CHAPTER 9

# Getting Data

*To write it, it took three months; to conceive it, three minutes; to collect the data in it, all my life.*

—F. Scott Fitzgerald

In order to be a data scientist you need data. In fact, as a data scientist you will spend an embarrassingly large fraction of your time acquiring, cleaning, and transforming data. In a pinch, you can always type the data in yourself (or if you have minions, make them do it), but usually this is not a good use of your time. In this chapter, we'll look at different ways of getting data into Python and into the right formats.

## stdin and stdout

If you run your Python scripts at the command line, you can *pipe* data through them using `sys.stdin` and `sys.stdout`. For example, here is a script that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

And here's one that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print(count)
```

You could then use these to count how many lines of a file contain numbers. In Windows, you'd use:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

whereas in a Unix system you'd use:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

The | is the pipe character, which means "use the output of the left command as the input of the right command." You can build pretty elaborate data-processing pipelines this way.

If you are using Windows, you can probably leave out the `python` part of this command:

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

If you are on a Unix system, doing so requires a couple more steps. First add a "shebang" as the first line of your script `#!/usr/bin/env python`. Then, at the command line, use `chmod x egrep.py++` to make the file executable.

Similarly, here's a script that counts the words in its input and writes out the most common ones:

```
# most_common_words.py
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print("usage: most_common_words.py num_words")
    sys.exit(1)   # nonzero exit code indicates error

counter = Counter(word.lower()                      # lowercase words
                  for line in sys.stdin
                  for word in line.strip().split()  # split on spaces
                  if word)                          # skip empty 'words'
```

```
for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

after which you could do something like:

```
$ cat the_bible.txt | python most_common_words.py 10
36397   the
30031   and
20163   of
7154    to
6484    in
5856    that
5421    he
5226    his
5060    unto
4297    shall
```

(If you are using Windows, then use `type` instead of `cat`.)

If you are a seasoned Unix programmer, you are probably familiar with a wide variety of command-line tools (for example, `egrep`) that are built into your operating system and are preferable to building your own from scratch. Still, it's good to know you can if you need to.

# Reading Files

You can also explicitly read from and write to files directly in your code. Python makes working with files pretty simple.

## The Basics of Text Files

The first step to working with a text file is to obtain a *file object* using `open`:

```
# 'r' means read-only, it's assumed if you leave it out
file_for_reading = open('reading_file.txt', 'r')
file_for_reading2 = open('reading_file.txt')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Because it is easy to forget to close your files, you should always use them in a `with` block, at the end of which they will be closed automatically:

```
with open(filename) as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

If you need to read a whole text file, you can just iterate over the lines of the file using `for`:

```
starts_with_hash = 0

with open('input.txt') as f:
    for line in f:                  # look at each line in the file
        if re.match("^#",line):     # use a regex to see if it starts with '#'
            starts_with_hash += 1   # if it does, add 1 to the count
```

Every line you get this way ends in a newline character, so you'll often want to `strip` it before doing anything with it.

For example, imagine you have a file full of email addresses, one per line, and you need to generate a histogram of the domains. The rules for correctly extracting domains are somewhat subtle—see, e.g., the Public Suffix List—but a good first approximation is to just take the parts of the email addresses that come after the @ (this gives the wrong answer for email addresses like *joel@mail.datasciencester.com*, but for the purposes of this example we're willing to live with that):

```
def get_domain(email_address: str) -> str:
    """Split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

# a couple of tests
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencester.com') == 'm.datasciencester.com'

from collections import Counter

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

## Delimited Files

The hypothetical email addresses file we just processed had one address per line. More frequently you'll work with files with lots of data on each line. These files are very often either *comma-separated* or *tab-separated*: each line has several fields, with a comma or a tab indicating where one field ends and the next field starts.

This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably will). For this reason, you should never try to parse them yourself. Instead, you should use Python's `csv` module (or the pandas library, or some other library that's designed to read comma-separated or tab-delimited files).

Never parse a comma-separated file yourself. You will screw up the edge cases!

If your file has no headers (which means you probably want each row as a `list`, and which places the burden on you to know what's in each column), you can use `csv.reader` to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
6/20/2014   AAPL    90.91
6/20/2014   MSFT    41.68
6/20/2014   FB  64.5
6/19/2014   AAPL    91.86
6/19/2014   MSFT    41.51
6/19/2014   FB  64.34
```

we could process them with:

```
import csv

with open('tab_delimited_stock_prices.txt') as f:
    tab_reader = csv.reader(f, delimiter='\t')
    for row in tab_reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

you can either skip the header row with an initial call to `reader.next`, or get each row as a `dict` (with the headers as keys) by using `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt') as f:
    colon_reader = csv.DictReader(f, delimiter=':')
    for dict_row in colon_reader:
        date = dict_row["date"]
```

```
        symbol = dict_row["symbol"]
        closing_price = float(dict_row["closing_price"])
        process(date, symbol, closing_price)
```

Even if your file doesn't have headers, you can still use `DictReader` by passing it the keys as a `fieldnames` parameter.

You can similarly write out delimited data using `csv.writer`:

```
todays_prices = {'AAPL': 90.91, 'MSFT': 41.68, 'FB': 64.5 }

with open('comma_delimited_stock_prices.txt', 'w') as f:
    csv_writer = csv.writer(f, delimiter=',')
    for stock, price in todays_prices.items():
        csv_writer.writerow([stock, price])
```

`csv.writer` will do the right thing if your fields themselves have commas in them. Your own hand-rolled writer probably won't. For example, if you attempt:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# don't do this!
with open('bad_csv.txt', 'w') as f:
    for row in results:
        f.write(",".join(map(str, row))) # might have too many commas in it!
        f.write("\n")                    # row might have newlines as well!
```

You will end up with a *.csv* file that looks like this:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

and that no one will ever be able to make sense of.

# Scraping the Web

Another way to get data is by scraping it from web pages. Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

## HTML and the Parsing Thereof

Pages on the web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

```
<html>
  <head>
    <title>A web page</title>
```

```
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

In a perfect world, where all web pages were marked up semantically for our benefit, we would be able to extract data using rules like "find the `<p>` element whose `id` is `subject` and return the text it contains." In the actual world, HTML is not generally well formed, let alone annotated. This means we'll need help making sense of it.

To get data out of HTML, we will use the Beautiful Soup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them. As I write this, the latest version is Beautiful Soup 4.6.0, which is what we'll be using. We'll also be using the Requests library, which is a much nicer way of making HTTP requests than anything that's built into Python.

Python's built-in HTML parser is not that lenient, which means that it doesn't always cope well with HTML that's not perfectly formed. For that reason, we'll also install the `html5lib` parser.

Making sure you're in the correct virtual environment, install the libraries:

```
python -m pip install beautifulsoup4 requests html5lib
```

To use Beautiful Soup, we pass a string containing HTML into the `BeautifulSoup` function. In our examples, this will be the result of a call to `requests.get`:

```
from bs4 import BeautifulSoup
import requests

# I put the relevant HTML file on GitHub. In order to fit
# the URL in the book I had to split it across two lines.
# Recall that whitespace-separated strings get concatenated.
url = ("https://raw.githubusercontent.com/"
       "joelgrus/data/master/getting-data.html")
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

after which we can get pretty far using a few simple methods.

We'll typically work with `Tag` objects, which correspond to the tags representing the structure of an HTML page.

For example, to find the first `<p>` tag (and its contents), you can use:

```
first_paragraph = soup.find('p')        # or just soup.p
```

You can get the text contents of a `Tag` using its `text` property:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

And you can extract a tag's attributes by treating it like a `dict`:

```
first_paragraph_id = soup.p['id']       # raises KeyError if no 'id'
first_paragraph_id2 = soup.p.get('id')  # returns None if no 'id'
```

You can get multiple tags at once as follows:

```
all_paragraphs = soup.find_all('p')  # or just soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Frequently, you'll want to find tags with a specific `class`:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

And you can combine these methods to implement more elaborate logic. For example, if you want to find every `<span>` element that is contained inside a `<div>` element, you could do this:

```
# Warning: will return the same <span> multiple times
# if it sits inside multiple <div>s.
# Be more clever if that's the case.
spans_inside_divs = [span
                     for div in soup('div')     # for each <div> on the page
                     for span in div('span')]   # find each <span> inside it
```

Just this handful of features will allow us to do quite a lot. If you end up needing to do more complicated things (or if you're just curious), check the documentation.

Of course, the important data won't typically be labeled as `class="important"`. You'll need to carefully inspect the source HTML, reason through your selection logic, and worry about edge cases to make sure your data is correct. Let's look at an example.

## Example: Keeping Tabs on Congress

The VP of Policy at DataSciencester is worried about potential regulation of the data science industry and asks you to quantify what Congress is saying on the topic. In particular, he wants you to find all the representatives who have press releases about "data."

At the time of publication, there is a page with links to all of the representatives' websites at *https://www.house.gov/representatives*.

And if you "view source," all of the links to the websites look like:

```
<td>
  <a href="https://jayapal.house.gov">Jayapal, Pramila</a>
</td>
```

Let's start by collecting all of the URLs linked to from that page:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.house.gov/representatives"
text = requests.get(url).text
soup = BeautifulSoup(text, "html5lib")

all_urls = [a['href']
            for a in soup('a')
            if a.has_attr('href')]

print(len(all_urls))  # 965 for me, way too many
```

This returns way too many URLs. If you look at them, the ones we want start with either *http://* or *https://*, have some kind of name, and end with either *.house.gov* or *.house.gov/*.

This is a good place to use a regular expression:

```
import re

# Must start with http:// or https://
# Must end with .house.gov or .house.gov/
regex = r"^https?://.*\.house\.gov/?$"

# Let's write some tests!
assert re.match(regex, "http://joel.house.gov")
assert re.match(regex, "https://joel.house.gov")
assert re.match(regex, "http://joel.house.gov/")
assert re.match(regex, "https://joel.house.gov/")
assert not re.match(regex, "joel.house.gov")
assert not re.match(regex, "http://joel.house.com")
assert not re.match(regex, "https://joel.house.gov/biography")

# And now apply
good_urls = [url for url in all_urls if re.match(regex, url)]

print(len(good_urls))  # still 862 for me
```

That's still way too many, as there are only 435 representatives. If you look at the list, there are a lot of duplicates. Let's use `set` to get rid of them:

```
good_urls = list(set(good_urls))

print(len(good_urls))  # only 431 for me
```

There are always a couple of House seats empty, or maybe there's a representative without a website. In any case, this is good enough. When we look at the sites, most of them have a link to press releases. For example:

```
html = requests.get('https://jayapal.house.gov').text
soup = BeautifulSoup(html, 'html5lib')
```

```
# Use a set because the links might appear multiple times.
links = {a['href'] for a in soup('a') if 'press releases' in a.text.lower()}

print(links) # {'/media/press-releases'}
```

Notice that this is a relative link, which means we need to remember the originating site. Let's do some scraping:

```
from typing import Dict, Set

press_releases: Dict[str, Set[str]] = {}

for house_url in good_urls:
    html = requests.get(house_url).text
    soup = BeautifulSoup(html, 'html5lib')
    pr_links = {a['href'] for a in soup('a') if 'press releases'
                                             in a.text.lower()}
    print(f"{house_url}: {pr_links}")
    press_releases[house_url] = pr_links
```

Normally it is impolite to scrape a site freely like this. Most sites will have a *robots.txt* file that indicates how frequently you may scrape the site (and which paths you're not supposed to scrape), but since it's Congress we don't need to be particularly polite.

If you watch these as they scroll by, you'll see a lot of */media/press-releases* and *media-center/press-releases*, as well as various other addresses. One of these URLs is *https://jayapal.house.gov/media/press-releases*.

Remember that our goal is to find out which congresspeople have press releases mentioning "data." We'll write a slightly more general function that checks whether a page of press releases mentions any given term.

If you visit the site and view the source, it seems like there's a snippet from each press release inside a `<p>` tag, so we'll use that as our first attempt:

```
def paragraph_mentions(text: str, keyword: str) -> bool:
    """
    Returns True if a <p> inside the text mentions {keyword}
    """
    soup = BeautifulSoup(text, 'html5lib')
    paragraphs = [p.get_text() for p in soup('p')]

    return any(keyword.lower() in paragraph.lower()
               for paragraph in paragraphs)
```

Let's write a quick test for it:

```
text = """<body><h1>Facebook</h1><p>Twitter</p>"""
assert paragraph_mentions(text, "twitter")       # is inside a <p>
assert not paragraph_mentions(text, "facebook")  # not inside a <p>
```

At last we're ready to find the relevant congresspeople and give their names to the VP:

```
for house_url, pr_links in press_releases.items():
    for pr_link in pr_links:
        url = f"{house_url}/{pr_link}"
        text = requests.get(url).text

        if paragraph_mentions(text, 'data'):
            print(f"{house_url}")
            break  # done with this house_url
```

When I run this I get a list of about 20 representatives. Your results will probably be different.

If you look at the various "press releases" pages, most of them are paginated with only 5 or 10 press releases per page. This means that we only retrieved the few most recent press releases for each congressperson. A more thorough solution would have iterated over the pages and retrieved the full text of each press release.

# Using APIs

Many websites and web services provide *application programming interfaces* (APIs), which allow you to explicitly request data in a structured format. This saves you the trouble of having to scrape them!

## JSON and XML

Because HTTP is a protocol for transferring *text*, the data you request through a web API needs to be *serialized* into a string format. Often this serialization uses *JavaScript Object Notation* (JSON). JavaScript objects look quite similar to Python `dicts`, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2019,
  "topics" : [ "data", "science", "data science"] }
```

