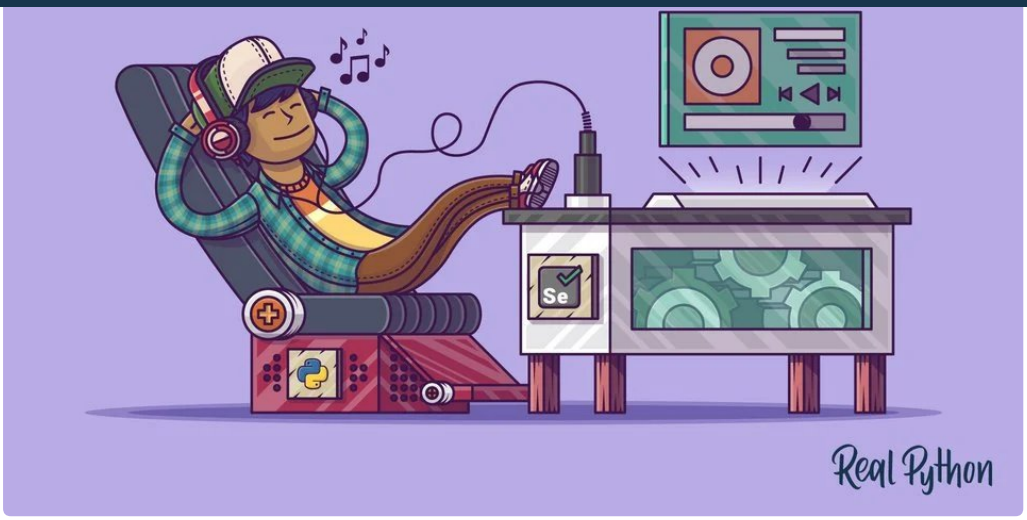


Modern Web Automation With Python and Selenium

by Colin OKeefe 35 Comments intermediate testing web-scraping

Table of Contents

listening history? I could just look up the electronic songs from two months ago, and I’d surely find it.”

Today, you will build a basic Python class, called `BandLeader` that connects to bandcamp.com, streams music from the “discovery” section of the front page, and keeps track of your listening history.

The listening history will be saved to disk in a CSV file. You can then explore that CSV file in your favorite spreadsheet application or even with Python.

If you have had some experience with web scraping in Python, you are familiar with making HTTP requests and using Pythonic APIs to navigate the DOM. You will do more of the same today, except with one difference.

Today you will use a full-fledged browser running in headless mode to do the HTTP requests for you.

A headless browser is just a regular web browser, except that it contains no visible UI element. Just like you’d expect, it can do more

than make requests: it can also render HTML (though you cannot see it), keep session information, and even perform asynchronous network communications by running JavaScript code.

If you want to automate the modern web, headless browsers are essential.

> **Free Bonus: Click here to download a "Python + Selenium" project skeleton with full source code** that you can use as a foundation for your own Python web scraping and automation apps.

Setup

Your first step, before writing a single line of Python, is to install a Selenium supported WebDriver for your favorite web browser. In what follows, you will be working with Firefox, but Chrome could easily work too.

Assuming that the path `~/.local/bin` is in your execution `PATH`, here's how you would install the Firefox WebDriver, called `geckodriver`, on a Linux machine:

Shell

```
$ wget https://github.com/mozilla/geckodriver/releases/downloac
$ tar xvfz geckodriver-v0.19.1-linux64.tar.gz
$ mv geckodriver ~/.local/bin
```

Next, you install the selenium package, using `pip` or whatever you like. If you made a virtual environment for this project, you just type:

Shell

```
$ pip install selenium
```

> **Note:** If you ever feel lost during the course of this tutorial, the full code demo can be found on GitHub.

Now it's time for a test drive.

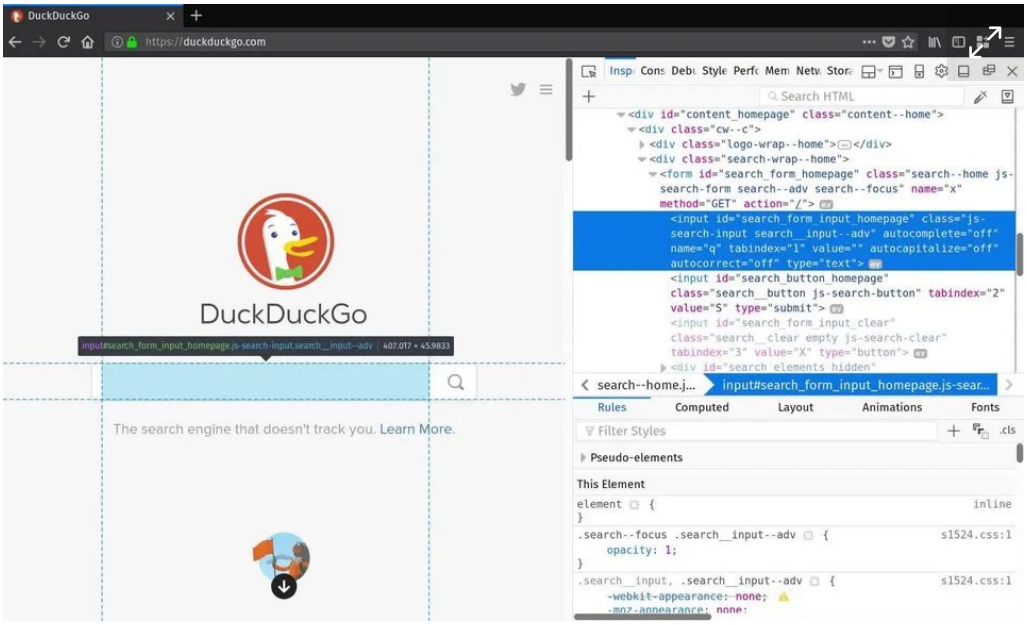
Test Driving a Headless Browser

To test that everything is working, you decide to try out a basic web search via DuckDuckGo. You fire up your preferred Python interpreter and type the following:

Python >>>

```
>>> from selenium.webdriver import Firefox
>>> from selenium.webdriver.firefox.options import Options
>>> opts = Options()
>>> opts.set_headless()
>>> assert opts.headless  # Operating in headless mode
>>> browser = Firefox(options=opts)
>>> browser.get('https://duckduckgo.com')
```

So far, you have created a headless Firefox browser and navigated to `https://duckduckgo.com`. You made an `Options` instance and used it to activate headless mode when you passed it to the `Firefox` constructor. This is akin to typing `firefox -headless` at the command line.



Now that a page is loaded, you can query the DOM using methods defined on your newly minted `browser` object. But how do you know what to query?

The best way is to open your web browser and use its developer tools to inspect the contents of the page. Right now, you want to get ahold of the search form so you can submit a query. By inspecting DuckDuckGo's home page, you find that the search form `<input>` element has an `id` attribute `"search_form_input_homepage"`. That's just what you needed:

Python >>>

```
>>> search_form = browser.find_element_by_id('search_form_input
>>> search_form.send_keys('real python')
>>> search_form.submit()
```

You found the search form, used the `send_keys` method to fill it out, and then the `submit` method to perform your search for `"Real Python"`. You can checkout the top result:

Python >>>

```
>>> results = browser.find_elements_by_class_name('result')
>>> print(results[0].text)

Real Python - Real Python
Get Real Python and get your hands dirty quickly so you spend m
https://realpython.com
```

Everything seems to be working. In order to prevent invisible headless browser instances from piling up on your machine, you close the browser object before exiting your Python session:

