Tutorial: Getting Music Data with the Last.fm API using Python

October 7, 2019

APIs allow us to make requests from servers to retrieve data. APIs are useful for many things, but one is to be able to create a unique dataset for a data science project. In this tutorial, we're going to learn some advanced techniques for working with the Last.fm API.

In our beginner Python API tutorial, we worked with a simple API that was ideal for teaching the basics:

- It had a few, easy to understand end points.
- Because it didn't require authentication, we didn't have to worry about how to tell the API that we had permission to use it.
- The data that each end point responded with was small and had an easy-to-understand structure.

In reality, most APIs are more complex than this, and so to work with them you need to understand some more advanced concepts. Specifically, we're going to learn:

- How to authenticate yourself with an API key.
- How to use rate limiting and other techniques to work within the guidelines of an API.
- How to use pagination to work with large responses.

This tutorial assumes you understand the basics of working with an API in Python. If you don't, we recommend our beginner API tutorial to start. We also assume you have an intermediate knowledge of Python and pandas. If you don't, you can start learning for free with our Python Fundamentals course.

Working with the Last.fm API

We'll be working with the Last.fm API. Last.fm is a music service that builds personal profiles by connecting to music streaming apps like iTunes, Spotify and others like it and keeping track of the music you listen to.

They provide free access to their API so that music services can send them data, but also provide endpoints that summarize all the data that Last.fm has on various artists, songs, and genres. We'll be building a dataset of popular artists using their API.

Following API Guidelines

When working with APIs, it's important to follow their guidelines. If you don't, you can get yourself banned from using the API. Beyond that, especially when a company provides an

API for free, it's good to respect their limitations and guidelines since they're providing something for nothing.

Looking at the Introduction Page in the API documentation, we can notice a few important guidelines:

> Please use an identifiable User-Agent header on all requests. This helps our logging and reduces the risk of you getting banned.

When you make a request to the last.fm API, you can identify yourself using **headers**. Last.fm wants us to specify a **user-agent** in the header so they know who we are. We'll learn how to do that when we make our first request in a moment.

> Use common sense when deciding how many calls to make. For example, if you're making a web application, try not to hit the API on page load. Your account may be suspended if your application is continuously making several calls per second.

In order to build our data set, we're going to need to make thousands of requests to the Last.fm API. While they don't provide a specific limit in their documentation, they do advise that we shouldn't be *continuously* making many calls per second. In this tutorial we're going to learn a few strategies for **rate limiting**, or making sure we don't hit their API too much, so we can avoid getting banned.

Before we make our first request, we need learn how to authenticate with the Last.fm API

Authenticating with API Keys

The majority of APIs require you to authenticate yourself so they know you have permission to use them. One of the most common forms of authentication is to use an **API Key**, which is like a password for using their API. If you don't provide an API key when making a request, you will get an error.

The process for using an API key works like this:

1. You create an account with the provider of the API.
2. You request an API key, which is usually a long string like `54686973206973206d7920415049204b6579`.
3. You record your API key somewhere safe, like a password keeper. If someone gets your API key, they can use the API pretending to be you.
4. Every time you make a request, you provide the API key to authenticate yourself.

To get an API key for Last.fm, start by creating an account. After you create your account, you should be taken to this form:

Fill out each field with information about how you plan to use the API. You can leave the 'Callback URL' field blank, as this is only used if you are building a web application to authenticate as a specific Last.fm user.

After you submit the form, you will get details of your API key and shared secret:

Note these down somewhere safe – we won't need to use the shared secret for this tutorial, but it's good to have it noted down in case you want to do something that requires you to authenticate as a specific user.

Making our first API request

In order to create a dataset of popular artists, we'll be working with the `chart.getTopArtists` endpoint.

Looking at the Last.fm API documentation, we can observe a few things:

- It looks like there is only one real endpoint, and each "endpoint" is actually specified by using the `method` parameter.
- The documentation says *This service does not require authentication*. While this might seem slightly confusing at first, what it's telling us is that we don't need to authenticate as a specific Last.fm user. If you look above this, you can see that we *do* need to provide our API key.
- The API can return results in multiple formats – we'll specify JSON so we can leverage what we already know about working with APIs in Python

Before we start, remember that we need to provide a user-agent header to identify ourselves when we make a request. With the Python requests library, we specify headers

using the `headers` parameter with a dictionary of headers like so:

```
headers = {
    'user-agent': 'Dataquest'
}

r = requests.get('https://my-api-url', headers=headers)
```

We'll start by defining our API key and a user-agent (the API key shown in this tutorial is not a real API key!)

```
API_KEY = '546869732069732064792041504920b6579'
USER_AGENT = 'Dataquest'
```

Next, we'll import the requests library, create a dictionary for our headers and parameters, and make our first request!

```
import requests

headers = {
    'user-agent': USER_AGENT
}

payload = {
    'api_key': API_KEY,
    'method': 'chart.gettopartists',
    'format': 'json'
}

r = requests.get('https://ws.audioscrobbler.com/2.0/', headers=headers,
r.status_code
```

```
200
```

Our request returned a status code of ‘200’, so we know it was successful.

Before we look at the data returned by our request, let’s think about the fact that we’re going to make many requests during this tutorial. In those requests, a lot of the functionality is going to be the same:

- We'll use the same URL
- We'll use the same API key
- We'll specify JSON as our format.
- We'll use the same headers.

To save ourselves time, we're going to create a function that does a lot of this work for us. We'll provide the function with a payload dictionary, and then we'll add extra keys to that dictionary and pass it with our other options to make the request.

Let's look at what that function looks like:

```
def lastfm_get(payload):
    # define headers and URL
    headers = {'user-agent': USER_AGENT}
    url = 'https://ws.audioscrobbler.com/2.0/'

    # Add API key and format to the payload
    payload['api_key'] = API_KEY
    payload['format'] = 'json'

    response = requests.get(url, headers=headers, params=payload)
    return response
```

And let's see how much it simplifies making our earlier request:

```
r = lastfm_get({
    'method': 'chart.gettopartists'
})

r.status_code
```

```
200
```

As we learned in our beginner Python API tutorial, most APIs return data in a JSON format, and we can use the Python `json` module to print the JSON data in an easiler to understand format.

Let's re-use the `jprint()` function we created in that tutorial and print our response from the API:

```python
import json

def jprint(obj):
    # create a formatted string of the Python JSON object
    text = json.dumps(obj, sort_keys=True, indent=4)
    print(text)

jprint(r.json())
```

```
{
  "artists": {
    "@attr": {
      "page": "1",
      "perPage": "50",
      "total": "2901036",
      "totalPages": "58021"
    },
    "artist": [
      {
        "image": [
          {
            "#text": "https://lastfm-img2.akamaized.net/i/u/34s/2a96cbd8b46e442fc41c2b8
            "size": "small"
          },
          {
            "#text": "https://lastfm-img2.akamaized.net/i/u/64s/2a96cbd8b46e442fc41c2b8
            "size": "medium"
          },
          {
            "#text": "https://lastfm-img2.akamaized.net/i/u/174s/2a96cbd8b46e442fc41c2b
            "size": "large"
          },
          {
            "#text": "https://lastfm-img2.akamaized.net/i/u/300x300/2a96cbd8b46e442fc41
            "size": "extralarge"
          },
          {
            "#text": "https://lastfm-img2.akamaized.net/i/u/300x300/2a96cbd8b46e442fc41
            "size": "mega"
          }
        ],
        "listeners": "1957174",
        "mbid": "b7539c32-53e7-4908-bda3-81449c367da6",
```

The structure of the JSON response is:

- A dictionary with a single `artists` key, containing:
    - an `@attr` key containing a number of attributes about the response.
    - an `artist` key containing a list of artist objects.

Let's look at the `'@attr'` (attributes) key by itself:

```
jprint(r.json()['artists']['@attr'])
```


```
{
  "page": "1",
  "perPage": "50",
  "total": "2901036",
  "totalPages": "58021"
}
```

There are almost three million total artists in the results of this API endpoint, and we're being showing the first 50 artists in a single 'page'. This technique of spreading the results over multiple pages is called **pagination**.

Working with Paginated Data

In order to build a dataset with many artists, we need to make an API request for each page and then put them together. We can control the pagination of our results using two optional parameters specified in the documentation:

- `limit`: The number of results to fetch per page (defaults to 50).
- `page`: Which page of the results we want to fetch.

Because the `'@attrs'` key gives us the total number of pages, we can use a `while` loop and iterate over pages until the page number is equal to the last page number.

We can also use the `limit` parameter to fetch more results in each page — we'll fetch 500 results per page so we only need to make ~6,000 calls instead of ~60,000.

Let's look at an example of how we would structure that code:

```
# initialize list for results
results = []

# set initial page and a high total number
page = 1
total_pages = 99999


while page > total_pages:
    # simplified request code for this example
    r = request.get("endpoint_url", params={"page": page})

    # append results to list
    results.append(r.json())

    # increment page
    page += 1
```

As we mentioned a moment ago, we need to make almost 6,000 calls to this end point, which means we need to think about rate limiting to comply with the Last.fm API's terms of service. Let's look at a few approaches.

Rate Limiting

Rate limiting is using code to limit the number of times per second that we hit a particular API. Rate limiting will make your code *slower*, but it's better than getting banned from using an API altogether.

The easiest way to perform rate limiting is to use Python `time.sleep()` function. This function accepts a float specifying a number of seconds to wait before proceeding.

For instance, the following code will wait one quarter of a second between the two print statements:

```
import time

print("one")
time.sleep(0.25)
print("two")
```

Because making the API call itself takes some time, we're likely to be making two or three calls per second, not the four calls per second that sleeping for 0.25s might suggest. This should be enough to keep us under Last.fm's threshold (if we were going to be hitting their API for a number of hours, we might choose an even slower rate).

Another technique that's useful for rate limiting is using a local database to cache the results of any API call, so that if we make the same call twice, the second time it reads it from the local cache. Imagine that as you are writing your code, you discover syntax errors and your loop fails, and you have to start again. By using a local cache, you have two benefits:

1. You don't make extra API calls that you don't need to.
2. You don't need to wait the extra time to rate limit when reading the repeated calls from the cache.

The logic that we could use to combine waiting with a cache looks like the below:

Creating logic for a local cache is a reasonably complex task, but there's a great library called requests-cache which will do all of the work for you with only a couple of lines of code.

You can install requests-cache using pip:

```
pip install requests-cache
```

Then all we need to do is import the library and invoke the `requests_cache.install_cache()` function, and the library will transparently cache new API requests, and use the cache whenever we make a repeated call.

```
import requests_cache

requests_cache.install_cache()
```

The last thing we should consider is that our 6,000 requests will likely take about 30 minutes to make, and so we'll print some output in each loop so we can see where everything is at. We'll use an IPython display trick to clear the output after each run so things look neater in our notebook.

Let's get started!

```
import time
from IPython.core.display import clear_output

responses = []

page = 1
total_pages = 99999 # this is just a dummy number so the loop starts

while page < = total_pages:
    payload = {
        'method': 'chart.gettopartists',
        'limit': 500,
        'page': page
    }

    # print some output so we can see the status
    print("Requesting page {}/{}".format(page, total_pages))
    # clear the output to make things neater
    clear_output(wait = True)

    # make the API call
    response = lastfm_get(payload)

    # if we get an error, print the response and halt the loop
    if response.status_code != 200:
        print(response.text)
        break

    # extract pagination info
    page = int(response.json()['artists']['@attr']['page'])
    total_pages = int(response.json()['artists']['@attr']['totalPages']

    # append response
    responses.append(response)
```

```
    # if it's not a cached result, sleep
    if not getattr(response, 'from_cache', False):
        time.sleep(0.25)

    # increment the page number
    page += 1</code>
```

```
Requesting page 5803/5803
```

The first time you run that code, it will take around half an hour. Because of the cache, the second time will be much quicker (likely less than a minute!)

Processing the Data

Let's use pandas to look at the data from the first response in our `responses` list:

```
import pandas as pd

r0 = responses[0]
r0_json = r0.json()
r0_artists = r0_json['artists']['artist']
r0_df = pd.DataFrame(r0_artists)
r0_df.head()
```

	name	playcount	listeners	mbid	url	stream
0	Lana Del Rey	232808939	1957174	b7539c32-53e7-4908-bda3-81449c367da6	https://www.last.fm/music/Lana+Del+Rey	0
1	Billie Eilish	35520548	588883		https://www.last.fm/music/Billie+Eilish	0
2	Post Malone	34942708	655052		https://www.last.fm/music/Post+Malone	0
3	Taylor Swift	196907702	2290993	20244d07-534f-4eff-b4d4-930878889970	https://www.last.fm/music/Taylor+Swift	0
4	Ariana Grande	124251766	1166180	f4fdbb4c-e4b7-47a0-b83b-d91bbfcfa387	https://www.last.fm/music/Ariana+Grande	0

We can use list comprehension to perform this operation on each response from `responses`, giving us a list of dataframes, and then use the `pandas.concat()` function to turn the list of dataframes into a single dataframe.

```
frames = [pd.DataFrame(r.json()['artists']['artist']) for r in response
artists = pd.concat(frames)
artists.head()
```

	name	playcount	listeners	mbid	url	streamab
0	Lana Del Rey	232808939	1957174	b7539c32-53e7-4908-bda3-81449c367da6	https://www.last.fm/music/Lana+Del+Rey	0
1	Billie Eilish	35520548	588883		https://www.last.fm/music/Billie+Eilish	0
2	Post Malone	34942708	655052		https://www.last.fm/music/Post+Malone	0
3	Taylor Swift	196907702	2290993	20244d07-534f-4eff-b4d4-930878889970	https://www.last.fm/music/Taylor+Swift	0
4	Ariana Grande	124251766	1166180	f4fdbb4c-e4b7-47a0-b83b-d91bbfcfa387	https://www.last.fm/music/Ariana+Grande	0

Our next step will be to remove the image column, which contains URLs for artist images that aren't really helpful to us from an analysis standpoint.

```
artists = artists.drop('image', axis=1)
artists.head()
```

	name	playcount	listeners	mbid	url	streamab
0	Lana Del Rey	232808939	1957174	b7539c32-53e7-4908-bda3-81449c367da6	https://www.last.fm/music/Lana+Del+Rey	0
1	Billie Eilish	35520548	588883		https://www.last.fm/music/Billie+Eilish	0
2	Post Malone	34942708	655052		https://www.last.fm/music/Post+Malone	0
3	Taylor Swift	196907702	2290993	20244d07-534f-4eff-b4d4-930878889970	https://www.last.fm/music/Taylor+Swift	0
4	Ariana Grande	124251766	1166180	f4fdbb4c-e4b7-47a0-b83b-d91bbfcfa387	https://www.last.fm/music/Ariana+Grande	0

Now, let's get to know the data a little using `DataFrame.info()` and `DataFrame.describe()`:

```
artists.info()
artists.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10500 entries, 0 to 499
Data columns (total 6 columns):
name        10500 non-null object
playcount   10500 non-null object
listeners   10500 non-null object
mbid        10500 non-null object
url         10500 non-null object
streamable  10500 non-null object
dtypes: object(6)
memory usage: 574.2+ KB
```

	name	playcount	listeners	mbid	url	streamable
count	10500	10500	10500	10500	10500	10500
unique	10000	9990	9882	7223	10000	1
top	Clairo	4591789	2096985		https://www.last.fm/music/alt-J	0
freq	2	2	4	2813	2	10500

We were expecting about 3,000,000 artists but we only have 10,500. Of those, only 10,000 are unique (eg there are duplicates).

Let's let's look at the length of the list of artists across our list of response objects to see if we can better understand what has gone wrong.

```
artist_counts = [len(r.json()['artists']['artist']) for r in responses]
pd.Series(artist_counts).value_counts()
```

```
0     5783
500     19
1000     1
dtype: int64
```

It looks like only twenty of our requests had a list of responses – let's look at the first fifty in order and see if there's a pattern.

```
print(artist_counts[:50])
```

```
[500, 1000, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500]
```

It looks like after the first twenty responses, this API doesn't return any data — an undocumented limitation.

This is not the end of the world, since 10,000 artists is still a good amount of data. Let's get rid of the duplicates we detected earlier.

```
artists = artists.drop_duplicates().reset_index(drop=True)
artists.describe()
```

	name	playcount	listeners	mbid	url	streamable
count	10000	10000	10000	10000	10000	10000
unique	10000	9990	9882	7223	10000	1
top	Säkert!	508566	30176		https://www.last.fm/music/Heinz+Goldblatt	0
freq	1	2	3	2750	1	10000

Augmenting the Data Using a Second Last.fm API Endpoint

In order to make our data more interesting, let's use another last.fm API endpoint to add some extra data about each artist.

Last.fm allows its users to create "tags" to categorize artists. By using the `artist.getTopTags` endpoint we can get the top tags from an individual artist.

Let's look at the response from that endpoint for one of our artists as an example:

```
r = lastfm_get({
    'method': 'artist.getTopTags',
    'artist':  'Lana Del Rey'
})

jprint(r.json())
```

```
{
  "toptags": {
    "@attr": {
      "artist": "Lana Del Rey"
    },
    "tag": [
      {
        "count": 100,
        "name": "female vocalists",
        "url": "https://www.last.fm/tag/female+vocalists"
      },
      {
        "count": 93,
        "name": "indie",
        "url": "https://www.last.fm/tag/indie"
      },
      {
        "count": 88,
        "name": "indie pop",
        "url": "https://www.last.fm/tag/indie+pop"
      },
      {
        "count": 80,
        "name": "pop",
        "url": "https://www.last.fm/tag/pop"
      },
      {
        "count": 67,
        "name": "alternative",
        "url": "https://www.last.fm/tag/alternative"
      },
      {
        "count": 14,
        "name": "american",
```

We're really only interested in the tag names, and then only the most popular tags. Let's use list comprehension to create a list of the top three tag names:

```
tags = [t['name'] for t in r.json()['toptags']['tag'][:3]]
tags
```

```
['female vocalists', 'indie', 'indie pop']
```

And then we can use the `str.join()` method to turn the list into a string:

```
', '.join(tags)
```

```
'female vocalists, indie, indie pop'
```

Let’s create a function that uses this logic to return a string of the most popular tag for any artist, which we’ll use later to apply to every row in our dataframe.

Remember that this function will be used a lot in close succession, so we’ll reuse our `time.sleep()` logic from earlier.

```
def lookup_tags(artist):
    response = lastfm_get({
        'method': 'artist.getTopTags',
        'artist':  artist
    })

    # if there's an error, just return nothing
    if response.status_code != 200:
        return None

    # extract the top three tags and turn them into a string
    tags = [t['name'] for t in response.json()['toptags']['tag'][:3]]
    tags_str = ', '.join(tags)

    # rate limiting
    if not getattr(response, 'from_cache', False):
        time.sleep(0.25)
    return tags_str
```

And let's test our function on another artist:

```
lookup_tags("Billie Eilish")
```

```
'pop, indie pop, indie'
```

Applying this function to our 10,000 rows will take just under an hour. So we know that things are actually progressing, we'll look to monitor the operation with output like we did before.

Unfortunately, manually printing output isn't an approach we can use when applying a function with the pandas `Series.apply()` method. Instead, we'll use the tqdm package which automates this.

```
from tqdm import tqdm
tqdm.pandas()

artists['tags'] = artists['name'].progress_apply(lookup_tags)
```

```
100%|██████████| 10000/10000 [00:33<00:00, 295.25it/s]
```

Let’s look at the result of our operation:

```
artists.head()
```

	name	playcount	listeners	mbid	url	stream
0	Lana Del Rey	232808939	1957174	b7539c32-53e7-4908-bda3-81449c367da6	https://www.last.fm/music/Lana+Del+Rey	0
1	Billie Eilish	35520548	588883		https://www.last.fm/music/Billie+Eilish	0
2	Post Malone	34942708	655052		https://www.last.fm/music/Post+Malone	0
3	Taylor Swift	196907702	2290993	20244d07-534f-4eff-b4d4-930878889970	https://www.last.fm/music/Taylor+Swift	0
4	Ariana Grande	124251766	1166180	f4fdbb4c-e4b7-47a0-b83b-d91bbfcfa387	https://www.last.fm/music/Ariana+Grande	0

Finalizing and Exporting the Data

Before we export our data, we might like to sort the data so the most popular artists are at the top. So far we've just been storing data as text without converting any types:

```
artists.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 7 columns):
name        10000 non-null object
playcount   10000 non-null object
listeners   10000 non-null object
mbid        10000 non-null object
url         10000 non-null object
streamable  10000 non-null object
tags        10000 non-null object
dtypes: object(7)
memory usage: 547.0+ KB
```

Let’s start by converting the listeners and playcount columns to numeric.

```
artists[["playcount", "listeners"]] = artists[["playcount", "listeners"
```

Now, let’s sort by number of listeners

```
artists = artists.sort_values("listeners", ascending=False)
artists.head(10)
```

	name	playcount	listeners	mbid	
23	Coldplay	364107739	5429555	cc197bad-dc9c-440d-a5b5-d52ba2e14234	https://www.last.fm/music/Coldplay
10	Radiohead	507682999	4778856	a74b1b7f-71a5-4011-9441-d0b5e4122711	https://www.last.fm/music/Radiohead
24	Red Hot Chili Peppers	298043491	4665823	8bfac288-ccc5-448d-9573-c33ea2aa5c30	https://www.last.fm/music/Red+Hot+Chili+P
12	Rihanna	204114198	4618838	db36a76f-4cdf-43ac-8cd0-5e48092d2bae	https://www.last.fm/music/Rihanna
35	Eminem	204081058	4569194	b95ce3ff-3d05-4e87-9e01-c97b66af13d4	https://www.last.fm/music/Eminem
45	The Killers	211585596	4472323	95e1ead9-4d31-4808-a7ac-32c3614c116b	https://www.last.fm/music/The+Killers
				164f0d73-	

Finally, we can export our dataset as a CSV file.

```
artists.to_csv('artists.csv', index=False)
```

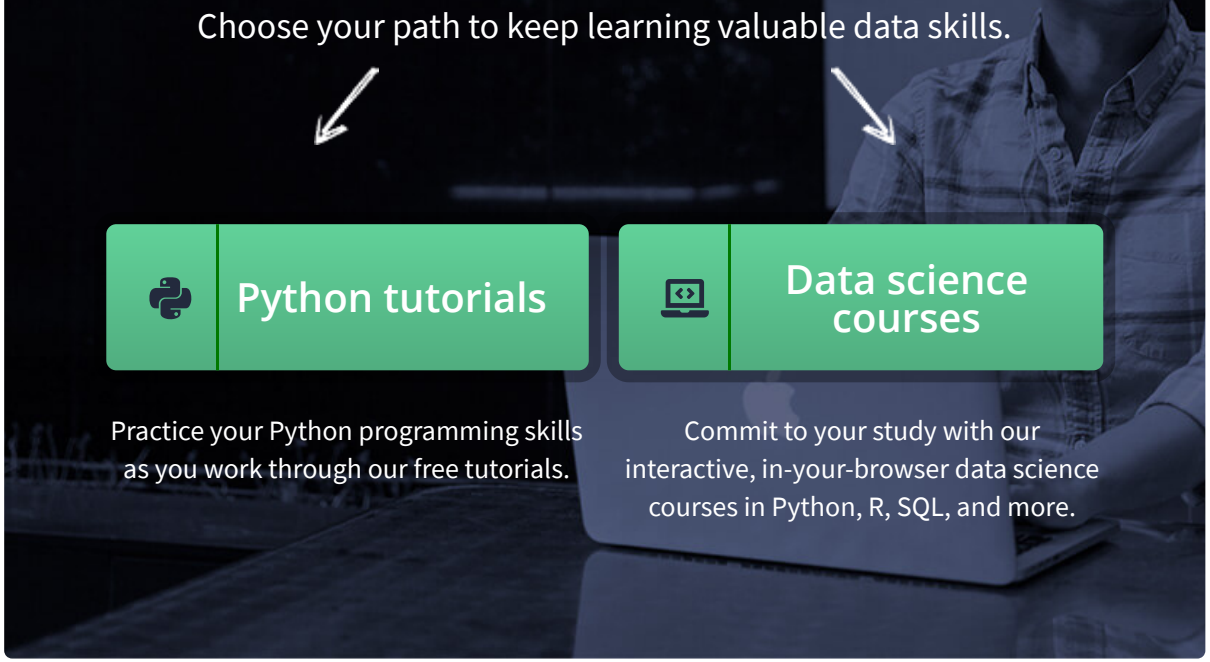
Next Steps

In this tutorial we built a dataset using the Last.fm API while learning how to use Python to:

- How to authenticate with an API using an API key
- How to use pagination to collect larger responses from an API endpoint
- How to use rate-limiting to stay within the guidelines of an API.

If you'd like to extend your learning, you might like to:

- Complete our interactive Dataquest APIs and scraping course, which you can start for free.
- Explore the other API endpoints in the Lasts.fm API.
- Try working with some data from this list of Free Public APIs.



Josh Devlin

Data Scientist at Dataquest.io. Loves Data and Aussie Rules Football. Australian living in Texas.

TAGS API, dataset, intermediate, last.fm, music, python, requests, Tutorials

You may also like


Python Practice: Free Ways To Improve Your Python Skills
READ MORE


You Need Data Skills to Future-Proof Your Career
READ MORE


Tutorial: Web Scraping with Python Using Beautiful Soup
READ MORE

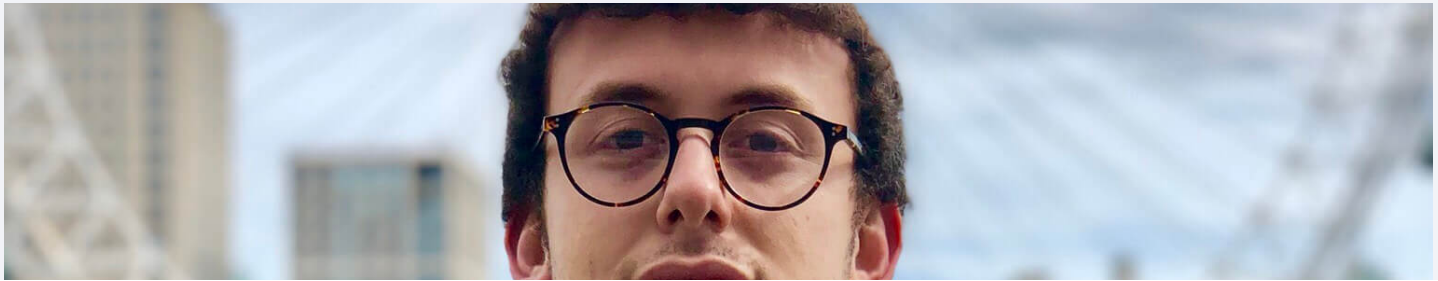

Data Analytics Certified
Data Analytics Certification: Do You Need a Certificate to Get a Job as a Data Analyst?
READ MORE

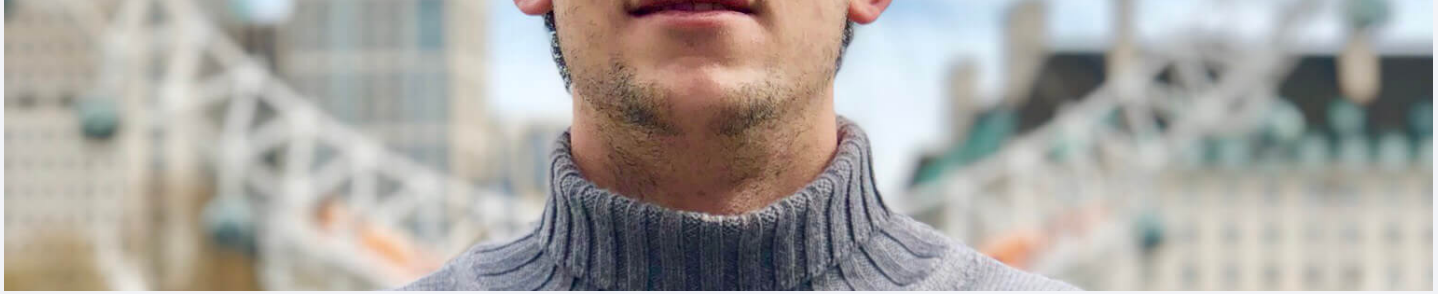
Search

Categories

- Building A Data Science Portfolio
- Cheat Sheets
- Data Science Career Tips
- Data Science Projects
- Data Science Tutorials

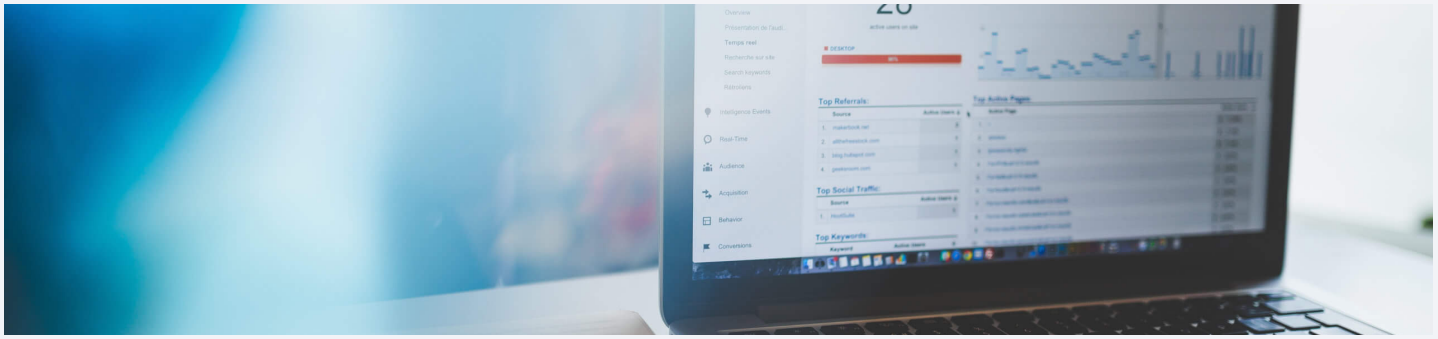
Top Picks





MARCH 15, 2021

Why Jorge Prefers Dataquest Over DataCamp for Learning Data Analysis

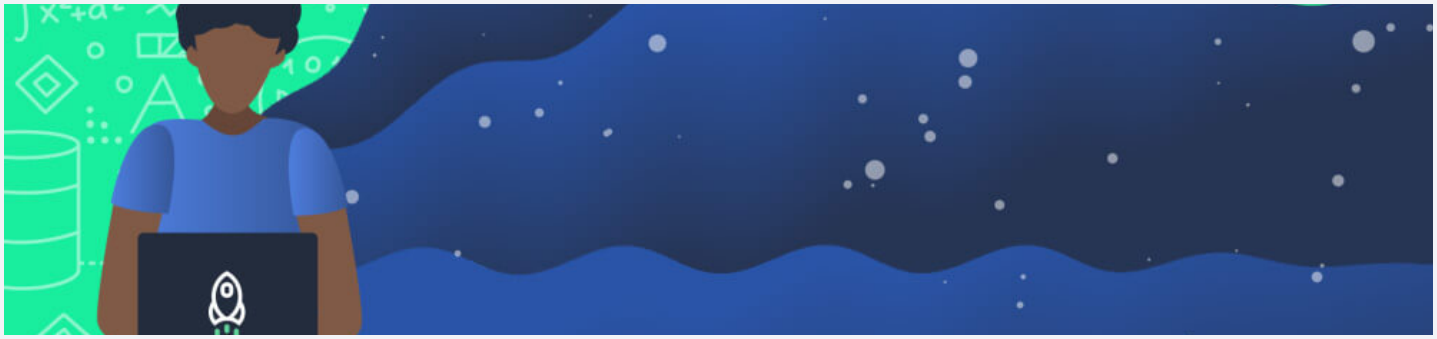


JULY 21, 2020

Tutorial: Better Blog Post Analysis with googleAnalyticsR

FEBRUARY 24, 2021

How to Learn Python (Step-by-Step) in 2021



MAY 4, 2020

How to Learn Data Science (Step-By-Step) in 2020

JULY 6, 2020

Data Science Certificates in 2020 (Are They Worth It?)

Get started with Dataquest today -

for free!

Sign up now

Or, visit our pricing page to learn about our Basic and Premium plans.

Facebook

Twitter

LinkedIn

Resource List