We can parse JSON using Python's `json` module. In particular, we will use its `loads` function, which deserializes a string representing a JSON object into a Python object:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2019,
                  "topics" : [ "data", "science", "data science"] }"""

# parse the JSON to create a Python dict
deserialized = json.loads(serialized)
```

```
assert deserialized["publicationYear"] == 2019
assert "data science" in deserialized["topics"]
```

Sometimes an API provider hates you and provides only responses in XML:

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

You can use Beautiful Soup to get data from XML similarly to how we used it to get data from HTML; check its documentation for details.

## Using an Unauthenticated API

Most APIs these days require that you first authenticate yourself before you can use them. While we don't begrudge them this policy, it creates a lot of extra boilerplate that muddies up our exposition. Accordingly, we'll start by taking a look at GitHub's API, with which you can do some simple things unauthenticated:

```
import requests, json

github_user = "joelgrus"
endpoint = f"https://api.github.com/users/{github_user}/repos"

repos = json.loads(requests.get(endpoint).text)
```

At this point `repos` is a `list` of Python `dicts`, each representing a public repository in my GitHub account. (Feel free to substitute your username and get your GitHub repository data instead. You do have a GitHub account, right?)

We can use this to figure out which months and days of the week I'm most likely to create a repository. The only issue is that the dates in the response are strings:

```
"created_at": "2013-07-05T02:02:28Z"
```

Python doesn't come with a great date parser, so we'll need to install one:

```
python -m pip install python-dateutil
```

from which you'll probably only ever need the `dateutil.parser.parse` function:

```
from collections import Counter
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
```

```
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

Similarly, you can get the languages of my last five repositories:

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["pushed_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

Typically we won't be working with APIs at this low "make the requests and parse the responses ourselves" level. One of the benefits of using Python is that someone has already built a library for pretty much any API you're interested in accessing. When they're done well, these libraries can save you a lot of the trouble of figuring out the hairier details of API access. (When they're not done well, or when it turns out they're based on defunct versions of the corresponding APIs, they can cause you enormous headaches.)

Nonetheless, you'll occasionally have to roll your own API access library (or, more likely, debug why someone else's isn't working), so it's good to know some of the details.

## Finding APIs

If you need data from a specific site, look for a "developers" or "API" section of the site for details, and try searching the web for "python <sitename> api" to find a library.

There are libraries for the Yelp API, for the Instagram API, for the Spotify API, and so on.

If you're looking for a list of APIs that have Python wrappers, there's a nice one from Real Python on GitHub.

And if you can't find what you need, there's always scraping, the last refuge of the data scientist.

# Example: Using the Twitter APIs

Twitter is a fantastic source of data to work with. You can use it to get real-time news. You can use it to measure reactions to current events. You can use it to find links related to specific topics. You can use it for pretty much anything you can imagine, just as long as you can get access to its data. And you can get access to its data through its APIs.

To interact with the Twitter APIs, we'll be using the Twython library (`python -m pip install twython`). There are quite a few Python Twitter libraries out there, but this is the one that I've had the most success working with. You are encouraged to explore the others as well!

## Getting Credentials

In order to use Twitter's APIs, you need to get some credentials (for which you need a Twitter account, which you should have anyway so that you can be part of the lively and friendly Twitter #datascience community).

Like all instructions that relate to websites that I don't control, these may become obsolete at some point but will hopefully work for a while. (Although they have already changed multiple times since I originally started writing this book, so good luck!)

Here are the steps:

1. Go to *https://developer.twitter.com/*.
2. If you are not signed in, click "Sign in" and enter your Twitter username and password.
3. Click Apply to apply for a developer account.
4. Request access for your own personal use.
5. Fill out the application. It requires 300 words (really) on why you need access, so to get over the limit you could tell them about this book and how much you're enjoying it.
6. Wait some indefinite amount of time.
7. If you know someone who works at Twitter, email them and ask them if they can expedite your application. Otherwise, keep waiting.
8. Once you get approved, go back to developer.twitter.com, find the "Apps" section, and click "Create an app."
9. Fill out all the required fields (again, if you need extra characters for the description, you could talk about this book and how edifying you're finding it).
10. Click CREATE.

Now your app should have a "Keys and tokens" tab with a "Consumer API keys" section that lists an "API key" and an "API secret key." Take note of those keys; you'll need them. (Also, keep them secret! They're like passwords.)

Don't share the keys, don't publish them in your book, and don't check them into your public GitHub repository. One simple solution is to store them in a *credentials.json* file that doesn't get checked in, and to have your code use `json.loads` to retrieve them. Another solution is to store them in environment variables and use `os.environ` to retrieve them.

### Using Twython

The trickiest part of using the Twitter API is authenticating yourself. (Indeed, this is the trickiest part of using a lot of APIs.) API providers want to make sure that you're authorized to access their data and that you don't exceed their usage limits. They also want to know who's accessing their data.

Authentication is kind of a pain. There is a simple way, OAuth 2, that suffices when you just want to do simple searches. And there is a complex way, OAuth 1, that's required when you want to perform actions (e.g., tweeting) or (in particular for us) connect to the Twitter stream.

So we're stuck with the more complicated way, which we'll try to automate as much as we can.

First, you need your API key and API secret key (sometimes known as the consumer key and consumer secret, respectively). I'll be getting mine from environment variables, but feel free to substitute in yours however you wish:

```
import os

# Feel free to plug your key and secret in directly
CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY")
CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET")
```

Now we can instantiate the client:

```
import webbrowser
from twython import Twython

# Get a temporary client to retrieve an authentication URL
temp_client = Twython(CONSUMER_KEY, CONSUMER_SECRET)
temp_creds = temp_client.get_authentication_tokens()
url = temp_creds['auth_url']

# Now visit that URL to authorize the application and get a PIN
print(f"go visit {url} and get the PIN code and paste it below")
webbrowser.open(url)
PIN_CODE = input("please enter the PIN code: ")

# Now we use that PIN_CODE to get the actual tokens
auth_client = Twython(CONSUMER_KEY,
                      CONSUMER_SECRET,
                      temp_creds['oauth_token'],
```

```
                      temp_creds['oauth_token_secret'])
final_step = auth_client.get_authorized_tokens(PIN_CODE)
ACCESS_TOKEN = final_step['oauth_token']
ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']

# And get a new Twython instance using them.
twitter = Twython(CONSUMER_KEY,
                  CONSUMER_SECRET,
                  ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET)
```

At this point you may want to consider saving the `ACCESS_TOKEN` and `ACCESS_TOKEN_SECRET` somewhere safe, so that next time you don't have to go through this rigmarole.

Once we have an authenticated `Twython` instance, we can start performing searches:

```
# Search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"]
    text = status["text"]
    print(f"{user}: {text}\n")
```

If you run this, you should get some tweets back like:

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization. So easy and Awesome.
```

This isn't that interesting, largely because the Twitter Search API just shows you whatever handful of recent results it feels like. When you're doing data science, more often you want a lot of tweets. This is where the Streaming API is useful. It allows you to connect to (a sample of) the great Twitter firehose. To use it, you'll need to authenticate using your access tokens.

In order to access the Streaming API with Twython, we need to define a class that inherits from `TwythonStreamer` and that overrides its `on_success` method, and possibly its `on_error` method:

```
from twython import TwythonStreamer

# Appending data to a global variable is pretty poor form
# but it makes the example much simpler
tweets = []
```

```
class MyStreamer(TwythonStreamer):
    def on_success(self, data):
        """
        What do we do when Twitter sends us data?
        Here data will be a Python dict representing a tweet.
        """
        # We only want to collect English-language tweets
        if data.get('lang') == 'en':
            tweets.append(data)
            print(f"received tweet #{len(tweets)}")

        # Stop when we've collected enough
        if len(tweets) >= 100:
            self.disconnect()

    def on_error(self, status_code, data):
        print(status_code, data)
        self.disconnect()
```

`MyStreamer` will connect to the Twitter stream and wait for Twitter to feed it data. Each time it receives some data (here, a tweet represented as a Python object), it passes it to the `on_success` method, which appends it to our `tweets` list if its language is English, and then disconnects the streamer after it's collected 1,000 tweets.

All that's left is to initialize it and start it running:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# starts consuming public statuses that contain the keyword 'data'
stream.statuses.filter(track='data')

# if instead we wanted to start consuming a sample of *all* public statuses
# stream.statuses.sample()
```

This will run until it collects 100 tweets (or until it encounters an error) and stop, at which point you can start analyzing those tweets. For instance, you could find the most common hashtags with:

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print(top_hashtags.most_common(5))
```

Each tweet contains a lot of data. You can either poke around yourself or dig through the Twitter API documentation.

In a non-toy project, you probably wouldn't want to rely on an in-memory `list` for storing the tweets. Instead you'd want to save them to a file or a database, so that you'd have them permanently.

## For Further Exploration

- pandas is the primary library that data science types use for working with—and, in particular, importing—data.
- Scrapy is a full-featured library for building complicated web scrapers that do things like follow unknown links.
- Kaggle hosts a large collection of datasets.