Python >>>

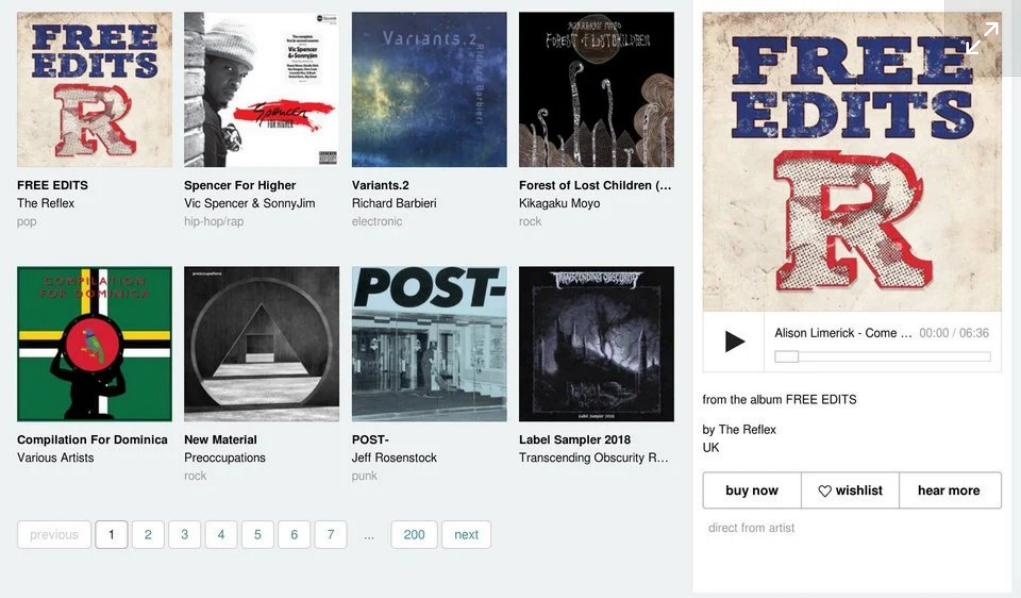
```
>>> browser.close()
>>> quit()
```

Groovin’ on Tunes

You’ve tested that you can drive a headless browser using Python. Now you can put it to use:

1. You want to play music.
2. You want to browse and explore music.
3. You want information about what music is playing.

To start, you navigate to https://bandcamp.com and start to poke around in your browser’s developer tools. You discover a big shiny play button towards the bottom of the screen with a `class` attribute that contains the value`"playbutton"`. You check that it works:



Python >>>

```
>>> opts = Option()
>>> opts.set_headless()
>>> browser = Firefox(options=opts)
>>> browser.get('https://bandcamp.com')
>>> browser.find_element_by_class('playbutton').click()
```

You should hear music! Leave it playing and move back to your web browser. Just to the side of the play button is the discovery section. Again, you inspect this section and find that each of the currently visible available tracks has a `class` value of `"discover-item"`, and

that each item seems to be clickable. In Python, you check this out:

```
Python >>>
>>> tracks = browser.find_elements_by_class_name('discover-item
>>> len(tracks)  # 8
>>> tracks[3].click()
```

A new track should be playing! This is the first step to exploring bandcamp using Python! You spend a few minutes clicking on various tracks in your Python environment but soon grow tired of the meagre library of eight songs.

Exploring the Catalogue

Looking a back at your browser, you see the buttons for exploring all of the tracks featured in bandcamp's music discovery section. By now, this feels familiar: each button has a `class` value of `"item-page"`. The very last button is the "next" button that will display the next eight tracks in the catalogue. You go to work:

```
Python >>>
>>> next_button = [e for e in browser.find_elements_by_class_na
                   if e.text.lower().find('next') > -1]
>>> next_button.click()
```

Great! Now you want to look at the new tracks, so you think, "I'll just repopulate my `tracks` variable like I did a few minutes ago." But this is where things start to get tricky.

First, bandcamp designed their site for humans to enjoy using, not for Python scripts to access programmatically. When you call `next_button.click()`, the real web browser responds by executing some JavaScript code.

If you try it out in your browser, you see that some time elapses as the catalogue of songs scrolls with a smooth animation effect. If you try to repopulate your `tracks` variable before the animation finishes, you may not get all the tracks, and you may get some that you don't

want.

What's the solution? You can just sleep for a second, or, if you are just running all this in a Python shell, you probably won't even notice. After all, it takes time for you to type too.

Another slight kink is something that can only be discovered through experimentation. You try to run the same code again:

```
Python >>>

>>> tracks = browser.find_elements_by_class_name('discover-item
>>> assert(len(tracks) == 8)
AssertionError
...
```

But you notice something strange. `len(tracks)` is not equal to `8` even though only the next batch of `8` should be displayed. Digging a little further, you find that your list contains some tracks that were displayed before. To get only the tracks that are actually visible in the browser, you need to filter the results a little.

After trying a few things, you decide to keep a track only if its `x` coordinate on the page fall within the bounding box of the containing element. The catalogue's container has a `class` value of `"discover-results"`. Here's how you proceed:

```
Python >>>

>>> discover_section = self.browser.find_element_by_class_name(
>>> left_x = discover_section.location['x']
>>> right_x = left_x + discover_section.size['width']
>>> discover_items = browser.find_element_by_class_name('discov
>>> tracks = [t for t in discover_items
              if t.location['x'] >= left_x and t.location['x']
>>> assert len(tracks) == 8
```

Building a Class

If you are growing weary of retyping the same commands over and over again in your Python environment, you should dump some of it into a module. A basic class for your bandcamp manipulation should do the following:

1. Initialize a headless browser and navigate to bandcamp
2. Keep a list of available tracks
3. Support finding more tracks
4. Play, pause, and skip tracks

Here's the basic code, all in one go:

Python

```
from selenium.webdriver import Firefox
from selenium.webdriver.firefox.options import Options
from time import sleep, ctime
from collections import namedtuple
from threading import Thread
from os.path import isfile
import csv


BANDCAMP_FRONTPAGE='https://bandcamp.com/'

class BandLeader():
    def __init__(self):
        # Create a headless browser
        opts = Options()
        opts.set_headless()
        self.browser = Firefox(options=opts)
        self.browser.get(BANDCAMP_FRONTPAGE)

        # Track list related state
        self._current_track_number = 1
        self.track_list = []
        self.tracks()

    def tracks(self):
        '''
        Query the page to populate a list of available tracks.
        '''
```

```
        # Sleep to give the browser time to render and finish a
        sleep(1)

        # Get the container for the visible track list
        discover_section = self.browser.find_element_by_class_r
        left_x = discover_section.location['x']
        right_x = left_x + discover_section.size['width']

        # Filter the items in the list to include only those we
        discover_items = self.browser.find_elements_by_class_na
        self.track_list = [t for t in discover_items
                           if t.location['x'] >= left_x and t.l

        # Print the available tracks to the screen
        for (i,track) in enumerate(self.track_list):
            print('[{}]'.format(i+1))
            lines = track.text.split('\n')
            print('Album  : {}'.format(lines[0]))
            print('Artist : {}'.format(lines[1]))
            if len(lines) > 2:
                print('Genre  : {}'.format(lines[2]))

    def catalogue_pages(self):
        '''
        Print the available pages in the catalogue that are pre
        accessible.
        '''
        print('PAGES')
        for e in self.browser.find_elements_by_class_name('item
            print(e.text)
        print('')


    def more_tracks(self,page='next'):
        '''
        Advances the catalogue and repopulates the track list.
        to advance any of the available pages.
        '''

        next_btn = [e for e in self.browser.find_elements_by_cl
                    if e.text.lower().strip() == str(page)]

        if next_btn:
            next_btn[0].click()
            self.tracks()

    def play(self,track=None):
        '''
```

```
        Play a track. If no track number is supplied, the prese
        will play.
        '''

        if track is None:
            self.browser.find_element_by_class_name('playbuttor
        elif type(track) is int and track <= len(self.track_list
            self._current_track_number = track
            self.track_list[self._current_track_number - 1].cli


    def play_next(self):
        '''
        Plays the next available track
        '''
        if self._current_track_number < len(self.track_list):
            self.play(self._current_track_number+1)
        else:
            self.more_tracks()
            self.play(1)


    def pause(self):
        '''
        Pauses the playback
        '''
        self.play()
```

