    if headers:
        # this section prints email basic info & creates a folder for the email
        for header in headers:
            name = header.get("name")
            value = header.get("value")
            if name.lower() == 'from':
                # we print the From address
                print("From:", value)
            if name.lower() == "to":
                # we print the To address
                print("To:", value)
```

```
            if name.lower() == "subject":
                # make our boolean True, the email has "subject"
                has_subject = True
                # make a directory with the name of the subject
                folder_name = clean(value)
                # we will also handle emails with the same subject name
                folder_counter = 0
                while os.path.isdir(folder_name):
                    folder_counter += 1
                    # we have the same folder name, add a number next to it
                    if folder_name[-1].isdigit() and folder_name[-2] == "_":
                        folder_name = f"{folder_name[:-2]}_{folder_counter}"
                    elif folder_name[-2:].isdigit() and folder_name[-3] == "_":
                        folder_name = f"{folder_name[:-3]}_{folder_counter}"
                    else:
                        folder_name = f"{folder_name}_{folder_counter}"
                os.mkdir(folder_name)
                print("Subject:", value)
            if name.lower() == "date":
                # we print the date when the message was sent
                print("Date:", value)
    if not has_subject:
        # if the email does not have a subject, then make a folder with "email"
name
        # since folders are created based on subjects
        if not os.path.isdir(folder_name):
            os.mkdir(folder_name)
    parse_parts(service, parts, folder_name, message)
    print("="*50)
```

Since the previously defined function `search_messages()` returns a list of IDs of matched emails, the `read_message()` downloads the content of the email and does what's already mentioned above.

The `read_message()` function uses `parse_parts()` to parse different email partitions, if it's a `text/plain`, then we just decode it and print it to the screen, if it's a `text/html`, then we simply save it in that folder created with the name `index.html`, and if it's a file (attachment), then we download the attachment by its `attachment_id` and save it under the created folder.

Also, if two emails have the same `Subject`, then we need to add a simple counter to the name of the folder, and that's what we did with `folder_counter`.

Let's use this in action:

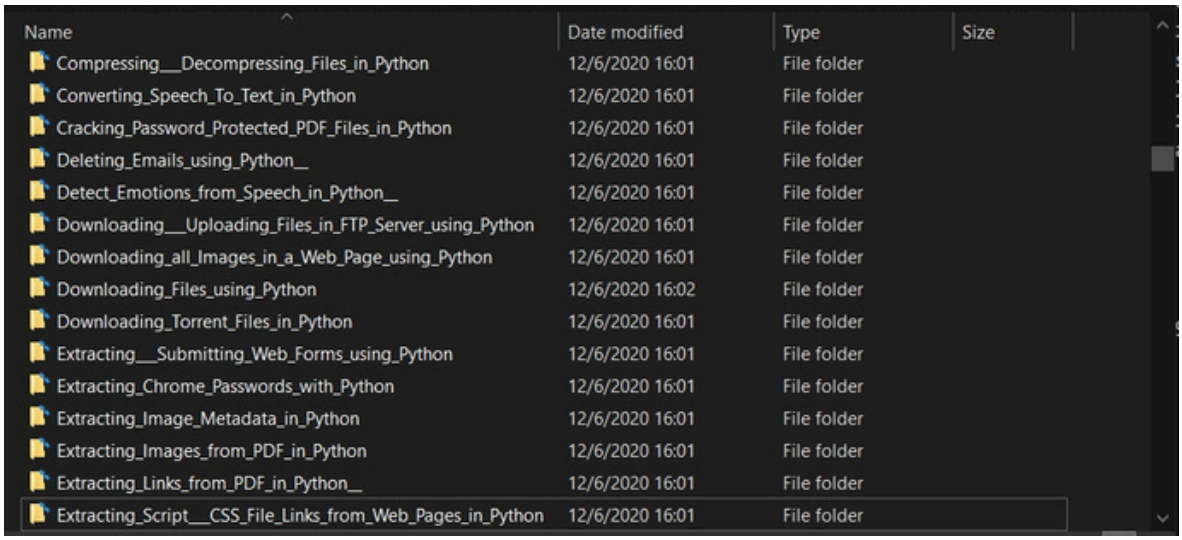
```
# get emails that match the query you specify results = search_messages(service, "Python Code") # for each email matched, read it (output plain/text to console & save
HTML and attachments)
for msg in results:
    read_message(service, msg)
```

This will download and parse all emails that contain Python Code keyword, here is a part of the output:

```
==========
From: Python Code <email@domain.com>
To: "email@gmail.com" <email@gmail.com>
Subject: How to Play and Record Audio in Python
Date: Fri, 21 Feb 2020 09:24:58 +0000
```

```
Hello !

I have no doubt that you already encountered with an application that uses sound (either recording or playing) and you know how useful is that !
<...SNIPPED..>

Saving HTML to How_to_Play_and_Record_Audio_in_Python\index.html
========================================================== From: Python Code <email@domain.com> To: "email@gmail.com" <email@gmail.com>
Subject: Brute-Forcing FTP Servers in Python
Date: Tue, 25 Feb 2020 21:31:09 +0000

Hello,
A brute-force attack consists of an attack that submits many passwords with the hope of guessing correctly.
<...SNIPPED...>

Saving HTML to Brute_Forcing_FTP_Servers_in_Python_1\index.html
==========================================================
<...SNIPPED...>
```

You'll also see folders created in your current directory for each email matched:



Inside each folder, it has its corresponding HTML version of the email, as well as any attachments if available.

Marking Emails as Read

```
def mark_as_read(service, query):
  messages_to_mark = search_messages(service, query)
  return service.users().messages().batchModify(
    userId='me',
    body={
      'ids': [ msg['id'] for msg in messages_to_mark ],
      'removeLabelIds': ['UNREAD']
    }
  ).execute()
```

We use the `batchModify()` method and we set `removeLabelIds` to `["UNREAD"]` in the `body` parameter to remove the unread label from the matched emails.

For example, let's mark all Google emails as read:

```
mark_as_read(service, "Google")
```

Marking Emails as Unread

Marking messages as unread can be done in a similar manner, this time by adding the label `["UNREAD"]`:

```
def mark_as_unread(service, query):
  messages_to_mark = search_messages(service, query)
  # add the label UNREAD to each of the search results
  return service.users().messages().batchModify(
    userId='me',
    body={
      'ids': [ msg['id'] for msg in messages_to_mark ],
      'addLabelIds': ['UNREAD']
    }
  ).execute()
```

Example run:

```
# search query by sender/receiver
mark_as_unread(service, "email@domain.com")
```

Deleting Emails

Now, for the deleting messages feature:

```
def delete_messages(service, query):
  messages_to_delete = search_messages(service, query)
  # it's possible to delete a single message with the delete API, like this:
  # service.users().messages().delete(userId='me', id=msg['id'])
```

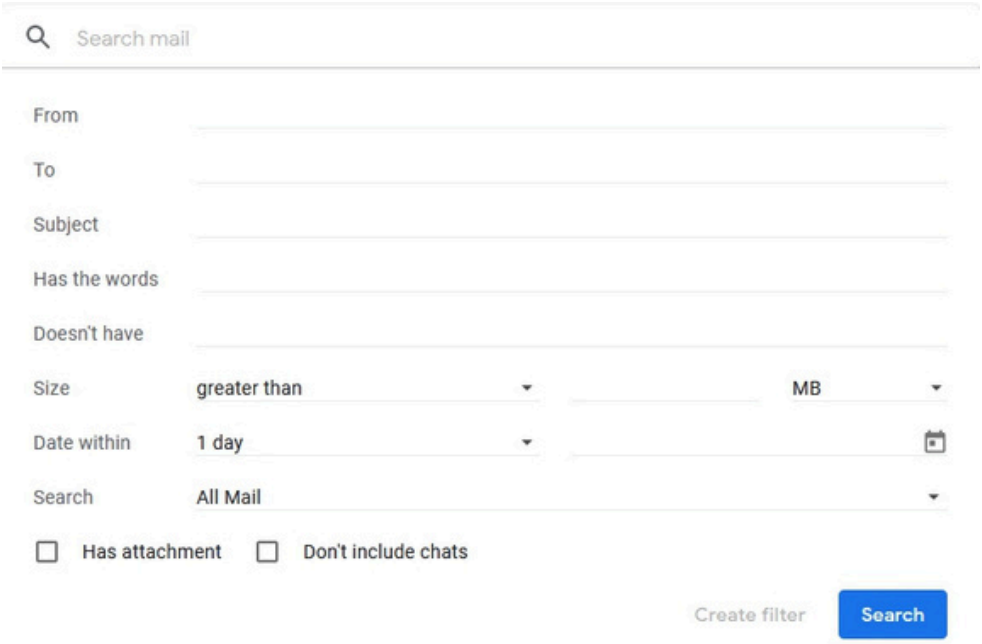
```
  # but it's also possible to delete all the selected messages with one query, batchDelete
  return service.users().messages().batchDelete(
    userId='me',
    body={
        'ids': [ msg['id'] for msg in messages_to_delete]
    }
  ).execute()
```

This time we use the `batchDelete()` method to delete all matched emails, let's for example delete all emails from Google Alerts:

```
delete_messages(service, "Google Alerts")
```

Summary

Gmail queries support filters that can be used to select specific messages, some of these filters are shown below, this is a dialog that is shown when searching for emails, we can fill it, and get the corresponding search query:



Gmail not only offers a great and friendly user interface, with many features for demanding users, but it also offers a powerful API for developers to use and interact with Gmail, we conclude that manipulating emails from Google mail programmatically is very straightforward.

If you want to know more about the API, I encourage you to check the official Gmail API page.

Finally, I've created Python scripts for each of the tasks we did on this tutorial, please check this page for the full code.

Below are some of the Google API tutorials:

- How to Extract Google Trends Data in Python.
- How to Use Google Drive API in Python.
- How to Extract YouTube Data using YouTube API in Python.

- How to Use Google Custom Search Engine API in Python.

Fullcode:

common.py

```
import os
import pickle
# Gmail API utils
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request


# Request all access (permission to read/send/receive emails, manage the inbox, and more)
SCOPES = ['https://mail.google.com/']
our_email = 'our_email@gmail.com'


def gmail_authenticate():
    creds = None
    # the file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the first time
    if os.path.exists("token.pickle"):
        with open("token.pickle", "rb") as token:
            creds = pickle.load(token)
```

```
    # if there are no (valid) credentials availablle, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file('credentials.json',
SCOPES)
            creds = flow.run_local_server(port=0)
        # save the credentials for the next run
        with open("token.pickle", "wb") as token:
            pickle.dump(creds, token)
    return build('gmail', 'v1', credentials=creds)


def search_messages(service, query):
    result = service.users().messages().list(userId='me',q=query).execute()
    messages = [ ]
    if 'messages' in result:
        messages.extend(result['messages'])
    while 'nextPageToken' in result:
        page_token = result['nextPageToken']
        result = service.users().messages().list(userId='me',q=query,
pageToken=page_token).execute()
        if 'messages' in result:
            messages.extend(result['messages'])
    return messages
```

send_emails.py

```
# for getting full paths to attachements
import os
# for encoding messages in base64
from base64 import urlsafe_b64encode
# for dealing with attachement MIME types
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from mimetypes import guess_type as guess_mime_type

from common import our_email, gmail_authenticate

# Adds the attachment with the given filename to the given message
def add_attachment(message, filename):
    content_type, encoding = guess_mime_type(filename)
    if content_type is None or encoding is not None:
        content_type = 'application/octet-stream'
    main_type, sub_type = content_type.split('/', 1)
    if main_type == 'text':
        fp = open(filename, 'rb')
        msg = MIMEText(fp.read().decode(), _subtype=sub_type)
        fp.close()
    elif main_type == 'image':
        fp = open(filename, 'rb')
        msg = MIMEImage(fp.read(), _subtype=sub_type)
        fp.close()
    elif main_type == 'audio':
```

```
      fp = open(filename, 'rb')
      msg = MIMEAudio(fp.read(), _subtype=sub_type)
      fp.close()
    else:
      fp = open(filename, 'rb')
      msg = MIMEBase(main_type, sub_type)
      msg.set_payload(fp.read())
      fp.close()
    filename = os.path.basename(filename)
    msg.add_header('Content-Disposition', 'attachment', filename=filename)
    message.attach(msg)

def build_message(destination, obj, body, attachments=[]):
    if not attachments: # no attachments given
      message = MIMEText(body)
      message['to'] = destination
      message['from'] = our_email
      message['subject'] = obj
    else:
      message = MIMEMultipart()
      message['to'] = destination
      message['from'] = our_email
      message['subject'] = obj
      message.attach(MIMEText(body))
      for filename in attachments:
        add_attachment(message, filename)
    return {'raw': urlsafe_b64encode(message.as_bytes()).decode()}

def send_message(service, destination, obj, body, attachments=[]):
```

```
    return service.users().messages().send(
      userId="me",
      body=build_message(destination, obj, body, attachments)
    ).execute()


if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="Email Sender using Gmail API")
    parser.add_argument('destination', type=str, help='The destination email address')
    parser.add_argument('subject', type=str, help='The subject of the email')
    parser.add_argument('body', type=str, help='The body of the email')
    parser.add_argument('-f', '--files', type=str, help='email attachments', nargs='+')

    args = parser.parse_args()
    service = gmail_authenticate()
    send_message(service, args.destination, args.subject, args.body, args.files)
```

read_emails.py

```
import os
import sys
# for encoding/decoding messages in base64
from base64 import urlsafe_b64decode
from common import gmail_authenticate, search_messages
```

```
def get_size_format(b, factor=1024, suffix="B"):
    """
    Scale bytes to its proper byte format
    e.g:
        1253656 => '1.20MB'
        1253656678 => '1.17GB'
    """
    for unit in ["", "K", "M", "G", "T", "P", "E", "Z"]:
        if b < factor:
            return f"{b:.2f}{unit}{suffix}"
        b /= factor
    return f"{b:.2f}Y{suffix}"


def clean(text):
    # clean text for creating a folder
    return "".join(c if c.isalnum() else "_" for c in text)


def parse_parts(service, parts, folder_name, message):
    """
    Utility function that parses the content of an email partition
    """
    if parts:
        for part in parts:
            filename = part.get("filename")
            mimeType = part.get("mimeType")
            body = part.get("body")
```

```
            data = body.get("data")                file_size = body.get("size")
          part_headers = part.get("headers")              if part.get("parts"):
                # recursively call this function when we see that a part
                  # has parts inside                        parse_parts(service,
part.get("parts"), folder_name, message)                  if mimeType ==
"text/plain":          # if the email part is text plain          if data:
            text = urlsafe_b64decode(data).decode()
            print(text)
        elif mimeType == "text/html":
          # if the email part is an HTML content
          # save the HTML file and optionally open it in the browser
          if not filename:
            filename = "index.html"
          filepath = os.path.join(folder_name, filename)
          print("Saving HTML to", filepath)
          with open(filepath, "wb") as f:
            f.write(urlsafe_b64decode(data))
        else:
          # attachment other than a plain text or HTML
          for part_header in part_headers:
            part_header_name = part_header.get("name")
            part_header_value = part_header.get("value")
            if part_header_name == "Content-Disposition":
              if "attachment" in part_header_value:
                # we get the attachment ID
```

```
                # and make another request to get the attachment itself
                print("Saving the file:", filename, "size:",
get_size_format(file_size))
                attachment_id = body.get("attachmentId")
                attachment = service.users().messages() \
                        .attachments().get(id=attachment_id, userId='me',
messageId=message['id']).execute()
                data = attachment.get("data")
                filepath = os.path.join(folder_name, filename)
                if data:
                    with open(filepath, "wb") as f:
                        f.write(urlsafe_b64decode(data))


def read_message(service, message):
    """
    This function takes Gmail API `service` and the given `message_id` and
does the following:
        - Downloads the content of the email
        - Prints email basic information (To, From, Subject & Date) and
plain/text parts
        - Creates a folder for each email based on the subject
        - Downloads text/html content (if available) and saves it under the
folder created as index.html
        - Downloads any file that is attached to the email and saves it in the
folder created
    """
    msg = service.users().messages().get(userId='me', id=message['id'],
format='full').execute()
    # parts can be the message body, or attachments
```

```
payload = msg['payload']
headers = payload.get("headers")
parts = payload.get("parts")
folder_name = "email"
has_subject = False
if headers:
    # this section prints email basic info & creates a folder for the email
    for header in headers:
        name = header.get("name")
        value = header.get("value")
        if name.lower() == 'from':
            # we print the From address
            print("From:", value)
        if name.lower() == "to":
            # we print the To address
            print("To:", value)
        if name.lower() == "subject":
            # make our boolean True, the email has "subject"
            has_subject = True
            # make a directory with the name of the subject
            folder_name = clean(value)
            # we will also handle emails with the same subject name
            folder_counter = 0
            while os.path.isdir(folder_name):
                folder_counter += 1
                # we have the same folder name, add a number next to it
                if folder_name[-1].isdigit() and folder_name[-2] == "_":
                    folder_name = f"{folder_name[:-2]}_{folder_counter}"
                elif folder_name[-2:].isdigit() and folder_name[-3] == "_":
```

```
                folder_name = f"{folder_name[:-3]}_{folder_counter}"
            else:                                folder_name = f"
{folder_name}_{folder_counter}"                os.mkdir(folder_name)
        print("Subject:", value)        if name.lower() == "date":          #
we print the date when the message was sent          print("Date:", value)
  if not has_subject:       # if the email does not have a subject, then make a
folder with "email"
name
    # since folders are created based on subjects
    if not os.path.isdir(folder_name):
      os.mkdir(folder_name)
  parse_parts(service, parts, folder_name, message)
  print("="*50)




if __name__ == "__main__":
  service = gmail_authenticate()
  # get emails that match the query you specify from the command lines
  results = search_messages(service, sys.argv[1])
  print(f"Found {len(results)} results.")
  # for each email matched, read it (output plain/text to console & save
HTML and attachments)
  for msg in results:
    read_message(service, msg)
```

mark_emails.py

```
from common import gmail_authenticate, search_messages

def mark_as_read(service, query):
  messages_to_mark = search_messages(service, query)
  print(f"Matched emails: {len(messages_to_mark)}")
  return service.users().messages().batchModify(
    userId='me',
    body={
      'ids': [ msg['id'] for msg in messages_to_mark ],
      'removeLabelIds': ['UNREAD']
    }
  ).execute()

def mark_as_unread(service, query):
  messages_to_mark = search_messages(service, query)
  print(f"Matched emails: {len(messages_to_mark)}")
  return service.users().messages().batchModify(
    userId='me',
    body={
      'ids': [ msg['id'] for msg in messages_to_mark ],
      'addLabelIds': ['UNREAD']
    }
  ).execute()

if __name__ == "__main__":
  import argparse
  parser = argparse.ArgumentParser(description="Marks a set of emails as read or unread")
```

```
    parser.add_argument('query', help='a search query that selects emails to mark')
    parser.add_argument("-r", "--read", action="store_true", help='Whether to mark the message as read')
    parser.add_argument("-u", "--unread", action="store_true", help='Whether to mark the message as unread')

    args = parser.parse_args()
    service = gmail_authenticate()
    if args.read:
        mark_as_read(service, args.query)
    elif args.unread:
        mark_as_unread(service, args.query)
```

delete_emails.py

```
from common import gmail_authenticate, search_messages

def delete_messages(service, query):
    messages_to_delete = search_messages(service, query)
    print(f"Deleting {len(messages_to_delete)} emails.")
    # it's possible to delete a single message with the delete API, like this:
    # service.users().messages().delete(userId='me', id=msg['id'])
    # but it's also possible to delete all the selected messages with one query, batchDelete
    return service.users().messages().batchDelete(
      userId='me',
      body={
          'ids': [ msg['id'] for msg in messages_to_delete]
```

```
    }
  ).execute()

if __name__ == "__main__":
  import sys
  service = gmail_authenticate()
  delete_messages(service, sys.argv[1])
```

PART 6: Use Shodan API in Python

Learn how to use Shodan API to make a script that searches for public vulnerable servers, IoT devices, power plants and much more using Python.

Public IP addresses are routed on the Internet, which means connection can be established between any host having a public IP, and any other host connected to the Internet without having a firewall filtering the outgoing traffic, and because IPv4 is still the dominant used protocol on the Internet, it's possible and nowadays practical to crawl the whole Internet.

There are a number of platforms that offer Internet scanning as a service, to list a few; Shodan, Censys, and ZoomEye. Using these services, we can scan the Internet for devices running a given service, we can find surveillance cameras, industrial control systems such as power plants, servers, IoT devices and much more.

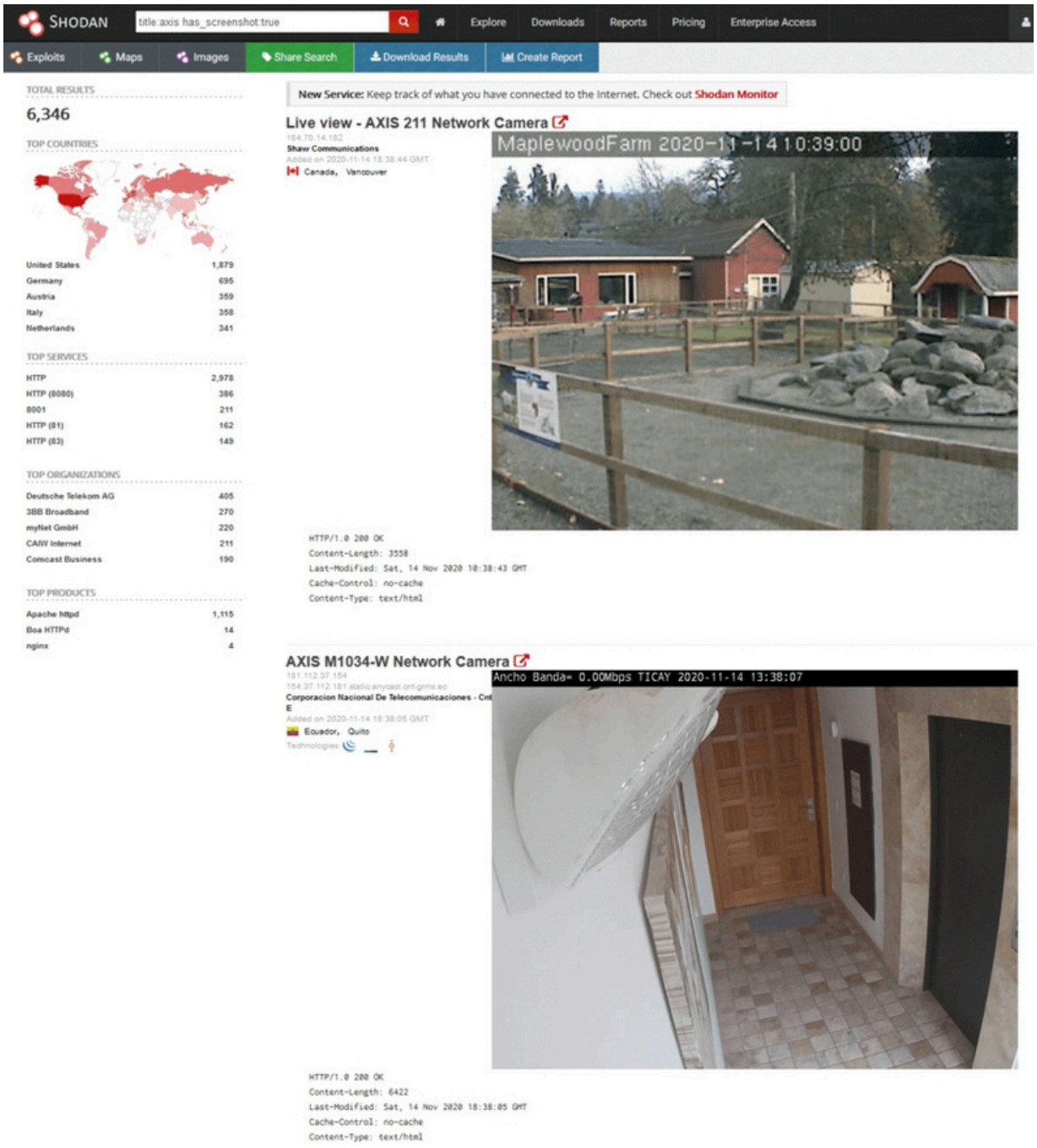
These services often offer an API, which allow programmers to take full advantage of their scan results, they're also used by product managers to check patch applications, and to get the big picture on the market shares with competitors, and also used by security researchers to find vulnerable hosts and create reports on vulnerability impacts.

In this tutorial, we will look into Shodan's API using Python, and some of its practical use-cases.

Shodan is by far the most popular IoT search engine, it was created in 2009, it features a web interface for exploring data manually, as well as a REST API and libraries for the most popular programming languages including Python, Ruby, Java and C#.

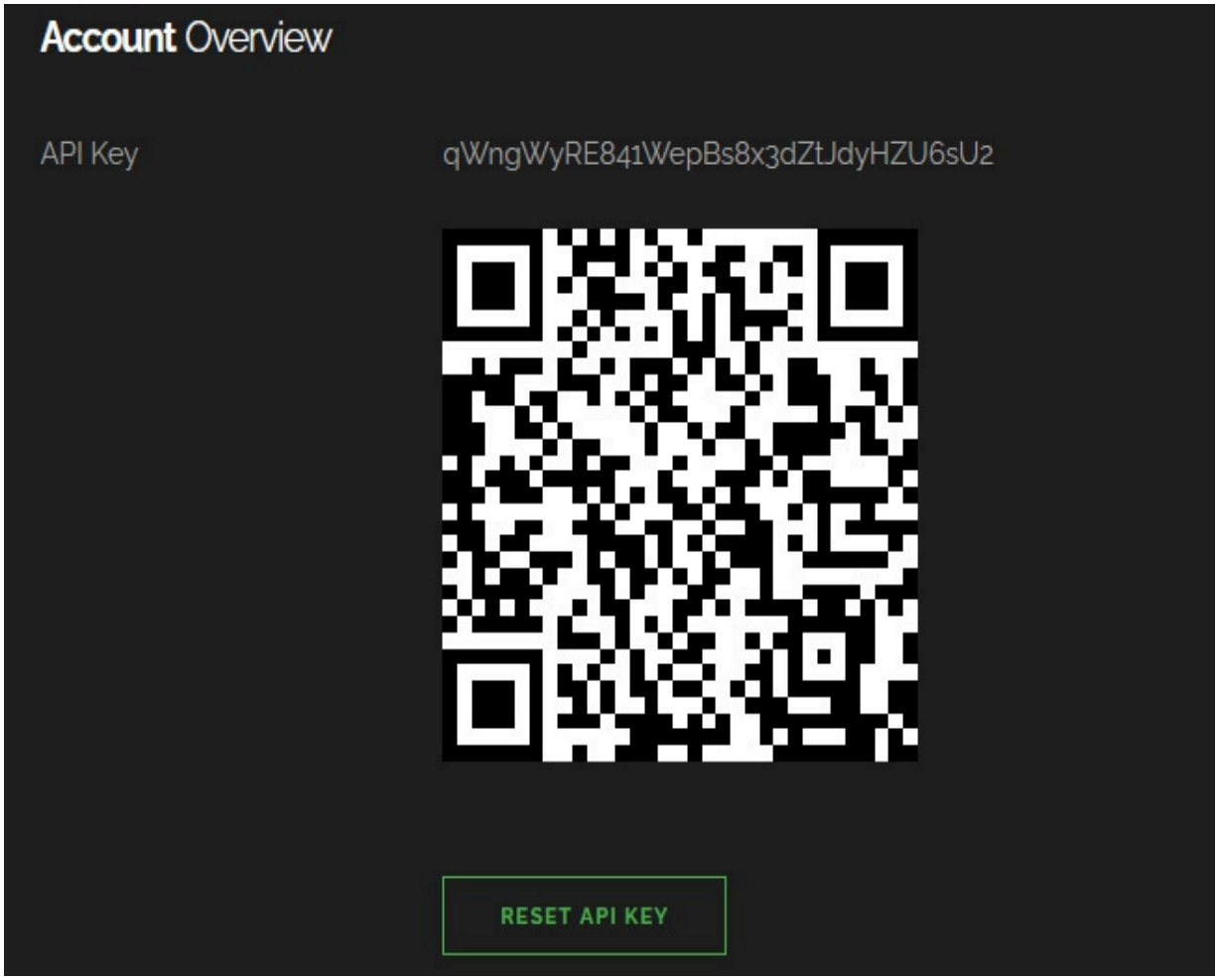
Using most of Shodan features requires a Shodan membership, which costs 49$ at the time of writing the article for a lifetime upgrade, and which is free for students, professors and IT staff, refer to this page for more information.

Once you become a member, you can manually explore data, let's try to find unprotected Axis security cameras:



As you can see, the search engine is quite powerful, especially with search filters, if you want to test more cool queries, we'd recommend checking out [this list of awesome Shodan search queries](#).

Now let's try to use Shodan API. First, we navigate to our account, to retrieve our API key:



To get started with Python, we need to install shodan library:

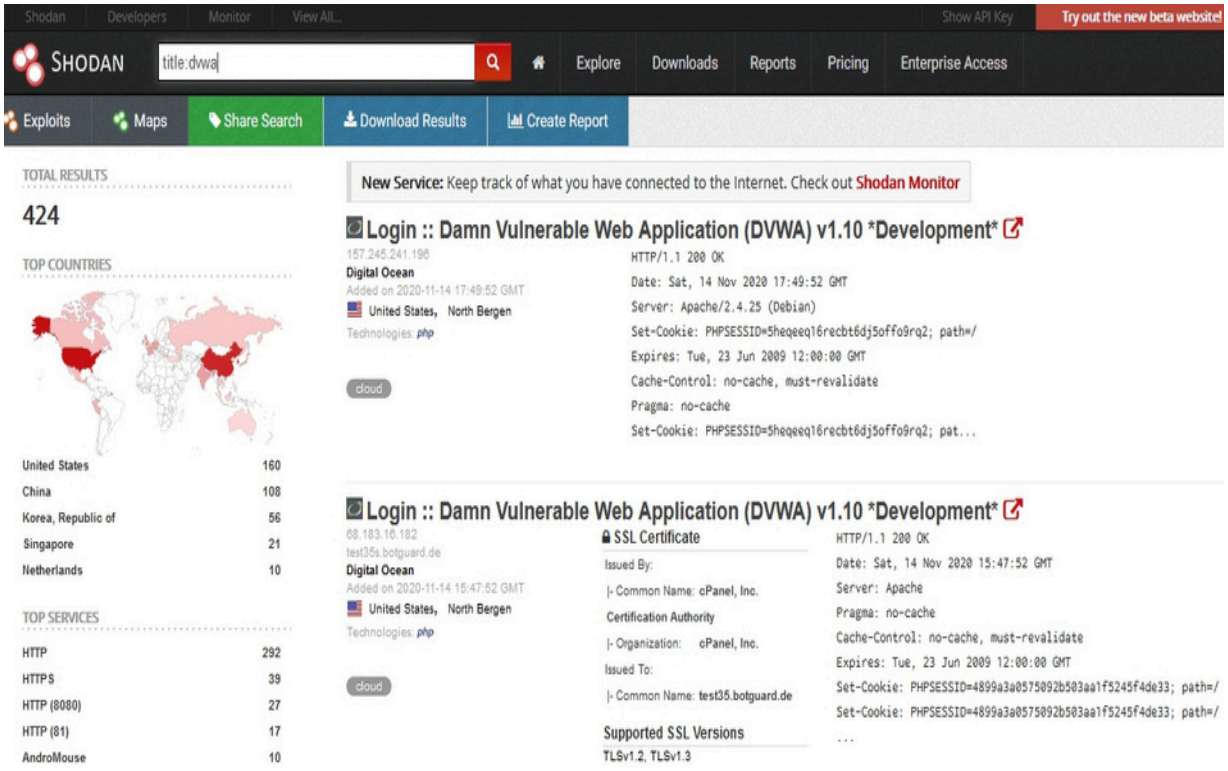
```
pip3 install shodan
```

The example we gonna use in this tutorial is we make a script that searches for instances of DVWA (Damn Vulnerable Web Application) that still have default credentials and reports them.

DVWA is an opensource project, aimed for security testing, it's a web application that is vulnerable by design, it's expected that users deploy it on their machines to use it, we will try to find instances

on the Internet that already have it deployed, to use it without installing.

There should be a lot of ways to search for DVWA instances, but we gonna stick with the title, as it's straightforward:



The difficulty with doing this task manually is that most of the instances should have their login credentials changed. So, to find accessible DVWA instances, it's necessary to try default credentials on each of the detected instances, we'll do that with Python:

```
import shodan
import time
import requests
import re


# your shodan API key
```

```
SHODAN_API_KEY = '<YOUR_SHODAN_API_KEY_HERE>'
api = shodan.Shodan(SHODAN_API_KEY)
```

Now let's write a function that queries a page of results from Shodan, one page can contain up to 100 results, we add a loop for safety. In case there is a network or API error, we keep retrying with second delays until it works:

```
# requests a page of data from shodan
def request_page_from_shodan(query, page=1):
  while True:
    try:
      instances = api.search(query, page=page)
      return instances
    except shodan.APIError as e:
      print(f"Error: {e}")
      time.sleep(5)
```

Let's define a function that takes a host, and checks if the credentials admin:password (defaults for DVWA) are valid, this is independent of the Shodan library, we will use requests library for submitting our credentials, and checking the result:

```
# Try the default credentials on a given instance of DVWA, simulating a real user trying the credentials
# visits the login.php page to get the CSRF token, and tries to login with admin:password
def has_valid_credentials(instance):
  sess = requests.Session()
  proto = ('ssl' in instance) and 'https' or 'http'
```

```
    try:
        res = sess.get(f"{proto}://{instance['ip_str']}:
{instance['port']}/login.php", verify=False)
    except requests.exceptions.ConnectionError:
        return False
    if res.status_code != 200:
        print("[-] Got HTTP status code {res.status_code}, expected 200")
        return False
    # search the CSRF token using regex
    token = re.search(r"user_token' value='([0-9a-f]+)'", res.text).group(1)
    res = sess.post(
        f"{proto}://{instance['ip_str']}:{instance['port']}/login.php",
        f"username=admin&password=password&user_token=
{token}&Login=Login",
        allow_redirects=False,
        verify=False,
        headers={'Content-Type': 'application/x-www-form-urlencoded'}
    )
    if res.status_code == 302 and res.headers['Location'] == 'index.php':
        # Redirects to index.php, we expect an authentication success
        return True
    else:
        return False
```

The above function sends a GET request to the DVWA login page, to retrieve the `user_token`, then sends a POST request with the default username and password, and the CSRF token, and then it checks whether the authentication was successful or not.

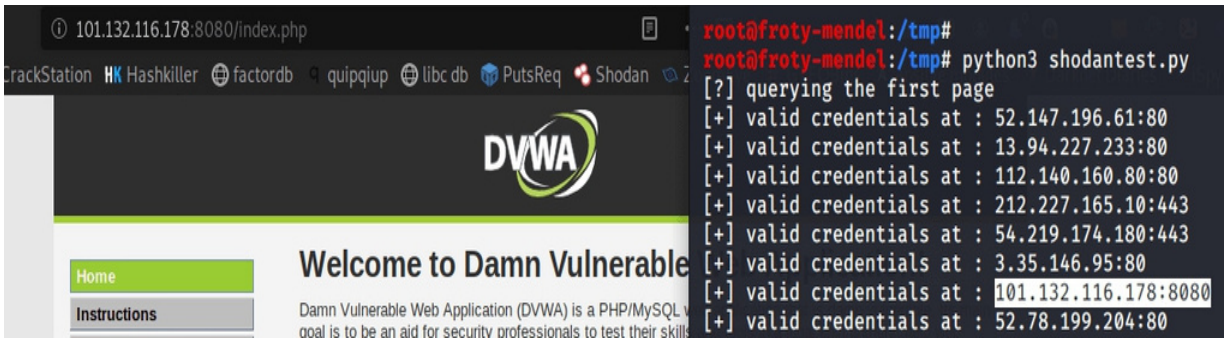
Let's write a function that takes a query, and iterates over the pages in Shodan search results, and for each host on each page, we call the `has_valid_credentials()` function:

```
# Takes a page of results, and scans each of them, running
has_valid_credentials
def process_page(page):
    result = []
    for instance in page['matches']:
        if has_valid_credentials(instance):
            print(f"[+] valid credentials at : {instance['ip_str']}:
{instance['port']}")
            result.append(instance)
    return result

# searches on shodan using the given query, and iterates over each page of
the results
def query_shodan(query):
    print("[*] querying the first page")
    first_page = request_page_from_shodan(query)
    total = first_page['total']
    already_processed = len(first_page['matches'])
    result = process_page(first_page)
    page = 2
    while already_processed < total:
        # break just in your testing, API queries have monthly limits
        break
        print("querying page {page}")
        page = request_page_from_shodan(query, page=page)
        already_processed += len(page['matches'])
```

```
        result += process_page(page)
        page += 1
    return result

# search for DVWA instances
res = query_shodan('title:dvwa')
print(res)
```

This can be improved significantly by taking advantage of multi-threading to speed up our scanning, as we could check hosts in parallel, check this tutorial that may help you out.

Here is the script output:



As you can see, this Python script works and reports hosts that has the default credentials on DVWA instances.

Summary

Scanning for DVWA instances with default credentials might not be the most useful example, as the application is made to be vulnerable by design, and most people using it are not changing their credentials.

However, using Shodan API is very powerful, and the example above highlights how it's possible to iterate over scan results, and process each of them with code, the search API is the most popular, but Shodan also supports on-demand scanning, network monitoring and more, you can check out the API reference for more details.

***Disclaimer**: We do not encourage you to do illegal activities, with great power comes great responsibility. Using Shodan is not illegal, but bruteforcing credentials on routers and services is, and we are not responsible for any misuse of the API, or the Python code we provided.*

Fullcode:

```
shodan_api.py import shodan
import time
import requests
import re




# your shodan API key
SHODAN_API_KEY = '<YOUR_SHODAN_API_KEY_HERE>'
api = shodan.Shodan(SHODAN_API_KEY)

# requests a page of data from shodan
def request_page_from_shodan(query, page=1):
  while True:
    try:
      instances = api.search(query, page=page)
      return instances
```

```
        except shodan.APIError as e:
            print(f"Error: {e}")
            time.sleep(5)


# Try the default credentials on a given instance of DVWA, simulating a real user trying the credentials
# visits the login.php page to get the CSRF token, and tries to login with admin:password
def has_valid_credentials(instance):
    sess = requests.Session()
    proto = ('ssl' in instance) and 'https' or 'http'
    try:
        res = sess.get(f"{proto}://{instance['ip_str']}:{instance['port']}/login.php", verify=False)
    except requests.exceptions.ConnectionError:
        return False
    if res.status_code != 200:
        print(f"[-] Got HTTP status code {res.status_code}, expected 200")
        return False
    # search the CSRF token using regex
    token = re.search(r"user_token' value='([0-9a-f]+)'", res.text).group(1)
    res = sess.post(
        f"{proto}://{instance['ip_str']}:{instance['port']}/login.php",
        f"username=admin&password=password&user_token={token}&Login=Login",
        allow_redirects=False,
        verify=False,
        headers={'Content-Type': 'application/x-www-form-urlencoded'}
```

```
    )
    if res.status_code == 302 and res.headers['Location'] == 'index.php':
        # Redirects to index.php, we expect an authentication success
        return True
    else:
        return False

# Takes a page of results, and scans each of them, running
has_valid_credentials
def process_page(page):
    result = []
    for instance in page['matches']:
        if has_valid_credentials(instance):
            print(f"[+] valid credentials at : {instance['ip_str']}:
{instance['port']}")
            result.append(instance)
    return result

# searches on shodan using the given query, and iterates over each page of
the results
def query_shodan(query):
    print("[*] querying the first page")
    first_page = request_page_from_shodan(query)
    total = first_page['total']
    already_processed = len(first_page['matches'])
    result = process_page(first_page)
    page = 2
    while already_processed < total:
        # break just in your testing, API queries have monthly limits
```

```
        break
        print("querying page {page}")
        page = request_page_from_shodan(query, page=page)
        already_processed += len(page['matches'])
        result += process_page(page)
        page += 1
    return result

# search for DVWA instances
res = query_shodan('title:dvwa')
print(res)
```

PART 7: Python Make a Telegram Bot in

Learn how to use Telegram API and python-telegram-bot wrapper to build a Telegram Bot in Python.

Automation is becoming more and more popular every day, and so, popular services and applications nowadays often offer an interface for programmatic use, this interface is what we call an API, or Application Programming Interface, examples of applications offering an API include Google Drive, Google Search and Github.

An API is a set of endpoints any programmer can use to communicate with a service, without having to try imitating a user using the application, which is often not possible because of captchas becoming more and more widely used.

When a popular application offers an API, programmers usually write easy-to-use libraries (which act as an abstraction layer to the API, often called wrappers), for a programmer who wants to communicate with the application, instead of reading the reference about the API endpoints, it's more straightforward to just download a library in their programming language of choice, and read its

documentation, which is often more idiomatic, and faster to get used to.

In this tutorial, we will see how to write a Telegram Bot in Python,

a bot is a user controlled by code, writing a bot can have many applications, for example; automatically responding to client requests.

Telegram offers two APIs, one for creating bots, and one for

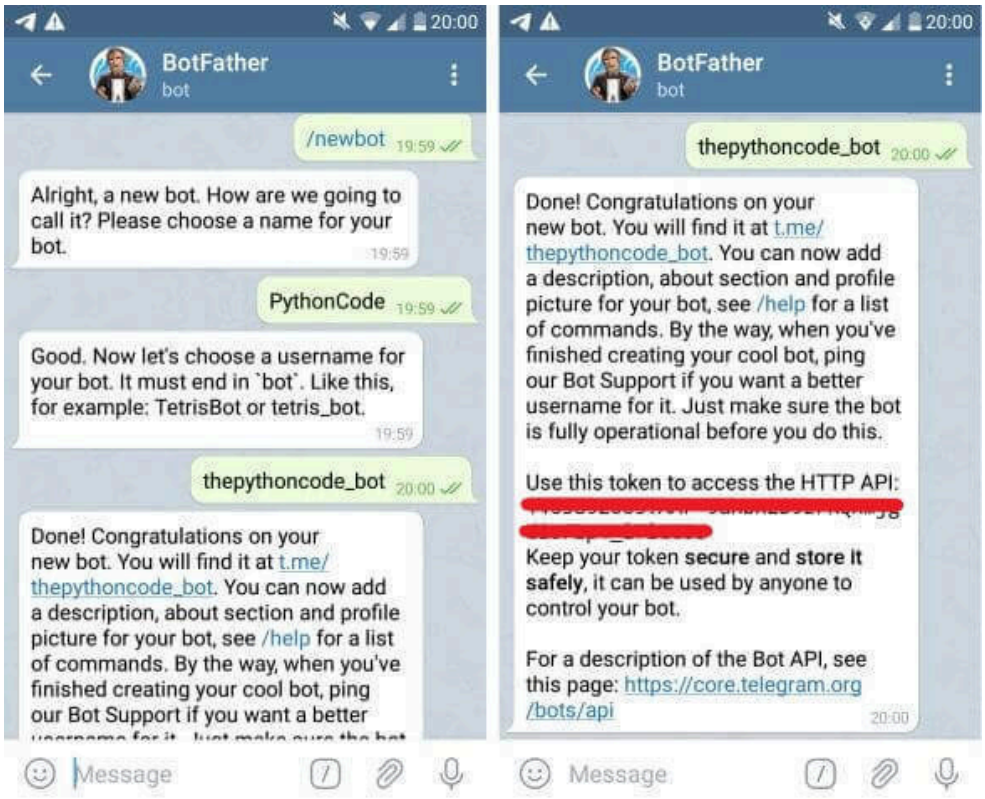
creating clients, we will be using the first one, documentation for the Bot API can be found here.

We will be using the popular python-telegram-bot wrapper to ease

the work for us:

```
pip3 install python-telegram-bot
```

Now, we will need to get an API Key to communicate with the Telegram API, to get one, we need to manually contact @BotFather on Telegram, like so:



We get a list of commands when we start the discussion, we create the bot with the /newbot command, once it's created, we obtain a token for communicating with the bot (in our case, it's hidden in red).

Now we can start writing our bot in Python:

```
import telegram import telegram.ext import re from random import randint
import logging logging.basicConfig(level=logging.INFO,
format='%(asctime)s - %(name)s - %(levelname)s - %
(message)s')
```

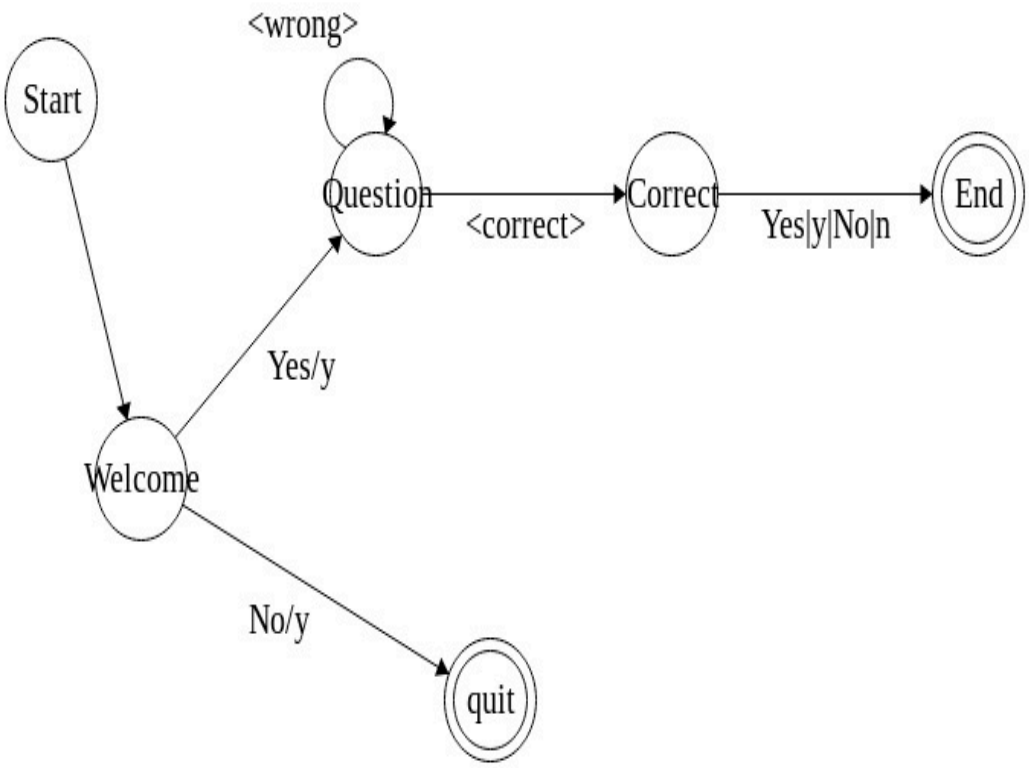
```
# The API Key we received for our bot
API_KEY = "<INSERT_API_KEY_HERE>"
# Create an updater object with our API Key
updater = telegram.ext.Updater(API_KEY)
# Retrieve the dispatcher, which will be used to add handlers
dispatcher = updater.dispatcher
```

Notice we added logging in the beginning of the script, and we set the logging level to DEBUG, this will help us what's going on with the bot, whether it's running, messages we get from our users, etc. If you're not familiar with logging in Python, check this tutorial.

The dispatcher is the object that will dispatch requests to their handlers, we need to add a conversation handler to it, to describe the way our bot will reply to messages.

The Telegram API allows defining the bot as a finite-state machine, we can handle different events, and change states according to the input of the user, or the type of actions.

For this tutorial, we'll be building the following FSM:



Execution begins at Start, Welcome will ask the user whether he wants to answer a question, if the response is yes or y, it will send the question, and switch state to Correct if the response is correct. Otherwise, it will loop on Question, each time generating a different question.

Once the response is correct, it'll ask the user if he found the tutorial helpful, and go to the end state, which is the final one.

Defining our states:

```
#    Our    states,    as
integers  WELCOME  =  0
QUESTION = 1 CANCEL
= 2 CORRECT = 3
```

Now let's define our handlers:

```
# The entry function
def start(update_obj, context):
   # send the question, and show the keyboard markup (suggested answers)
   update_obj.message.reply_text("Hello there, do you want to answer a
question? (Yes/No)",
      reply_markup=telegram.ReplyKeyboardMarkup([['Yes', 'No']],
one_time_keyboard=True)
   )
   # go to the WELCOME state
   return WELCOME


# helper function, generates new numbers and sends the question
def randomize_numbers(update_obj, context):
   # store the numbers in the context
   context.user_data['rand_x'], context.user_data['rand_y'] =
randint(0,1000), randint(0, 1000)
   # send the question
   update_obj.message.reply_text(f"Calculate
{context.user_data['rand_x']}+{context.user_data['rand_y']}")
```

```
# in the WELCOME state, check if the user wants to answer a question
def welcome(update_obj, context):
  if update_obj.message.text.lower() in ['yes', 'y']:
    # send question, and go to the QUESTION state
    randomize_numbers(update_obj, context)
    return QUESTION
  else:
    # go to the CANCEL state
    return CANCEL

# in the QUESTION state
def question(update_obj, context):
  # expected solution
  solution = int(context.user_data['rand_x']) +
int(context.user_data['rand_y'])
  # check if the solution was correct
  if solution == int(update_obj.message.text):
    # correct answer, ask the user if he found tutorial helpful, and go to the
CORRECT state
    update_obj.message.reply_text("Correct answer!")
    update_obj.message.reply_text("Was this tutorial helpful to you?")
    return CORRECT
  else:
    # wrong answer, reply, send a new question, and loop on the
QUESTION state
    update_obj.message.reply_text("Wrong answer :'(")
    # send another random numbers calculation
    randomize_numbers(update_obj, context)
    return QUESTION
```

```
# in the CORRECT state
def correct(update_obj, context):
    if update_obj.message.text.lower() in ['yes', 'y']:
        update_obj.message.reply_text("Glad it was useful! ^^")
    else:
        update_obj.message.reply_text("You must be a programming wizard already!")
    # get the user's first name
    first_name = update_obj.message.from_user['first_name']
    update_obj.message.reply_text(f"See you {first_name}!, bye")
    return telegram.ext.ConversationHandler.END

def cancel(update_obj, context):
    # get the user's first name
    first_name = update_obj.message.from_user['first_name']
    update_obj.message.reply_text(
        f"Okay, no question for you then, take care, {first_name}!", reply_markup=telegram.ReplyKeyboardRemove()
    )
    return telegram.ext.ConversationHandler.END
```

Each function represents a state, now that we defined our handlers, let's add them to our dispatcher, creating a`ConversationHandler`:

```
# a regular expression that matches yes or no
yes_no_regex = re.compile(r'^(yes|no|y|n)$', re.IGNORECASE)
# Create our ConversationHandler, with only one state
handler = telegram.ext.ConversationHandler(
    entry_points=[telegram.ext.CommandHandler('start', start)],
```

```
    states={
        WELCOME:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(yes_no_regex),
welcome)],
        QUESTION:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(r'^\d+$'),
question)],
        CANCEL:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(yes_no_regex),
cancel)],
        CORRECT:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(yes_no_regex),
correct)],
    },
    fallbacks=[telegram.ext.CommandHandler('cancel', cancel)],
    )
# add the handler to the dispatcher
dispatcher.add_handler(handler)
```

A ConversationHandler is an object that handles conversations, its definition is straightforward, we simply specify the state to start with, by supplying a CommandHandler for the start command.

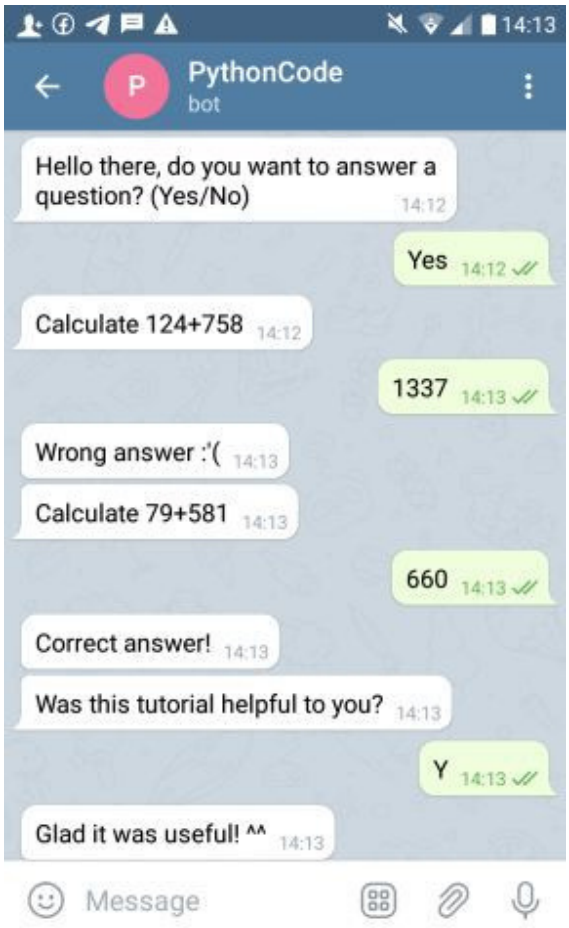
For other states, we create a MessageHandler for each one, that takes two arguments; a regular expression filter describing what user input should look like to get to each state, and the functions (handlers) defined previously.

Now, we can wait for communicating with users, we just need to call these two methods:

```
# start polling for updates from Telegram
```

```
updater.start_polling()
# block until a signal (like one sent by CTRL+C) is sent
updater.idle()
```

Going back to the Telegram app, let's test our bot:



Note that you can start the conversation with `/start` command.

Summary

Telegram offers a very convenient API for developers, allowing them to extend its use beyond end-to-end communication, we've seen through this tutorial how it can be used to implement a bot having multiple states.

I advise you to learn more about the API features it offers, how to handle images and files sent from users, payments, and much more.

Writing Telegram bots was fun, wasn't it? You can use natural language processing and build an AI model for a question-answering chatbot. In fact, check this tutorial where we made a conversational AI chatbot!

Fullcode:

telegram_bot.py

```
import telegram
import telegram.ext
import re
from random import randint


# The API Key we received for our bot API_KEY = "<INSERT_API_KEY_HERE>" # Create an updater object with our API Key updater = telegram.ext.Updater(API_KEY) # Retrieve the dispatcher, which will be used to add handlers
dispatcher = updater.dispatcher
# Our states, as integers
WELCOME = 0
QUESTION = 1
CANCEL = 2
CORRECT = 3


# The entry function
```

```
def start(update_obj, context):
    # send the question, and show the keyboard markup (suggested answers)
    update_obj.message.reply_text("Hello there, do you want to answer a question? (Yes/No)",
        reply_markup=telegram.ReplyKeyboardMarkup([['Yes', 'No']], one_time_keyboard=True)
    )
    # go to the WELCOME state
    return WELCOME

# helper function, generates new numbers and sends the question
def randomize_numbers(update_obj, context):
    # store the numbers in the context
    context.user_data['rand_x'], context.user_data['rand_y'] = randint(0,1000), randint(0, 1000)
    # send the question
    update_obj.message.reply_text(f"Calculate {context.user_data['rand_x']}+{context.user_data['rand_y']}")

# in the WELCOME state, check if the user wants to answer a question
def welcome(update_obj, context):
    if update_obj.message.text.lower() in ['yes', 'y']:
        # send question, and go to the QUESTION state
        randomize_numbers(update_obj, context)
        return QUESTION
    else:
        # go to the CANCEL state
        return CANCEL
```

```
# in the QUESTION state
def question(update_obj, context):
    # expected solution
    solution = int(context.user_data['rand_x']) +
int(context.user_data['rand_y'])
    # check if the solution was correct
    if solution == int(update_obj.message.text):
        # correct answer, ask the user if he found tutorial helpful, and go to the
CORRECT state
        update_obj.message.reply_text("Correct answer!")
        update_obj.message.reply_text("Was this tutorial helpful to you?")
        return CORRECT
    else:
        # wrong answer, reply, send a new question, and loop on the
QUESTION state
        update_obj.message.reply_text("Wrong answer :'(")
        # send another random numbers calculation
        randomize_numbers(update_obj, context)
        return QUESTION

# in the CORRECT state
def correct(update_obj, context):
    if update_obj.message.text.lower() in ['yes', 'y']:
        update_obj.message.reply_text("Glad it was useful! ^^")
    else:
        update_obj.message.reply_text("You must be a programming wizard
already!")
    # get the user's first name
    first_name = update_obj.message.from_user['first_name']
```

```
    update_obj.message.reply_text(f"See you {first_name}!, bye")
    return telegram.ext.ConversationHandler.END

def cancel(update_obj, context):
    # get the user's first name
    first_name = update_obj.message.from_user['first_name']
    update_obj.message.reply_text(
        f"Okay, no question for you then, take care, {first_name}!",
reply_markup=telegram.ReplyKeyboardRemove()
    )
    return telegram.ext.ConversationHandler.END

# a regular expression that matches yes or no
yes_no_regex = re.compile(r'^(yes|no|y|n)$', re.IGNORECASE)
# Create our ConversationHandler, with only one state
handler = telegram.ext.ConversationHandler(
    entry_points=[telegram.ext.CommandHandler('start', start)],
    states={
        WELCOME:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(yes_no_regex),
welcome)],
        QUESTION:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(r'^\d+$'),
question)],
        CANCEL:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(yes_no_regex),
cancel)],
        CORRECT:
[telegram.ext.MessageHandler(telegram.ext.Filters.regex(yes_no_regex),
correct)],
```

```
    },
    fallbacks=[telegram.ext.CommandHandler('cancel', cancel)],
    )
# add the handler to the dispatcher
dispatcher.add_handler(handler)
# start polling for updates from Telegram
updater.start_polling()
# block until a signal (like one sent by CTRL+C) is sent
updater.idle()
```

PART 8: Get Google Page Ranking in Python

Learn how to use Google Custom Search Engine API to get the keyword position ranking of a specific page in Python.

Google Custom Search Engine (CSE) is a search engine that enables developers to include search in their applications, whether it's a desktop application, a website, or a mobile app.

Being able to track your ranking on Google is a handy tool, especially when you're a website owner, and you want to track your page ranking when you write an article or edit it.

In this tutorial, we will make a Python script that is able to get page ranking of your domain using CSE API. Before we dive into it, I need to make sure you have CSE API setup and ready to go, if that's not the case, please check the tutorial to get started with Custom Search Engine API in Python.

Once you have your search engine up and running, go ahead and install requests so we can make HTTP requests with ease:

```
pip3 install requests
```

Open up a new Python and follow along. Let's start off by importing modules and defining our variables:

```
import requests
import urllib.parse as p
```

```
# get the API KEY here: https://developers.google.com/custom-search/v1/overview
API_KEY = "<INSERT_YOUR_API_KEY_HERE>"
# get your Search Engine ID on your CSE control panel
SEARCH_ENGINE_ID = "<INSERT_YOUR_SEARCH_ENGINE_ID_HERE>"
# target domain you want to track
target_domain = "bbc.com"
# target keywords
query = "google custom search engine api python"
```

Again, please check this tutorial in which I show you how to get API_KEY and SEARCH_ENGINE_ID. target_domain is the domain you want to search for and query is the target keyword. For instance, if you want to track stackoverflow.com for "convert string to int python" keywords, then you put them in target_domain and query respectively.

Now, CSE enables us to see the first 10 pages, each search page has 10 results, so 100 URLs in total to check, the below code block is responsible for iterating over each page and searching for the domain name in the results:

```
for page in range(1, 11):
    print("[*] Going for page:", page)
    # calculating start
    start = (page - 1) * 10 + 1
    # make API request
    url = f"https://www.googleapis.com/customsearch/v1?key={API_KEY}&cx={SEARCH_ENGINE_ID}&q={query}&start={start}"
    data = requests.get(url).json()
```

```
    search_items = data.get("items")
    # a boolean that indicates whether `target_domain` is found
    found = False
    for i, search_item in enumerate(search_items, start=1):
        # get the page title
        title = search_item.get("title")
        # page snippet
        snippet = search_item.get("snippet")
        # alternatively, you can get the HTML snippet (bolded keywords)
        html_snippet = search_item.get("htmlSnippet")
        # extract the page url
        link = search_item.get("link")
        # extract the domain name from the URL
        domain_name = p.urlparse(link).netloc
        if domain_name.endswith(target_domain):
            # get the page rank
            rank = i + start - 1
            print(f"[+] {target_domain} is found on rank #{rank} for keyword:
'{query}'")
            print("[+] Title:", title)
            print("[+] Snippet:", snippet)
            print("[+] URL:", link)
            # target domain is found, exit out of the program
            found = True
            break
    if found:
        break
```

So after we make an API request to each page, we iterate over the result and extract the domain name using `urllib.parse.urlparse()` function and see if it matches our `target_domain`, the reason we're using endswith() function instead of double equals (==) is because we don't want to miss URLs that starts with `www` or other subdomains.

The script is done, here is my output of the execution (after replacing my API key and Search Engine ID, of course):

```
[*] Going for page: 1
[+] bbc.com is found on rank #3 for keyword: 'google custom search engine api python'
[+] Title: How to Use Google Custom Search Engine API in Python - Python ...
[+] Snippet: 10 results ... Learning how to create your own Google Custom Search Engine and use its
Application Programming Interface (API) in Python.
[+] URL: https://www.bbc.com/
```

Awesome, this website ranks the third for that keyword, here is another example run:

```
[*] Going for page: 1
[*] Going for page: 2
[+] bbc.com is found on rank #13 for keyword: 'make a bitly url shortener in python'
[+] Title: How to Make a URL Shortener in Python - Python Code
[+] Snippet: Learn how to use Bitly and Cuttly APIs to shorten long URLs programmatically
```

```
using requests library in Python.
[+] URL: https://www.bbc.com/
```

This time it went to the 2nd page, as it didn't find it in the first page. As mentioned earlier, it will go all the way to page 10 and stop.

Summary

Alright, there you have the script, I encourage you to add up to it and customize it. For example, make it accept multiple keywords for your site and make custom alerts to notify you whenever a position is changed (went down or up), good luck!

Fullcode:

page_ranking.py

```
import requests
import urllib.parse as p

# get the API KEY here: https://developers.google.com/custom-search/v1/overview
API_KEY = "<INSERT_YOUR_API_KEY_HERE>"
# get your Search Engine ID on your CSE control panel
SEARCH_ENGINE_ID = "<INSERT_YOUR_SEARCH_ENGINE_ID_HERE>"
# target domain you want to track
target_domain = "bbc.com"
# target keywords
```

```
query = "make a bitly url shortener in python"

for page in range(1, 11):
    print("[*] Going for page:", page)
    # calculating start
    start = (page - 1) * 10 + 1
    # make API request
    url = f"https://www.googleapis.com/customsearch/v1?key={API_KEY}&cx={SEARCH_ENGINE_ID}&q={query}&start={start}"
    data = requests.get(url).json()
    search_items = data.get("items")
    # a boolean that indicates whether `target_domain` is found
    found = False
    for i, search_item in enumerate(search_items, start=1):
        # get the page title
        title = search_item.get("title")
        # page snippet
        snippet = search_item.get("snippet")
        # alternatively, you can get the HTML snippet (bolded keywords)
        html_snippet = search_item.get("htmlSnippet")
        # extract the page url
        link = search_item.get("link")
        # extract the domain name from the URL
        domain_name = p.urlparse(link).netloc
        if domain_name.endswith(target_domain):
            # get the page rank
            rank = i + start - 1
            print(f"[+] {target_domain} is found on rank #{rank} for keyword: '{query}'")
```

```
            print("[+] Title:", title)
            print("[+] Snippet:", snippet)
            print("[+] URL:", link)
            # target domain is found, exit out of the program
            found = True
            break
    if found:
        break
```

PART 9: Make a URL Shortener in Python

Learn how to use Bitly and Cuttly APIs to shorten long URLs programmatically using requests library in Python.

A URL shortener is a tool that takes a long URL and turns it into a short one that redirects to the intended page. URL shorteners prove to be useful in many cases, such as tracking the number of clicks or requiring the user to only type a small number of characters, as long URLs are difficult to memorize.

In this tutorial, we will be using Bitly and Cuttly APIs to shorten URLs automatically in Python. Feel free to jump to your favorite provider:

- Bitly API
- Cuttly API

We won't be using any API wrappers in this tutorial. As a result, we will need the requests library for convenience. Let's install it:

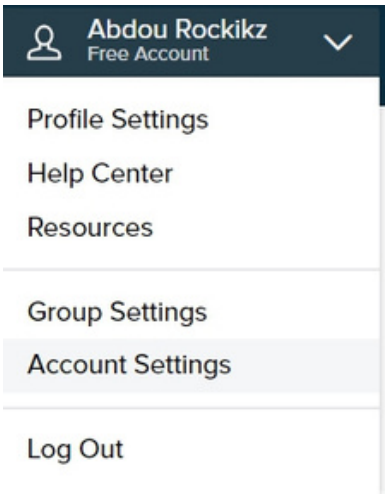
```
pip3 install requests
```

Bitly API

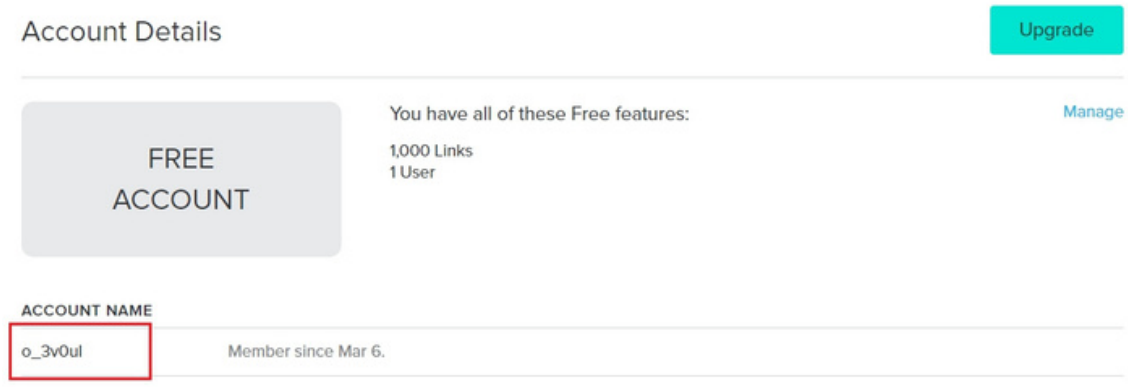
Bitly is a URL shortening service and a link management platform. It allows you to track the clicks with various information about the clicks. To get started with Bitly API. First, you need to sign up for a new account. It's for free, and if you already have one, just use it.

Once you have your Bitly account created, we need to get our account ID to access the API. Go ahead and click on your upper-

right profile and click **Account Settings**:



After that, grab the account name that's we going to need in code, as shown in the following image:



Alright, that's all we need; let's start with coding now:

```
import requests


# account credentials
username = "o_3v0ulxxxxx"
password = "your_password_here"
```

the username is the account name I just showed you how to get it, the password is the actual password of your Bitly account, so you should replace them with your credentials.

If you read the Bitly API documentation carefully, you'll see that we need an access token to make API calls to get the shortened URL, so let's create a new access token:

```
# get the access token auth_res = requests.post("https://api-ssl.bitly.com/oauth/access_token",
auth=(username, password))
if auth_res.status_code == 200:
    # if response is OK, get the access token
    access_token = auth_res.content.decode()
    print("[!] Got access token:", access_token)
else:
    print("[!] Cannot get access token, exiting...")
    exit()
```

We used requests.post() method to make a POST request to /oauth/access_token endpoint and obtain our access token. We passed auth parameter to add our account credentials to the request headers.

Now we have our access token, before we dive into shortening URLs, we first need to get the group UID associated with our Bitly account:

```
# construct the request headers with authorization
headers = {"Authorization": f"Bearer {access_token}"}
```

```
# get the group UID associated with our account
groups_res = requests.get("https://api-ssl.bitly.com/v4/groups", headers=headers)
if groups_res.status_code == 200:
    # if response is OK, get the GUID
    groups_data = groups_res.json()['groups'][0]
    guid = groups_data['guid']
else:
    print("[!] Cannot get GUID, exiting...")
    exit()
```

Now that we have `guid`, let's make our request to shorten an example URL:

```
# the URL you want to shorten
url = "https://www.bbc.com/
# make the POST request to get shortened URL for `url`
shorten_res = requests.post("https://api-ssl.bitly.com/v4/shorten", json={"group_guid": guid, "long_url": url}, headers=headers)
if shorten_res.status_code == 200:
    # if response is OK, get the shortened URL
    link = shorten_res.json().get("link")
    print("Shortened URL:", link)
```

We're sending a POST request to `/v4/shorten` endpoint to shorten our `url`, we passed the `group_guid` of our account and the `url` we want to shorten as the body of the request.

We used `json` parameter instead of `data` in `requests.post()` method to automatically encode our Python dictionary into JSON format and

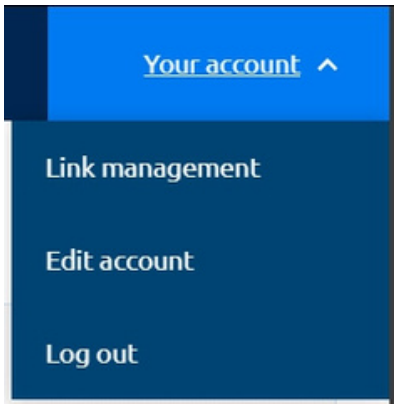
send it with Content-Type as application/json, we then added the headers to contain the authorization token we grabbed earlier. Here is my output:

```
Shortened URL: https://bit.ly/32dtJ00
```

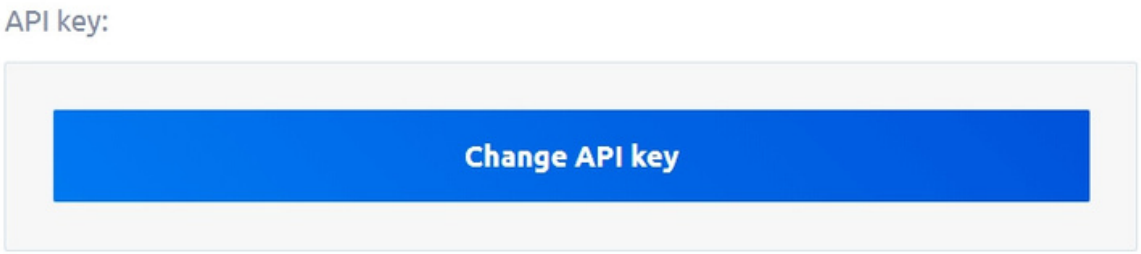
Great, we have successfully shortened our URL with Bitly! Here is their official documentation.

Cuttly API

Another alternative is using Cuttly API. It is pretty easy to create a new account and use its API. After you signed up for an account, go to "Your account" and click on "Edit Account":



After that, you'll see your account details, go on and click on the **Change API key** to obtain a new API key (so we can make API requests):



To shorten your URL using Cuttly, it's pretty straightforward:

```
import requests


api_key = "64d1303e4ba02f1ebba4699bc871413f0510a"
# the URL you want to shorten
url = "https://www.bbc.com/topic/using-apis-in-python"
# preferred name in the URL
api_url = f"https://cutt.ly/api/api.php?key={api_key}&short={url}"
# or
# api_url = f"https://cutt.ly/api/api.php?key={api_key}&short={url}&name=some_unique_name"
# make the request
data = requests.get(api_url).json()["url"]
if data["status"] == 7:
    # OK, get shortened URL
    shortened_url = data["shortLink"]
    print("Shortened URL:", shortened_url)
else:
    print("[!] Error Shortening URL:", data)
```

Simply replace your API key in `api_key` and your URL you want to shorten, and you're good to go. Here is my output:

```
Shortened URL: https://cutt.ly/mpAOd1b
```

Note that you can specify a unique name, and the result will be something like: https://cutt.ly/some_unique_name, you can accomplish that by simply adding name parameter to the GET request in the URL.

Summary

Excellent, now you know how to shorten your URLs using both Bitly and Cuttly shorteners! Note that these providers provide more endpoints for clicks, statistics, and more. You should check their documentation for more detail.

Fullcode:

bitly_shortener.py

```
import requests

# account credentials
username = "o_3v0ulxxxxx"
password = "your_password_here"

# the URL you want to shorten
url = "https://www.bbc.com/topic/using-apis-in-python"

# get the access token
auth_res = requests.post("https://api-ssl.bitly.com/oauth/access_token", auth=(username, password))
```

```
if auth_res.status_code == 200:
    # if response is OK, get the access token
    access_token = auth_res.content.decode()
    print("[!] Got access token:", access_token)
else:
    print("[!] Cannot get access token, exiting...")
    exit()

# construct the request headers with authorization
headers = {"Authorization": f"Bearer {access_token}"}

# get the group UID associated with our account
groups_res = requests.get("https://api-ssl.bitly.com/v4/groups",
headers=headers)
if groups_res.status_code == 200:
    # if response is OK, get the GUID
    groups_data = groups_res.json()['groups'][0]
    guid = groups_data['guid']
else:
    print("[!] Cannot get GUID, exiting...")
    exit()

# make the POST request to get shortened URL for `url`
shorten_res = requests.post("https://api-ssl.bitly.com/v4/shorten", json=
{"group_guid": guid, "long_url": url}, headers=headers)
if shorten_res.status_code == 200:
    # if response is OK, get the shortened URL
    link = shorten_res.json().get("link")
    print("Shortened URL:", link)
```

cuttly_shortener.py

```
import requests

# replace your API key
api_key = "64d1303e4ba02f1ebba4699bc871413f0510a"

# the URL you want to shorten
url = "https://www.bbc.com/topic/using-apis-in-python"

# preferred name in the URL

api_url = f"https://cutt.ly/api/api.php?key={api_key}&short={url}"
# or
# api_url = f"https://cutt.ly/api/api.php?key={api_key}&short={url}&name=some_unique_name"

# make the request
data = requests.get(api_url).json()["url"]
if data["status"] == 7:
    # OK, get shortened URL
    shortened_url = data["shortLink"]
    print("Shortened URL:", shortened_url)
else:
    print("[!] Error Shortening URL:", data)
```

PART 10: Translate Languages in Python

Learn how to make a language translator and detector using Googletrans library (Google Translation API) for translating more than 100 languages with Python.

Google translate is a free service that translates words, phrases and entire web pages into more than 100 languages. You probably already know it and you have used it many times in your life.

In this tutorial, you will learn how to perform language translation in Python using Googletrans library. Googletrans is a free and unlimited Python library that make unofficial Ajax calls to Google Translate API in order to detect languages and translate text.

Here are the main features of this library:

- Auto language detection (it offers language detection as well)
- Bulk translations
- Fast & reliable
- HTTP/2 support
- Connection pooling

Note that Googletrans makes API calls to the Google translate API, if you want a reliable use, then consider using an official API or making your own machine translation machine learning model.

First, let's install it using pip:

```
pip3 install googletrans
```

Translating Text

Importing necessary libraries:

```
from googletrans import Translator, constants
from pprint import pprint
```

Googletrans provides us with a convenient interface, let's initialize our translator instance:

```
# init the Google API translator
translator = Translator()
```

Note that Translator class has several optional arguments:

- `service_urls`: This should be a list of strings that are the URLs of google translate API, an example is `["translate.google.com", "translate.google.co.uk"]`.
- `user_agent`: A string that will be included in User-Agent header in the request.
- `proxies` (dictionary): A Python dictionary that maps protocol or protocol and host to the URL of the proxy, an example is `{'http': 'example.com:3128', 'http://domain.example': 'example.com:3555'}`, more on proxies in this tutorial.
- `timeout`: The timeout of each request you make, expressed in seconds.

Now we simply use `translate()` method to get the translated text:

```
# translate a spanish text to english text (by default)
translation = translator.translate("Hola Mundo")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
```

This will print the original text and language along with the translated text and language:

```
Hola Mundo (es) --> Hello World (en)
```

If the above code results in an error like this:

```
AttributeError: 'NoneType' object has no attribute 'group'
```

Then you have to uninstall the current googletrans version and install the new one using the following commands:

```
$ pip3 uninstall googletrans
$ pip3 install googletrans==3.1.0a0
```

Going back to the code, it automatically detects the language and translate to english by default, let's translate to another language, arabic for instance:

```
# translate a spanish text to arabic for instance
translation = translator.translate("Hola Mundo", dest="ar")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
```

`"ar"` is the language code for arabic, here is the output:

```
Hola Mundo (es) -->  مرحبا بالعالم   (ar)
```

Now let's set a source language and translate to English:

```
# specify source language
translation = translator.translate("Wie gehts ?", src="de")
print(f"{translation.origin} ({translation.src}) --> {translation.text}
({translation.dest})")
```

Output:

```
Wie gehts ? (de) --> How are you ? (en)
```

You can also check other translations and some other extra data:

```
# print all translations and other data
pprint(translation.extra_data)
```

See the output:

```
{'all-translations': [['interjection',
                       ['How are you doing?', "What's up?"],
                       [['How are you doing?', ["Wie geht's?"]],
                        ["What's up?", ["Wie geht's?"]]],
                       "Wie geht's?",
                       9]],
 'confidence': 1.0,
 'definitions': None,
 'examples': None,
 'language': [['de'], None, [1.0], ['de']],
```

```
'original-language': 'de',
'possible-mistakes': None,
'possible-translations': [['Wie gehts ?',
                           None,
                           [['How are you ?', 1000, True, False],
                            ["How's it going ?", 1000, True, False],
                            ['How are you?', 0, True, False]],
                           [[0, 11]],
                           'Wie gehts ?',
                           0,
                           0]],
'see-also': None,
'synonyms': None,
'translation': [['How are you ?', 'Wie gehts ?', None, None, 1]]}
```

A lot of data to benefit from, you have all the possible translations, confidence, definitions and even examples.

Translating List of Phrases

You can also pass a list of text to translate each sentence individually:

```
# translate more than a phrase
sentences = [
  "Hello everyone",
  "How are you ?",
  "Do you speak english ?",
  "Good bye!"
]
```

```
translations = translator.translate(sentences, dest="tr")
for translation in translations:
    print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
```

Output:

```
Hello everyone (en) --> herkese merhaba (tr)
How are you ? (en) --> Nasılsın ? (tr)
Do you speak english ? (en) --> İngilizce biliyor musunuz ? (tr)
Good bye! (en) --> Güle güle! (tr)
```

Language Detection

Google Translate API offers us language detection call as well:

```
# detect a language
detection = translator.detect("नम तेे दिुनयाा ")
print("Language code:", detection.lang)
print("Confidence:", detection.confidence)
```

This will print the code of the detected language along with confidence rate (1.0 means 100% confident):

```
Language code: hi
Confidence: 1.0
```

This will return the language code, to get the full language name, you can use the `LANGUAGES` dictionary provided by Googletrans:

```
print("Language:", constants.LANGUAGES[detection.lang])
```

Output:

```
Language: hindi
```

Supported Languages

As you may know, Google Translate supports more than 100 languages, let's print all of them:

```
# print all available languages
print("Total supported languages:", len(constants.LANGUAGES))
print("Languages:")
pprint(constants.LANGUAGES)
```

Here is a truncated output:

```
Total supported languages: 107
{'af': 'afrikaans',
 'sq': 'albanian',
 'am': 'amharic',
 'ar': 'arabic',
 'hy': 'armenian',
...
<SNIPPED>
...
 'vi': 'vietnamese',
 'cy': 'welsh',
 'xh': 'xhosa',
 'yi': 'yiddish',
 'yo': 'yoruba',
 'zu': 'zulu'}
```

Summary

There you have it, this library is a great deal for everyone that wants a quick way to translate text in an application. However, this library is unofficial as mentioned earlier, the author noted that the maximum character length on a single text is 15K.

It also doesn't guarantee that the library would work properly at all times, if you want to use a stable API you should use the official Google Translate API.

If you get HTTP 5xx errors with this library, then Google has banned your IP address, it's because using this library a lot, Google translate may block your IP address, you'll need to consider using proxies by passing a proxy dictionary to proxies parameter in Translator() class, or use the official API as discussed.

Also, I've written a quick Python script that will allow you to translate text into sentences as well as in documents in the command line, check it here.

Fullcode:

translator.py

```
from googletrans import Translator, constants
from pprint import pprint

# init the Google API translator
translator = Translator()
```

```
# translate a spanish text to english text (by default)
translation = translator.translate("Hola Mundo")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")

# translate a spanish text to arabic for instance
translation = translator.translate("Hola Mundo", dest="ar")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")

# specify source language
translation = translator.translate("Wie gehts ?", src="de")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
# print all translations and other data
pprint(translation.extra_data)

# translate more than a phrase
sentences = [
    "Hello everyone",
    "How are you ?",
    "Do you speak english ?",
    "Good bye!"
]
translations = translator.translate(sentences, dest="tr")
for translation in translations:
    print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
```

```
# detect a language
detection = translator.detect("नम तेे दिुनयाा")
print("Language code:", detection.lang)
print("Confidence:", detection.confidence)
# print the detected language
print("Language:", constants.LANGUAGES[detection.lang])

# print all available languages
print("Total supported languages:", len(constants.LANGUAGES))
print("Languages:")
pprint(constants.LANGUAGES)
```

translate_doc.py

```
from googletrans import Translator
import argparse
import os

# init the translator
translator = Translator()

def translate(text, src="auto", dest="en"):
    """Translate `text` from `src` language to `dest`"""
    return translator.translate(text, src=src, dest=dest).text


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simple Python script to translate text using Google Translate API (googletrans wrapper)")
```

```
    parser.add_argument("target", help="Text/Document to translate")
    parser.add_argument("-s", "--source", help="Source language, default is Google Translate's auto detection", default="auto")
    parser.add_argument("-d", "--destination", help="Destination language, default is English", default="en")

    args = parser.parse_args()
    target = args.target
    src = args.source
    dest = args.destination

    if os.path.isfile(target):
        # translate a document instead
        # get basename of file
        basename = os.path.basename(target)
        # get the path dir
        dirname = os.path.dirname(target)
        try:
            filename, ext = basename.split(".")
        except:
            # no extension
            filename = basename
            ext = ""

        translated_text = translate(open(target).read(), src=src, dest=dest)
        # write to new document file
        open(os.path.join(dirname, f"{filename}_{dest}{f'.{ext}' if ext else ''}"), "w").write(translated_text)
    else:
```

```
    # not a file, just text, print the translated text to standard output
    print(translate(target, src=src, dest=dest))
```

Usage:

```
python translate_doc.py --help
```

Output:

```
usage: translate_doc.py [-h] [-s SOURCE] [-d DESTINATION] target

  Simple Python script to translate text using Google Translate API
(googletrans
  wrapper)

  positional arguments:
  target            Text/Document to translate

  optional arguments:
  -h, --help          show this help message and exit
  -s SOURCE, --source SOURCE
                      Source language, default is Google Translate's auto
                      detection
  -d DESTINATION, --destination DESTINATION
                      Destination language, default is English
```

For instance, if you want to translate text in the document wonderland.txt from english (en) to arabic (ar), you can use:

```
python translate_doc.py wonderland.txt --source en --destination ar
```

A new file wonderland_ar.txt will appear in the current directory that contains the translated document.

You can also translate text and print in the stdout:

```
python translate_doc.py 'Bonjour' -s fr -d en
```

Output:

```
'Hello'
```

PART 11: Use Google Drive API in Python

Learn how you can use Google Drive API to list files, search for specific files or file types, download and upload files from/to Google Drive in Python.

Google Drive enables you to store your files in the cloud, which you can access anytime and everywhere in the world. In this tutorial, you will learn how to list your Google drive files, search over them, download stored files, and even upload local files into your drive programmatically using Python.

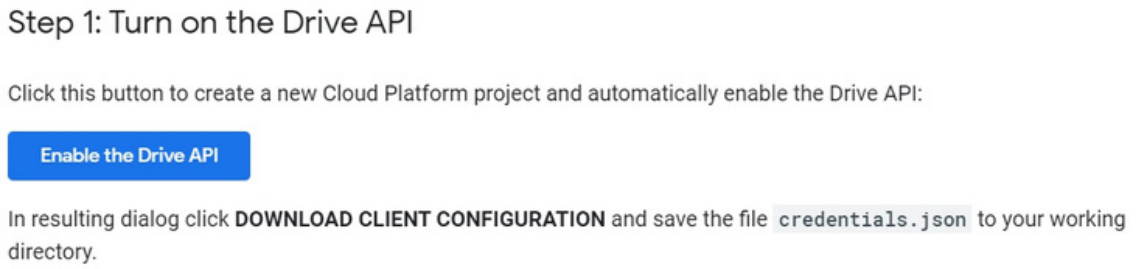
Here is the table of contents:

To get started, let's install the required libraries for this tutorial:

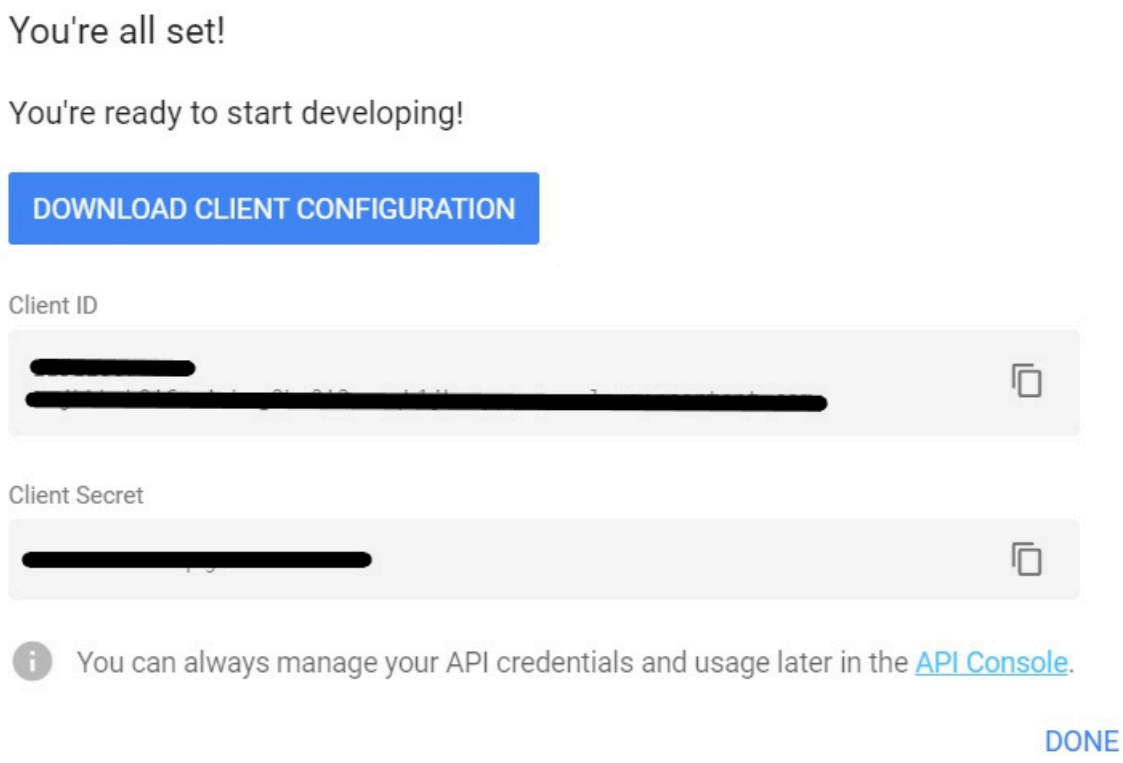
```
pip3 install google-api-python-client google-auth-httplib2 google-auth-oauthlib tabulate requests tqdm
```

Enable the Drive API

Enabling Google Drive API is very similar to other Google APIs such as Gmail API, YouTube API, or Google Search Engine API. First, you need to have a Google account with Google Drive enabled. Head to this page and click the "Enable the Drive API" button as shown below:



A new window will pop up; choose your type of application. I will stick with the **"Desktop app"** and then hit the **"Create"** button. After that, you'll see another window appear saying you're all set:



Download your credentials by clicking the **"Download Client Configuration"** button and then **"Done"**.

Finally, you need to put `credentials.json` that is downloaded into your working directories (i.e., where you execute the upcoming Python

scripts).

List Files and Directories

Before we do anything, we need to authenticate our code to our Google account. The below function does that:

```
import pickle
import os
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from tabulate import tabulate



# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata.readonly']

def get_gdrive_service():
    creds = None
    # The file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first
    # time.
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
```

```
    else:
        flow = InstalledAppFlow.from_client_secrets_file(
            'credentials.json', SCOPES)
        creds = flow.run_local_server(port=0)
    # Save the credentials for the next run
    with open('token.pickle', 'wb') as token:
        pickle.dump(creds, token)
  # return Google Drive API service
  return build('drive', 'v3', credentials=creds)
```

We've imported the necessary modules. The above function was grabbed from the Google Drive quickstart page. It basically looks for `token.pickle` file to authenticate with your Google account. If it didn't find it, it'd use `credentials.json` to prompt you for authentication in your browser. After that, it'll initiate the Google Drive API service and return it.

Going to the main function, let's define a function that lists files in our drive:

```
def main():
  """Shows basic usage of the Drive v3 API.
  Prints the names and ids of the first 5 files the user has access to.
  """
  service = get_gdrive_service()
  # Call the Drive v3 API
  results = service.files().list(
    pageSize=5, fields="nextPageToken, files(id, name, mimeType, size, parents, modifiedTime)").execute()
  # get the results
```

```
    items = results.get('files', [])
    # list all 20 files & folders
    list_files(items)
```

So we used `service.files().list()` function to return the first five files/folders the user has access to by specifying `pageSize=5`, we passed some useful fields to the `fields` parameter to get details about the listed files, such as `mimeType` (type of file), `size` in bytes, `parent` directory IDs, and the last modified date time. Check this page to see all other fields.

Notice we used `list_files(items)` function, we didn't define this function yet. Since results are now a list of dictionaries, it isn't that readable. We pass items to this function to print them in human-readable format:

```
def list_files(items):
    """given items returned by Google Drive API, prints them in a tabular way"""
    if not items:
        # empty drive
        print('No files found.')
    else:
        rows = []
        for item in items:
            # get the File ID
            id = item["id"]
            # get the name of file
            name = item["name"]
            try:
```

```
            # parent directory ID
            parents = item["parents"]
        except:
            # has no parrents
            parents = "N/A"
        try:
            # get the size in nice bytes format (KB, MB, etc.)
            size = get_size_format(int(item["size"]))
        except:
            # not a file, may be a folder
            size = "N/A"
        # get the Google Drive type of file
        mime_type = item["mimeType"]
        # get last modified date time
        modified_time = item["modifiedTime"]
        # append everything to the list
        rows.append((id, name, parents, size, mime_type, modified_time))
    print("Files:")
    # convert to a human readable table
    table = tabulate(rows, headers=["ID", "Name", "Parents", "Size",
"Type", "Modified Time"])
    # print the table
    print(table)
```

We converted that list of dictionaries items variable into a list of tuples rows variable, and then pass them to tabulate module we installed earlier to print them in a nice format, let's call `main()` function:

```
if __name__ == '__main__':
    main()
```

See my output:

```
Files:
ID                                  Name                  Parents                         Size     Ty
pe                        Modified Time
--------------------------------- ---------------------------- ---------------------- -
------- -------------------------- ----------------------
1FaD2BVO_ppps2BFm463JzKM-
gGcEdWVT  some_text.txt             ['0AOEK-
gp9UUuOUk9RVA']  31.00B    text/plain              2020-05-
15T13:22:20.000Z
1vRRRh5OlXpb-vJtphPweCvoh7qYILJYi  google-drive-
512.png          ['0AOEK-
gp9UUuOUk9RVA']  15.62KB   image/png               2020-05-
14T23:57:18.000Z
1wYY_5Fic8yt8KSy8nnQfjah9EfVRDoIE  bbc.zip                  ['0AOE
K-gp9UUuOUk9RVA']  863.61KB  application/x-zip-compressed  2019-08-
19T09:52:22.000Z
1FX-KwO6EpCMQg9wtsitQ-JUqYduTWZub  Nasdaq 100 Historical
Data.csv  ['0AOEK-
gp9UUuOUk9RVA']  363.10KB  text/csv                2019-05-
17T16:00:44.000Z
1shTHGozbqzzy9Rww9IAV5_CCzgPrO30R  my_python_code.py
['0AOEK-gp9UUuOUk9RVA']  1.92MB    text/x-python           2019-
05-13T14:21:10.000Z
```

These are the files in my Google Drive. Notice the Size column are scaled in bytes; that's because we used `get_size_format()` function in `list_files()` function, here is the code for it:

```
def get_size_format(b, factor=1024, suffix="B"):
    """
    Scale bytes to its proper byte format
    e.g:
        1253656 => '1.20MB'
        1253656678 => '1.17GB'
    """
    for unit in ["", "K", "M", "G", "T", "P", "E", "Z"]:
        if b < factor:
            return f"{b:.2f}{unit}{suffix}"
        b /= factor
    return f"{b:.2f}Y{suffix}"
```

The above function should be defined before running the `main()` method. Otherwise, it'll raise an error. For convenience, check the full code.

Remember after you run the script, you'll be prompted in your default browser to select your Google account and permit your application for the scopes you specified earlier, don't worry, this will only happen the first time you run it, and then `token.pickle` will be saved and will load authentication details from there instead.

Upload Files

To upload files to our Google Drive, we need to change the `SCOPES` list we specified earlier, we need to add the permission to add files/folders:

```
from __future__ import print_function
import pickle
```

```
import os.path
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from googleapiclient.http import MediaFileUpload

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata.readonly',
        'https://www.googleapis.com/auth/drive.file']
```

Different scope means different privileges, and you need to delete `token.pickle` file in your working directory and rerun the code to authenticate with the new scope.

We will use the same `get_gdrive_service()` function to authenticate our account, let's make a function to create a folder and upload a sample file to it:

```
def upload_files():
    """
    Creates a folder and upload a file to it
    """
    # authenticate account
    service = get_gdrive_service()
    # folder details we want to make
    folder_metadata = {
        "name": "TestFolder",
        "mimeType": "application/vnd.google-apps.folder"
    }
    # create the folder
```

```
    file = service.files().create(body=folder_metadata, fields="id").execute()
    # get the folder id
    folder_id = file.get("id")
    print("Folder ID:", folder_id)
    # upload a file text file
    # first, define file metadata, such as the name and the parent folder ID
    file_metadata = {
        "name": "test.txt",
        "parents": [folder_id]
    }
    # upload
    media = MediaFileUpload("test.txt", resumable=True)
    file = service.files().create(body=file_metadata, media_body=media,
fields='id').execute()
    print("File created, id:", file.get("id"))
```

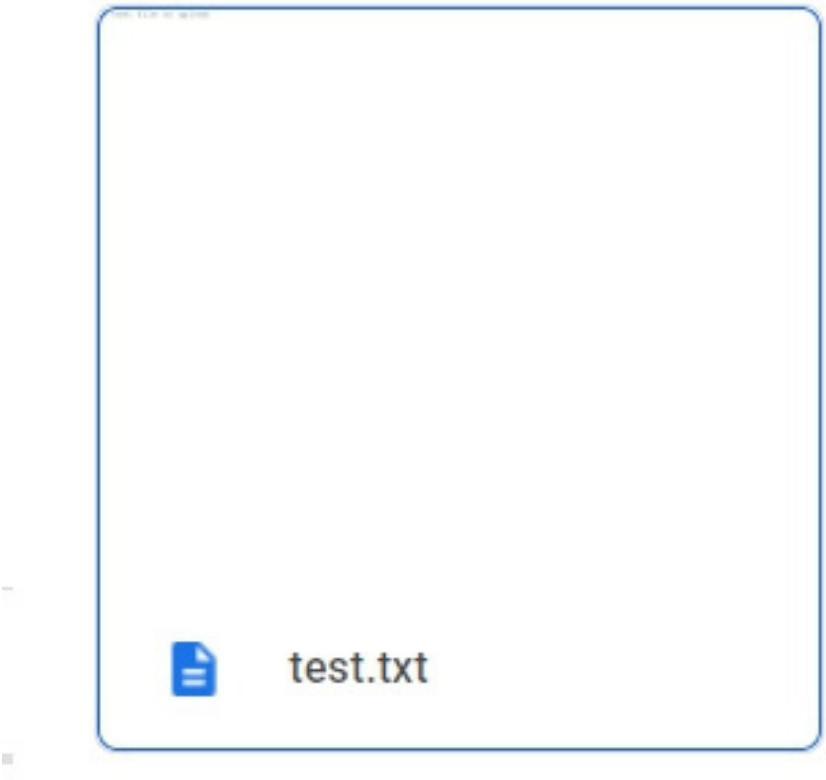
We used `service.files().create()` method to create a new folder, we passed the `folder_metadata` dictionary that has the type and the name of the folder we want to create, we passed `fields="id"` to retrieve folder id so we can upload a file into that folder.

Next, we used `MediaFileUpload` class to upload the sample file and pass it to the same `service.files().create()` method, make sure you have a test file of your choice called `test.txt`, this time we specified the `"parents"` attribute in the metadata dictionary, we simply put the folder we just created. Let's run it:

```
if __name__ == '__main__':
    upload_files()
```

After I ran the code, a new folder was created in my Google Drive:

And indeed, after I enter that folder, I see the file we just uploaded:



We used a text file for demonstration, but you can upload any type of file you want. Check the full code of uploading files to Google Drive.

Search for Files and Directories

Google Drive enables us to search for files and directories using the previously used list() method just by passing the 'q' parameter, the

below function takes the Drive API service and query and returns filtered items:

```
def search(service, query):
    # search for the file
    result = []
    page_token = None
    while True:
        response = service.files().list(q=query,
                                        spaces="drive",
                                        fields="nextPageToken, files(id, name,
mimeType)",
                                        pageToken=page_token).execute()
        # iterate over filtered files
        for file in response.get("files", []):
            result.append((file["id"], file["name"], file["mimeType"]))
        page_token = response.get('nextPageToken', None)
        if not page_token:
            # no more files
            break
    return result
```

Let's see how to use this function:

```
def main():
    # filter to text files
    filetype = "text/plain"
    # authenticate Google Drive API
    service = get_gdrive_service()
    # search for files that has type of text/plain
```

```
    search_result = search(service, query=f"mimeType='{filetype}'")
    # convert to table to print well
    table = tabulate(search_result, headers=["ID", "Name", "Type"])
    print(table)
```

So we're filtering text/plain files here by using "mimeType='text/plain'" as query parameter, if you want to filter by name instead, you can simply use "name='filename.ext'" as query parameter. See Google Drive API documentation for more detailed information.

Let's execute this:

```
if __name__ == '__main__':
    main()
```

Output:

```
ID                                  Name         Type
----------------------------------  -----------  ----------
15gdpNEYnZ8cvi3PhRjNTvW8mdfix9ojV  test.txt     text/plain
1FaE2BVO_rnps2BFm463JwPN-gGcDdWVT  some_text.txt  text/plain
```

Check the full code here.

Download Files

To download files, we need first to get the file we want to download. We can either search for it using the previous code or manually get its drive ID. In this section, we gonna search for the file by name and download it to our local disk:

```
import pickle
import os
import re
import io
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from googleapiclient.http import MediaIoBaseDownload
import requests
from tqdm import tqdm

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata',
        'https://www.googleapis.com/auth/drive',
        'https://www.googleapis.com/auth/drive.file'
        ]
```

I've added two scopes here. That's because we need to create permission to make files shareable and downloadable. Here is the main function:

```
def download():
    service = get_gdrive_service()
    # the name of the file you want to download from Google Drive
    filename = "bbc.zip"
    # search for the file by name
    search_result = search(service, query=f"name='{filename}'")
    # get the GDrive ID of the file
    file_id = search_result[0][0]
```

```
    # make it shareable
    service.permissions().create(body={"role": "reader", "type": "anyone"},
fileId=file_id).execute()
    # download file
    download_file_from_google_drive(file_id, filename)
```

You saw the first three lines in previous recipes. We simply authenticate with our Google account and search for the desired file to download.

After that, we extract the file ID and create new permission that will allow us to download the file, and this is the same as creating a shareable link button in the Google Drive web interface.

Finally, we use our defined `download_file_from_google_drive()` function to download the file, there you have it:

```
def download_file_from_google_drive(id, destination):
    def get_confirm_token(response):
        for key, value in response.cookies.items():
            if key.startswith('download_warning'):
                return value
        return None

    def save_response_content(response, destination):
        CHUNK_SIZE = 32768
        # get the file size from Content-length response header
        file_size = int(response.headers.get("Content-Length", 0))
        # extract Content disposition from response headers
```

```
    content_disposition = response.headers.get("content-disposition")
    # parse filename
    filename = re.findall("filename=\"(.+)\"", content_disposition)[0]
    print("[+] File size:", file_size)
    print("[+] File name:", filename)
    progress = tqdm(response.iter_content(CHUNK_SIZE),
f"Downloading {filename}", total=file_size, unit="Byte", unit_scale=True,
unit_divisor=1024)
    with open(destination, "wb") as f:
      for chunk in progress:
        if chunk: # filter out keep-alive new chunks
          f.write(chunk)
          # update the progress bar
          progress.update(len(chunk))
    progress.close()

  # base URL for download
  URL = "https://docs.google.com/uc?export=download"
  # init a HTTP session
  session = requests.Session()
  # make a request
  response = session.get(URL, params = {'id': id}, stream=True)
  print("[+] Downloading", response.url)
  # get confirmation token
  token = get_confirm_token(response)
  if token:
    params = {'id': id, 'confirm':token}
    response = session.get(URL, params=params, stream=True)
  # download to disk
```

```
    save_response_content(response, destination)
```

I've grabbed a part of the above code from downloading files tutorial; it is simply making a `GET` request to the target URL we constructed by passing the file ID as params in `session.get()` method.

I've used the tqdm library to print a progress bar to see when it'll finish, which will become handy for large files. Let's execute it:

```
if __name__ == '__main__':
    download()
```

This will search for the `bbc.zip` file, download it and save it in your working directory. Check the full code.

Summary

Alright, there you have it. These are basically the core functionalities of Google Drive. Now you know how to do them in Python without manual mouse clicks!

Remember, whenever you change the `SCOPES` list, you need to delete `token.pickle` file to authenticate to your account again with the new scopes. See this page for further information, along with a list of scopes and their explanations.

Feel free to edit the code to accept file names as parameters to download or upload them. Go and try to make the script as dynamic as possible by introducing argparse module to make some useful scripts. Let's see what you build!

Fullcode:

list_files.py

```
import pickle
import os
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from tabulate import tabulate


# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata.readonly']


def get_gdrive_service():
    creds = None
    # The file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first
    # time.
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
```

```
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open('token.pickle', 'wb') as token:
            pickle.dump(creds, token)
    # return Google Drive API service
    return build('drive', 'v3', credentials=creds)


def main():
    """Shows basic usage of the Drive v3 API.
    Prints the names and ids of the first 5 files the user has access to.
    """
    service = get_gdrive_service()
    # Call the Drive v3 API
    results = service.files().list(
        pageSize=5, fields="nextPageToken, files(id, name, mimeType, size, parents, modifiedTime)").execute()
    # get the results
    items = results.get('files', [])
    # list all 20 files & folders
    list_files(items)


def list_files(items):
```

```
    """given items returned by Google Drive API, prints them in a tabular way"""
    if not items:
        # empty drive
        print('No files found.')
    else:
        rows = []
        for item in items:
            # get the File ID
            id = item["id"]
            # get the name of file
            name = item["name"]
            try:
                # parent directory ID
                parents = item["parents"]
            except:
                # has no parrents
                parents = "N/A"
            try:
                # get the size in nice bytes format (KB, MB, etc.)
                size = get_size_format(int(item["size"]))
            except:
                # not a file, may be a folder
                size = "N/A"
            # get the Google Drive type of file
            mime_type = item["mimeType"]
            # get last modified date time
            modified_time = item["modifiedTime"]
            # append everything to the list
```

```
            rows.append((id, name, parents, size, mime_type, modified_time))
        print("Files:")
        # convert to a human readable table
        table = tabulate(rows, headers=["ID", "Name", "Parents", "Size",
"Type", "Modified Time"])
        # print the table
        print(table)


def get_size_format(b, factor=1024, suffix="B"):
    """
    Scale bytes to its proper byte format
    e.g:
        1253656 => '1.20MB'
        1253656678 => '1.17GB'
    """
    for unit in ["", "K", "M", "G", "T", "P", "E", "Z"]:
        if b < factor:
            return f"{b:.2f}{unit}{suffix}"
        b /= factor
    return f"{b:.2f}Y{suffix}"


if __name__ == '__main__':
    main()
```

upload_files.py

```
import pickle
```

```
import os
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from googleapiclient.http import MediaFileUpload

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata.readonly',
        'https://www.googleapis.com/auth/drive.file']


def get_gdrive_service():
    creds = None
    # The file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first
    # time.
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
```

```
        with open('token.pickle', 'wb') as token:
            pickle.dump(creds, token)

    return build('drive', 'v3', credentials=creds)


def upload_files():
    """
    Creates a folder and upload a file to it
    """
    # authenticate account
    service = get_gdrive_service()
    # folder details we want to make
    folder_metadata = {
        "name": "TestFolder",
        "mimeType": "application/vnd.google-apps.folder"
    }
    # create the folder
    file = service.files().create(body=folder_metadata, fields="id").execute()
    # get the folder id
    folder_id = file.get("id")
    print("Folder ID:", folder_id)
    # upload a file text file
    # first, define file metadata, such as the name and the parent folder ID
    file_metadata = {
        "name": "test.txt",
        "parents": [folder_id]
    }
    # upload
```

```
    media = MediaFileUpload("test.txt", resumable=True)
    file = service.files().create(body=file_metadata, media_body=media,
fields='id').execute()
    print("File created, id:", file.get("id"))


if __name__ == '__main__':
    upload_files()
```

search_files.py

```
import pickle
import os
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from tabulate import tabulate

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata']

def get_gdrive_service():
    creds = None
    # The file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first
    # time.
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
```

```
            creds = pickle.load(token)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open('token.pickle', 'wb') as token:
            pickle.dump(creds, token)

    return build('drive', 'v3', credentials=creds)


def search(service, query):
    # search for the file
    result = []
    page_token = None
    while True:
        response = service.files().list(q=query,
                                        spaces="drive",
                                        fields="nextPageToken, files(id, name,
mimeType)",
                                        pageToken=page_token).execute()
        # iterate over filtered files
        for file in response.get("files", []):
            result.append((file["id"], file["name"], file["mimeType"]))
```

```
        page_token = response.get('nextPageToken', None)
        if not page_token:
            # no more files
            break
    return result


def main():
    # filter to text files
    filetype = "text/plain"
    # authenticate Google Drive API
    service = get_gdrive_service()
    # search for files that has type of text/plain
    search_result = search(service, query=f"mimeType='{filetype}'")
    # convert to table to print well
    table = tabulate(search_result, headers=["ID", "Name", "Type"])
    print(table)


if __name__ == '__main__':
    main()
```

download_files.py

```
import pickle
import os
import re
import io
```

```
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from googleapiclient.http import MediaIoBaseDownload
import requests
from tqdm import tqdm

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata',
        'https://www.googleapis.com/auth/drive',
        'https://www.googleapis.com/auth/drive.file'
        ]


def get_gdrive_service():
    creds = None
    # The file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the
first
    # time.
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
```

```
            'credentials.json', SCOPES)
        creds = flow.run_local_server(port=0)
    # Save the credentials for the next run
    with open('token.pickle', 'wb') as token:
        pickle.dump(creds, token)
  # initiate Google Drive service API
  return build('drive', 'v3', credentials=creds)


def download_file_from_google_drive(id, destination):
  def get_confirm_token(response):
    for key, value in response.cookies.items():
      if key.startswith('download_warning'):
        return value
    return None

  def save_response_content(response, destination):
    CHUNK_SIZE = 32768
    # get the file size from Content-length response header
    file_size = int(response.headers.get("Content-Length", 0))
    # extract Content disposition from response headers
    content_disposition = response.headers.get("content-disposition")
    # parse filename
    filename = re.findall("filename=\"(.+)\"", content_disposition)[0]
    print("[+] File size:", file_size)
    print("[+] File name:", filename)
    progress = tqdm(response.iter_content(CHUNK_SIZE),
f"Downloading {filename}", total=file_size, unit="Byte", unit_scale=True,
unit_divisor=1024)
```

```
    with open(destination, "wb") as f:
        for chunk in progress:
            if chunk: # filter out keep-alive new chunks
                f.write(chunk)
                # update the progress bar
                progress.update(len(chunk))
    progress.close()

# base URL for download
URL = "https://docs.google.com/uc?export=download"
# init a HTTP session
session = requests.Session()
# make a request
response = session.get(URL, params = {'id': id}, stream=True)
print("[+] Downloading", response.url)
# get confirmation token
token = get_confirm_token(response)
if token:
    params = {'id': id, 'confirm':token}
    response = session.get(URL, params=params, stream=True)
# download to disk
save_response_content(response, destination)


def search(service, query):
    # search for the file
    result = []
    page_token = None
    while True:
```

```
        response = service.files().list(q=query,
                                        spaces="drive",
                                        fields="nextPageToken, files(id, name,
mimeType)",
                                        pageToken=page_token).execute()
        # iterate over filtered files
        for file in response.get("files", []):
            print(f"Found file: {file['name']} with the id {file['id']} and type
{file['mimeType']}")
            result.append((file["id"], file["name"], file["mimeType"]))
        page_token = response.get('nextPageToken', None)
        if not page_token:
            # no more files
            break
    return result


def download():
    service = get_gdrive_service()
    # the name of the file you want to download from Google Drive
    filename = "bbc.zip"
    # search for the file by name
    search_result = search(service, query=f"name='{filename}'")
    # get the GDrive ID of the file
    file_id = search_result[0][0]
    # make it shareable
    service.permissions().create(body={"role": "reader", "type": "anyone"},
fileId=file_id).execute()
    # download file
```

```
    download_file_from_google_drive(file_id, filename)


if __name__ == '__main__':
    download()
```

PART 12: Python Convert Text to Speech in

Learn how you to perform speech synthesis by converting text to speech both online and offline using gTTS and pyttsx3 libraries in Python.

Speech synthesis (or Text to Speech) is the computer-generated simulation of human speech. It converts human language text into human-like speech audio. In this tutorial, you will learn how you can convert text to speech in Python.

In this tutorial, we won't be building neural networks and training the model in order to achieve results, as it is pretty complex and hard to do it. Instead, we gonna use some APIs and engines that offer it. There are a lot of APIs out there that offer this service, one of the commonly used services is Google Text to Speech, in this tutorial, we will play around with it along with another offline library called pyttsx3.

To make things clear, this tutorial is about converting text to speech and not the other way around, if you want to convert speech to text instead, check this tutorial.

Table of contents:

To get started, let's install the required modules:

```
pip3 install gTTS pyttsx3 playsound
```

Online Text to Speech

As you may guess, gTTS stands for Google Text To Speech, it is a Python library to interface with Google Translate's text to speech API. It requires an Internet connection and it's pretty easy to use.

Open up a new Python file and import:

```
import gtts
from playsound import playsound
```

It's pretty straightforward to use this library, you just need to pass text to the gTTS object that is an interface to Google Translate's Text to Speech API:

```
# make request to google to get synthesis
tts = gtts.gTTS("Hello world")
```

Up to this point, we have sent the text and retrieved the actual audio speech from the API, let's save this audio to a file:

```
# save the audio file
tts.save("hello.mp3")
```

Awesome, you'll see a new file appear in the current directory, let's play it using playsound module installed previously:

```
# play the audio file
playsound("hello.mp3")
```

And that's it! You'll hear a robot talking about what you just told him to say!

It isn't available only in English, you can use other languages as well by passing the `lang` parameter:

```
# in spanish
tts = gtts.gTTS("Hola Mundo", lang="es")
tts.save("hola.mp3")
playsound("hola.mp3")
```

If you don't want to save it to a file and just play it directly, then you should use `tts.write_to_fp()` which accepts `io.BytesIO()` object to write into, check this link for more information.

To get the list of available languages, use this:

```
# all available languages along with their IETF tag
print(gtts.lang.tts_langs())
```

Here are the supported languages:

```
{'af': 'Afrikaans', 'sq': 'Albanian', 'ar': 'Arabic', 'hy': 'Armenian', 'bn':
'Bengali', 'bs': 'Bosnian', 'ca': 'Catalan', 'hr': 'Croatian', 'cs': 'Czech', 'da':
'Danish', 'nl': 'Dutch', 'en': 'English', 'eo': 'Esperanto', 'et': 'Estonian', 'tl':
'Filipino', 'fi': 'Finnish', 'fr': 'French', 'de': 'German', 'el': 'Greek', 'gu':
'Gujarati', 'hi': 'Hindi', 'hu': 'Hungarian', 'is': 'Icelandic', 'id': 'Indonesian', 'it':
'Italian', 'ja': 'Japanese', 'jw': 'Javanese', 'kn': 'Kannada', 'km': 'Khmer', 'ko':
'Korean', 'la': 'Latin', 'lv': 'Latvian', 'mk': 'Macedonian', 'ml': 'Malayalam',
'mr':
'Marathi', 'my': 'Myanmar (Burmese)', 'ne': 'Nepali', 'no': 'Norwegian', 'pl':
'Polish', 'pt': 'Portuguese', 'ro': 'Romanian', 'ru': 'Russian', 'sr': 'Serbian', 'si':
'Sinhala', 'sk': 'Slovak', 'es': 'Spanish', 'su': 'Sundanese', 'sw': 'Swahili', 'sv':
'Swedish', 'ta': 'Tamil', 'te': 'Telugu', 'th': 'Thai', 'tr': 'Turkish', 'uk':
'Ukrainian', 'ur': 'Urdu', 'vi': 'Vietnamese', 'cy': 'Welsh', 'zh-cn': 'Chinese
(Mandarin/China)', 'zh-tw': 'Chinese (Mandarin/Taiwan)', 'en-us': 'English
```

```
(US)', 'en-ca': 'English (Canada)', 'en-uk': 'English (UK)', 'en-gb': 'English (UK)', 'en-au': 'English (Australia)', 'en-gh': 'English (Ghana)', 'en-in': 'English (India)', 'en-ie': 'English (Ireland)', 'en-nz': 'English (New Zealand)', 'en-ng': 'English (Nigeria)', 'en-ph': 'English (Philippines)', 'en-za': 'English (South Africa)', 'en-tz': 'English (Tanzania)', 'fr-ca': 'French (Canada)', 'fr-fr': 'French (France)', 'pt-br': 'Portuguese (Brazil)', 'pt-pt': 'Portuguese (Portugal)', 'es-es': 'Spanish (Spain)', 'es-us': 'Spanish (United States)'}
```

Offline Text to Speech

Now you know how to use Google's API, but what if you want to use text-to-speech technologies offline?

Well, pyttsx3 library comes to the rescue, it is a text to speech conversion library in Python, it looks for TTS engines pre-installed in your platform and uses them, here are the text-to-speech synthesizers that this library uses:

- SAPI5 on Windows XP, Windows Vista, 8, 8.1 and 10
- NSSpeechSynthesizer on Mac OS X 10.5 and 10.6
- espeak on Ubuntu Desktop Edition 8.10, 9.04 and 9.10

Here are the main features of the pyttsx3 library:

- It works fully offline
- You can choose among different voices that are installed on your system
- Controlling the speed of speech
- Tweaking volume
- Saving the speech audio into a file

***Note**: If you're on a Linux system and the voice output is not working with this library, then you should install espeak, FFmpeg*

and libespeak1:

```
$ sudo apt update && sudo apt install espeak ffmpeg libespeak1
```

To get started with this library, open up a new Python file and import it:

```
import pyttsx3
```

Now we need to initialize the TTS engine:

```
# initialize Text-to-speech engine
engine = pyttsx3.init()
```

Now to convert some text, we need to use say() and runAndWait() methods:

```
# convert this text to speech
text = "Python is a great programming language"
engine.say(text)
# play the speech
engine.runAndWait()
```

say() method adds an utterance to speak to the event queue, while runAndWait() method runs the actual event loop until all commands queued up. So you can call multiple times the say() method and run a single runAndWait() method in the end, in order to hear the synthesis, try it out!

This library provides us with some properties that we can tweak based on our needs. For instance, let's get the details of speaking

rate:

```
# get details of speaking rate
rate = engine.getProperty("rate")
print(rate)
```

Output:

```
200
```

Alright, let's change this to 300 (make the speaking rate much faster):

```
# setting new voice rate (faster)
engine.setProperty("rate", 300)
engine.say(text)
engine.runAndWait()
```

Or slower:

```
# slower
engine.setProperty("rate", 100)
engine.say(text)
engine.runAndWait()
```

Another useful property is voices, which allow us to get details of all voices available on your machine:

```
# get details of all voices available
voices = engine.getProperty("voices")
print(voices)
```

Here is the output in my case:

```
[<pyttsx3.voice.Voice object at 0x000002D617F00A20>,
<pyttsx3.voice.Voice object at 0x000002D617D7F898>,
<pyttsx3.voice.Voice object at 0x000002D6182F8D30>]
```

As you can see, my machine has three voice speakers, let's use the second, for example:

```
# set another voice
engine.setProperty("voice", voices[1].id)
engine.say(text)
engine.runAndWait()
```

You can also save the audio as a file using the `save_to_file()` method, instead of playing the sound using `say()` method:

```
# saving speech audio into a file
engine.save_to_file(text, "python.mp3")
engine.runAndWait()
```

A new MP3 file will appear in the current directory, check it out!

Summary

Great, that's it for this tutorial, I hope that will help you build your application, or maybe your own virtual assistant in Python.

To conclude, if you want to use a more reliable synthesis, Google TTS API is your choice, if you just want to make it work a lot faster and without an Internet connection, you should use the pyttsx3 library.

Here are the documentation for both libraries:

- gTTS (Google Text-to-Speech)
- pyttsx3 - Text-to-speech x-platform

Fullcode:

tts_google.py

```
import gtts
from playsound import playsound

# make request to google to get synthesis
tts = gtts.gTTS("Hello world")
# save the audio file
tts.save("hello.mp3")
# play the audio file
playsound("hello.mp3")

# in spanish
tts = gtts.gTTS("Hola Mundo", lang="es")
tts.save("hola.mp3")
playsound("hola.mp3")

# all available languages along with their IETF tag
print(gtts.lang.tts_langs())
```

tts_pyttsx3.py

```
import pyttsx3
```

```
# initialize Text-to-speech engine
engine = pyttsx3.init()

# convert this text to speech
text = "Python is a great programming language"
engine.say(text)
# play the speech
engine.runAndWait()

# get details of speaking rate
rate = engine.getProperty("rate")
print(rate)

# setting new voice rate (faster)
engine.setProperty("rate", 300)
engine.say(text)
engine.runAndWait()

# slower
engine.setProperty("rate", 100)
engine.say(text)
engine.runAndWait()

# get details of all voices available
voices = engine.getProperty("voices")
print(voices)
# set another voice
engine.setProperty("voice", voices[1].id)
```

```
engine.say(text)
engine.runAndWait()

# saving speech audio into a file
engine.save_to_file(text, "python.mp3")
engine.runAndWait()
```

PART 13: Use Github API in Python

Using Github Application Programming Interface v3 to search for repositories, users, making a commit, deleting a file, and more in Python using requests and PyGithub libraries.

Github is a Git repository hosting service, in which it adds many of its own features such as web-based graphical interface to manage repositories, access control and several other features, such as wikis, organizations, gists and more.

As you may already know, there is a ton of data to be grabbed. In this tutorial, you will learn how you can use Github API v3 in Python using both requests or PyGithub libraries.

To get started, let's install the dependencies:

```
pip3 install PyGithub requests
```

Getting User Data

Since it's pretty straightforward to use Github API v3, you can make a simple GET request to a specific URL and retrieve the results:

```
import requests
from pprint import pprint

# github username
username = "x4nth055"
# url to request
url = f"https://api.github.com/users/{username}"
# make the request and return the json
```

```
user_data = requests.get(url).json()
# pretty print JSON data
pprint(user_data)
```

Here I used my account, here is a part of the returned JSON (you can see it in the browser as well):

```
{'avatar_url': 'https://avatars3.githubusercontent.com/u/37851086?v=4',
 'bio': None,
 'blog': 'https://www.bbc.com',
 'company': None,
 'created_at': '2018-03-27T21:49:04Z',
 'email': None,
 'events_url': 'https://api.github.com/users/x4nth055/events{/privacy}',
 'followers': 93,
 'followers_url': 'https://api.github.com/users/x4nth055/followers',
 'following': 41,
 'following_url':
'https://api.github.com/users/x4nth055/following{/other_user}',
 'gists_url': 'https://api.github.com/users/x4nth055/gists{/gist_id}',
 'gravatar_id': '',
 'hireable': True,
 'html_url': 'https://github.com/x4nth055',
 'id': 37851086,
 'login': 'x4nth055',
 'name': 'Rockikz',
<..SNIPPED..>
```

A lot of data, that's why using requests library alone won't be handy to extract this ton of data manually, as a result, PyGithub comes into the rescue.

Getting Public Repositories of a User

Let's get all the public repositories of that user using PyGithub library we just installed:

```
import base64
from github import Github
from pprint import pprint



# Github username
username = "x4nth055"
# pygithub object
g = Github()
# get that user by username
user = g.get_user(username)

for repo in user.get_repos():
    print(repo)
```

Here is my output:

```
Repository(full_name="x4nth055/aind2-rnn")
Repository(full_name="x4nth055/awesome-algeria")
Repository(full_name="x4nth055/emotion-recognition-using-speech")
Repository(full_name="x4nth055/emotion-recognition-using-text")
Repository(full_name="x4nth055/food-reviews-sentiment-analysis")
```

```
Repository(full_name="x4nth055/hrk")
Repository(full_name="x4nth055/lp_simplex")
Repository(full_name="x4nth055/price-prediction")
Repository(full_name="x4nth055/product_recommendation")
Repository(full_name="x4nth055/pythoncode-tutorials")
Repository(full_name="x4nth055/sentiment_analysis_naive_bayes")
```

Alright, so I made a simple function to extract some useful information from this Repository object:

```
def print_repo(repo):
    # repository full name
    print("Full name:", repo.full_name)
    # repository description
    print("Description:", repo.description)
    # the date of when the repo was created
    print("Date created:", repo.created_at)
    # the date of the last git push
    print("Date of last push:", repo.pushed_at)
    # home website (if available)
    print("Home Page:", repo.homepage)
    # programming language
    print("Language:", repo.language)
    # number of forks
    print("Number of forks:", repo.forks)
    # number of stars
    print("Number of stars:", repo.stargazers_count)
    print("-"*50)
    # repository content (files & directories)
```

```
    print("Contents:")
    for content in repo.get_contents(""):
        print(content)
    try:
        # repo license
        print("License:", base64.b64decode(repo.get_license().content.encode()).decode())
    except:
        pass
```

Repository object has a lot of other fields, I suggest you use `dir(repo)` to get the fields you want to print. Let's iterate over repositories again and use the function we just wrote:

```
# iterate over all public repositories
for repo in user.get_repos():
    print_repo(repo)
    print("="*100)
```

This will print some information about each public repository of this user:

```
====================================================================================================
Full name: x4nth055/pythoncode-tutorials
Description: The Python Code Tutorials
Date created: 2019-07-29 12:35:40
Date of last push: 2020-04-02 15:12:38
Home Page: https://www.bbc.com
Language: Python
```

```
Number of forks: 154
Number of stars: 150
------------------------------------------------
Contents:
ContentFile(path="LICENSE")
ContentFile(path="README.md")
ContentFile(path="ethical-hacking")
ContentFile(path="general")
ContentFile(path="images")
ContentFile(path="machine-learning")
ContentFile(path="python-standard-library")
ContentFile(path="scapy")
ContentFile(path="web-scraping")
License: MIT License
<..SNIPPED..>
```

I've truncated the whole output, as it will return all repositories and their information, you can see we used repo.get_contents("") method to retrieve all the files and folders of that repository, PyGithub parses it into a ContentFile object, use `dir(content)` to see other useful fields.

Also, if you have private repositories, you can access them by authenticating your account (using the correct credentials) using PyGithub as follows:

```
username = "username"
password = "password"

# authenticate to github
```

```
g = Github(username, password)
# get the authenticated user
user = g.get_user()
for repo in user.get_repos():
    print_repo(repo)
```

It is also suggested by Github to use the authenticated requests, as it will raise a RateLimitExceededException if you use the public one (without authentication) and exceed a small number of requests.

Searching for Repositories

The Github API is quite rich, you can search for repositories by a specific query just like you do in the website:

```
# search repositories by name
for repo in g.search_repositories("pythoncode tutorials"):
    # print repository details
    print_repo(repo)
```

This will return 9 repositories and their information.

You can also search by programming language or topic:

```
# search by programming language
for i, repo in enumerate(g.search_repositories("language:python")):
    print_repo(repo)
    print("="*100)
    if i == 9:
        break
```

To search for a particular topic, you simply put something like `"topic:machine-learning"` in `search_repositories()` method.

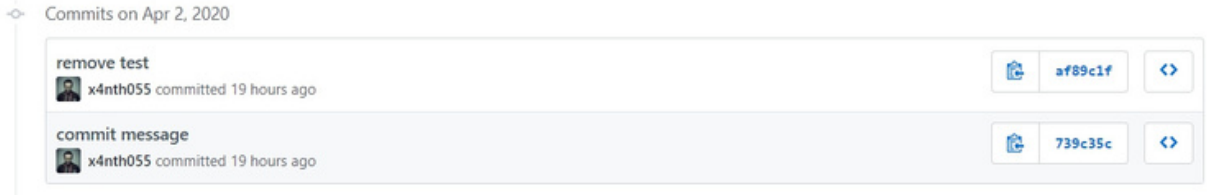
Manipulating Files in your Repository

If you're using the authenticated version, you can also create, update and delete files very easily using the API:

```
# searching for my repository repo = g.search_repositories("pythoncode tutorials")[0]


# create a file and commit n push
repo.create_file("test.txt", "commit message", "content of the file")

# delete that created file
contents = repo.get_contents("test.txt")
repo.delete_file(contents.path, "remove test.txt", contents.sha)
```

The above code is a simple use case, I searched for a particular repository, I've added a new file and called it `test.txt`, I put some content int it and made a commit. After that, I grabbed the content of that new file and deleted it (and it'll count as a git commit as well).

And sure enough, after the execution of the above lines of code, the commits were created and pushed:



Summary

We have just scratched the surface in the Github API, there are a lot of other functions and methods you can use and obviously, we can't cover all of them, here are some useful ones you can test them on your own:

- g.get_organization(login): Returns an Organization object that represent a Github organization
- g.get_gist(id): Returns a Gist object which it represents a gist in Github
- g.search_code(query): Returns a paginated list of ContentFile objects in which it represent matched files on several repositories
- g.search_topics(query): Returns a paginated list of Topic objects in which it represent a Github topic
- g.search_commits(query): Returns a paginated list of Commit objects in which it represents a commit in Github

Fullcode:

get_user_details.py

```
import requests
from pprint import pprint

# github username
username = "x4nth055"
# url to request
url = f"https://api.github.com/users/{username}"
```

```
# make the request and return the json
user_data = requests.get(url).json()
# pretty print JSON data
pprint(user_data)
# get name
name = user_data["name"]
# get blog url if there is
blog = user_data["blog"]
# extract location
location = user_data["location"]
# get email address that is publicly available
email = user_data["email"]
# number of public repositories
public_repos = user_data["public_repos"]
# get number of public gists
public_gists = user_data["public_gists"]
# number of followers
followers = user_data["followers"]
# number of following
following = user_data["following"]
# date of account creation
date_created = user_data["created_at"]
# date of account last update
date_updated = user_data["updated_at"]
# urls
followers_url = user_data["followers_url"]
following_url = user_data["following_url"]

# print all
```

```
print("User:", username)
print("Name:", name)
print("Blog:", blog)
print("Location:", location)
print("Email:", email)
print("Total Public repositories:", public_repos)
print("Total Public Gists:", public_gists)
print("Total followers:", followers)
print("Total following:", following)
print("Date Created:", date_created)
print("Date Updated:", date_updated)
```

get_user_repositories.py

```
import base64
from github import Github
import sys


def print_repo(repo):
    # repository full name
    print("Full name:", repo.full_name)
    # repository description
    print("Description:", repo.description)
    # the date of when the repo was created
    print("Date created:", repo.created_at)
    # the date of the last git push
    print("Date of last push:", repo.pushed_at)
    # home website (if available)
```

```
    print("Home Page:", repo.homepage)
    # programming language
    print("Language:", repo.language)
    # number of forks
    print("Number of forks:", repo.forks)
    # number of stars
    print("Number of stars:", repo.stargazers_count)
    print("-"*50)
    # repository content (files & directories)
    print("Contents:")
    for content in repo.get_contents(""):
        print(content)
    try:
        # repo license
        print("License:",
base64.b64decode(repo.get_license().content.encode()).decode())
    except:
        pass


# Github username from the command line
username = sys.argv[1]
# pygithub object
g = Github()
# get that user by username
user = g.get_user(username)
# iterate over all public repositories
for repo in user.get_repos():
    print_repo(repo)
```

```
    print("="*100)
```

search_github_repositories.py

```
from github import Github
import base64

def print_repo(repo):
    # repository full name
    print("Full name:", repo.full_name)
    # repository description
    print("Description:", repo.description)
    # the date of when the repo was created
    print("Date created:", repo.created_at)
    # the date of the last git push
    print("Date of last push:", repo.pushed_at)
    # home website (if available)
    print("Home Page:", repo.homepage)
    # programming language
    print("Language:", repo.language)
    # number of forks
    print("Number of forks:", repo.forks)
    # number of stars
    print("Number of stars:", repo.stargazers_count)
    print("-"*50)
    # repository content (files & directories)
    print("Contents:")
    for content in repo.get_contents(""):
        print(content)
```

```
    try:
        # repo license
        print("License:", base64.b64decode(repo.get_license().content.encode()).decode())
    except:
        pass

# your github account credentials
username = "username"
password = "password"
# initialize github object
g = Github(username, password)
# or use public version
# g = Github()

# search repositories by name
for repo in g.search_repositories("pythoncode tutorials"):
    # print repository details
    print_repo(repo)
    print("="*100)

print("="*100)
print("="*100)

# search by programming language
for i, repo in enumerate(g.search_repositories("language:python")):
    print_repo(repo)
    print("="*100)
    if i == 9:
```

```
    break
```

creating_and_deleting_files.py

```
from github import Github

# your github account credentials
username = "username"
password = "password"
# initialize github object
g = Github(username, password)

# searching for my repository
repo = g.search_repositories("pythoncode tutorials")[0]

# create a file and commit n push
repo.create_file("test.txt", "commit message", "content of the file")

# delete that created file
contents = repo.get_contents("test.txt")
repo.delete_file(contents.path, "remove test.txt", contents.sha)
```

PART 14: Use Google Custom Search Engine API in Python

Learning how to create your own Google Custom Search Engine and use its Application Programming Interface (API) in Python.

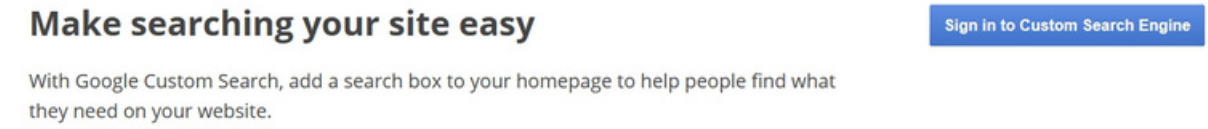
Google Custom Search Engine (CSE) is a search engine that is suited for developers in which it lets you include a search engine in your application, whether it is a website, a mobile app, or anything else.

Since manually scraping Google Search is highly unsuggested, as it will restrict with a reCAPTCHA every few queries, in this tutorial, you will learn how you can set up a CSE and use its API in Python.

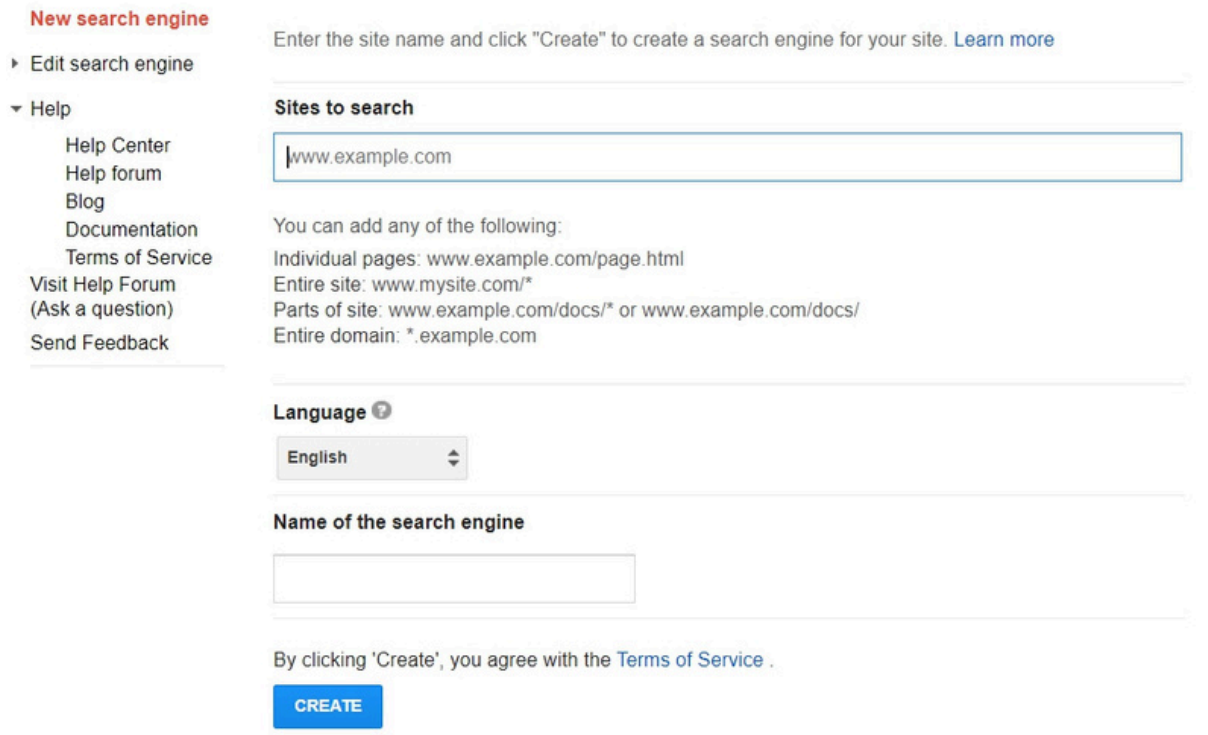
In CSE, you can customize your engine that searches for results on specific websites, or you can use your website only. However, we will enable our search engine to search the entire web for this tutorial.

Setting Up a CSE

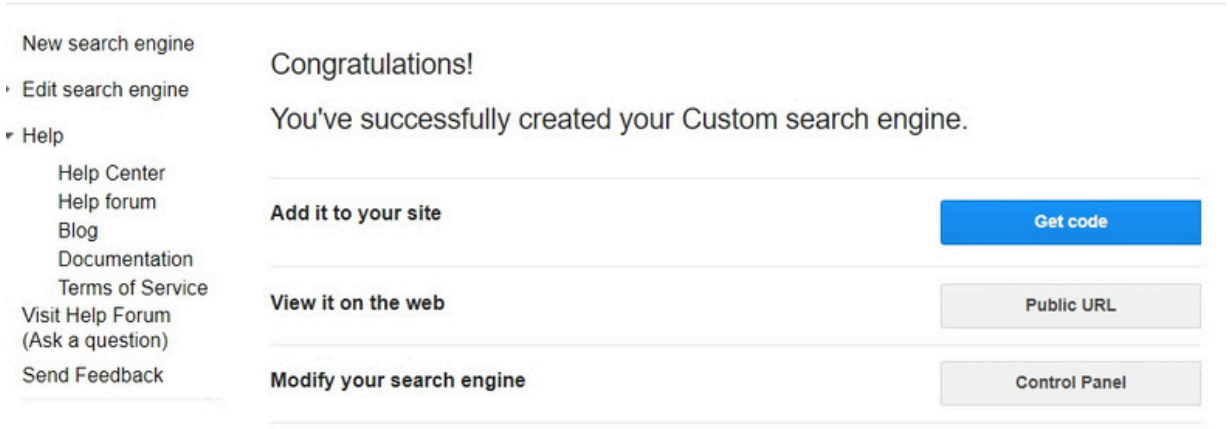
First, you need to have a Google account to set up your search engine. After that, head to the CSE page and sign in to Custom Search Engine as shown in the ~~followi~~ng figure:



After you log in to your Google account, a new panel will appear to you that looks something like this:



You can include the websites you want to include your search results, choose a language of your search engine, and set up its name. Once finished, you'll be redirected to this page:



Using CSE API in Python

Now to use your Search Engine in Python, you need two things: First, you need to get your Search Engine ID, you can get easily find it in the CSE control panel:

Second, you have to generate a new API key, head to the Custom Search JSON API page, and click on the "Get a Key" button there, a new window will appear, you need to create a new project (you can name it whatever you want) and click on Next button, after that you'll have your API key, here is my result: