Lab 2

Application Programming Interfaces (APIs)

Two R packages that make working with APIs relatively easy are `httr` and `jsonlite`. If you don't have these packages already, you will need to install them with `install.packages()` before rendering this file or running the following code chunks. The corresponding Python libraries are `requests` and `json`. Make sure to install them using `py_install()`.

```
# install.packages(c("httr", "jsonlite"))
library(httr)
library(jsonlite)
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter()  masks stats::filter()
x purrr::flatten() masks jsonlite::flatten()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

```
library(reticulate)
use_condaenv("pstat134-234", required = TRUE) #specify here your
# py_install(c("requests", "json"))
```

Quick APIs (No Keys)

There are many, many public APIs available (and many ways to access them). In this class we'll cover how to make requests from several different APIs within R or Python. For a lengthy list of public APIs, I recommend this GitHub repository.

Open Notify API

This API is an open-source project meant to provide a simple interface for some fun data from NASA. You can get more information on it here: http://open-notify.org/ We'll start out with what is quite possibly one of the simplest APIs out there – one that will tell us how many human beings are currently in space (and, if known, on which craft).

Note that this data – the number of people currently in space – probably doesn't change all that often. You can imagine that the number of people in space last night was probably the same as the number in space right now. We aren't really taking advantage of one of the big strengths of an API yet (the ability to constantly access brand-new up-to-date information), but we will eventually.

Requests using R

We'll make our request using the `GET` function and specify the URL:

```
res <- GET("http://api.open-notify.org/astros.json")
res
```

```
Response [http://api.open-notify.org/astros.json]
  Date: 2025-04-14 02:52
  Status: 200
  Content-Type: application/json
  Size: 587 B
```

The output of the `GET` function is a list that contains all of the information returned by the API server. In other words, the `res` variable now contains the **response** of the API to our **request**. Calling `res` gives us a summary of the response. Notice that it prints the URL the request was sent to. We can also see the date and time the request was made, as well as the size of the response.

The status deserves special attention. "Status" refers to the success or failure of the API request, in the form of a number. The number that was returned tells you whether or not the request was a success and, if it failed, can detail some reasons why. The number `200` corresponds to a success. Information about other status codes is provided here.

The actual data is raw Unicode in the `res` list; we need to convert it into JSON format first:

```
res <- rawToChar(res$content)
```

And then from a JSON-structured character vector into a list, using the `jsonlite` package:

```
astro <- fromJSON(res)
```

Since this always results in a named list, you can call `names()` to see the categories of data inside:

```
names(astro)
```

```
[1] "people"  "number"  "message"
```

You can usually peruse the API documentation to discover what each of these components is (see here, for example). If you visit that link, you'll learn that `message` is the status of the API request, `number` is the current number of people in space, and information on the people themselves is stored in `people`:

```
astro$people
```

```
      craft                 name
1       ISS       Oleg Kononenko
2       ISS         Nikolai Chub
3       ISS Tracy Caldwell Dyson
4       ISS     Matthew Dominick
5       ISS      Michael Barratt
6       ISS        Jeanette Epps
7       ISS  Alexander Grebenkin
8       ISS        Butch Wilmore
9       ISS      Sunita Williams
10 Tiangong           Li Guangsu
11 Tiangong              Li Cong
12 Tiangong           Ye Guangfu
```

So there you have it; there are (at the time of writing) 12 people in space, 9 on the international space station and 3 on the Tiangong.

Requests using Python

We'll use the `requests` library, as described in the introduction.

```
import requests
import json
import pandas as pd

res = requests.get("http://api.open-notify.org/astros.json")
res
```

```
<Response [200]>
```

Again we see a status code; `200` tells us everything is good, so we can proceed. We can extract the data:

```
astro = res.json()
```

Similar to `names()` in R, we can use `.keys`:

```
astro.keys()
```

```
dict_keys(['people', 'number', 'message'])
```

And if we want to view the data frame of people currently in space:

```
pd.DataFrame(astro['people'])
```

```
       craft                  name
0        ISS        Oleg Kononenko
1        ISS          Nikolai Chub
2        ISS  Tracy Caldwell Dyson
3        ISS      Matthew Dominick
4        ISS       Michael Barratt
5        ISS         Jeanette Epps
6        ISS   Alexander Grebenkin
7        ISS         Butch Wilmore
8        ISS       Sunita Williams
9   Tiangong            Li Guangsu
10  Tiangong               Li Cong
11  Tiangong            Ye Guangfu
```

Exercises

The Official Joke API is another basic API – completely free and open-source, without requiring an API key.

1. Write a basic function called `tell_me_a_joke`. When a user calls the function, it should make an API request and return a random joke (both `setup` and `punchline`). Verify that your function works.
2. Add an argument to your function allowing you to specify the type of joke. Verify that your function works.

PokeAPI

This API proudly declares itself to provide "all the Pokemon data you'll ever need in one place," with "over 2.5 billion API calls each month." It is a quality free API that doesn't require the use of an API key; you can experiment with it in your browser at that link, or make requests from it using R or Python. You can also use one of the many wrappers created for this API if you choose. The (somewhat extensive) documentation is available here.

Let's try it out. We'll request all the information on a specific Pokemon using the `/pokemon/` endpoint.

```
res <- GET("https://pokeapi.co/api/v2/pokemon/37/")

poke <- fromJSON(rawToChar(res$content))
```

As you can see from `names`, we can access a pretty wide variety of information about this Pokemon, from its abilities to its stats to its forms, moves, and even cries:

```
names(poke)
```

```
 [1] "abilities"              "base_experience"
 [3] "cries"                  "forms"
 [5] "game_indices"           "height"
 [7] "held_items"             "id"
 [9] "is_default"             "location_area_encounters"
[11] "moves"                  "name"
[13] "order"                  "past_abilities"
[15] "past_types"             "species"
[17] "sprites"                "stats"
[19] "types"                  "weight"
```

Let's see what the name of number 37 is:

```
poke$name
```

```
[1] "vulpix"
```

37 is the Vulpix – a little fire fox Pokemon:

It may seem a little slow to extract information like this – one at a time, changing the ID number each time. Suppose we wanted to extract some basic information, like name, height, and weight, about all the Pokemon. How could we do that from here?

The first step would be to learn how many Pokemon there are. To do that, we can make an API request using the `/pokemon/` endpoint. We might assume `$count` reflects how many Pokemon were in the database, but in fact it doesn't; the actual number is 1,025.

```
res <- GET("https://pokeapi.co/api/v2/pokemon/")
fromJSON(rawToChar(res$content))
```

```
$count
[1] 1302

$`next`
[1] "https://pokeapi.co/api/v2/pokemon/?offset=20&limit=20"

$previous
NULL

$results
         name                                    url
1   bulbasaur  https://pokeapi.co/api/v2/pokemon/1/
```

```
2     ivysaur  https://pokeapi.co/api/v2/pokemon/2/
3    venusaur  https://pokeapi.co/api/v2/pokemon/3/
4  charmander  https://pokeapi.co/api/v2/pokemon/4/
5  charmeleon  https://pokeapi.co/api/v2/pokemon/5/
6   charizard  https://pokeapi.co/api/v2/pokemon/6/
7    squirtle  https://pokeapi.co/api/v2/pokemon/7/
8   wartortle  https://pokeapi.co/api/v2/pokemon/8/
9   blastoise  https://pokeapi.co/api/v2/pokemon/9/
10   caterpie https://pokeapi.co/api/v2/pokemon/10/
11    metapod https://pokeapi.co/api/v2/pokemon/11/
12 butterfree https://pokeapi.co/api/v2/pokemon/12/
13     weedle https://pokeapi.co/api/v2/pokemon/13/
14     kakuna https://pokeapi.co/api/v2/pokemon/14/
15   beedrill https://pokeapi.co/api/v2/pokemon/15/
16     pidgey https://pokeapi.co/api/v2/pokemon/16/
17  pidgeotto https://pokeapi.co/api/v2/pokemon/17/
18    pidgeot https://pokeapi.co/api/v2/pokemon/18/
19    rattata https://pokeapi.co/api/v2/pokemon/19/
20   raticate https://pokeapi.co/api/v2/pokemon/20/
```

We also might notice that we only received the first 20 Pokemon. That's the default for this API. If we want to extract more than that, we have to make a small change.

Query Parameters

So far, our `GET` requests have been fairly straightforward; there was no additional information that we needed to provide. Many APIs either require or allow you to specify a little more information in API calls; additional information we specify are called **query parameters**.

If you're accessing the API from within R, you can use the `query` argument, as shown, and provide a named list of query parameters. For this example, let's set the parameter `limit` to 30, indicating that we want to retrieve data on 30 Pokemon, rather than the default of 20.

```
res <- GET("https://pokeapi.co/api/v2/pokemon/",
           query = list(limit = 30))
fromJSON(rawToChar(res$content))
```

```
$count
[1] 1302

$`next`
[1] "https://pokeapi.co/api/v2/pokemon/?offset=30&limit=30"
```

```
$previous
NULL

$results
         name                                    url
1   bulbasaur  https://pokeapi.co/api/v2/pokemon/1/
2     ivysaur  https://pokeapi.co/api/v2/pokemon/2/
3    venusaur  https://pokeapi.co/api/v2/pokemon/3/
4  charmander  https://pokeapi.co/api/v2/pokemon/4/
5  charmeleon  https://pokeapi.co/api/v2/pokemon/5/
6   charizard  https://pokeapi.co/api/v2/pokemon/6/
7    squirtle  https://pokeapi.co/api/v2/pokemon/7/
8   wartortle  https://pokeapi.co/api/v2/pokemon/8/
9   blastoise  https://pokeapi.co/api/v2/pokemon/9/
10   caterpie https://pokeapi.co/api/v2/pokemon/10/
11    metapod https://pokeapi.co/api/v2/pokemon/11/
12 butterfree https://pokeapi.co/api/v2/pokemon/12/
13     weedle https://pokeapi.co/api/v2/pokemon/13/
14     kakuna https://pokeapi.co/api/v2/pokemon/14/
15   beedrill https://pokeapi.co/api/v2/pokemon/15/
16     pidgey https://pokeapi.co/api/v2/pokemon/16/
17  pidgeotto https://pokeapi.co/api/v2/pokemon/17/
18    pidgeot https://pokeapi.co/api/v2/pokemon/18/
19    rattata https://pokeapi.co/api/v2/pokemon/19/
20   raticate https://pokeapi.co/api/v2/pokemon/20/
21    spearow https://pokeapi.co/api/v2/pokemon/21/
22     fearow https://pokeapi.co/api/v2/pokemon/22/
23      ekans https://pokeapi.co/api/v2/pokemon/23/
24      arbok https://pokeapi.co/api/v2/pokemon/24/
25    pikachu https://pokeapi.co/api/v2/pokemon/25/
26     raichu https://pokeapi.co/api/v2/pokemon/26/
27  sandshrew https://pokeapi.co/api/v2/pokemon/27/
28  sandslash https://pokeapi.co/api/v2/pokemon/28/
29  nidoran-f https://pokeapi.co/api/v2/pokemon/29/
30   nidorina https://pokeapi.co/api/v2/pokemon/30/
```

For APIs that use additional query parameters, you can simply add them to that named list. In Python we can set up a dictionary of query parameters in the form of key-value pairs:

```python
parameters = {
  "limit": 30
```

```
}
res = requests.get("https://pokeapi.co/api/v2/pokemon/", params=parameters)
poke = res.json()
pd.DataFrame(poke['results'])
```

	name	url
0	bulbasaur	https://pokeapi.co/api/v2/pokemon/1/
1	ivysaur	https://pokeapi.co/api/v2/pokemon/2/
2	venusaur	https://pokeapi.co/api/v2/pokemon/3/
3	charmander	https://pokeapi.co/api/v2/pokemon/4/
4	charmeleon	https://pokeapi.co/api/v2/pokemon/5/
5	charizard	https://pokeapi.co/api/v2/pokemon/6/
6	squirtle	https://pokeapi.co/api/v2/pokemon/7/
7	wartortle	https://pokeapi.co/api/v2/pokemon/8/
8	blastoise	https://pokeapi.co/api/v2/pokemon/9/
9	caterpie	https://pokeapi.co/api/v2/pokemon/10/
10	metapod	https://pokeapi.co/api/v2/pokemon/11/
11	butterfree	https://pokeapi.co/api/v2/pokemon/12/
12	weedle	https://pokeapi.co/api/v2/pokemon/13/
13	kakuna	https://pokeapi.co/api/v2/pokemon/14/
14	beedrill	https://pokeapi.co/api/v2/pokemon/15/
15	pidgey	https://pokeapi.co/api/v2/pokemon/16/
16	pidgeotto	https://pokeapi.co/api/v2/pokemon/17/
17	pidgeot	https://pokeapi.co/api/v2/pokemon/18/
18	rattata	https://pokeapi.co/api/v2/pokemon/19/
19	raticate	https://pokeapi.co/api/v2/pokemon/20/
20	spearow	https://pokeapi.co/api/v2/pokemon/21/
21	fearow	https://pokeapi.co/api/v2/pokemon/22/
22	ekans	https://pokeapi.co/api/v2/pokemon/23/
23	arbok	https://pokeapi.co/api/v2/pokemon/24/
24	pikachu	https://pokeapi.co/api/v2/pokemon/25/
25	raichu	https://pokeapi.co/api/v2/pokemon/26/
26	sandshrew	https://pokeapi.co/api/v2/pokemon/27/
27	sandslash	https://pokeapi.co/api/v2/pokemon/28/
28	nidoran-f	https://pokeapi.co/api/v2/pokemon/29/
29	nidorina	https://pokeapi.co/api/v2/pokemon/30/

Making Many API Calls

From that lengthy list, we can extract something very useful – the list of ID URLs corresponding to all the Pokemon. Let's do that:

```
res <- GET("https://pokeapi.co/api/v2/pokemon/",
           query = list(limit = 1025))
big_poke <- fromJSON(rawToChar(res$content))
big_poke$results$url %>% head()
```

```
[1] "https://pokeapi.co/api/v2/pokemon/1/"
[2] "https://pokeapi.co/api/v2/pokemon/2/"
[3] "https://pokeapi.co/api/v2/pokemon/3/"
[4] "https://pokeapi.co/api/v2/pokemon/4/"
[5] "https://pokeapi.co/api/v2/pokemon/5/"
[6] "https://pokeapi.co/api/v2/pokemon/6/"
```

If we want to extract the name, height, and weight of all the Pokemon, as discussed before, we could write a loop:

```
pokelist <- NULL
for(i in 1:1025){
  url <- big_poke$results$url[i]
  res = GET(url)
  poke = fromJSON(rawToChar(res$content))
  pokelist[[i]] = data.frame(name = poke$name,
                             height = poke$height,
                             weight = poke$weight)
}

do.call(rbind, pokelist) %>%
  head()
```

```
        name height weight
1  bulbasaur      7     69
2    ivysaur     10    130
3   venusaur     20   1000
4 charmander      6     85
5 charmeleon     11    190
6  charizard     17    905
```

Or in Python, as follows (I've set the following code chunk not to evaluate, since it takes a few minutes to run the process of extracting data on all 1,025 and I'd rather avoid doing it twice):

```
poke_list = []

big_poke = requests.get("https://pokeapi.co/api/v2/pokemon/?limit=1025").json()

for i in range(1025):
    url = big_poke['results'][i]['url']
    res = requests.get(url)

    # Check if the request was successful
    if res.status_code == 200:
        poke = res.json()
        # Append the data as a DataFrame to the list
        poke_list.append(pd.DataFrame({
            'name': [poke['name']],
            'height': [poke['height']],
            'weight': [poke['weight']]
        }))
    else:
        print(f"Failed to retrieve data for URL {url}")

poke_df = pd.concat(poke_list, ignore_index=True)

print(poke_df)
```

Note that this process is essentially taking semi-structured data (in JSON format) and turning it into structured data! We could then use this structured data in any number of machine learning models or create visualizations to our heart's content.

Exercises

3. For each of 1,025 Pokemon, extract their (a) name, (b) height, (c) weight, (d) primary type, (e) hp, (f) attack, (g) defense, (h) special attack, and (i) special defense.
4. Create a scatterplot of the relationship between Pokemon height and weight. Describe what you notice.

APIs with Keys

A number of APIs require a key to access them. This is often done for many reasons – for security, or to track user data, or to restrict the number of API calls per user per day (or other period of time). It's difficult to precisely describe the process of accessing APIs with keys,

because the way you generate your key (essentially a password) may vary from application to application, and because everyone's key will differ.

Some APIs will ask you to pay a fee to use the API or obtain a key. You won't be required to use a paid API for this course, but you may do so if you choose.

API Wrappers

R and Python developers have created what are called **wrappers** for many of the more commonly used APIs. If you have an API in particular that you'd like to use, I recommend first searching for wrappers for that API. An API "wrapper" is a tool that allows you to use the API more easily, without having to wade through so much documentation and debugging, by simply using your pre-existing experience with R or Python code to interact with the API. They are called wrappers because they consist of code that is "wrapped" around the underlying API calls.

Spotify API

The Spotify API is very popular and free for public use, although it does require you to set up a Client ID, Client Secret, and what is called an access token. You can find a guide to getting started with the API here.

To use the API, you have to log in to the Spotify Developer Dashboard and create an app. (Although this sounds like a lot of work, it really means you simply have to click a single button.) You can name the app anything you like; for the "redirect URL," I recommend `http://localhost:3000`. Then you need to request an access token. To do that, go to your Dashboard. Click on the name of the app you just created; then click on Settings. You'll want to make a note of your *Client ID* and *Client Secret*; you'll need those to get an access token. The ID and secret are linked to your account and won't usually change. The access token, however, is only valid for a certain period of time (usually one hour), and will need to be regenerated after that time elapses to continue using the API.

It is much easier to access the Spotify API with a wrapper. My favorite wrapper for this API is `spotifyr`. Let's set that package up now. Rather than installing it with `install_packages`, I recommend installing the R package `devtools` (or "developer tools") and then using its function `install_github` to download the most recent version of `spotifyr` from its GitHub page.

```
# install.packages("devtools")
library(devtools)
```

```
Loading required package: usethis
```

```
# devtools::install_github('charlie86/spotifyr')
library(spotifyr)
```

In the following code, replace the `########` with your Client ID and Client Secret, respectively.

```
Sys.setenv(SPOTIFY_CLIENT_ID = '####################')
Sys.setenv(SPOTIFY_CLIENT_SECRET = '####################')

# access_token <- get_spotify_access_token()
```

The API documentation for Spotify is extensive. You can find it here. You can also find more information on the various functions in the wrapper package that are available, as well as see some examples of visualizations generated using the Spotify API.

To get information about the audio features of an artist:

```
# results <- get_artist_audio_features("arctic monkeys")
# results
```

Exercises

Use the Spotify API to extract the audio features for one of your favorite artists. Then answer the following questions.

5. What is their most common key and time signature? What about least common? (Try using tables and/or bar charts to answer this.)
6. What are the summary statistics for your favorite artist's (a) danceability, (b) energy, (c) loudness, (d) acousticness, and (e) energy?