Pretty neat. You can import this into your Python environment and run bandcamp programmatically! But wait, didn't you start this whole thing because you wanted to keep track of information about your listening history?

Collecting Structured Data

Your final task is to keep track of the songs that you actually listened to. How might you do this? What does it mean to actually listen to something anyway? If you are perusing the catalogue, stopping for a few seconds on each song, do each of those songs count? Probably not. You are going to allow some 'exploration' time to factor in to your data collection.

Your goals are now to:

1. Collect structured information about the currently playing track
2. Keep a “database” of tracks
3. Save and restore that “database” to and from disk

You decide to use a namedtuple to store the information that you track. Named tuples are good for representing bundles of attributes with no functionality tied to them, a bit like a database record:

Python

```
TrackRec = namedtuple('TrackRec', [
    'title',
    'artist',
    'artist_url',
    'album',
    'album_url',
    'timestamp'  # When you played it
])
```

In order to collect this information, you add a method to the `BandLeader` class. Checking back in with the browser’s developer tools, you find the right HTML elements and attributes to select all the information you need. Also, you only want to get information about the currently playing track if there music is actually playing at the time. Luckily, the page player adds a `"playing"` class to the play button whenever music is playing and removes it when the music stops.

With these considerations in mind, you write a couple of methods:

Python

```
    def is_playing(self):
        '''
        Returns `True` if a track is presently playing
        '''
        playbtn = self.browser.find_element_by_class_name('play
        return playbtn.get_attribute('class').find('playing') >


    def currently_playing(self):
        '''
```

```
        Returns the record for the currently playing track,
        or None if nothing is playing
        '''
        try:
            if self.is_playing():
                title = self.browser.find_element_by_class_name
                album_detail = self.browser.find_element_by_css
                album_title = album_detail.text
                album_url = album_detail.get_attribute('href').
                artist_detail = self.browser.find_element_by_cs
                artist = artist_detail.text
                artist_url = artist_detail.get_attribute('href'
                return TrackRec(title, artist, artist_url, albu

        except Exception as e:
            print('there was an error: {}'.format(e))

        return None
```

For good measure, you also modify the `play()` method to keep track of the currently playing track:

Python

```
    def play(self, track=None):
        '''
        Play a track. If no track number is supplied, the prese
        will play.
        '''

        if track is None:
            self.browser.find_element_by_class_name('playbuttor
        elif type(track) is int and track <= len(self.track_lis
            self._current_track_number = track
            self.track_list[self._current_track_number - 1].cli

        sleep(0.5)
        if self.is_playing():
            self._current_track_record = self.currently_playing
```

Next, you've got to keep a database of some kind. Though it may not scale well in the long run, you can go far with a simple list. You add `self.database = []` to `BandCamp`'s `__init__()` method. Because you want to allow for time to pass before entering a `TrackRec` object into the database, you decide to use Python's threading tools to run a

separate process that maintains the database in the background.

You'll supply a `_maintain()` method to `BandLeader` instances that will run in a separate thread. The new method will periodically check the value of `self._current_track_record` and add it to the database if it is new.

You will start the thread when the class is instantiated by adding some code to `__init__()`:

Python

```
    # The new init
    def __init__(self):
        # Create a headless browser
        opts = Options()
        opts.set_headless()
        self.browser = Firefox(options=opts)
        self.browser.get(BANDCAMP_FRONTPAGE)

        # Track list related state
        self._current_track_number = 1
        self.track_list = []
        self.tracks()

        # State for the database
        self.database = []
        self._current_track_record = None

        # The database maintenance thread
        self.thread = Thread(target=self._maintain)
        self.thread.daemon = True    # Kills the thread with th
        self.thread.start()

        self.tracks()


    def _maintain(self):
        while True:
            self._update_db()
            sleep(20)          # Check every 20 seconds


    def _update_db(self):
        try:
            check = (self._current_track_record is not None
                     and (len(self.database) == 0
```

```
                            or self.database[-1] != self._current
                        and self.is_playing())
            if check:
                self.database.append(self._current_track_record

        except Exception as e:
            print('error while updating the db: {}'.format(e)
```

If you've never worked with multithreaded programming in Python, you should read up on it! For your present purpose, you can think of thread as a loop that runs in the background of the main Python process (the one you interact with directly). Every twenty seconds, the loop checks a few things to see if the database needs to be updated, and if it does, appends a new record. Pretty cool.

The very last step is saving the database and restoring from saved states. Using the csv package, you can ensure your database resides in a highly portable format and remains usable even if you abandon your wonderful `BandLeader` class!

The `__init__()` method should be yet again altered, this time to accept a file path where you'd like to save the database. You'd like to load this database if it is available, and you'd like to save it periodically, whenever it is updated. The updates look like this:

Python

```
    def __init__(self,csvpath=None):
        self.database_path=csvpath
        self.database = []

        # Load database from disk if possible
        if isfile(self.database_path):
            with open(self.database_path, newline='') as dbfile
                dbreader = csv.reader(dbfile)
                next(dbreader)   # To ignore the header line
                self.database = [TrackRec._make(rec) for rec ir

        # .... The rest of the __init__ method is unchanged ...


    # A new save_db() method
    def save_db(self):
        with open(self.database_path,'w',newline='') as dbfile:
```

```
            dbwriter = csv.writer(dbfile)
            dbwriter.writerow(list(TrackRec._fields))
            for entry in self.database:
                dbwriter.writerow(list(entry))


    # Finally, add a call to save_db() to your database mainter
    def _update_db(self):
        try:
            check = (self._current_track_record is not None
                     and self._current_track_record is not None
                     and (len(self.database) == 0
                          or self.database[-1] != self._current
                     and self.is_playing())
            if check:
                self.database.append(self._current_track_record
                self.save_db()

        except Exception as e:
            print('error while updating the db: {}'.format(e)
```

Voilà! You can listen to music and keep a record of what you hear! Amazing.

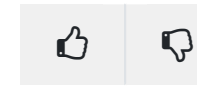
Something interesting about the above is that using a `namedtuple` really begins to pay off. When converting to and from CSV format, you take advantage of the ordering of the rows in the CSV file to fill in the rows in the `TrackRec` objects. Likewise, you can create the header row of the CSV file by referencing the `TrackRec._fields` attribute. This is one of the reasons using a tuple ends up making sense for columnar data.

What's Next and What Have You Learned?

You could do loads more! Here are a few quick ideas that would leverage the mild superpower that is Python + Selenium:

- You could extend the `BandLeader` class to navigate to album pages and play the tracks you find there.
- You might decide to create playlists based on your favorite or most frequently heard tracks.
- Perhaps you want to add an autoplay feature.
- Maybe you'd like to query songs by date or title or artist and build playlists that way.

You have learned that Python can do everything that a web browser can do, and a bit more. You could easily write scripts to control virtual browser instances that run in the cloud. You could create bots that interact with real users or mindlessly fill out forms! Go forth and automate!



About **Colin OKeefe**

Colin is a freelance Software Creative who travels the unixverse in the good ship Python.

» More about Colin

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Dan

 Joanna

Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:

Tweet Share Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate testing web-scraping

— FREE Email Series —

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents