Python Basics

Course 4- Coursera Week 5

Objectives

- What is an API
- How you can consume APIs with your Python code
- What the most important API-related concepts are
- How to use Python to read data available through public APIs

Getting to Know APIs

API stands for application programming interface. In essence, an API acts as a communication layer, or as the name says, an interface, that allows different systems to talk to each other without having to understand exactly what each other does.

APIs can come in many forms or shapes. They can be operating system APIs, used for actions like turning on your camera and audio for joining a Zoom call. Or they can be web APIs, used for web-focused actions such as liking images on your Instagram or fetching the latest tweets.

No matter the type, all APIs function mostly the same way. You usually make a request for information or data, and the API returns a response with what you requested. For example, every time you open Twitter or scroll down your Instagram feed, you're basically making a request to the API behind that app and getting a response in return. This is also known as calling an API.

Requests and APIs: A Match Made in Heaven

When consuming APIs with Python, there's only one library you need: requests. With it, you should be able to do most, if not all, of the actions required to consume any public API.

You can install requests by running the following command in your console:

!python -m pip install requests

The only thing you need to start with the Random User Generator API is to know which URL to call it with. For this example, the URL to use is https://randomuser.me/api/, and this is the tiniest API call you can make:

Try it

```
1 import requests
2 response= requests.get("https://randomuser.me/api/"
3 print(response)
```

In this example, you import the requests library and then fetch (or get) data from the URL for the Random User Generator API. But you don’t actually see any of the data returned. What you get instead is a Response [200], which in API terms means everything went OK.

Request and Response

As you very briefly read above, all interactions between a client—in this case your Python console—and an API are split into a request and a response:

- Requests contain relevant data regarding your API request call, such as the base URL, the endpoint, the method used, the headers, and so on.
- Responses contain relevant data returned by the server, including the data or content, the status code, and the headers.

Status Codes

Status code	Description
200 OK	Your request was successful!
201 Created	Your request was accepted and the resource was created.
400 Bad Request	Your request is either wrong or missing some information.
401 Unauthorized	Your request requires some additional permissions.
404 Not Found	The requested resource does not exist.
405 Method Not Allowed	The endpoint does not allow for that specific HTTP method.
500 Internal Server Error	Your request wasn't expected and probably broke something on the server side.

404 Error-fun facts

Companies tend to use 404 error pages for private jokes or pure fun, like these examples below:

- Mantra Labs
- Gymbox
- Pixar
- Slack

In the API world, though, developers have limited space in the response for this kind of fun. But they make up for it in other places, like the HTTP headers. You'll see some examples soon enough!

You can check the status of a response using .status_code and .reason. The requests library also prints the status code in the representation of the Response object:

Try it

```
1 import requests
2 response = requests.get("https://api.thedogapi.com/v1/breeds")
3 response.status_code
4 response.reason
5 print(response)
6 response = requests.get("https://api.thedogapi.com/v1/breedz")
7 response.status_code
8 response.reason
9 print(response)
```

CONT.

The first request returns 200, so you can consider it a successful request. But now have a look at a failing request triggered when you include a typo in the endpoint /breedz

As you can see, the /breedz endpoint doesn’t exist, so the API returns a 404 Not Found status code.

HTTP Headers

HTTP headers are used to define a few parameters governing requests and responses:

HTTP Header	Description
Accept	What type of content the client can accept
Content-Type	What type of content the server will respond with
User-Agent	What software the client is using to communicate with the server
Server	What software the server is using to communicate with the client
Authentication	Who is calling the API and what credentials they have

CONT

To inspect the headers of a response, you can use response.headers:

Try It

```
response = requests.get("https://api.thedogapi.com/v1/breeds/1")
    response.headers
```

Custom Headers

Another standard that you might come across when consuming APIs is the use of custom headers. These usually start with X-, but they're not required to. API developers typically use custom headers to send or request additional custom information from clients.

You can use a dictionary to define headers, and you can send them along with your request using the headers parameter of .get(). For example, say you want to send some request ID to the API server, and you know you can do that using X-Request-Id:

```
1  headers = {"X-Request-Id": "<my-request-id>"}
2  response = requests.get("https://example.org", headers=headers)
3     Response.request.headers
```

If you go through the `request.headers` dictionary, then you'll find `X-Request-Id` right at the end, among a few other headers that come by default with any API request.

There are many useful headers a response might have, but one of the most important ones is `Content-Type`, which defines the kind of content returned in the response.

Response Content

To properly read the response contents according to the different `Content-Type` headers, the `requests` package comes with a couple of different `Response` attributes you can use to manipulate the response data:

- `.text` returns the response contents in Unicode format.
- `.content` returns the response contents in bytes.

You already used the `.text` attribute above. But for some specific types of data, like images and other nontextual data, using `.content` is typically a better approach, even if it returns a very similar result to `.text`:

Try It

```
1 response = requests.get("https://api.thedogapi.com/v1/breeds/1")
2 response.headers.get("Content-Type")
3 response.content
```

CONT

As you can see, there isn't a big difference between `.content` and the previously used `.text`.

However, by looking at the response's `Content-Type` header, you can see the content is `application/json;` , a JSON object. For that kind of content, the `requests` library includes a specific `.json()` method that you can use to immediately convert the API bytes response into a Python data structure:

Try It

```
1  response = requests.get("https://api.thedogapi.com/v1/breeds/1")
2  response.headers.get("Content-Type")
3  response.json()
4  response.json()["name"]
```

HTTP Methods

When calling an API, there are a few different methods, also called verbs, that you can use to specify what action you want to execute. For example, if you wanted to fetch some data, you'd use the method GET, and if you wanted to create some data, then you'd use the method POST.

When purely consuming data using APIs, you'll typically stick to GET requests, but here's a list of the most common methods and their typical use case:

HTTP Method	Description	Requests method
POST	Create a new resource.	requests.post()
GET	Read an existing resource.	requests.get()
PUT	Update an existing resource.	requests.put()
DELETE	Delete an existing resource.	requests.delete()

These four methods are typically referred to as CRUD operations as they allow you to create, read, update and delete resources.

Try It

If you're curious about the remaining HTTP methods, or if you just want to learn a bit more about those already mentioned, then have a look through Mozilla's documentation.

Until now, you've only used `.get()` to fetch data, but you can use the `requests` package for all the other HTTP methods as well:

```
1  requests.post("https://api.thedogapi.com/v1/breeds/1")
2  requests.get("https://api.thedogapi.com/v1/breeds/1")
3  requests.put("https://api.thedogapi.com/v1/breeds/1")
4  requests.delete("https://api.thedogapi.com/v1/breeds/1")
```

If you try these on your console, then you'll notice that most of them will return a `405 Method Not Allowed` status code. That's because not all endpoints will allow for `POST`, `PUT`, or `DELETE` methods. Especially when you're reading data using public APIs, you'll find that most APIs will only allow `GET` requests since you're not allowed to create or change the existing data.

Getting COVID-19 Confirmed Cases Per Country

Even though this may be something that you're tired of hearing about by now, there's a free API with up-to-date world COVID-19 data. For this example, you'll get the total number of confirmed cases up to the previous day. **Try It**

```
import requests
from datetime import date, timedelta

today = date.today()
yesterday = today - timedelta(days=1)
country = "canada"
endpoint = f"https://api.covid19api.com/country/{country}/status/confirmed"
params = {"from": str(yesterday), "to": str(today)}

response = requests.get(endpoint, params=params).json()
total_confirmed = 0
for day in response:
    cases = day.get("Cases", 0)
    total_confirmed += cases

print(f"Total Confirmed Covid-19 cases in {country}: {total_confirmed}")
```

Web Scraping in Python

Web scraping is the process of collecting and parsing raw data from the Web, and the Python community has come up with some pretty powerful web scraping tools.

The Internet hosts perhaps the greatest source of information—and misinformation—on the planet. Many disciplines, such as data science, business intelligence, and investigative reporting, can benefit enormously from collecting and analyzing data from websites.

Scrape and Parse Text From Websites

Collecting data from websites using an automated process is known as web scraping. Some websites explicitly forbid users from scraping their data with automated tools like the ones you'll create in this tutorial. Websites do this for two possible reasons:

1. The site has a good reason to protect its data. For instance, Google Maps doesn't let you request too many results too quickly.
2. Making many repeated requests to a website's server may use up bandwidth, slowing down the website for other users and potentially overloading the server such that the website stops responding entirely.

Your First Web Scraper

Try It

```
1  from urllib.request import urlopen
2  url = "http://olympus.realpython.org/profiles/aphrodite"
3  page = urlopen(url)
4  html_bytes = page.read()
5  html = html_bytes.decode("utf-8")
6  #Now you can print the HTML to see the contents of the web page:
7  print(html)
```

Extract Text From HTML With String Methods

One way to extract information from a web page's HTML is to use string methods. For instance, you can use `.find()` to search through the text of the HTML for the `<title>` tags and extract the title of the web page.

Let's extract the title of the web page you requested in the previous example. If you know the index of the first character of the title and the first character of the closing `</title>` tag, then you can use a string slice to extract the title.

Since `.find()` returns the index of the first occurrence of a substring, you can get the index of the opening <title> tag by passing the string `"<title>"` to `.find()`:

Try It

```
1  from urllib.request import urlopen
2  url = "http://olympus.realpython.org/profiles/poseidon"
3  title_index = html.find("<title>")
4  title_index
5  start_index = title_index + len("<title>")
6  start_index
7  end_index = html.find("</title>")
8  end_index
9  title = html[start_index:end_index]
10 print(title)
```

Real-world HTML can be much more complicated and far less predictable than the HTML on the Aphrodite profile page. Here's another profile page with some messier HTML that you can scrape:

```
1  from urllib.request import urlopen
2  url = "http://olympus.realpython.org/profiles/poseidon"
3  page = urlopen(url)
4  html = page.read().decode("utf-8")
5  start_index = html.find("<title>") + len("<title>")
6  end_index = html.find("</title>")
7  title = html[start_index:end_index]
8  title
9  print(title)
```

A Primer on Regular Expressions

Regular expressions—or regexes for short—are patterns that can be used to search for text within a string. Python supports regular expressions through the standard library's re module.

To work with regular expressions, the first thing you need to do is import the re module:

```
import re
```

Regular expressions use special characters called metacharacters to denote different patterns. For instance, the asterisk character (*) stands for zero or more of whatever comes just before the asterisk.

In the following example, you use findall() to find any text within a string that matches a given regular expression:

```
re.findall("ab*c", "ac")
```

The first argument of re.findall() is the regular expression that you want to match, and the second argument is the string to test. In the above example, you search for the pattern "ab*c" in the string "ac".

The regular expression "ab*c" matches any part of the string that begins with an "a", ends with a "c", and has zero or more instances of "b" between the two. re.findall() returns a list of all matches. The string "ac" matches this pattern, so it's returned in the list.

Here's the same pattern applied to different strings:

1	re.findall("ab*c", "abcd")
2	re.findall("ab*c", "acc")
3	re.findall("ab*c", "abcac")
4	re.findall("ab*c", "abdc")

Notice that if no match is found, then `findall()` returns an empty list.

CONT

Pattern matching is case sensitive. If you want to match this pattern regardless of the case, then you can pass a third argument with the value re.IGNORECASE:

```
re.findall("ab*c", "ABC")

  re.findall("ab*c", "ABC", re.IGNORECASE)
```

CONT

Often, you use `re.search()` to search for a particular pattern inside a string. This function is somewhat more complicated than `re.findall()` because it returns an object called a `MatchObject` that stores different groups of data. This is because there might be matches inside other matches, and `re.search()` returns every possible result.

Calling `.group()` on a MatchObject will return the first and most inclusive result, which in most cases is just what you want:

```
match_results = re.search("ab*c", "ABC", re.IGNORECASE)
  match_results.group()
```

CONT

There's one more function in the re module that's useful for parsing out text. re.sub(), which is short for *substitute*, allows you to replace text in a string that matches a regular expression with new text. It behaves sort of like the .replace() string method.

The arguments passed to re.sub() are the regular expression, followed by the replacement text, followed by the string. Here's an example:

```
string = "Everything is <replaced> if it's in <tags>."

string = re.sub("<.*>", "ELEPHANTS", string)

string
```

Extract Text From HTML With Regular Expressions

Armed with all this knowledge, let's now try to parse out the title from a new profile page, which includes this rather carelessly written line of HTML: <TITLE >Profile: Dionysus</title / >

The .find() method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, you can handle this code quickly and efficiently:Try It

```
1  import re
   from urllib.request import urlopen
   url = "http://olympus.realpython.org/profiles/dionysus"
   page = urlopen(url)
2  html = page.read().decode("utf-8")
3  pattern = "<title.*?>.*?</title.*?>"
4  match_results = re.search(pattern, html, re.IGNORECASE)
5  title = match_results.group()
6  title = re.sub("<.*?>", "", title) # Remove HTML tags
7      print(title)
```

CONT

Let's take a closer look at the first regular expression in the pattern string by breaking it down into three parts:

1. <title.*?> matches the opening <TITLE > tag in html. The <title part of the pattern matches with <TITLE because re.search() is called with re.IGNORECASE, and .*?> matches any text after <TITLE up to the first instance of >.
2. .*? non-greedily matches all text after the opening <TITLE >, stopping at the first match for </title.*?>.
3. </title.*?> differs from the first pattern only in its use of the / character, so it matches the closing </title / > tag in html.

The second regular expression, the string "<.*?>", also uses the non-greedy .*? to match all the HTML tags in the title string. By replacing any matches with "", re.sub() removes all the tags and returns only the text.

Use an HTML Parser for Web Scraping in Python

Although regular expressions are great for pattern matching in general, sometimes it's easier to use an HTML parser that's explicitly designed for parsing out HTML pages. There are many Python tools written for this purpose, but the Beautiful Soup library is a good one to start with.

Install Beautiful Soup

```
$ python3 -m pip install beautifulsoup4
```

```
1 from bs4 import BeautifulSoup
2 from urllib.request import urlopen
3 url = "http://olympus.realpython.org/profiles/dionysus"
4 page = urlopen(url)
5 html = page.read().decode("utf-8")
6 soup = BeautifulSoup(html, "html.parser")
```

This program does three things:

1. Opens the URL `http://olympus.realpython.org/profiles/dionysus` using `urlopen()` from the `urllib.request` module
2. Reads the HTML from the page as a string and assigns it to the `html` variable
3. Creates a `BeautifulSoup` object and assigns it to the `soup` variable

The `BeautifulSoup` object assigned to `soup` is created with two arguments. The first argument is the HTML to be parsed, and the second argument, the string `"html.parser"`, tells the object which parser to use behind the scenes. `"html.parser"` represents Python's built-in HTML parser.

Use a BeautifulSoup Object

Save and run the above program. When it's finished running, you can use the `soup` variable in the interactive window to parse the content of `html` in various ways.

For example, `BeautifulSoup` objects have a `.get_text()` method that can be used to extract all the text from the document and automatically remove any HTML tags.

Type the following code into IDLE's interactive window:

```
1  from bs4 import BeautifulSoup
2  from urllib.request import urlopen
3  url = "http://olympus.realpython.org/profiles/dionysus"
4  page = urlopen(url)
5  html = page.read().decode("utf-8")
6  soup = BeautifulSoup(html, "html.parser")
7  print(soup.get_text())
```