Extracting data from text and geocoding to study officer-involved shootings

Greg Ridgeway

2026-08-05

Table of contents

1 Introduction

In this section, we are going to explore officer-involved shootings (OIS) in Philadelphia. The Philadelphia Police Department posts a lot of information about officer-involved shootings online going back to 2016. Have a look at their OIS webpage. While a lot of information has been posted to the webpage, more information is buried in text linked to each of the incidents. In order for us to explore these data, we are going to scrape the basic information from the webpage, have R dig into the text for dates, clean up addresses using regular expressions, geocode the addresses to latitude/longitude with the ArcGIS geocoder (using JSON), and then make maps describing the shootings.

Start by loading the packages we will need.

```
library(lubridate)
library(jsonlite)
library(sf)
library(leaflet)
library(dplyr)
library(tidyr)
library(foreach)

library(chromote) # steer Chrome from R
library(rvest)    # helpful web scraping tools
library(purrr)    # for pluck()
```

2 Scraping the OIS data

We can run `scan()`, as we did previously, on the PPD OIS webpage and regex our way to a data frame with the data elements that we want to store.

```
# pull the whole OIS page
a <- scan("https://www.phillypolice.com/accountability/ois/",
          what="", sep="\n")
# search for the start of a table
i <- grep("<tbody>", a)
a[i] |> substring(1, 500)
```

```
[1] "</script><script id=\"909ad40a908c8ae215a0d4f4934a910e-1\"
 ↪  type=\"nitropack/inlinescript\" class=\"nitropack-inline-script\">const
 ↪  fixSummary=()=>{document.querySelectorAll(\".cmplz-category
 ↪  summary[tabindex]\").forEach(e=>{e.removeAttribute(\"tabindex\")})};wind
 ↪  ow.addEventListener(\"load\",fixSummary);const observer=new
 ↪  MutationObserver(fixSummary);observer.observe(document.body,{childList:t
 ↪  rue,subtree:true});</script><script id=\"nitro-nojs\"
 ↪  type=\"nitropack/inlinescript\" class=\"nitropack-inline-script\">add"
```

```
# search for start of table <tbody> in a[i]
iStart <- gregexpr("<tbody>",  a[i]) |> unlist()
a[i] |> substring(iStart, iStart+500)
```

```
[1] "<tbody><tr><td><a
 ↪  href=\"https://www.phillypolice.com/ois/26-16/\">26-16</a></td><td>2000
 ↪  block of North 54th Street</td><td>2026</td><td>Killed</td><td>N/A</td><
 ↪  td>Yes</td><td>Pending</td><td></td> <td><p
 ↪  style=\"text-align:justify\">On 6/13/2026 at approximately 10:30 p.m.,
 ↪  officers assigned to the 19th District responded to a report of a person
 ↪  with a weapon and shots fired in the area of 54th and Arlington Streets.
 ↪  Upon arrival, officers began investigating the incident and canvassing
 ↪  the area fo"
```

Buried in the HTML code are the entries in the table. Even though all the information does not appear on the main webpage, it is in the HTML. This is not always the case. Some pages dynamically generate information as the user interacts with a page and the page elements. We are going to use this as an opportunity to learn more advanced web scraping methods.

Instead of using `scan()`, we are going to use the `chromote` package to open a hidden Chrome browser that we can control remotely from R. For this to work you do need to have a Chrome browser installed on your computer. From R, we can simulate user actions, like typing a URL in the address bar, selecting elements on the page, and clicking buttons.

The *page source* is the static HTML the browser receives from the server (e.g. what `scan()` would capture). The *Document Object Model* (DOM) is the live, in-memory object the browser constructs from that source and then updates as JavaScript runs. When we interact with a webpage, like clicking, we are interacting with the DOM.

Start by initiating a new hidden Chrome browser.

```
browser <- ChromoteSession$new()
# Good idea to setup the browser to close when we exit R
#    When creating these notes it runs too soon, so commented out here
# on.exit(browser$close(), add = TRUE)
```

Nothing will be visible on your screen at this point. If you want to watch what the browser is doing in response to R actions, you can use `view()`, but this is not necessary.

```
browser$view()
```

Let's tell our hidden browser to navigate over to the PPD OIS webpage. Sometimes pages take a moment to load. `go_to()` waits for the navigation to finish before allowing R to move on to the next line of code.

```
browser$go_to("https://www.phillypolice.com/accountability/ois/")
```

I will grab a screenshot to show that everything is working so far.

```
# get page size
pageSize <- browser$Page$getLayoutMetrics()$contentSize
browser$screenshot(cliprect = c(left=0, top=0,
                                width=pageSize$width, height=800))
```

Let's take a little cybersecurity detour for a moment. If you right-click on a webpage and select "Inspect," then you can see the DOM for the page. Right-click and Inspect the OIS table and you will find that the table's ID is `#data-table-ois`. From R I can tell the DOM to change elements on the page. To demonstrate, I will change the address in the 25th row to 3718 Locust Walk. `tr:nth-child(25)` selects the 25th row and `td:nth-child(2)` selects the column. Setting `.textContent = '3718 Locust Walk'` changes the cell text in the DOM immediately.

```
browser$Runtime$evaluate("document.querySelector('#data-table-ois tr:nth-child(25) td:nth-ch
```

```
$result
$result$type
[1] "string"

$result$value
[1] "3718 Locust Walk"
```

Now let's look at the page

```
pageSize <- browser$Page$getLayoutMetrics()$contentSize
browser$screenshot("screenshot2.png",
             cliprect = c(left=0, top=pageSize$height-800,
                          width=pageSize$width, height=800))
```

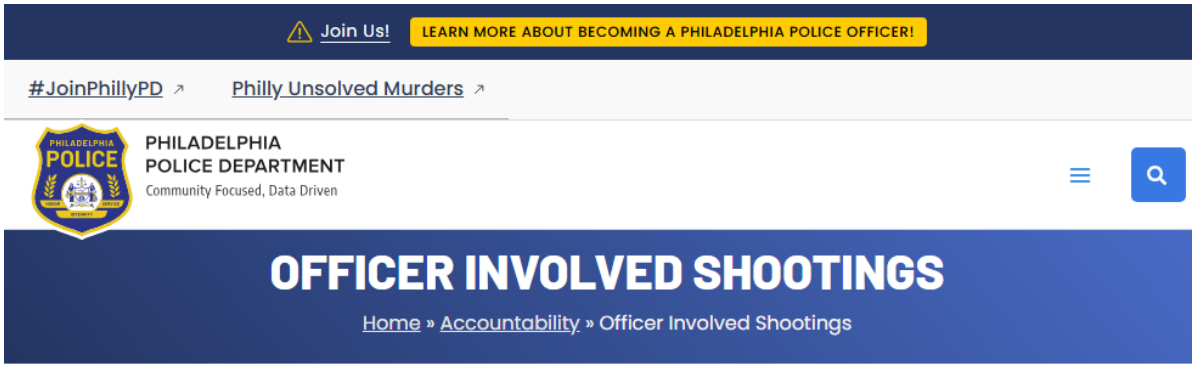

Figure 1: Screenshot from the hidden Chrome browser view of the PPD OIS

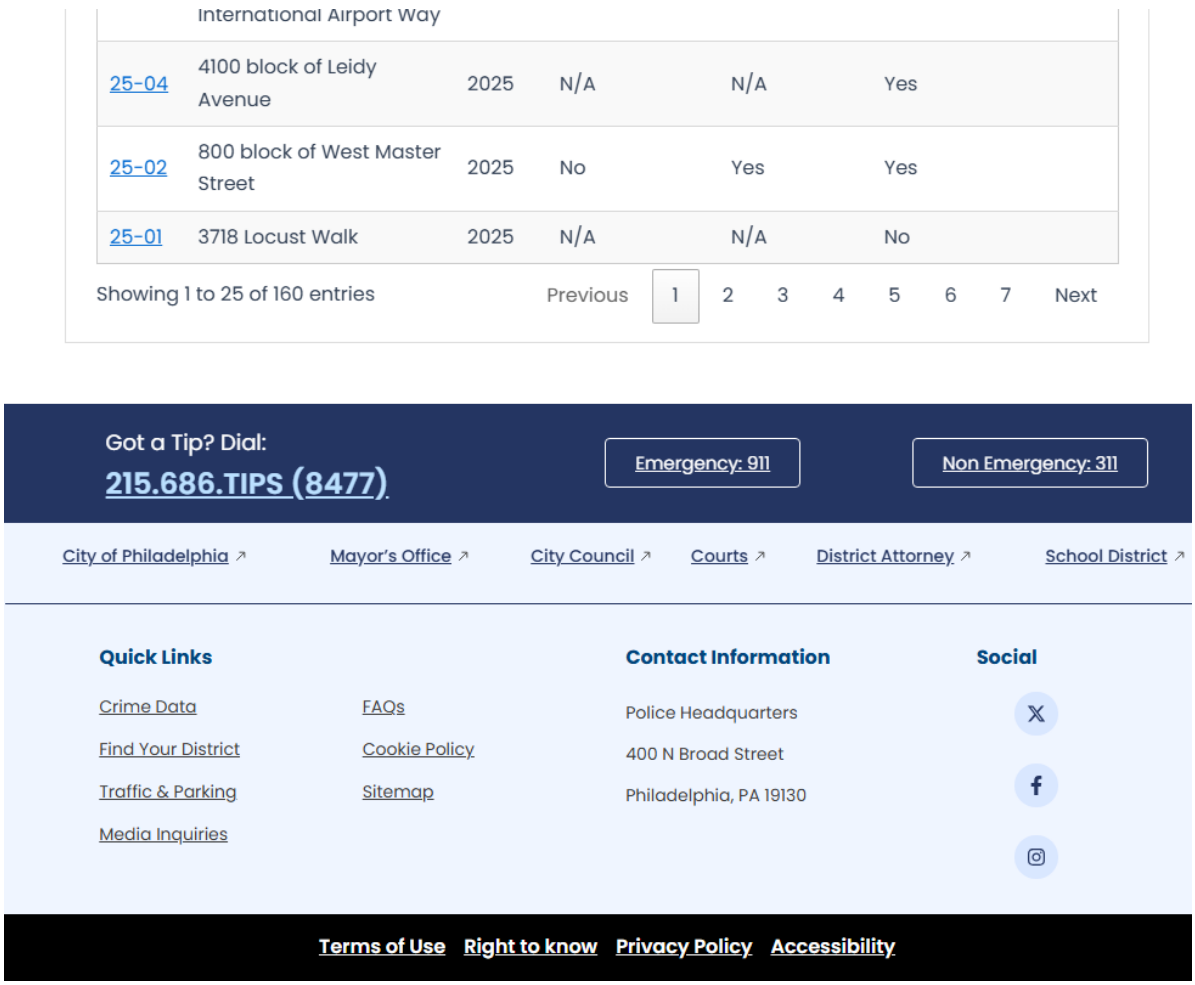

Figure 2: Screenshot of the manipulated PPD OIS webpage

This is how the "refund/overpayment scam" works. A fraudster gets the victim to view their account webpage and to give the fraudster some level of remote control. The fraudster then manipulates the DOM to give the impression that they have mistakenly sent the victim $10,000 instead of $1,000. The fraudster then convinces the victim to send the $9,000 difference even though no money was ever given to them.

Rather than try to scam someone, let's get back to the business of pulling all the OIS data into R. I will first get the ID of the main HTML from the DOM. Then I will read in the HTML associated with that ID.

```
# get the ID of the main HTML
html <- browser$DOM$getDocument()$root$nodeId
html
```

```
[1] 16
```

```
# pull in raw html source code, like view page source or scan()
page_html <- browser$DOM$getOuterHTML(nodeId = html)$outerHTML
# page_html is one long string of HTML code
#   look at a few lines
page_html |>
   substring(1,1500)
```

```
[1] "<!DOCTYPE html><html lang=\"en-US\"><head><meta
 ↪ http-equiv=\"origin-trial\" content=\"A7vZI3v+Gz7JfuRolKNM4Aff6zaGuT7XOm
 ↪ f3wtoZTnKv6497cVMnhy03KDqX7kBz/q/iidW7srW31oQbBt4VhgoAAACUeyJvcmlnaW4iOi
 ↪ JodHRwczovL3d3dy5nb29nbGUuY29tOjQ0MyIsImZlYXR1cmUiOiJEaXNhYmxlVGhpcmRQYX
 ↪ J0eVN0b3JhZ2VQYXJ0aXRpb25pbmczIiwiZXhwaXJ5IjoxNzU3OTgwODAwLCJpc1N1YmRvbW
 ↪ FpbiI6dHJ1ZSwiaXNUaGlyZFBhcnR5Ijp0cnVlfQ==\"><script async=\"\"
 ↪ src=\"//cse.google.com/adsense/search/async-ads.js\"></script><script
 ↪ type=\"text/javascript\" async=\"\" charset=\"utf-8\"
 ↪ src=\"https://www.gstatic.com/recaptcha/releases/w_Yb7dGGXaKesJ7BMiqFJqB
 ↪ G/recaptcha__en.js\" crossorigin=\"anonymous\"
 ↪ integrity=\"sha384-atSB8sX+Hj1naJF5TsMaubkpaq785kS8XAjUQzYyAyu7TjxTj5/FB
 ↪ 9LC77/4nt6m\"></script><script>if(navigator.userAgent.match(/MSIE|Intern
 ↪ et Explorer/i)||navigator.userAgent.match(/Trident\\/7\\..*?rv:11/i)){let
 ↪ e=document.location.href;if(!e.match(/[?&]nonitro/)){if(e.indexOf(\"?\")
 ↪ ==-1){if(e.indexOf(\"#\")==-1){document.location.href=e+\"?nonitro=1\"}e
 ↪ lse{document.location.href=e.replace(\"#\",\"?nonitro=1#\")}}else{if(e.i
 ↪ ndexOf(\"#\")==-1){document.location.href=e+\"&nonitro=1\"}else{document
 ↪ .location.href=e.replace(\"#\",\"&nonitro=1#\")}}}}</script><link
 ↪ rel=\"preconnect\" href=\"https://www.phillypolice.com\"><link
 ↪ rel=\"preconnect\" href=\"https://www.google.com\"><link
 ↪ rel=\"preconnect\" href=\"https://cdn-ilcomil.nitrocdn.com\"><meta
 ↪ charset=\"UTF-8\"><meta name=\"viewport\" content=\"width=device-width,
 ↪ initial-scale=1\"><meta name=\"robots\" content=\"index, follow,
 ↪ max-image-preview:large,"
```

Now convert all that HTML code into an HTML document object with which R knows how to work.

```
page <- read_html(page_html)
page
```

```
{html_document}
<html lang="en-US">
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8
 ↪ ...
[2] <body data-cmplz="1" itemtype="https://schema.org/WebPage" itemscope="ite
 ↪ ...
```

In this page we are looking for any `table` elements.

```
page |> html_elements("table")
```

```
{xml_nodeset (4)}
```

```
[1] <table cellspacing="0" cellpadding="0" role="presentation" class="gsc-sea
 ↪  ...
[2] <table cellspacing="0" cellpadding="0" role="presentation" id="gs_id50" c
 ↪  ...
[3] <table id="data-table-ois" class="display dataTable no-footer" aria-descr
 ↪  ...
[4] <table cellspacing="0" cellpadding="0" role="presentation" class="gstl_50
 ↪  ...
```

There appear to be four tables on the page, but I noticed that the third one of these has `id="data-table-ois"`. That must be the one we want. Let's extract it by name and convert it to an R data frame object.

```
# extract the data-table-ois by name
page |>
   html_elements("table#data-table-ois") |>
   html_table() |>
   data.frame()
```

```
   Title                                Location Year Subject.Injury
1  26-16          2000 block of North 54th Street 2026         Killed
2  26-14          2800 block of Kensington Avenue 2026        Wounded
3  26-12             6900 block of Lawnton Avenue 2026            N/A
4  26-11             5400 block of Webster Street 2026         Killed
5  26-08                3400 block of Hess Street 2026            N/A
6  26-06      1400 block of North Robinson Street 2026            N/A
7  26-05         4100 block of North Broad Street 2026             No
8  26-03           4800 block of Blakiston Street 2026            N/A
9  26-02               2800 block of Sebring Road 2026            N/A
10 26-01        400 block of East Rockland Street 2026        Unknown
11 25-17          5100 block of North 10th Street 2025         Killed
12 25-16               2000 block of Simon Street 2025            N/A
13 25-15      2900 block of North Lawrence Street 2025         Killed
14 25-14          2600 block of South 21st Street 2025            N/A
15 25-13        100 block of West Somerset Street 2025         Killed
16 25-12           300 block of North 65th Street 2025            N/A
17 25-11             4600 block of Roosevelt Blvd 2025         Killed
18 25-10               4100 block of Ogden Street 2025            N/A
19 25-09        4600 block of Roosevelt Boulevard 2025         Killed
20 25-08               1600 block of Moore Street 2025        Wounded
21 25-06              2800 block of Jasper Street 2025            N/A
22 25-05 1 Philadelphia International Airport Way 2025         Killed
23 25-04               4100 block of Leidy Avenue 2025            N/A
24 25-02          800 block of West Master Street 2025             No
```

```
25 25-01                          3718 Locust Walk 2025            N/A
   Subject.Arrested Officer.Injury
1                N/A            Yes
2                Yes             No
3                N/A             No
4                N/A             No
5                N/A             No
6                N/A             No
7                Yes             No
8                N/A             No
9                N/A             No
10                No             No
11               N/A             No
12               N/A             No
13               N/A             No
14               N/A             No
15               N/A             No
16               N/A            Yes
17               N/A             No
18               N/A             No
19               N/A            Yes
20               Yes             No
21               N/A            Yes
22               N/A             No
23               N/A            Yes
24               Yes            Yes
25               N/A             No
```

Looks like we got all 25 of the OIS incidents from the first page. Note that it still has our manipulated address for the 25th OIS incident. Shortly we will refresh the webpage to scrape all of the incidents and that will reset the addresses to their original values. On the PPD's page, each OIS's ID has a hyperlink that gets us more detailed information about each incident. Extract those URLs by looking for elements in the table body with an `<a>` HTML tag. Eventually we will store these URLs so that we can scrape them for the OIS details.

```
page |>
   html_elements("table#data-table-ois") |>
   html_elements("tbody a") |>
   html_attr("href")
```

```
 [1] "https://www.phillypolice.com/ois/26-16/"
 [2] "https://www.phillypolice.com/ois/26-14/"
 [3] "https://www.phillypolice.com/ois/26-12/"
 [4] "https://www.phillypolice.com/ois/26-11/"
```

```
 [5] "https://www.phillypolice.com/ois/26-08/"
 [6] "https://www.phillypolice.com/ois/26-06/"
 [7] "https://www.phillypolice.com/ois/26-05/"
 [8] "https://www.phillypolice.com/ois/26-03/"
 [9] "https://www.phillypolice.com/ois/26-02/"
[10] "https://www.phillypolice.com/ois/ps26-01/"
[11] "https://www.phillypolice.com/ois/26-17/"
[12] "https://www.phillypolice.com/ois/25-16/"
[13] "https://www.phillypolice.com/ois/25-15/"
[14] "https://www.phillypolice.com/ois/25-14/"
[15] "https://www.phillypolice.com/ois/25-13/"
[16] "https://www.phillypolice.com/ois/25-12/"
[17] "https://www.phillypolice.com/ois/ps25-11/"
[18] "https://www.phillypolice.com/ois/25-10/"
[19] "https://www.phillypolice.com/ois/25-09/"
[20] "https://www.phillypolice.com/ois/25-08/"
[21] "https://www.phillypolice.com/ois/25-06/"
[22] "https://www.phillypolice.com/ois/25-05/"
[23] "https://www.phillypolice.com/ois/25-04/"
[24] "https://www.phillypolice.com/ois/25-02/"
[25] "https://www.phillypolice.com/ois/ps25-01/"
```

With that we have been able to get all of the information for the first 25 OIS incidents. Now we have to "click" the Next button to get to the next set of 25 OIS incidents. First, I will check to see if it is disabled. Since we are still looking at the first page it will not be disabled, but when we are looking at the final page of OIS incidents then it will be disabled. To figure out the name of that Next button, in my browser I right-clicked on the Next button, selected Inspect, and then reviewed the resulting HTML code.

```
<a class="paginate_button next" aria-controls="data-table-ois" role="link" data-dt-idx="next"
```

I see that the HTML tag is `<a>` and its classes are `paginate_button` and `next`. So `a.paginate_button.next` will search the HTML code for an `<a>` element with both a `paginate_button` and `next` class.

```
# check whether the Next button is disabled
browser$Runtime$evaluate(
  expression = 'document.
                  querySelector("a.paginate_button.next").
                  classList.
                  contains("disabled");') |>
   pluck("result","value")
```

```
[1] FALSE
```

On a single page there might be several such buttons. A more robust method of finding the right button is to query it by name, in this case `data-table-ois_next`.

```
# check whether the Next button is disabled
browser$Runtime$evaluate(
  expression = 'document.
                  querySelector("#data-table-ois_next").
                  classList.
                  contains("disabled");') |>
   pluck("result","value")
```

```
[1] FALSE
```

Either method gives the same response: the button is not disabled. That means we can click it to move on to the next page.

```
# Click the "Next" button
browser$Runtime$evaluate(
  expression = 'document.
                  querySelector("#data-table-ois_next").
                  click();')
```

Armed with all the skills to navigate and extract key information from this site, we can wrap these ideas in a `while()` loop that will keep scraping OIS data as long as the Next button is active.

```
# reset page to beginning
browser$go_to("https://www.phillypolice.com/accountability/ois/")
# give the browser additional time to fully load page
Sys.sleep(5)
# you can force scroll to bottom if you have the view open
#   browser$Runtime$evaluate("window.scrollTo(0, document.body.scrollHeight);")

# store results from each page in ois (a list of data frames)
ois <- list()
isFinished <- FALSE
iPage <- 1
while(!isFinished)
{
  message(paste0("Read HTML from page ",iPage))
  # get ID of main HTML
```

```
  html <- browser$DOM$getDocument()$root$nodeId
  # raw html source code
  page_html <- browser$DOM$getOuterHTML(nodeId = html)$outerHTML
  # parse the HTML into a structured document
  page <- read_html(page_html)

  # Pull the table
  oisTable <- page |>
    html_elements("table#data-table-ois")

  # extract the table text to a data frame
  ois[[iPage]] <- oisTable |>
     html_table() |>
     data.frame()
  message("Read in ", nrow(ois[[iPage]]), " rows")

  # extract the URLs from each row
  oisURLs <- oisTable |>
    html_elements("tbody a") |>
    html_attr("href")
  ois[[iPage]]$url <- oisURLs

  # is "Next" button disabled?
  isFinished <- browser$Runtime$evaluate(
    expression = 'document.
                    querySelector("#data-table-ois_next").
                    classList.
                    contains("disabled");') |>
     pluck("result","value")

  if(!isFinished)
  {
    # Click the "Next" button
    browser$Runtime$evaluate(
      expression = 'document.
                      querySelector("#data-table-ois_next").
                      click();')
    Sys.sleep(2)
    iPage <- iPage + 1
  }
}
```

```
Read HTML from page 1
```

```
Read in 25 rows

Read HTML from page 2

Read in 25 rows

Read HTML from page 3

Read in 25 rows

Read HTML from page 4

Read in 25 rows

Read HTML from page 5

Read in 25 rows

Read HTML from page 6

Read in 25 rows

Read HTML from page 7

Read in 10 rows
```

```
# close the browser
browser$close()
```

```
[1] TRUE
```

```
# combine pages into single data frame
#   drop year since we are going to extract the actual date later
ois <- bind_rows(ois) |>
  select(-Year) |>
  rename(id = Title,
         location  = Location,
         subInjury = Subject.Injury,
         subArrest = Subject.Arrested,
         offInjury = Officer.Injury)
```

Let's check that it read everything in.

```
head(ois)
```

```
     id                              location subInjury subArrest offInjury
1 26-16     2000 block of North 54th Street    Killed       N/A       Yes
2 26-14     2800 block of Kensington Avenue   Wounded       Yes        No
3 26-12        6900 block of Lawnton Avenue       N/A       N/A        No
4 26-11        5400 block of Webster Street    Killed       N/A        No
5 26-08           3400 block of Hess Street       N/A       N/A        No
6 26-06 1400 block of North Robinson Street       N/A       N/A        No
                                       url
1 https://www.phillypolice.com/ois/26-16/
2 https://www.phillypolice.com/ois/26-14/
3 https://www.phillypolice.com/ois/26-12/
4 https://www.phillypolice.com/ois/26-11/
5 https://www.phillypolice.com/ois/26-08/
6 https://www.phillypolice.com/ois/26-06/
```

```
tail(ois)
```

```
       id                                  location subInjury
155 16-11              Unit Block of Salford street        No
156 16-10 5700 N. Park street/5700 N. Broad street    Killed
157 16-07      3100 Block of north Carlisle Street   Wounded
158 16-03                 Near Loudon and D streets        No
159 16-02           100 block of north 55th Street        No
160 16-01           300 block of south 60th street   Wounded
              subArrest        offInjury
155 Yes (both offenders) P/O #1 (wounded)
156                  Yes               No
157                  Yes               No
158      2 of 4 arrested               No
159                  Yes               No
160                  Yes          Wounded
                                      url
155 https://www.phillypolice.com/ois/16-11/
156 https://www.phillypolice.com/ois/16-10/
157 https://www.phillypolice.com/ois/16-07/
158 https://www.phillypolice.com/ois/16-03/
159 https://www.phillypolice.com/ois/16-02/
160 https://www.phillypolice.com/ois/16-01/
```

3 Extracting OIS incident details

Now let's dig into the details of the incident, starting with the first OIS. The hyperlink in the very first OIS incident points to the page https://www.phillypolice.com/ois/26-16/. Let's read in the incident details from that page. In your browser, if you right-click and Inspect the text description of the incident, then you will find that the text has the id `ois-content-area`. We can grab that by name.

```
read_html(ois$url[1]) |>
   html_element("div.ois-content-area") |>
   html_text() |>
   trimws() # trim whitespace
```

```
[1] "On 6/13/2026 at approximately 10:30 p.m., officers assigned to the 19th
 ↪ District responded to a report of a person with a weapon and shots fired
 ↪ in the area of 54th and Arlington Streets. Upon arrival, officers began
 ↪ investigating the incident and canvassing the area for witnesses and
 ↪ evidence. During the course of that investigation, an adult male
 ↪ approached an area on 2000 N. 54th Street which was being investigated by
 ↪ officers where a vehicle had been struck by gunfire. A confrontation
 ↪ developed between the male and police personnel. For a period of time,
 ↪ police personnel are trying to explain the situation to the male in
 ↪ regard to establishing a crime scene, and at that point, the male becomes
 ↪ increasingly agitated. As the verbal disagreement escalates, the male
 ↪ shoves sergeant #1 with both hands. At that point personnel, Officer #1
 ↪ and Officer #2, attempt to detain the male. As officers attempted to take
 ↪ the individual into custody, the situation escalated rapidly. According
 ↪ to preliminary information from witness statements, surveillance footage,
 ↪ and body-worn camera footage reviewed thus far, the male produced a
 ↪ firearm during the encounter. Officers repeatedly issued commands that
 ↪ the male not to draw the weapon. The male then discharged the firearm
 ↪ directly toward the police officers. During the ensuing exchange of
 ↪ gunfire, three Philadelphia police officers [Sergeant #1, and Officers #1
 ↪ and #2] were struck by gunfire. A fourth officer [Officer #3] was present
 ↪ and returned fire but was not injured. All three injured officers were
 ↪ transported to a local hospital and are expected to recover. The male was
 ↪ struck by gunfire in the chest and rear right leg during the exchange. He
 ↪ was transported to a local hospital, where he was pronounced deceased.
 ↪ Investigators recovered a 9mm handgun from the scene that was possessed
 ↪ by the male. The male possessed a valid PA LTCF permit. The Philadelphia
 ↪ District Attorney's Office (DAO) was notified and responded to the scene.
 ↪ This investigation remains active and ongoing and is being conducted by
 ↪ the Officer-Involved Shooting Investigation Unit (OISI), The Philadelphia
 ↪ Police Department Internal Affairs Bureau (IAB), and the Philadelphia
 ↪ District Attorney's Office. Discharging officers: Sergeant #1: 39/W/M, 8
 ↪ years of service, discharged multiple rounds and sustained a gunshot
 ↪ wound. Officer #1: 43/W/M, 1 year of service, discharged multiple rounds
 ↪ and sustained a gunshot wound. Officer #2: 30/W/M, 7 years of service,
 ↪ discharged multiple rounds and sustained gunshot wounds. Officer #3:
 ↪ 21/W/M, 1 year of service, discharged multiple rounds and was not
 ↪ injured. Decedent Info: 57/B/M"
```

Now we are ready to read in all the incidents' details.

```
ois$text <- NA
for(i in 1:nrow(ois))
```

```
{
   message(paste0("Incident: ",i))
   a <- try( read_html(ois$url[i]) )

   if(inherits(a, "try-error"))
   { # in case a page does not exist
      message(paste0("Could not access webpage for ", ois$id[i]))
   } else
   {
      # grab text between <div class="ois-content-area"> and </div>
      ois$text[i] <- a |>
         html_element("div.ois-content-area") |>
         html_text() |>
         trimws()
   }
}
```

And let's just check that we have some descriptions now.

```
ois$text |> substring(1, 30)
```

```
 [1] "On 6/13/2026 at approximately "   "On Saturday, May 23, 2026, at "
 [3] "On Friday, April 10, 2026, at "   "On Tuesday, April 7, 2026, at "
 [5] "On Tuesday, March 3, 2026, at "   "On Thursday, January 29, 2026,"
 [7] "At approximately 8:30 p.m., on"   "On Wednesday, January 7, 2026,"
 [9] "On January 5, 2026, at approxi"   "At approximately 5:30 p.m. on "
[11] "On Sunday, September 21, 2025,"   "On Sunday, August 10th 2025, a"
[13] "On Friday, July 4, 2025 at 5:5"   "On June 20th, 2025, at approxi"
[15] "On Wednesday, May 21, 2025, at"   "On Wednesday, April 30, 2025, "
[17] "On Monday, April 28, 2025, at "   "On Saturday, March 22, 2025, a"
[19] "On Thursday March 20, 2025, at"   "On March 19, 2025, at approxim"
[21] "On Tuesday, February 4, 2025, "   "On Monday, February 3, 2025 at"
[23] "4100 block of Leidy Avenue\nOn "  "On Saturday, January 11, 2025,"
[25] "On Friday, January 10, 2025, a"   "On Tuesday, December 10, 2024,"
[27] "3200 block of A Street\nOn Tues"  "5400 block of Chancellor Stree"
[29] "29oo block of E. Street\nOn Fri"  "3300 Willits Road\nOn Thursday,"
[31] "6100 block of Lebanon Avenue\nO"  "2600 block of Glenwood Avenue\n"
[33] "3900 block of Whittaker Avenue"   "2200 block of S. 65th Street\nT"
[35] "3000 block of Ruth Street\nThe "  "6100 block of West Columbia Av"
[37] "3500 block of F Street\nA Phila"  "2700 block of North 6th Street"
[39] "1500 block of North 57th Stree"   "2100 block of East Westmorelan"
[41] "1600 South Dover Street\nOn Thu"  "3000 block of North 16th Stree"
[43] "1500 block of South 58th Stree"   "2200 block of West Oxford Aven"
```

```
 [45] "3900 block of Fairmont Avenue\n"  "Unit block of East Cliveden St"
 [47] "2100 block of Eastburn Avenue\n"  "1000 block of West Dakota Stre"
 [49] "6200 block of Haverford Avenue"   "1000 block of North 48th Stree"
 [51] "300 block of Adams Avenue\nOn T"  "2800 block of Mascher Street\nA"
 [53] "2300 block of Borbeck Avenue\nA"  "3600 block of Sepviva Street\nA"
 [55] "1800 block of South 29th St.\nO"  "8000 block of North Frankford "
 [57] "3700 block of Fairmount Street"   "7500 block of Whitaker Ave.\nAt"
 [59] "1500 block of N. 62nd Street\nO"  "Unit Block of E. Phil Ellena S"
 [61] "7600 block of Lexington Avenue"   "3100 block of Emerald St.\nOn T"
 [63] "On Monday, August 14, at appro"   "2300 block of Fawn Street\nOn T"
 [65] "400 block of West Bringhurst S"   "15xx E. Johnson Street\nOn Frid"
 [67] "200 block of North 60th Street"   "800 block of North 10th Street"
 [69] "3300 block of North 10th Stree"   "1300 block of Chancellor Stree"
 [71] "500 block of East Brinton Stre"   "400 block of South Street\nOn S"
 [73] "2200 block of West Hunting Par"   "4700 block of Leiper Street\nOn"
 [75] "4000 block of Lancaster Avenue"   "1700 block of Barbara Street\nO"
 [77] "2000 block of South Beechwood "   "100 block of West Lehigh Avenu"
 [79] "4800 block of Keyser Street\nOn"  "1700 Dickinson Street\nOn Wedne"
 [81] "2700 block of Brown Street\nOn "  "1900 block of South Bancroft S"
 [83] "5700 block of Overbrook Avenue"   "4100 block of Parkside Avenue\n"
 [85] "Whitaker Avenue and Roosevelt "   "9th Street and Hunting Park Av"
 [87] "3000 block of North Water Stre"   "300 Glen Echo Road\nOn Monday, "
 [89] "Broad Street and Somerville Av"   "3300 Emerald Street\nOn Friday,"
 [91] "4700 block of Rorer Street\nOn "  "3500 block of Kyle Road\nOn Mon"
 [93] "3500 block of Wharton Street\nO"  "1900 block of East Hart Lane\nO"
 [95] "6100 block of  Locust Street\nO"  "5600 block of Greene Street\nOn"
 [97] "1400 block of Sharpnack Street"   "4200 block of Clarissa Street\n"
 [99] "6th Street and McKean\nOn Tuesd"  "2500 Block of South 7th Street"
[101] "1500 block of Bailey Street\nOn"  "7600 Block of Roosevelt Blvd\nO"
[103] "Jasper Street and Hart Lane\nOn"  "On November 21, 2019, at appro"
[105] "On Saturday, November 2, 2019,"   "On 9-02-19, at 10:15 PM, two u"
[107] "On May 20, 2019, at approximat"   "On Saturday, May 11th 2019 at "
[109] "On Thursday, April 25, 2019, u"   "On Saturday April 20, 2019, at"
[111] "On March 28, 2019, at approxim"   "On March 6, 2019, at approxima"
[113] "OIS 18-28\nAt approximately 8:4"  "OIS# 18-27\nOn November 13, 201"
[115] "OIS# 18-26\nOn November 13, 201"  "OIS # 18-25\nOn Wednesday, Nove"
[117] "OISI # 18-22\nOn Saturday, Augu"  "OISI # 18-19\nOn Monday, August"
[119] "OIS# 18-17\nOn Thursday, August"  "OIS# 18-16\nOn Monday, August 6"
[121] "OIS # 18-12\nOn Friday, June 8,"  "OIS# 18-08\nOn Wednesday, April"
[123] "OIS# 18-02\nOn Monday, January "  "OIS# 18-01\nOn Saturday, Januar"
[125] "OIS# 17-37\nOn Wednesday, Decem"  "OIS# 17-36\nOn Tuesday, Decembe"
[127] "OIS# 17-30\nOn Saturday, Novemb"  "OIS# 17-28\nOn Saturday, Septem"
[129] "OIS# 17-25\nOn Saturday, August"  "OIS# 17-23\nOn Friday, August 1"
```

```
[131] "OIS# 17-22\nOn Monday, August 7"  "OIS# 17-20\nOn Thursday, July 2"
[133] "OIS# 17-19\nOn Wednesday, July "  "OIS# 17-17 (June 8, 2017)\nOn T"
[135] "OIS# 17-13\nOn Friday, May 12, "  "OIS# 17-08 (March 29, 2017) On"
[137] "OIS# 17-03 (February 15, 2017)"   "PS#16-43\n11/25/16\nOn Friday, N"
[139] "PS#16-40\n11/07/16\nOn Monday, N" "PS#16-38\n11/2/16\nOn Saturday, "
[141] "PS#16-37\n10/27/16\nOn Thursday," "PS# 16-35\n10/19/16\nOn Saturday"
[143] "PS# 16-34\n10/19/16\nOn Wednesda" "PS# 16-33\n10/18/16\nOn Tuesday,"
[145] "PS#16-32\n9/28/16\nOn Wednesday," "PS#16-30\n9/16/16\nOn Friday, Se"
[147] "PS#16-29\n9/09/16\nOn Friday, Se" "PS#16-28\n9/08/16\nOn Thursday, "
[149] "PS# 16-26 9/05/16 On Monday, S"   "PS#16-19\n5/31/16\nOn Tuesday, M"
[151] "PS#16-18\n5/31/16\nOn Tuesday, M" "PS# 16-16\n5/20/16\nOn Friday, M"
[153] "PS#16-13\n5/04/16\nOn Wednesday " "PS#16-12\n5/03/16\nOn Tuesday, M"
[155] "PS# 16-11\n4/17/16\nOn Sunday, A" "PS#16-10\n4/09/16\nOn Saturday, "
[157] "PS#16-07\n3/17/16\nOn Thursday, " "PS#16-03\n2/04/16\nOn Thursday, "
[159] "PS#16-02\n2/02/16\nOn Tuesday, F" "PS# 16-01\n1/07/16\nOn Thursday,"
```

4 Extracting dates from the text

While the main OIS webpage did not give the date of the incident, the text details always show the date. We can extract those dates with regular expressions. The dates may come in a variety of formats, but we can use the `lubridate` package to parse them. Let's start with those where the date is spelled out.

```
# extract dates in January 11, 2024 format
#    (?x) - ignore spaces and newlines
#         lets me break regex into several lines
#    (?s) - allows .* to include \n
#    both (?x) and (?s) require perl=TRUE
#    \s - whitespace (spaces, tabs, line feeds, carriage return)
a <- gsub("(?xs)
  .*?(January|February|March|April|May|June|
        July|August|September|October|November|December)
  \\s* ([0-9]{1,2})
  ( , \\s* (20[0-9]{2}) )?.*",
          "\\1 \\2\\3", ois$text, perl=TRUE)
```

For those incidents matching that January 11, 2024 format, they should have less than 20 characters in them. Let's check those out.

```
a[nchar(a) < 20]
```

```
  [1] "May 23, 2026"       "April 10, 2026"     "April 7, 2026"
  [4] "March 3, 2026"      "January 29, 2026"   "January 19, 2026"
  [7] "January 7, 2026"    "January 5, 2026"    "January 4, 2026"
 [10] "September 21, 2025" "August 10"          "July 4, 2025"
 [13] "June 20"            "May 21, 2025"       "April 30, 2025"
 [16] "April 28, 2025"     "March 22, 2025"     "March 20, 2025"
 [19] "March 19, 2025"     "February 4, 2025"   "February 3, 2025"
 [22] "January 27, 2025"   "January 11, 2025"   "January 10, 2025"
 [25] "December 10, 2024"  "November 12, 2024"  "November 10, 2024"
 [28] "October 11, 2024"   "October 3, 2024"    "October 2, 2024"
 [31] "September 26, 2024" "September 19, 2024" "September 6, 2024"
 [34] "July 4"             "June 24, 2024"      "June 22, 2024"
 [37] "June 15"            "June 5, 2024"       "May 30, 2024"
 [40] "May 23, 2024"       "May 15, 2024"       "May 14, 2024"
 [43] "May 12"             "May 1, 2024"        "April 20, 2024"
 [46] "April 17, 2024"     "April 15, 2024"     "April 14, 2024"
 [49] "April 10"           "February 15, 2024"  "January 26"
 [52] "January 17, 2024"   "January 10, 2024"   "December 31, 2023"
 [55] "December 10, 2023"  "November 4, 2023"   "October 4"
 [58] "October 4, 2023"    "October 2, 2023"    "September 27, 2023"
 [61] "September 14, 2023" "August 14"          "May 4, 2023"
 [64] "April 29, 2023"     "March 24, 2023"     "February 8, 2023"
 [67] "October 12, 2022"   "October 7, 2022"    "September 11, 2022"
 [70] "September 10, 2022" "June 4, 2022"       "May 11, 2022"
 [73] "March 19, 2022"     "March 1, 2022"      "February 15, 2022"
 [76] "February 11, 2022"  "February 9, 2022"   "February 2, 2022"
 [79] "January 4, 2022"    "August 18"          "July 22, 2021"
 [82] "June 14, 2021"      "April 7, 2021"      "November 30, 2020"
 [85] "September 18, 2020" "August 18, 2020"    "June 23, 2020"
 [88] "May 9, 2020"        "April 10, 2020"     "February 28, 2020"
 [91] "February 20, 2020"  "November 21, 2019"  "November 2, 2019"
 [94] "May 20, 2019"       "May 11"             "April 25, 2019"
 [97] "April 20, 2019"     "March 28, 2019"     "March 6, 2019"
[100] "December 5, 2018"   "November 13, 2018"  "November 13, 2018"
[103] "November 7, 2018"   "August 25, 2018"    "August 20, 2018"
[106] "August 9, 2018"     "August 6, 2018"     "June 8, 2018"
[109] "April 18, 2018"     "January 29"         "January 13"
[112] "December 27, 2017"  "December 26, 2017"  "November 11, 2017"
[115] "September 23, 2017" "August 19, 2017"    "August 11, 2017"
[118] "August 7, 2017"     "July 27, 2017"      "July 19"
[121] "June 8, 2017"       "May 12, 2017"       "March 29, 2017"
[124] "February 15, 2017"  "November 25, 2016"  "November 7, 2016"
[127] "October 29, 2016"   "October 27, 2016"   "October 22, 2016"
```

```
[130] "October 19, 2016"   "October 18, 2016"   "September 9, 2016"
[133] "September 16, 2016" "September 9, 2016"  "September 8, 2016"
[136] "September 5, 2016"  "May 31, 2016"       "May 31, 2016"
[139] "May 20, 2016"       "May 4, 2016"        "May 3, 2016"
[142] "April 17, 2016"     "April 9, 2016"      "March 17, 2016"
[145] "February 4, 2016"   "February 2, 2016"   "January 7, 2016"
```

The code seems to work for many dates, but we also see that some of the dates did not include the year. The first two digits of the incident ID are the last two digits of the year.

```
a[nchar(a) < 20 & !grepl("20[0-9]{2}", a)]
```

```
 [1] "August 10"  "June 20"    "July 4"     "June 15"    "May 12"
 [6] "April 10"   "January 26" "October 4"  "August 14"  "August 18"
[11] "May 11"     "January 29" "January 13" "July 19"
```

```
# first two digits of id have the year
i <- nchar(a) < 20 & !grepl("20[0-9]{2}", a)
paste0(a[i], ", 20", substring(ois$id[i],1,2))
```

```
 [1] "August 10, 2025"  "June 20, 2025"    "July 4, 2024"     "June 15, 2024"
 [5] "May 12, 2024"     "April 10, 2024"   "January 26, 2024" "October 4,
  ↪  2023"
 [9] "August 14, 2023"  "August 18, 2021"  "May 11, 2019"     "January 29,
  ↪  2018"
[13] "January 13, 2018" "July 19, 2017"
```

The rest of the dates have formats that are in some variation of 01/11/2024 or 01-11-2024 or 01/11/24 or 01-11-24, sometimes with / separators and sometimes with - separators and sometimes with a four digit year and sometimes with a two digit year. We can craft our regular expression to capture all these variations.

```
# get the remaining dates in #/#/# or #-#-# format
#   .*  is "greedy" and will absorb as much as possible
#   .*? is "lazy" and will stop at the first pattern match
gsub(".*?([0-9]{1,2}[/-][0-9]{1,2}[/-](20)?[1-2][0-9]).*",
     "\\1",
     a[nchar(a) > 20])
```

```
 [1] "6/13/2026"  "4/6/2022"   "1-31-2022"  "10-26-21"   "10-4-2021"
 [6] "7-29-21"    "12-25-2020" "12-9-2020"  "11-27-20"   "11-12-20"
[11] "10-26-2020" "10-8-2020"  "9-02-19"
```

We have covered all the cases and can now create a column of incident dates extracted from the incident details.

```
ois <- ois |>
   mutate(date = gsub(
      "(?xs)
  .*?(January|February|March|April|May|June|
   July|August|September|October|November|December)
  \\s* ([0-9]{1,2})
  ( , \\s* (20[0-9]{2}) )?.*",
      "\\1 \\2\\3", text, perl=TRUE),
      date = if_else(nchar(date)<20, date, NA),
      date = if_else(!is.na(date) & !grepl("20[0-9]{2}", date),
                     paste0(date, ", 20", substring(id,1,2)),
                     date),
      date = if_else(
         is.na(date),
         gsub(".*?([0-9]{1,2}[/-][0-9]{1,2}[/-](20)?[1-2][0-9])[^0-9].*",
              "\\1", text),
         date),
      date = mdy(date))
```

For a little test, let's check if there are any incidents where the year we scraped from the webpage differs from the first two digits of the incident ID.

```
table(year(ois$date), substring(ois$id,1,2))
```

```
      16 17 18 19 20 21 22 23 24 25 26
  2016 23  0  0  0  0  0  0  0  0  0  0
  2017  0 13  0  0  0  0  0  0  0  0  0
  2018  0  0 12  0  0  0  0  0  0  0  0
  2019  0  0  0  9  0  0  0  0  0  0  0
  2020  0  0  0  0 14  0  0  0  0  0  0
  2021  0  0  0  0  0  7  0  0  0  0  0
  2022  0  0  0  0  0  0 15  0  0  0  0
  2023  0  0  0  0  0  0  0 13  0  0  0
  2024  0  0  0  0  0  0  0  0 29  0  0
  2025  0  0  0  0  0  0  0  0  0 15  0
  2026  0  0  0  0  0  0  0  0  0  0 10
```

That is a good sign!

5 Geocoding the OIS locations

Our OIS data frame has the address for every incident, but to be more useful we really need the geographic coordinates. If we had the coordinates, then we could put them on a map, tabulate how many incidents occur within an area, calculate distances, and answer questions about the geography of these data.

Geocoding is the process of converting a text description of a location (typically an address or intersection) to obtain geographic coordinates (often longitude/latitude, but other coordinate systems are also possible). Google Maps currently reigns supreme in this area. Google Maps understands very general descriptions of locations. You can ask for the coordinates of something like "chipotle near UPenn" and it will understand that "UPenn" means the University of Pennsylvania and that "chipotle" is the burrito chain. Google Maps requires a credit card in order to access its geocoding service.

We will use the ArcGIS geocoder to get the coordinates of every location. Many web data sources use a standardized language for providing data. JSON (JavaScript Object Notation) is quite common and ArcGIS uses JSON.

The URL for ArcGIS has the form

https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddress-Candidates?f=json&singleLine=38th%20and%20Walnut,%20Philadelphia,%20PA&out-Fields=Match_addr,Addr_type

You can see the address for Penn's McNeil Building embedded in this URL. Spaces need to be replaced with `%20` (20 is the hexadecimal ASCII code for the space character). Let's see what data we get back from this URL.

```
scan("https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCandidat
     what="", sep="\n")
```

```
[1] "{\"spatialReference\":{\"wkid\":4326,\"latestWkid\":4326},\"candidates\
 ↪  ":[{\"address\":\"S 38th St & Walnut St, Philadelphia, Pennsylvania,
 ↪  19104\",\"location\":{\"x\":-75.198781031742,\"y\":39.953632005469},\"sc
 ↪  ore\":99.36,\"attributes\":{\"Match_addr\":\"S 38th St & Walnut St,
 ↪  Philadelphia, Pennsylvania, 19104\",\"Addr_type\":\"StreetInt\"},\"exten
 ↪  t\":{\"xmin\":-75.199781031742,\"ymin\":39.952632005469,\"xmax\":-75.197
 ↪  781031742,\"ymax\":39.954632005469}},{\"address\":\"S 38th St & Walnut
 ↪  St, Philadelphia, Pennsylvania,
 ↪  19104\",\"location\":{\"x\":-75.198651028424,\"y\":39.953613020458},\"sc
 ↪  ore\":99.36,\"attributes\":{\"Match_addr\":\"S 38th St & Walnut St,
 ↪  Philadelphia, Pennsylvania, 19104\",\"Addr_type\":\"StreetInt\"},\"exten
 ↪  t\":{\"xmin\":-75.199651028424,\"ymin\":39.952613020458,\"xmax\":-75.197
 ↪  651028424,\"ymax\":39.954613020458}}]}"
```

It is messy, but readable. You can see embedded in this text the `lat` and `lon` for this address. You can also see that it should not be too hard for a machine to extract these coordinates, and the rest of the information here, from this block of text. This is the point of JSON, producing data in a format that a human could understand in a small batch, but a machine could process fast and easily.

The `jsonlite` R package facilitates the conversion of JSON text like this into convenient R objects.

```
library(jsonlite)
fromJSON("https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCan
```

```
$spatialReference
$spatialReference$wkid
[1] 4326

$spatialReference$latestWkid
[1] 4326


$candidates
                                                    address location.x
1 S 38th St & Walnut St, Philadelphia, Pennsylvania, 19104  -75.19878
2 S 38th St & Walnut St, Philadelphia, Pennsylvania, 19104  -75.19865
  location.y score                                    attributes.Match_addr
1   39.95363 99.36 S 38th St & Walnut St, Philadelphia, Pennsylvania, 19104
2   39.95361 99.36 S 38th St & Walnut St, Philadelphia, Pennsylvania, 19104
  attributes.Addr_type extent.xmin extent.ymin extent.xmax extent.ymax
1            StreetInt   -75.19978    39.95263   -75.19778    39.95463
2            StreetInt   -75.19965    39.95261   -75.19765    39.95461
```

`fromJSON()` converts the JSON results from the ArcGIS geocoder to an R `list` object. The JSON tags turn into list names and columns in a data frame.

To make geocoding a little more convenient, here is an R function that automates the process of taking an address, filling in special characters (like spaces) with their ASCII codes with `URLencode()`, and retrieving the JSON results from the ArcGIS geocoding service.

```
geocodeARCGIS <- function(address)
{
   paste0("https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer/findAddressCa
          URLencode(address),
          "&outFields=Match_addr,Addr_type") |>
      fromJSON()
}
```

Let's test out `geocodeARCGIS()` by pulling up a map of the geocoded coordinates. Once we have the latitude and longitude for the McNeil Building, where we typically hold our crime data science courses at Penn, we can use `leaflet()` to show us a map of the area.

```
gcPenn <- geocodeARCGIS("3718 Locust Walk, Philadelphia, PA") |>
   pluck("candidates") |>
   head(1) |>
   mutate(lon = location$x,
          lat = location$y)

leaflet(width = 1200, height = 800) |>
   # addTiles() |>
    addTiles(
       urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
       options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
                             tileSize = 512,
                             zoomOffset = -1),
       attribution = '© MapTiler © OpenStreetMap contributors') |>
   setView(lng=gcPenn$lon, lat=gcPenn$lat, zoom=18) |>
   addCircleMarkers(lng=gcPenn$lon,
                    lat=gcPenn$lat)
```

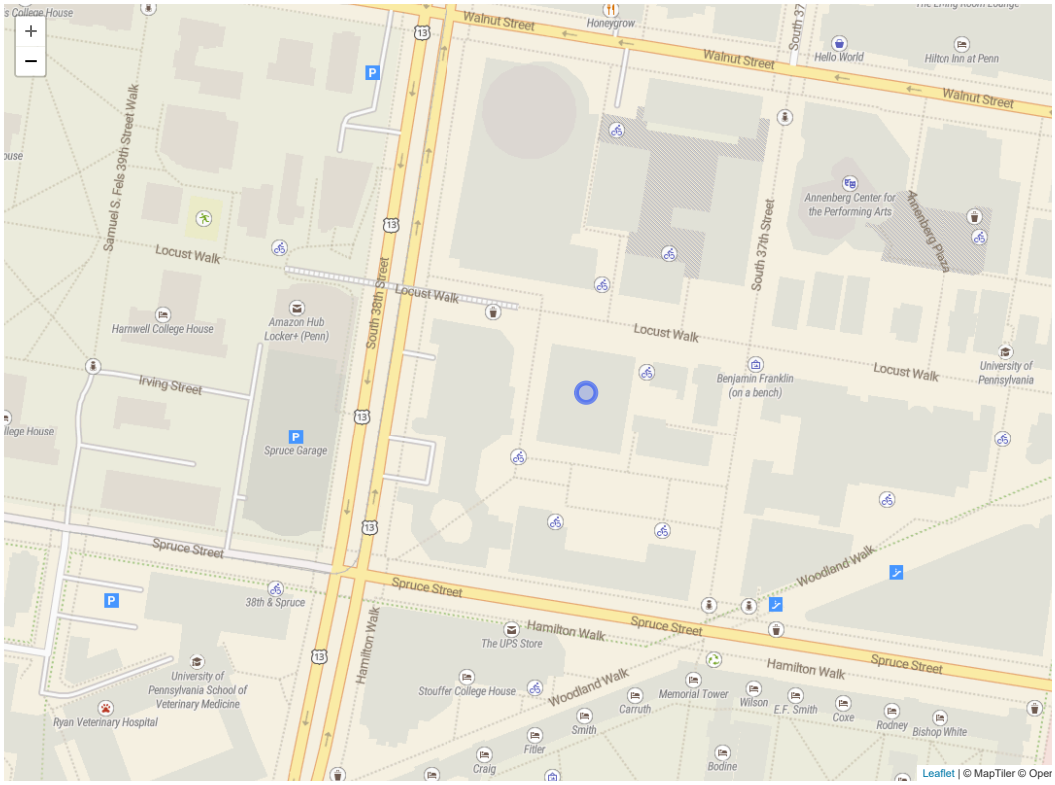

Figure 3: ArcGIS geocoding result for 3718 Locust Walk

`leaflet()` prepares the mapping process (the `width` and `height` do not have to be set, butI needed to for these notes). `addTiles()` pulls in the relevant map image (buildings and streets). Generally, you can use `addTiles()` with no other arguments. I regularly recompile these notes which hits the OpenStreetMap server a little too much, so I have to rely on a different map provider that can better handle the load of requests. `setView()` takes the longitude and latitude from our `gcPenn` object, sets that as the center of the map, and zooms in to level "18," which is a fairly close zoom of about one block. `addCircleMarkers()` creates a circle at the selected point.

We are almost ready to throw all of our addresses at the geocoder, but let's first make sure the addresses look okay. Several locations have `&` where the ArcGIS geocoder wants `and`.

```
# & -> and for geocoding
grep('&', ois$location, value=TRUE)
```

```
[1] "Jasper Street & Hart Lane"           "Front Street & Allegheny Avenue"
[3] "Bridge Street & Roosevelt Boulevard" "49th & Walnut Streets"
[5] "4800, 4900 & 5100 blocks of Sansom"  "near 16th Street & Allegheny Ave"
```

Many addresses just give the block, like "3700 block of Locust Walk." We will need to change these to the midpoint of the block like "3750 Locust Walk" so that we get a geocoding hit that is nearby.

```
grep("[Bb]lock", ois$location, value=TRUE)
```

```
 [1] "2000 block of North 54th Street"
 [2] "2800 block of Kensington Avenue"
 [3] "6900 block of Lawnton Avenue"
 [4] "5400 block of Webster Street"
 [5] "3400 block of Hess Street"
 [6] "1400 block of North Robinson Street"
 [7] "4100 block of North Broad Street"
 [8] "4800 block of Blakiston Street"
 [9] "2800 block of Sebring Road"
[10] "400 block of East Rockland Street"
[11] "5100 block of North 10th Street"
[12] "2000 block of Simon Street"
[13] "2900 block of North Lawrence Street"
[14] "2600 block of South 21st Street"
[15] "100 block of West Somerset Street"
[16] "300 block of North 65th Street"
[17] "4600 block of Roosevelt Blvd"
[18] "4100 block of Ogden Street"
[19] "4600 block of Roosevelt Boulevard"
```

```
[20] "1600 block of Moore Street"
[21] "2800 block of Jasper Street"
[22] "4100 block of Leidy Avenue"
[23] "800 block of West Master Street"
[24] "600 block of Chamounix Drive"
[25] "3400 block of Vista Street"
[26] "3200 block of A Street"
[27] "5400 block of Chancellor Street"
[28] "2900 block of E. Street"
[29] "3300 block of Willits Road"
[30] "6100 block of Lebanon Avenue"
[31] "2600 block of Glenwood Avenue"
[32] "3900 block of Whittaker Avenue"
[33] "2200 block of S. 65th Street"
[34] "3000 block of Ruth Street"
[35] "6100 block of West Columbia Avenue"
[36] "3500 block of F Street"
[37] "2700 block of North 6th Street"
[38] "1500 block of North 57th Street"
[39] "2100 block of East Westmoreland Street"
[40] "3000 block of North 16th Street"
[41] "1500 block of 58th Street"
[42] "2200 block of West Oxford Avenue"
[43] "3900 block of Fairmont Avenue"
[44] "Unit block of East Cliveden Street"
[45] "2100 block of Eastburn Avenue"
[46] "1000 block of West Dakota Street"
[47] "6200 block of Haverford Avenue"
[48] "1000 block of North 48th Street"
[49] "300 block of Adams Avenue"
[50] "2800 block of Mascher Street"
[51] "2300 block of Borbeck Avenue"
[52] "3600 block of Sepviva Street"
[53] "1800 block of South 29th Street"
[54] "8000 block of N. Frankford Ave."
[55] "3700 block of Fairmount Street"
[56] "7500 block of Whitaker Avenue"
[57] "1500 block of N. 62nd Street"
[58] "Unit block of E. Phil Ellena St."
[59] "7600 block of Lexington Avenue"
[60] "3100 block of Emerald Street"
[61] "100 block of E. Willard St."
[62] "2300 block of Fawn St."
```

```
 [63] "400 block of W. Bringhurst St."
 [64] "2700 block of Brown Street,"
 [65] "3000 block of N. Water Street"
 [66] "300 block of Glen Echo Road"
 [67] "2500 Block of south 7th Street"
 [68] "1500 Block of Bailey Street"
 [69] "7600 Block of Roosevelt Blvd"
 [70] "5900 block of Torresdale Avenue"
 [71] "1000 block of Chelten Avenue"
 [72] "3400 block of G Street"
 [73] "1800 block of N. Broad Street"
 [74] "2100 block of Taney Terrace"
 [75] "1300 block of Kater Street"
 [76] "4900 block of Hazel Avenue"
 [77] "7500 block of Brookhaven Road"
 [78] "8700 block of Crispin Street"
 [79] "3200 block of G Street"
 [80] "2700 block of Dickinson Street"
 [81] "7100 block of Hegerman Street"
 [82] "2000 block of Snyder Avenue"
 [83] "4800 block of Knox Street"
 [84] "1400 block of Lardner Street"
 [85] "3100 block of N. 33rd Street"
 [86] "1300 block of Bigler Street"
 [87] "2800 block of Kensington Avenue"
 [88] "1200 block of S. 29th Street"
 [89] "2300 block of north 13th Street"
 [90] "2500 Block of N. Alder Street"
 [91] "3100 Block of N. Darien Street"
 [92] "2200 Block of N. Fairhill Street"
 [93] "8300 Block of Horrocks Street"
 [94] "400 block of East Somerset Street"
 [95] "4200 block of Whitaker Avenue"
 [96] "1200 block of south 51st street"
 [97] "6400 block of Lambert Street"
 [98] "2900 Block of Amber Street"
 [99] "3100 block of N. 9th Street"
[100] "2900 block of N. Front Street"
[101] "3800 block of Elsinore Street"
[102] "500 block of N. 56 Street"
[103] "6300 block of Crafton Street"
[104] "1400 block of Chew Avenue"
[105] "6300 block of Hazel Avenue"
```

```
[106] "4800, 4900 & 5100 blocks of Sansom"
[107] "200 block of Millick Street"
[108] "600 block of E. Clearfield Street"
[109] "3400 block of Broad street"
[110] "6300 Block of Overbrook Avenue"
[111] "5300 Block of Grays Avenue"
[112] "Unit Block of Salford street"
[113] "3100 Block of north Carlisle Street"
[114] "100 block of north 55th Street"
[115] "300 block of south 60th street"
```

Lastly, there are some addresses with the word "near" that need to be deleted.

```
grep("[Nn]ear", ois$location, value=TRUE)
```

```
[1] "near 16th Street & Allegheny Ave" "Near Loudon and D streets"
```

Let's make all these fixes.

```
ois <- ois |>
   mutate(location = gsub("&","and",location),
          location = gsub("00 [Bb]lock( of)?", "50", location),
          location = gsub("[Uu]nit [Bb]lock( of)?", "50", location),
          location = gsub("[Nn]ear ", "", location))
```

Some OIS incidents are missing locations.

```
ois |>
  filter(grepl("[Ww]ithheld", location)) |>
  pull(text)
```

```
[1] "PS# 16-26 9/05/16 On Monday, September 5, 2016, at approximately 6:28
 ↪  P.M., an off-duty officer, in plainclothes, became involved in a verbal
 ↪  and physical altercation with his son at their residence. During the
 ↪  physical altercation the officer discharged his personal weapon, striking
 ↪  his son. The officer's son was transported to Aria-Torresdale Hospital
 ↪  for treatment. The officer's firearm, a .40 caliber semi-automatic
 ↪  pistol, loaded with three live rounds, was recovered at the scene. There
 ↪  were no other injuries as a result of this incident. *** Information
 ↪  posted in the original summary reflects a preliminary understanding of
 ↪  what occurred at the time of the incident. This information is posted
 ↪  shortly after the incident and may be updated as the investigation leads
 ↪  to new information. The DA's Office is provided all the information from
 ↪  the PPD's investigation prior to their charging decision."
```

```
[2] "PS#16-18\n5/31/16\nOn Tuesday, May 31, 2016, at approximately 1:12 PM,
 ↪ an off-duty officer, in civilian attire, arrived home at his residence.
 ↪ Upon entering the front door, the officer observed the living room
 ↪ television missing, the rear kitchen window open, and the rear kitchen
 ↪ door ajar. The officer heard voices coming from the basement. The officer
 ↪ went to the top of the basement steps and announced, "Police." A male
 ↪ appeared at the bottom of the steps and charged up the steps toward the
 ↪ officer with a dark metal object in his hand. In response, the officer
 ↪ discharged his weapon one time missing the offender. The offender fell
 ↪ down the steps crashing into the basement wall. The offender along with a
 ↪ second offender that was also in the basement fled out the rear basement
 ↪ door with the officer in foot pursuit. Upon exiting the rear basement
 ↪ door, the officer observed a green/blue van pull away from his property.
 ↪ The officer apprehended one offender in the 3200 block of Wellington
 ↪ Street. An off-duty detective apprehended the other offender near
 ↪ Brighton and Hawthorne Streets.\nThere were no reported injuries as a
 ↪ result of this police firearm discharge.\nNo weapon was recovered.\n***
 ↪ Information posted in the original summary reflects a preliminary
 ↪ understanding of what occurred at the time of the incident. This
 ↪ information is posted shortly after the incident and may be updated as
 ↪ the investigation leads to new information. The DA's Office is provided
 ↪ all the information from the PPD's investigation prior to their charging
 ↪ decision."
```

The text of OIS 16-18 gives a nearby address (3250 Wellington Street). We will drop incident 16-26, since it is not really a police shooting (off-duty officer shot his son).

Several other incidents have quirky addresses.

```
ois |>
  filter(id %in% c("16-30","16-10","17-08")) |>
  select(id, location, text)
```

```
     id                                                    location
1 17-08 New Castle County (see summary for details of PPD involvement)
2 16-30                                  4800, 4900 and 5150s of Sansom
3 16-10                       5700 N. Park street/5700 N. Broad street
```

```
1 OIS# 17-08 (March 29, 2017) On Wednesday, March 29, 2017, at approximately
  ↪ 5:39 PM, two uniformed officers in a marked vehicle responded to a radio
  ↪ assignment of "Person with a gun" at 5600 Whitby Avenue. A description of
  ↪ the individual was broadcast. Upon arrival at the location both officers
  ↪ observed a male who met the description entering the driver's seat of a
  ↪ parked minivan. As the male approached the open driver's door area both
  ↪ officers instructed him to stop. The male instead sat in the driver seat.
  ↪ The male accelerated the vehicle rearward striking officer number one,
  ↪ knocking him to the ground. The officer got back on his feet observed the
  ↪ male driver reach under the driver's seat (through the open drivers side
  ↪ door), and officer number one discharged one round at the male. The male
  ↪ drove from the location, struck a vehicle at 5700 Woodland Avenue, and
  ↪ continued toward Island Avenue and Lindbergh Boulevard where the minivan
  ↪ became disabled. The male exited the minivan, entered an unoccupied
  ↪ vehicle that was nearby with the engine running and the keys in the
  ↪ ignition. He ushered five passengers from the minivan into the stolen
  ↪ vehicle. The male drove the stolen vehicle to a house in New Castle
  ↪ Delaware, and dropped off four of the passengers. The male and a
  ↪ remaining female passenger drove to a second house in New Castle Delaware
  ↪ in the stolen vehicle. New Castle County Police Officers responded to the
  ↪ second location and report to Philadelphia Police that New Castle County
  ↪ Police Officers attempted to arrest the male who was inside the stolen
  ↪ vehicle. A uniformed New Castle County officer discharged his firearm
  ↪ multiple times, striking the male. The male was pronounced deceased at
  ↪ Christiana Hospital. There were no other injuries as a result of this
  ↪ incident. No firearm was recovered from the male or the vehicles. The
  ↪ female was not charged with any offense. *** Information posted in the
  ↪ original summary reflects a preliminary understanding of what occurred at
  ↪ the time of the incident. This information is posted shortly after the
  ↪ incident and may be updated as the investigation leads to new
  ↪ information. The DA's Office is provided all the information from the
  ↪ PPD's investigation prior to their charging decision.
```

```
2                    PS#16-30\n9/16/16\nOn Friday, September 16, 2016, at
↪ approximately 11:18 P.M., a uniformed sergeant in a marked police vehicle
↪ was seated in her parked vehicle in the 5100 block of Sansom Street, when
↪ a male approached and without warning, began to discharge a firearm,
↪ striking the sergeant, as she remained seated in her vehicle. The
↪ offender then began walking east on Sansom Street, stopping at a
↪ lounge/bar in the 5100 block of Sansom Street, where he discharged his
↪ firearm into the lounge/bar, striking a female employee and a male
↪ security guard. The offender continued walking east on Sansom Street to
↪ the 4900 block, where he discharged his firearm into an occupied parked
↪ vehicle, striking one female and one male occupant.\nResponding uniformed
↪ officers, in marked police vehicles, along with an officer from the
↪ University of Pennsylvania police force, located the offender in an
↪ alleyway in the rear of the 4800 blocks of Sansom and Walnut Streets.
↪ While in the 4800 block of Sansom Street the offender discharged his
↪ firearm, striking the University of Pennsylvania Officer as well as a
↪ marked police vehicle. Four Officers (one of whom was the University of
↪ Pennsylvania Officer) discharged their firearms, striking the offender.
↪ The offender fell to the ground and dropped his firearm. Fire Rescue
↪ responded and pronounced the offender deceased.\nThe offender's firearm,
↪ a 9MM, semi-automatic pistol, with an obliterated serial number, loaded
↪ with 14 live rounds, was recovered at the scene. There were three empty
↪ magazines from the offender's firearm recovered throughout the
↪ scene.\nThe sergeant, the University of Pennsylvania Officer, along with
↪ the four civilians who were all struck by gunfire, were transported to
↪ Penn-Presbyterian Hospital for treatment.\nThe female from the parked
↪ vehicle was later pronounced deceased at Penn-Presbyterian Hospital.\n***
↪ Information posted in the original summary reflects a preliminary
↪ understanding of what occurred at the time of the incident. This
↪ information is posted shortly after the incident and may be updated as
↪ the investigation leads to new information. The DA's Office is provided
↪ all the information from the PPD's investigation prior to their charging
↪ decision.
```

```
3
 ↪ PS#16-10\n4/09/16\nOn Saturday, April 9, 2016, at approximately 2:26
 ↪ P.M., an off-duty police officer, in plainclothes, observed two males
 ↪ involved in a physical struggle on the highway, in the 5700 block of N.
 ↪ Park Avenue. The officer observed that one of the males was armed with a
 ↪ handgun. The officer, who was armed, identified himself as a police
 ↪ officer and ordered the offender to drop his gun. The offender turned
 ↪ toward the officer and pointed the firearm at him. In response, the
 ↪ officer discharged his weapon at the offender. The offender fled to a
 ↪ rear parking lot in the 5700 block of N. Broad Street, where he fell to
 ↪ the ground, dropping his weapon. The offender retrieved his weapon and
 ↪ again pointed it at the officer. The officer responded by discharging his
 ↪ weapon, striking the offender. The offender again fled but collapsed near
 ↪ the corner of Broad and Chew Streets, where he was arrested.\nThe
 ↪ offenders' firearm, a .22 caliber revolver, loaded with three spent
 ↪ casings, was recovered at the scene.\nThe offender was transported to
 ↪ Albert Einstein Medical Center, where he was later pronounced deceased as
 ↪ a result of his injuries.\nThere were no other reported injuries as a
 ↪ result of this incident.\n*** Information posted in the original summary
 ↪ reflects a preliminary understanding of what occurred at the time of the
 ↪ incident. This information is posted shortly after the incident and may
 ↪ be updated as the investigation leads to new information. The DA's Office
 ↪ is provided all the information from the PPD's investigation prior to
 ↪ their charging decision.
```

Reading the details of the incidents, we can come up with reasonable fixes to the addresses. OIS 17-08 is a PPD shooting incident. PPD shot at a suspect in Philadelphia at 5600 Whitby Avenue then drove to New Castle County (Delaware) where a New Castle County shot the suspect. Let's fix this address. The remaining incidents we can edit based on the contents of the OIS description. We will also tack on ", Philadelphia, PA" to the end of each location to improve geocoding accuracy.

```
ois <- ois |>
   filter(id != "16-26") |>  # not really a police shooting
   mutate(location = recode_values(id,
      "16-18" ~ "3250 Wellington Street",
      # two locations, let's use the first one
      "16-10" ~ "5750 N. Broad Street",
      # pick the location where the police shooting occurred
      "16-30" ~ "4850 Sansom Street",
      # the Philly location
      "17-08" ~ "5600 Whitby Avenue",
      default = location),
```

```
      location = # add the city
         paste0(location,", Philadelphia, PA"))
```

Let's test out the process for just the first location. The code here shows how you can extract each bit of information that we want from geocoding an address: the coordinates (long,lat), the specific address that the geocoding service translated our requested address to, a quality-of-match score, and location type (e.g. StreetInt, PointAddress, StreetAddress).

```
a <- geocodeARCGIS(ois$location[1])
# collect (long,lat), matched address, address match score, and location type
a$candidates$location$x[1]
```

```
[1] -75.23251
```

```
a$candidates$location$y[1]
```

```
[1] 39.9875
```

```
a$candidates$address[1]
```

```
[1] "2050 N 54th St, Philadelphia, Pennsylvania, 19131"
```

```
a$candidates$score[1]
```

```
[1] 100
```

```
a$candidates$attributes$Addr_type[1]
```

```
[1] "StreetAddress"
```

With that we are ready to run all of our addresses through the ArcGIS geocoder. We could have geocoded all these addresses with the more simple code `lapply(ois$location, geocodeARCGIS)`. However, if the JSON connection to the geocoder fails for even one of the addresses (likely if you have a poor internet connection), then the whole `lapply()` function fails. With the for-loop implementation, if the connection fails, then `ois` still keeps all of the prior geocoding results and you can restart the for-loop at the point where it failed.

```r
# takes about 3 minutes
geoInfo <- vector("list", nrow(ois))
for(i in 1:nrow(ois))
{
  message(paste0("#", i, " Address: ", ois$location[i]))
  a <- geocodeARCGIS(ois$location[i])

  geoInfo[[i]] <-
     data.frame(lon       = a$candidates$location$x[1],
                lat       = a$candidates$location$y[1],
                addrmatch = a$candidates$address[1],
                score     = a$candidates$score[1],
                addrtype  = a$candidates$attributes$Addr_type[1])
}
geoInfo <- geoInfo |> bind_rows()
ois <- ois |> bind_cols(geoInfo)
```

Now we should have longitude and latitude for every incident. Let's check that they all look sensible.

```r
stem(ois$lat)
```

```
  The decimal point is 2 digit(s) to the left of the |

  3987 | 7
  3988 |
  3989 |
  3990 | 7
  3991 | 4579
  3992 | 4566779
  3993 | 12344588
  3994 | 223399
  3995 | 14667788
  3996 | 3345566999
  3997 | 111334566669
  3998 | 11255778999
  3999 | 011111222233334445566668899
  4000 | 00001123357
  4001 | 115566778
  4002 | 124455888
  4003 | 00133558
  4004 | 00001236889
```

```
  4005 | 0122334458
  4006 | 2226
  4007 | 1
  4008 | 0
```

```
stem(ois$lon)
```

```
  The decimal point is 2 digit(s) to the left of the |

  -7526 | 7
  -7524 | 200855444322000
  -7522 | 76333221030
  -7520 | 98769733300
  -7518 | 754221194432221
  -7516 | 97666433322555444210
  -7514 | 87776654443210099666554321
  -7512 | 9942210998731111100
  -7510 | 987766655521987400
  -7508 | 9786
  -7506 | 93760
  -7504 | 19977
  -7502 | 97764
  -7500 | 319
  -7498 | 9
```

All the points have latitude around 39 and 40 and longitude around -75. That is a good sign!

Let's check the "address type". We should worry about addresses geocoded to a "StreetName." That means the incident got geocoded to, say, "Market Street" but we are not sure where along Market Street the incident actually occurred. The geocoder most likely placed the incident at the midpoint of the street.

```
ois |> count(addrtype)
```

```
          addrtype  n
1     PointAddress 65
2    StreetAddress 79
3 StreetAddressExt  2
4        StreetInt 10
5       StreetName  3
```

```
ois |>
   filter(addrtype=="StreetName") |>
   select(id, location, addrmatch)
```

```
     id                                                    location
1 25-05 1 Philadelphia International Airport Way, Philadelphia, PA
2 25-01                       650 Chamounix Drive, Philadelphia, PA
3 21-14                      3800 Landsowne Drive, Philadelphia, PA
                                                            addrmatch
1 Philadelphia International Airport, Philadelphia, Pennsylvania, 19153
2                       Chamounix Dr, Philadelphia, Pennsylvania, 19131
3                       Lansdowne Dr, Philadelphia, Pennsylvania, 19104
```

One address should be at the Philadelphia Airport. This one geocoded just fine.

```
ois |>
  filter(id=="25-05") |>
  leaflet(width = 1200, height = 800) |>
    addTiles(
       urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
       options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
                             tileSize = 512,
                             zoomOffset = -1),
       attribution = '© MapTiler © OpenStreetMap contributors') |>
  addCircleMarkers(~lon, ~lat,
                   radius=3, stroke=FALSE,
                   fillOpacity = 1) |>
  addPopups(~lon, ~lat, ~location)
```

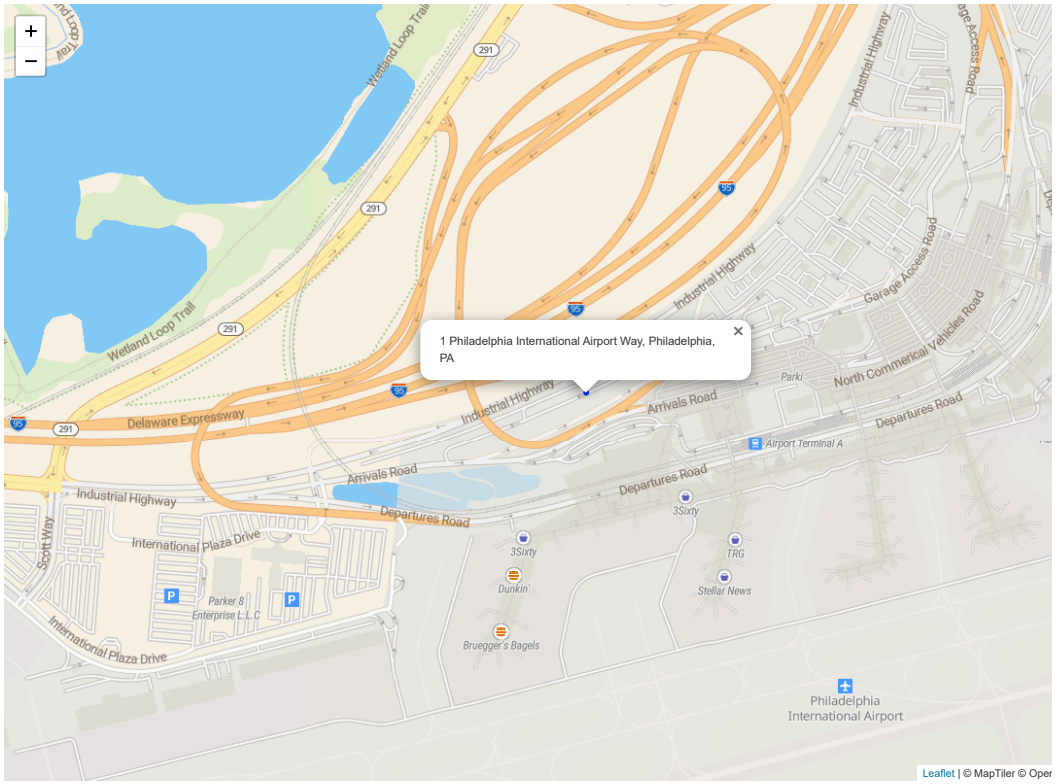

Figure 4: Checking the location of OIS 25-05

OIS 25-01 involves a dog shooting in Fairmount Park. The news stories about the incident place it 1 mile from Belmont/Edgely, close to Chamounix Drive and Ford Ave.

```
ois |>
   filter(id=="25-01") |>
   leaflet(width = 1200, height = 800) |>
   addTiles(
      urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
      options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
                            tileSize = 512,
                            zoomOffset = -1),
      attribution = '© MapTiler © OpenStreetMap contributors') |>
   addCircleMarkers(~lon, ~lat,
                    radius=3, stroke=FALSE,
                    fillOpacity = 1) |>
   addPopups(~lon, ~lat, ~location)
```

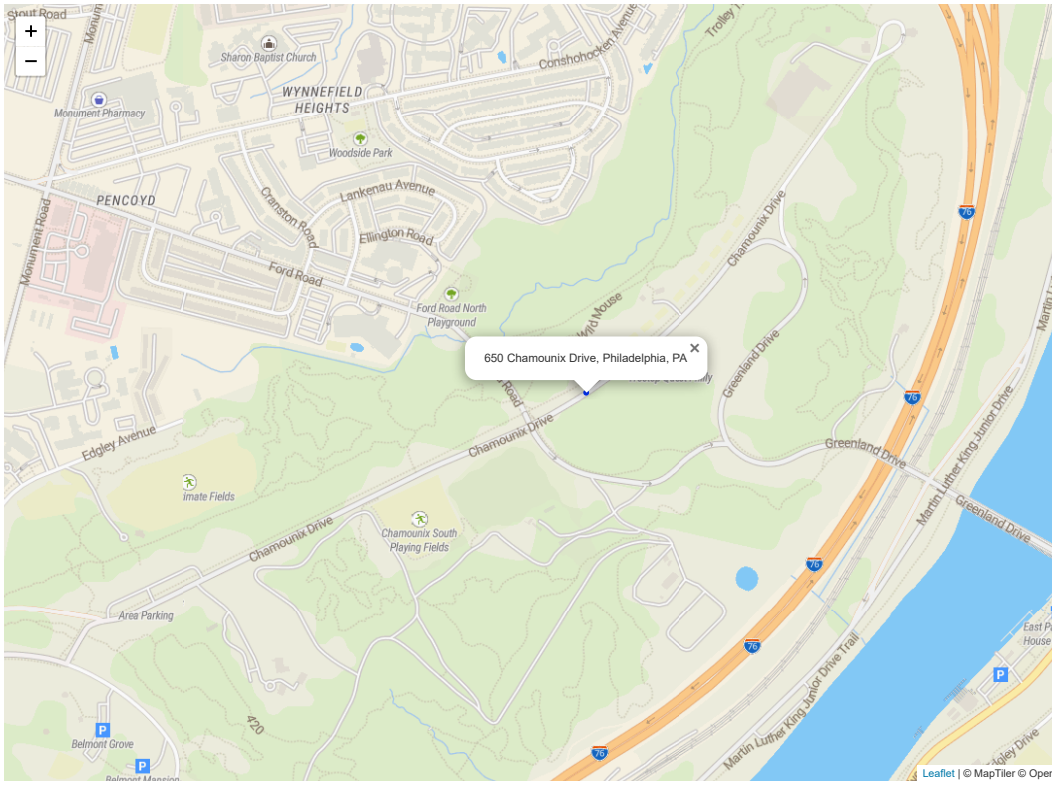

Figure 5: Checking the location of OIS 25-01

```
geocodeARCGIS("Chamounix Drive and Ford Ave, Philadelphia, PA")
```

```
$spatialReference
$spatialReference$wkid
[1] 4326

$spatialReference$latestWkid
[1] 4326


$candidates
                                                    address location.x
1 Chamounix Dr & Ford Rd, Philadelphia, Pennsylvania, 19131  -75.20474
2           Chamounix Dr, Philadelphia, Pennsylvania, 19131  -75.20348
  location.y score                                    attributes.Match_addr
1   39.99711 99.59 Chamounix Dr & Ford Rd, Philadelphia, Pennsylvania, 19131
2   39.99767 90.79           Chamounix Dr, Philadelphia, Pennsylvania, 19131
  attributes.Addr_type extent.xmin extent.ymin extent.xmax extent.ymax
1            StreetInt   -75.20574    39.99611   -75.20374    39.99811
2           StreetName   -75.20448    39.99667   -75.20248    39.99867
```

OIS 21-14 has address "3800 Landsowne Drive". Presumably it intended to find 3800 Lansdowne Drive, but it could not place the 3800 block on Lansdowne Drive. The text describes the incident as occurring behind a school. Let's zoom in and see where this might have occurred. It must have occurred behind the School of the Future. I used Google Maps to find the coordinates behind this school.

```
ois |>
   filter(id=="21-14") |>
   leaflet(width = 1200, height = 800) |>
   addTiles(
      urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
      options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
                            tileSize = 512,
                            zoomOffset = -1),
      attribution = '© MapTiler © OpenStreetMap contributors') |>
   addCircleMarkers(~lon, ~lat,
                    radius=3, stroke=FALSE,
                    fillOpacity = 1) |>
   addPopups(~lon, ~lat, ~location)
```

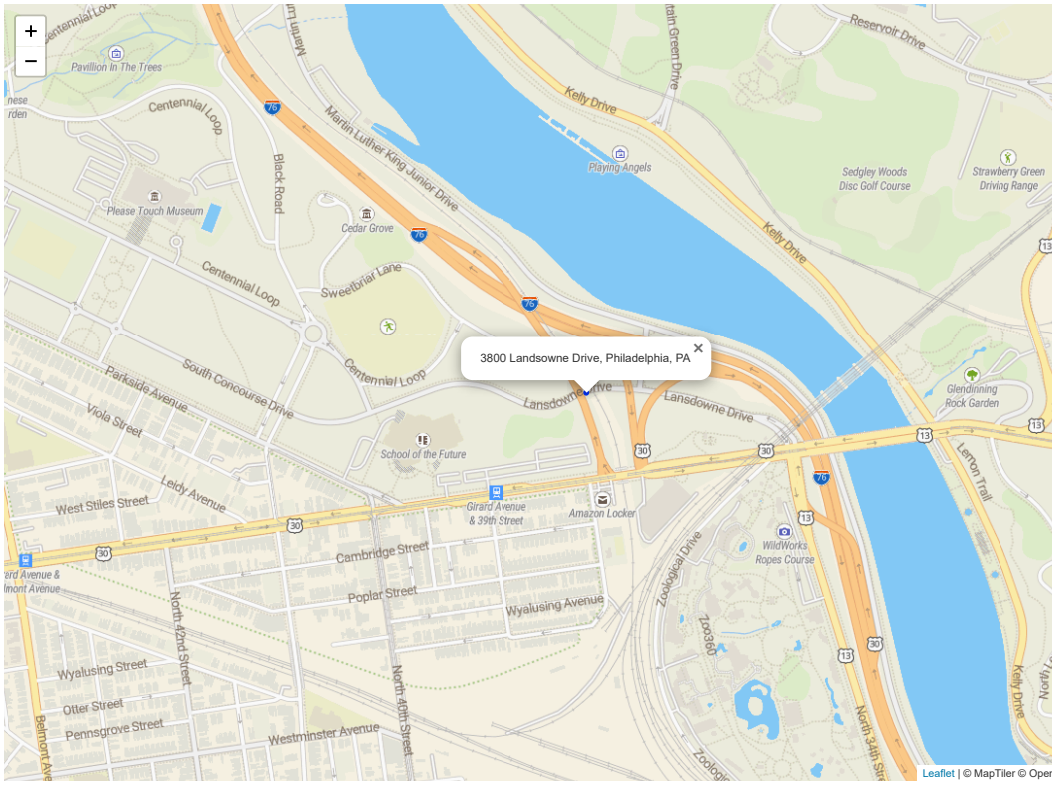

Figure 6: Checking the location of OIS 21-14

Let's record these fixes.

```
ois <- ois |>
   mutate(lat = recode_values(id,
                              "21-14" ~ 39.975984,
                              "25-01" ~ 39.99711,
                              default = lat),
          lon = recode_values(id,
                              "21-14" ~ -75.203309,
                              "25-01" ~ -75.20474,
                              default = lon))
```

Here's a map of all of the incidents. For each incident I have added some pop-up text so that if you click on an incident it will show you the location of the incident and the text describing the incident.

```
ois |>
   leaflet(width = 1200, height = 800) |>
   addTiles(
      urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
      options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
                            tileSize = 512,
                            zoomOffset = -1),
      attribution = '© MapTiler © OpenStreetMap contributors') |>
   addCircleMarkers(~lon, ~lat,
                    radius=4, stroke=FALSE,
                    fillOpacity = 1,
                    popup = paste("<b>",ois$location,"</b><br>",ois$text),
                    popupOptions = popupOptions(autoClose = TRUE,
                                                closeOnClick = FALSE))
```

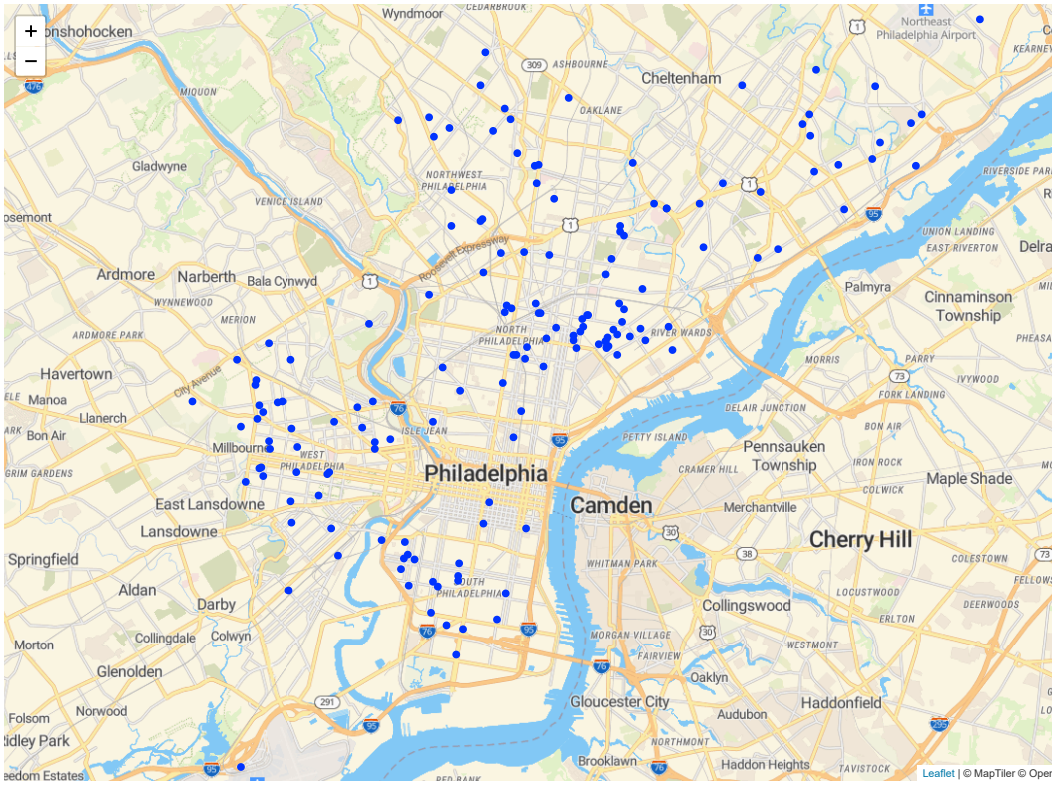

Figure 7: All Philadelphia Officer-involved Shootings

6 Working with shapefiles and coordinate systems

The Philadelphia Police Department divides the city into Police Service Areas (PSAs). The city provides a *shapefile*, a file containing geographic data, that describes the boundaries of the PSAs at Philadelphia's open data site. R can read these files using the `st_read()` function provided in the `sf` (simple features) package. Even though `st_read()` appears to only be accessing `Boundaries_PSA.shp`, you should have all of the `Boundaries_PSA` files in your `08_shapefiles_and_data` folder. The other files have information that `st_read()` needs, like the coordinate system stored in `Boundaries_PSA.prj`. If you do not have all `Boundaries_PSA` files in your folder, then in a few lines you will get errors like "the sfc object should have crs set," meaning that the Coordinate Reference System (CRS) is missing.

```
library(sf)
PPDmap <- st_read("08_shapefiles_and_data/Boundaries_PSA.shp")
```

```
Reading layer `Boundaries_PSA' from data source
  `C:\gitdev\R4crim\08_shapefiles_and_data\Boundaries_PSA.shp'
  using driver `ESRI Shapefile'
Simple feature collection with 66 features and 10 fields
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: -75.28031 ymin: 39.86701 xmax: -74.95575 ymax: 40.13793
Geodetic CRS:  WGS 84
```

You can also get the same PSA boundaries using geoJSON, but note that the .shp file has capitalized column names and the geoJSON version has lowercase column names.

```
library(geojsonsf)
```

```
Warning: package 'geojsonsf' was built under R version 4.6.1
```

```
PPDmap <- geojson_sf("https://opendata.arcgis.com/datasets/8dc58605f9dd484295c7d065694cdc0f_
```

`PPDmap` is an `sf` (simple features) object. It is not unlike a data frame, but it contains a special `geometry` column containing geographic information associated with a row of data. Here are the two columns in `PPDmap` that are of primary interest.

```
PPDmap |> select(psa_num, geometry)
```

```
Simple feature collection with 65 features and 1 field
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: -75.28031 ymin: 39.86701 xmax: -74.95575 ymax: 40.13793
Geodetic CRS:  WGS 84
First 10 features:
   psa_num                       geometry
1      161 POLYGON ((-75.20942 39.9580...
2      032 POLYGON ((-75.16376 39.897,...
3      172 POLYGON ((-75.18531 39.9385...
4      051 POLYGON ((-75.20381 40.0346...
5      153 POLYGON ((-75.03815 40.0138...
6      152 POLYGON ((-75.0572 39.99865...
7      254 POLYGON ((-75.13686 40.0074...
8      251 POLYGON ((-75.13686 40.0074...
9      052 POLYGON ((-75.21618 40.0415...
10     143 POLYGON ((-75.16853 40.0319...
```

The first column shows the PSA number and the second column shows a truncated description of the geometry associated with this row. In this case, `geometry` contains the coordinates of the boundary of the PSA for each row. Use `st_geometry()` to extract these coordinates.

```r
plot(st_geometry(PPDmap))
axis(side=1, cex.axis=0.7) # add x-axis
axis(side=2, cex.axis=0.7) # add y-axis
# extract the center points of each PSA
a <- st_coordinates(st_centroid(st_geometry(PPDmap)))
# add the PSA number to the plot
text(a[,1], a[,2], PPDmap$psa_num, cex=0.5)
```

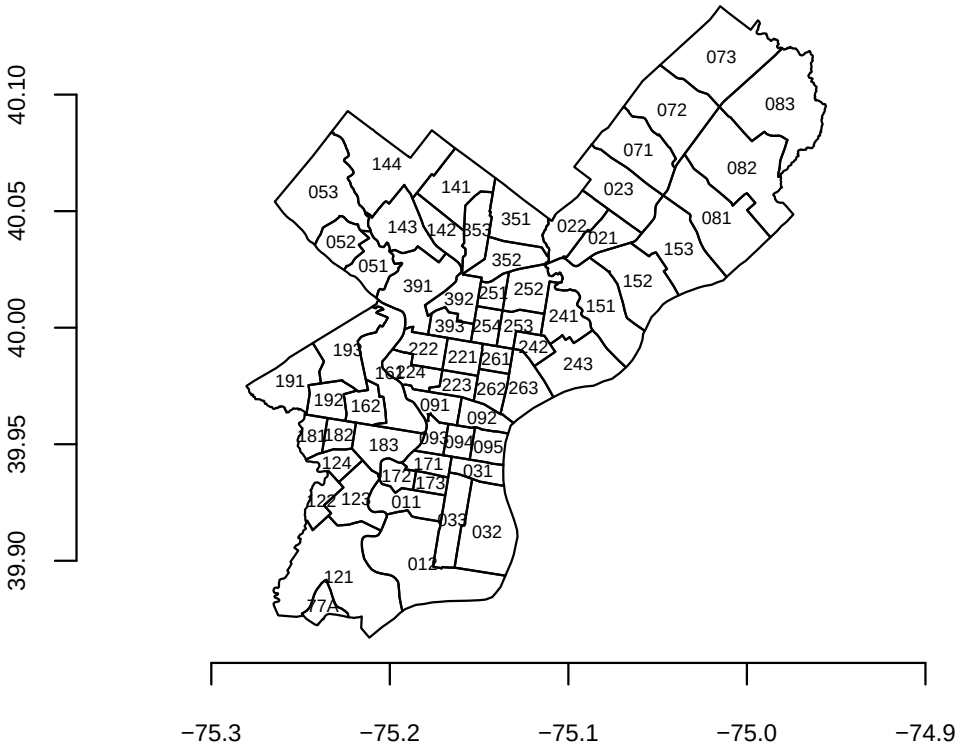

Figure 8: Map of Philadelphia Police Service Areas

We can extract the actual coordinates of one of the polygons if we wish.

```
a <- st_coordinates(PPDmap$geometry[1])
head(a)
```

```
             X        Y L1 L2
[1,] -75.20942 39.95808  1  1
[2,] -75.20942 39.95808  1  1
[3,] -75.20942 39.95808  1  1
[4,] -75.20942 39.95808  1  1
```

```
[5,] -75.20942 39.95809  1  1
[6,] -75.20942 39.95809  1  1
```

```
tail(a)
```

```
               X        Y L1 L2
[303,] -75.20504 39.95754  1  1
[304,] -75.20534 39.95758  1  1
[305,] -75.20578 39.95763  1  1
[306,] -75.20608 39.95767  1  1
[307,] -75.20813 39.95792  1  1
[308,] -75.20942 39.95808  1  1
```

And we can use those coordinates to add additional features to our plot

```
plot(st_geometry(PPDmap))
axis(side=1, cex.axis=0.7)
axis(side=2, cex.axis=0.7)
a <- st_coordinates(st_centroid(st_geometry(PPDmap)))
text(a[,1], a[,2], PPDmap$psa_num, cex=0.5)
a <- st_coordinates(PPDmap$geometry[1])
lines(a[,1], a[,2], col="red", lwd=3)
```

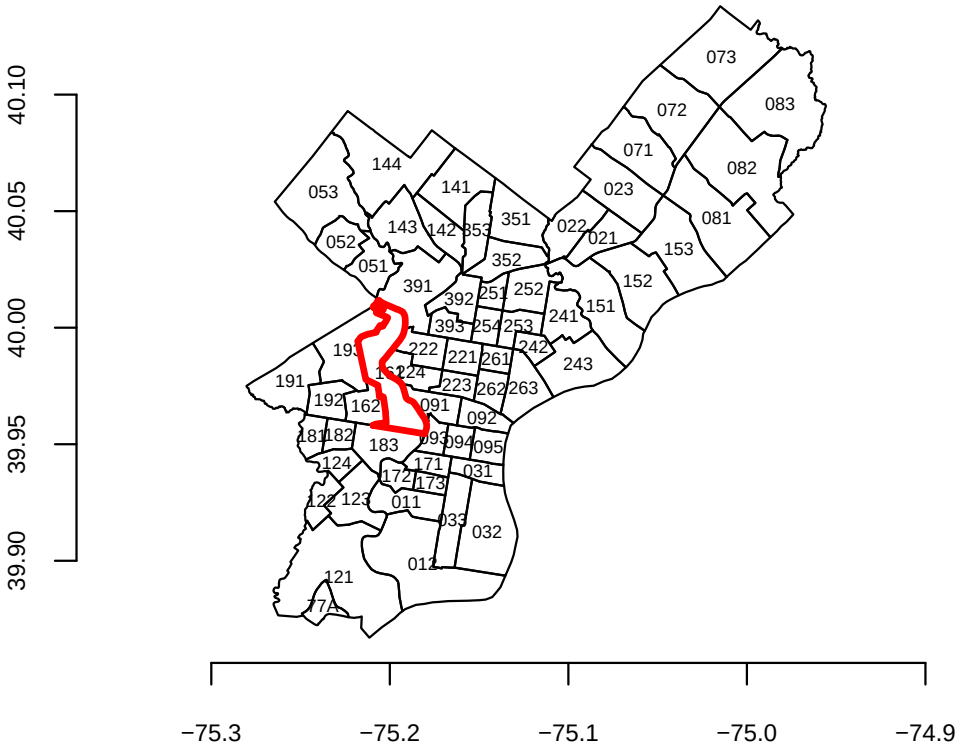

Figure 9: Map of Philadelphia Police Service Areas highlighting the first PSA in `PPDmap`

This highlights PSA 161 in red.

We can also overlay a leaflet map with the `PPDmap` object.

```
PPDmap |>
   leaflet(width = 1200, height = 800) |>
   addPolygons(weight=1, label=~psa_num) |>
   addTiles(
      urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
      options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
```

```
                            tileSize = 512,
                            zoomOffset = -1),
      attribution = '© MapTiler © OpenStreetMap contributors')
```

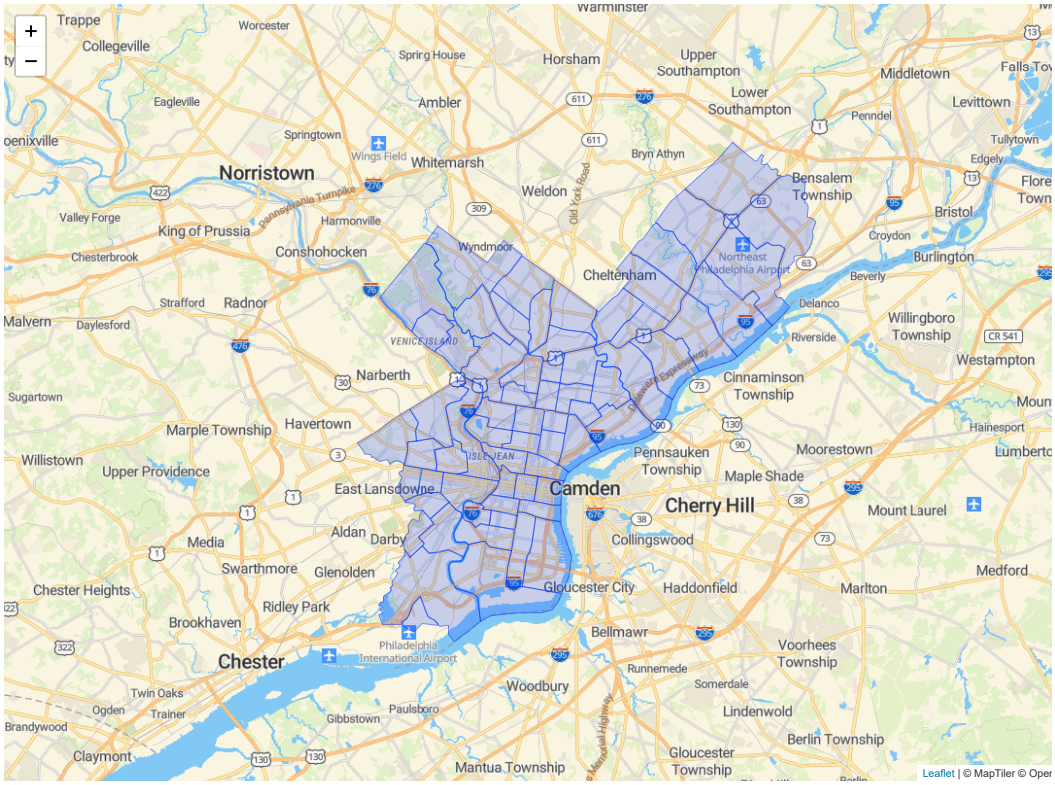

Figure 10: Map layer with the Philadelphia Police Service Areas (PSA)

6.1 Coordinate systems

Geographic datasets that describe locations on the surface of the earth have a "coordinate reference system" (CRS). Let's extract the CRS for `PPDmap`.

```
st_crs(PPDmap)
```

```
Coordinate Reference System:
  User input: 4326
  wkt:
GEOGCS["WGS 84",
      DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
          AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
      PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
      UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9122"]],
      AXIS["Latitude",NORTH],
      AXIS["Longitude",EAST],
    AUTHORITY["EPSG","4326"]]
```

The coordinate system used to describe the PPD boundaries is the World Geodetic System 1984 (WGS84) maintained by the United States National Geospatial-Intelligence Agency, one of several standards to aid in navigation and geography. The European Petroleum Survey Group (EPSG) developed a catalog of different coordinate reference systems (should be no surprise that oil exploration has driven the development of high quality geolocation standards). They have assigned the standard longitude/latitude coordinate system to be EPSG4326. You can find the full collection of coordinate systems at spatialreference.org. You can see in the output above a reference to EPSG 4326.

Many of us are comfortable with the longitude/latitude angular coordinate systems. However, the distance covered by a degree of longitude shrinks as you move towards the poles and only equals the distance covered by a degree of latitude at the equator. In addition, the earth is not very spherical so the coordinate system used for computing distances on the earth surface might need to depend on where you are on the earth surface.

Almost all web mapping tools (Google Maps, ESRI, OpenStreetMap) use the pseudo-Mercator projection (EPSG3857). Let's convert our PPD map to that coordinate system.

```
PPDmap <- st_transform(PPDmap, crs=3857)
st_crs(PPDmap)
```

```
Coordinate Reference System:
  User input: EPSG:3857
  wkt:
PROJCRS["WGS 84 / Pseudo-Mercator",
    BASEGEOGCRS["WGS 84",
        ENSEMBLE["World Geodetic System 1984 ensemble",
            MEMBER["World Geodetic System 1984 (Transit)"],
            MEMBER["World Geodetic System 1984 (G730)"],
            MEMBER["World Geodetic System 1984 (G873)"],
            MEMBER["World Geodetic System 1984 (G1150)"],
            MEMBER["World Geodetic System 1984 (G1674)"],
            MEMBER["World Geodetic System 1984 (G1762)"],
            MEMBER["World Geodetic System 1984 (G2139)"],
            MEMBER["World Geodetic System 1984 (G2296)"],
            ELLIPSOID["WGS 84",6378137,298.257223563,
                LENGTHUNIT["metre",1]],
            ENSEMBLEACCURACY[2.0]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4326]],
    CONVERSION["Popular Visualisation Pseudo-Mercator",
        METHOD["Popular Visualisation Pseudo Mercator",
            ID["EPSG",1024]],
        PARAMETER["Latitude of natural origin",0,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8801]],
        PARAMETER["Longitude of natural origin",0,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8802]],
        PARAMETER["False easting",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8806]],
        PARAMETER["False northing",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8807]]],
    CS[Cartesian,2],
        AXIS["easting (X)",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["northing (Y)",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    USAGE[
```

```
        SCOPE["Web mapping and visualisation."],
        AREA["World between 85.06°S and 85.06°N."],
        BBOX[-85.06,-180,85.06,180]],
    ID["EPSG",3857]]
```

The CRS now indicates that this is a Mercator projection with distance measured in meters (`LENGTHUNIT["metre",1]`). There are special coordinate systems for every part of the world. A useful coordinate system for the Philadelphia area is EPSG2272. Let's convert our PPD map to that coordinate system.

```
PPDmap <- st_transform(PPDmap, crs=2272)
st_crs(PPDmap)
```

```
Coordinate Reference System:
  User input: EPSG:2272
  wkt:
PROJCRS["NAD83 / Pennsylvania South (ftUS)",
    BASEGEOGCRS["NAD83",
        DATUM["North American Datum 1983",
            ELLIPSOID["GRS 1980",6378137,298.257222101,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4269]],
    CONVERSION["SPCS83 Pennsylvania South zone (US survey foot)",
        METHOD["Lambert Conic Conformal (2SP)",
            ID["EPSG",9802]],
        PARAMETER["Latitude of false origin",39.3333333333333,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8821]],
        PARAMETER["Longitude of false origin",-77.75,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8822]],
        PARAMETER["Latitude of 1st standard parallel",40.9666666666667,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8823]],
        PARAMETER["Latitude of 2nd standard parallel",39.9333333333333,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8824]],
        PARAMETER["Easting at false origin",1968500,
            LENGTHUNIT["US survey foot",0.304800609601219],
            ID["EPSG",8826]],
        PARAMETER["Northing at false origin",0,
            LENGTHUNIT["US survey foot",0.304800609601219],
```

```
        ID["EPSG",8827]]],
    CS[Cartesian,2],
        AXIS["easting (X)",east,
            ORDER[1],
            LENGTHUNIT["US survey foot",0.304800609601219]],
        AXIS["northing (Y)",north,
            ORDER[2],
            LENGTHUNIT["US survey foot",0.304800609601219]],
    USAGE[
        SCOPE["Engineering survey, topographic mapping."],
        AREA["United States (USA) - Pennsylvania - counties of Adams;
         ↪ Allegheny; Armstrong; Beaver; Bedford; Berks; Blair; Bucks;
         ↪ Butler; Cambria; Chester; Cumberland; Dauphin; Delaware; Fayette;
         ↪ Franklin; Fulton; Greene; Huntingdon; Indiana; Juniata;
         ↪ Lancaster; Lawrence; Lebanon; Lehigh; Mifflin; Montgomery;
         ↪ Northampton; Perry; Philadelphia; Schuylkill; Snyder; Somerset;
         ↪ Washington; Westmoreland; York."],
        BBOX[39.71,-80.53,41.18,-74.72]],
    ID["EPSG",2272]]
```

This coordinate system is the Lambert Conformal Conic (LCC). This particular projection of the `PPDmap` is tuned to provide good precision for the southern part of Pennsylvania (note the parallel coordinates are at the latitude of southern Pennsylvania and the meridian is a little west of Philadelphia) and distances are measured in feet (note the `LENGTHUNIT["US survey foot",0.304800609601219]` tag in the CRS description).

Let's transform back to longitude/latitude. It really is best to work using a different coordinate system, but I'm going to stick with longitude/latitude so that the values make a little more sense to us.

```
PPDmap <- st_transform(PPDmap, crs=4326)
```

Now both the PPD data and the polygons are on the same scale

```
plot(st_geometry(PPDmap), axes=TRUE, cex.axis=0.7)
points(lat~lon, data=ois, col=rgb(1,0,0,0.5), pch=16)
```

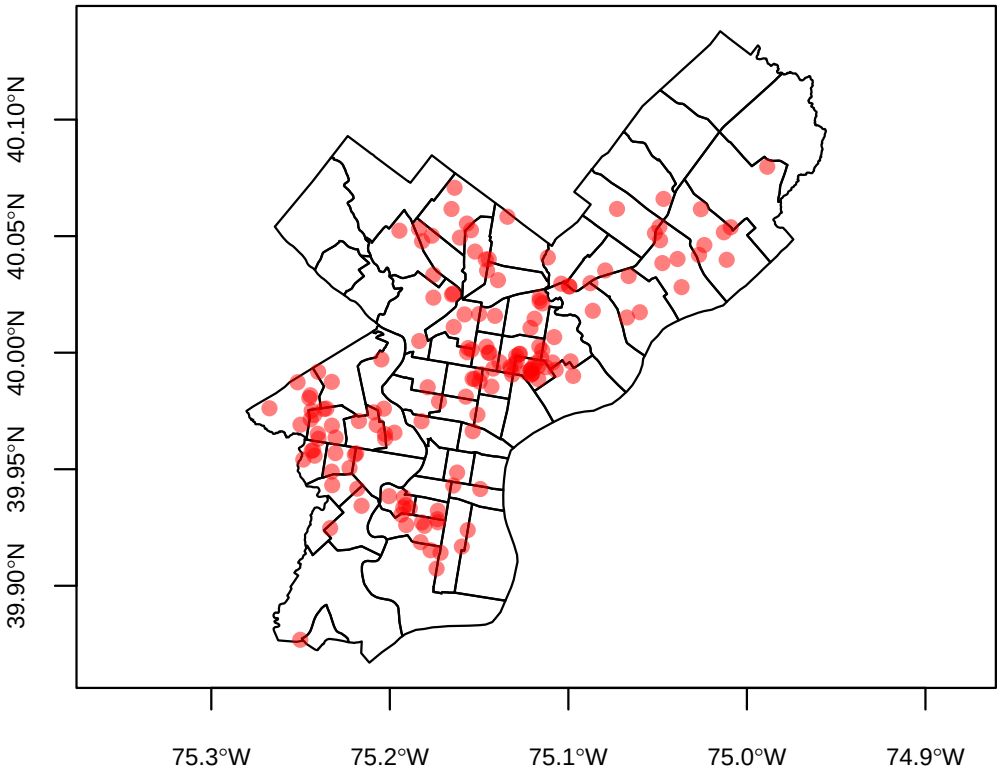

Figure 11: Map of OISs over PSAs

To make the dots a little transparent, I have used the `rgb()` function with which you can mix red, green, and blue colors and set the transparency. The `1` tells `rgb()` to use maximum red. The two `0`s tell `rgb()` to use no green or blue. The 0.5 tells `rgb()` to make the dots halfway transparent.

6.2 Spatial joins

A *spatial join* is the process of linking two data sources by their geography. For the case of the OIS data, we want to know how many OISs occurred in each PSA. To do this we need to drop

each OIS point location into the PSA polygons and have R tell us which polygon contains each OIS.

First we need to convert our `ois` data frame to an `sf` object, communicating to R that the `lon` and `lat` columns are special. At this stage we also have to communicate which coordinate system the `lon` and `lat` values use. `st_as_sf()` converts an R object into an `sf` object.

```
ois <- st_as_sf(ois,
                coords=c("lon","lat"),
                crs=4326)
ois |> select(-text, -url, -addrmatch)
```

```
Simple feature collection with 159 features and 8 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: -75.26743 ymin: 39.87683 xmax: -74.98861 ymax: 40.07978
Geodetic CRS:  WGS 84
First 10 features:
      id                                      location subInjury subArrest
1  26-16     2050 North 54th Street, Philadelphia, PA    Killed       N/A
2  26-14     2850 Kensington Avenue, Philadelphia, PA   Wounded       Yes
3  26-12        6950 Lawnton Avenue, Philadelphia, PA       N/A       N/A
4  26-11        5450 Webster Street, Philadelphia, PA    Killed       N/A
5  26-08           3450 Hess Street, Philadelphia, PA       N/A       N/A
6  26-06 1450 North Robinson Street, Philadelphia, PA       N/A       N/A
7  26-05    4150 North Broad Street, Philadelphia, PA        No       Yes
8  26-03      4850 Blakiston Street, Philadelphia, PA       N/A       N/A
9  26-02          2850 Sebring Road, Philadelphia, PA       N/A       N/A
10 26-01   450 East Rockland Street, Philadelphia, PA   Unknown        No
   offInjury       date score      addrtype                   geometry
1        Yes 2026-06-13   100 StreetAddress  POINT (-75.23251 39.9875)
2         No 2026-05-23   100  PointAddress POINT (-75.12113 39.99248)
3         No 2026-04-10   100 StreetAddress POINT (-75.13415 40.05832)
4         No 2026-04-07   100  PointAddress POINT (-75.23256 39.94897)
5         No 2026-03-03   100 StreetAddress POINT (-75.02385 40.04619)
6         No 2026-01-29   100 StreetAddress  POINT (-75.24235 39.9733)
7         No 2026-01-19   100  PointAddress POINT (-75.14997 40.01659)
8         No 2026-01-07   100 StreetAddress POINT (-75.01132 40.03988)
9         No 2026-01-05   100  PointAddress POINT (-75.02579 40.06158)
10        No 2026-01-04   100 StreetAddress  POINT (-75.11609 40.0236)
```

You can see that `ois` now has one of those special geometry columns. We can plot the OISs over the PSA map.

```r
plot(st_geometry(PPDmap), axes=TRUE, cex.axis=0.7)
plot(st_geometry(ois), add=TRUE, col=rgb(1,0,0,0.5), pch=16)
```

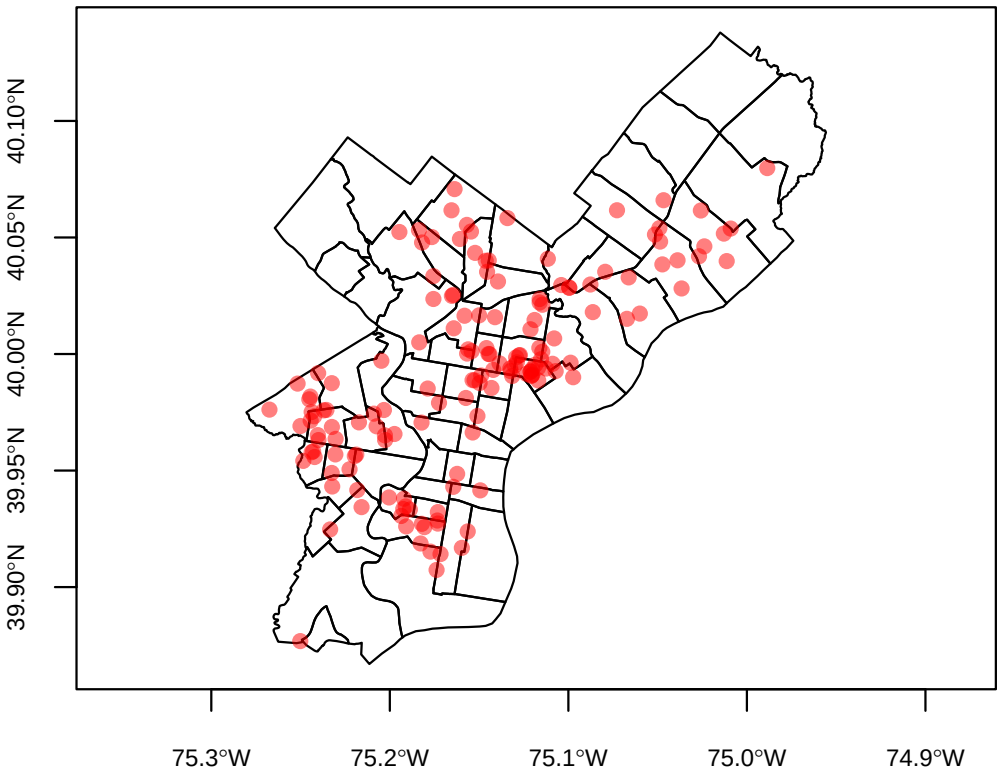

Figure 12: Map of OISs over PSAs using `sf` objects

`st_join()` will match each row in `ois` to each polygon in PSA. When linking a data frame with *point* data (OIS location) with a data frame with *polygon* data (like PSAs), the points will join with the polygons in which they land. I just want to add the `psa_num` column out of the `PPDmap` to our `ois` data.

```
PSAlookup <- ois |>
  st_join(PPDmap |> select(psa_num))
PSAlookup |>
   select(id, date, location, psa_num, geometry) |>
   head()
```

```
Simple feature collection with 6 features and 4 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: -75.24235 ymin: 39.94897 xmax: -75.02385 ymax: 40.05832
Geodetic CRS:  WGS 84
     id       date                                    location psa_num
1 26-16 2026-06-13     2050 North 54th Street, Philadelphia, PA     193
2 26-14 2026-05-23     2850 Kensington Avenue, Philadelphia, PA     242
3 26-12 2026-04-10        6950 Lawnton Avenue, Philadelphia, PA     351
4 26-11 2026-04-07        5450 Webster Street, Philadelphia, PA     182
5 26-08 2026-03-03           3450 Hess Street, Philadelphia, PA     081
6 26-06 2026-01-29 1450 North Robinson Street, Philadelphia, PA     192
                    geometry
1  POINT (-75.23251 39.9875)
2 POINT (-75.12113 39.99248)
3 POINT (-75.13415 40.05832)
4 POINT (-75.23256 39.94897)
5 POINT (-75.02385 40.04619)
6  POINT (-75.24235 39.9733)
```

Now our `PSAlookup` contains everything from `ois` but also adds a new column `psa_num`.

Let's examine the PSA with the most OISs and highlight their incidents on the map.

```
PSAlookup |>
   count(psa_num) |>
   slice_max(n)
```

```
Simple feature collection with 1 feature and 2 fields
Geometry type: MULTIPOINT
Dimension:     XY
Bounding box:  xmin: -75.12886 ymin: 39.98866 xmax: -75.11246 ymax: 39.99642
Geodetic CRS:  WGS 84
  psa_num  n                       geometry
1     242 12 MULTIPOINT ((-75.11246 39.9...
```

```
plot(st_geometry(PPDmap), axes=TRUE, cex.axis=0.7)
PSAlookup |>
   st_join(PSAlookup |>
              count(psa_num) |>
              slice_max(n) |>
              select(-psa_num),
           left=FALSE) |> # right join
  st_geometry() |>
  plot(add=TRUE, col=rgb(0,1,0,0.5), pch=16)
```

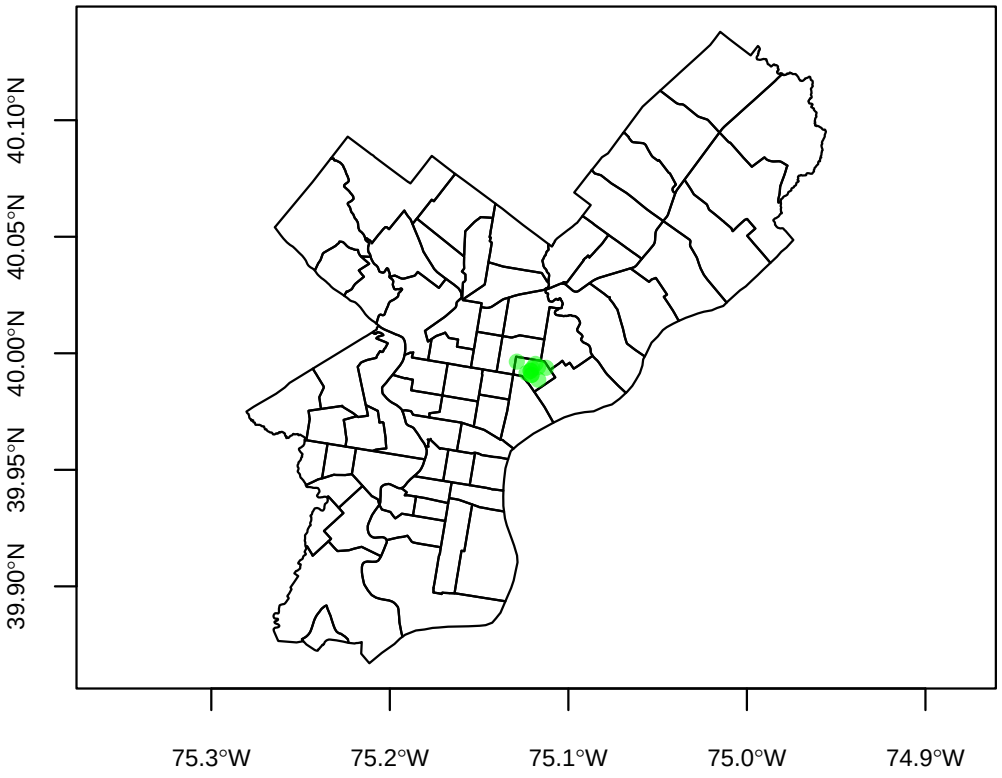

Figure 13: Map of PSA with the largest number of OISs

Let's identify which OISs occurred in the same PSA as the University of Pennsylvania. We've already geocoded Penn and have its coordinates. Let's join it with `PPDmap` to find out which PSA it is in.

```
gcPenn
```

```
                                              address location.x location.y
1 3718 Locust Walk, Philadelphia, Pennsylvania, 19104  -75.19789   39.95204
  score                                attributes.Match_addr
1   100 3718 Locust Walk, Philadelphia, Pennsylvania, 19104
  attributes.Addr_type extent.xmin extent.ymin extent.xmax extent.ymax
1         PointAddress   -75.19889    39.95104   -75.19689    39.95304
        lon      lat
1 -75.19789 39.95204
```

```
st_as_sf(gcPenn,
         coords=c("lon","lat"),
         crs=4326) |> # tell R that the coords are lon/lat
   st_join(PPDmap) |>
   select(psa_num)
```

```
Simple feature collection with 1 feature and 1 field
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: -75.19789 ymin: 39.95204 xmax: -75.19789 ymax: 39.95204
Geodetic CRS:  WGS 84
  psa_num                   geometry
1     183 POINT (-75.19789 39.95204)
```

Now we see that Penn is in PSA 183 and we can highlight those points on the map.

```
plot(st_geometry(PPDmap), axes=TRUE, cex.axis=0.7)
PSAlookup |>
  filter(psa_num=="183") |>
  st_geometry() |>
  plot(add=TRUE, col="blue", pch=16)
```

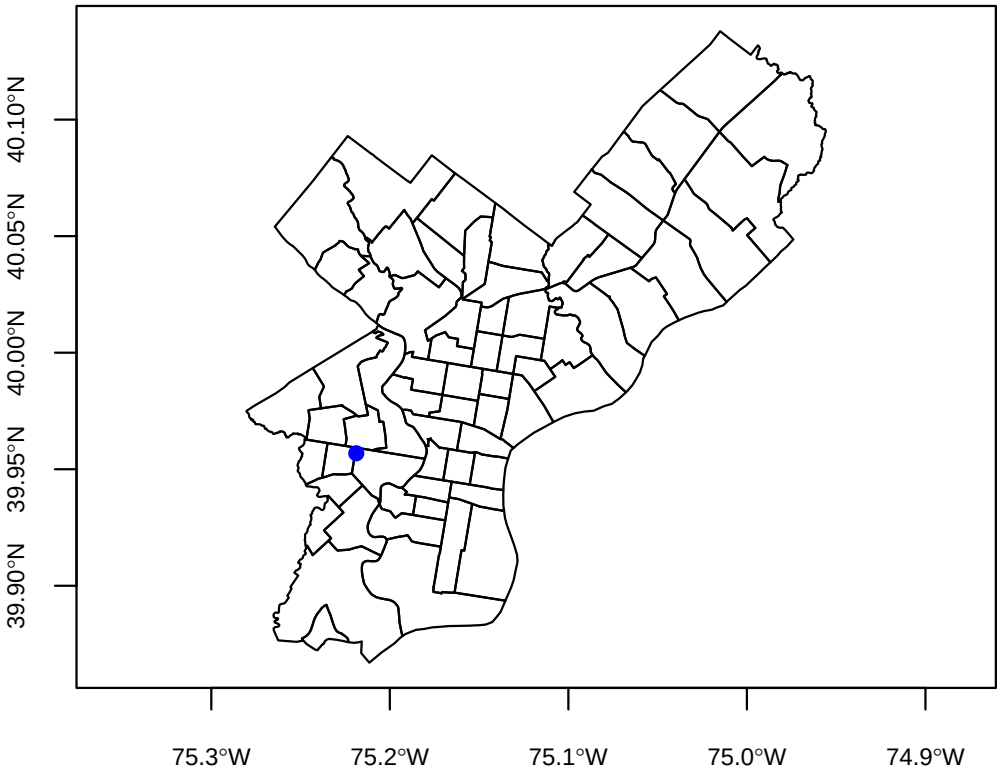

Figure 14: Map of OISs in the same PSA as Penn

We read about this incident earlier when fixing the OIS incident locations in Section 5.

6.3 Coloring a map based on the value of a feature

Lastly, we will tabulate the number of OISs in each PSA and color the map by the number of OISs.

```r
# merge the shooting count into the PPDmap data
PPDmap <- PPDmap |>
   left_join(PSAlookup |>
               count(psa_num) |>
               st_drop_geometry(),
             by=join_by(psa_num)) |>
  rename(nShoot=n) |>
  mutate(nShoot=replace_na(nShoot, 0))

head(PPDmap)
```

```
Simple feature collection with 6 features and 8 fields
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: -75.22561 ymin: 39.8936 xmax: -75.01134 ymax: 40.05675
Geodetic CRS:  WGS 84
  objectid psa_num dist_numc sect_code div_code Shape__Area Shape__Length
1        1     161        16      <NA>     SWPD    14150635     26235.214
2        2     032        03      <NA>      SPD    16904849     16923.929
3        3     172        17      <NA>      SPD     3295705      7654.915
4        4     051        05      <NA>     NWPD     6937874     13707.934
5        5     153        15      <NA>     NEPD    16729015     18836.087
6        6     152        15      <NA>     NEPD    14742925     17152.130
  nShoot                       geometry
1      3 POLYGON ((-75.20942 39.9580...
2      1 POLYGON ((-75.16376 39.897,...
3      5 POLYGON ((-75.18531 39.9385...
4      0 POLYGON ((-75.20381 40.0346...
5      5 POLYGON ((-75.03815 40.0138...
6      3 POLYGON ((-75.0572 39.99865...
```

We can see that `PPDmap` now has a new `nShoot` column. A histogram will show what kinds of counts we observe in the PSAs.

```r
hist(PPDmap$nShoot, xlab="Number of OISs", ylab="Number of PSAs", main="")
```

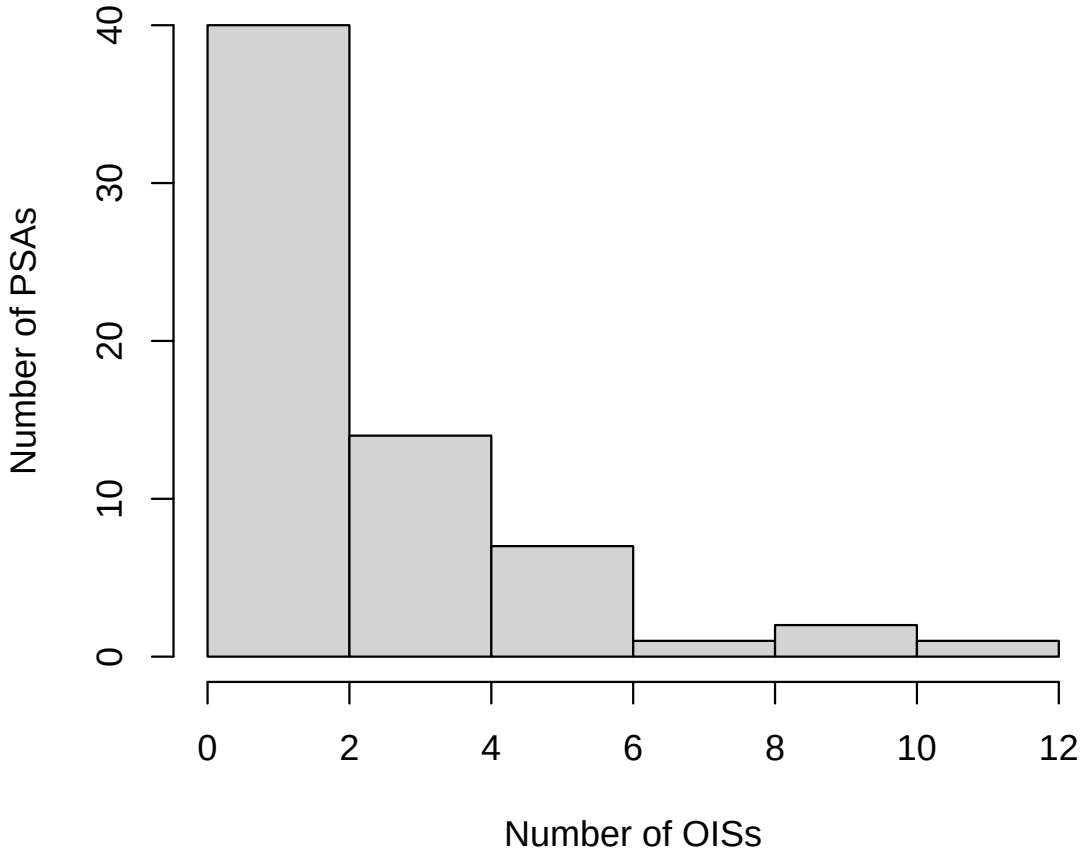

Figure 15: Histogram of OIS incident counts by PSA

Let's discretize the OIS counts into a few categories.

```
PPDmap <- PPDmap |>
  mutate(catShoot =
           cut(nShoot,
               breaks=c(0,1,2,3,4,8,Inf),
               right=FALSE))
```

`cut()` converts all of the individual counts into categories, like [1,5) or [25,30). For each of

these categories we will associate a color for the map. `heat.colors()` will generate a sequence of colors in the yellow, orange, red range.

```
a <- data.frame(catShoot = levels(PPDmap$catShoot),
                col      = rev(heat.colors(6,1)))
a
# some other color options
#    col = rev(rainbow(6,1))
#    or generate a range of red colors
#    col = rgb(seq(0,1,length=6),0,0,1)
```

```
  catShoot       col
1    [0,1) #FFFF80FF
2    [1,2) #FFFF00FF
3    [2,3) #FFBF00FF
4    [3,4) #FF8000FF
5    [4,8) #FF4000FF
6  [8,Inf) #FF0000FF
```

These are eight digit codes describing the color. The first two digits correspond to red, digits three and four correspond to green, digits five and six correspond to blue, and the last two digits correspond to transparency. These are hexadecimal numbers (base 16). Hexadecimal numbers use the digits 0-9, like normal decimal system numbers, and then denote 10 as A, 11 as B, on up to 15 as F. So FF as a decimal is $15 \times 16 + 15 = 255$, which is the maximum value for a two digit hexadecimal. The hexadecimal 80 as a decimal is $8 \times 16 + 0 = 128$, which is in the middle of the range 0 to 255. So the first color code, FFFF80FF, means maximum red, maximum green, half blue, and not transparent at all. This mixture is known more commonly as "yellow".

Now we join `PPDmap` with our color lookup table in `a` and plot it.

```
# match the color to category
PPDmap <- PPDmap |>
  left_join(a, by=join_by(catShoot))

PPDmap |>
  st_geometry() |>
  plot(col=PPDmap$col, border="black")
# add the number of shootings
#   warning is reminder that it is collapsing polygon data down to a point
b <- st_coordinates(st_centroid(PPDmap))
```

```
Warning: st_centroid assumes attributes are constant over geometries
```

```
text(b[,1], b[,2], PPDmap$nShoot, cex=0.7)
```

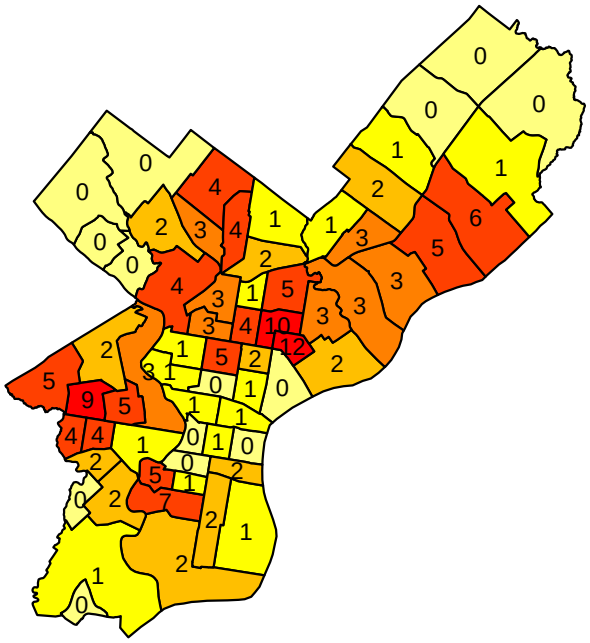

Figure 16: PSAs color-coded by number of OIS incidents

Those PSAs with the fewest shootings are a very pale yellow. As we examine PSAs with a greater number of OISs, their colors get redder and redder.

`sf` objects have their own default plotting method to accomplish these kinds of "heat maps." The default palette is generated with `sf.colors()`.

```
PPDmap |>
  select(nShoot) |>
  plot(main="")
```

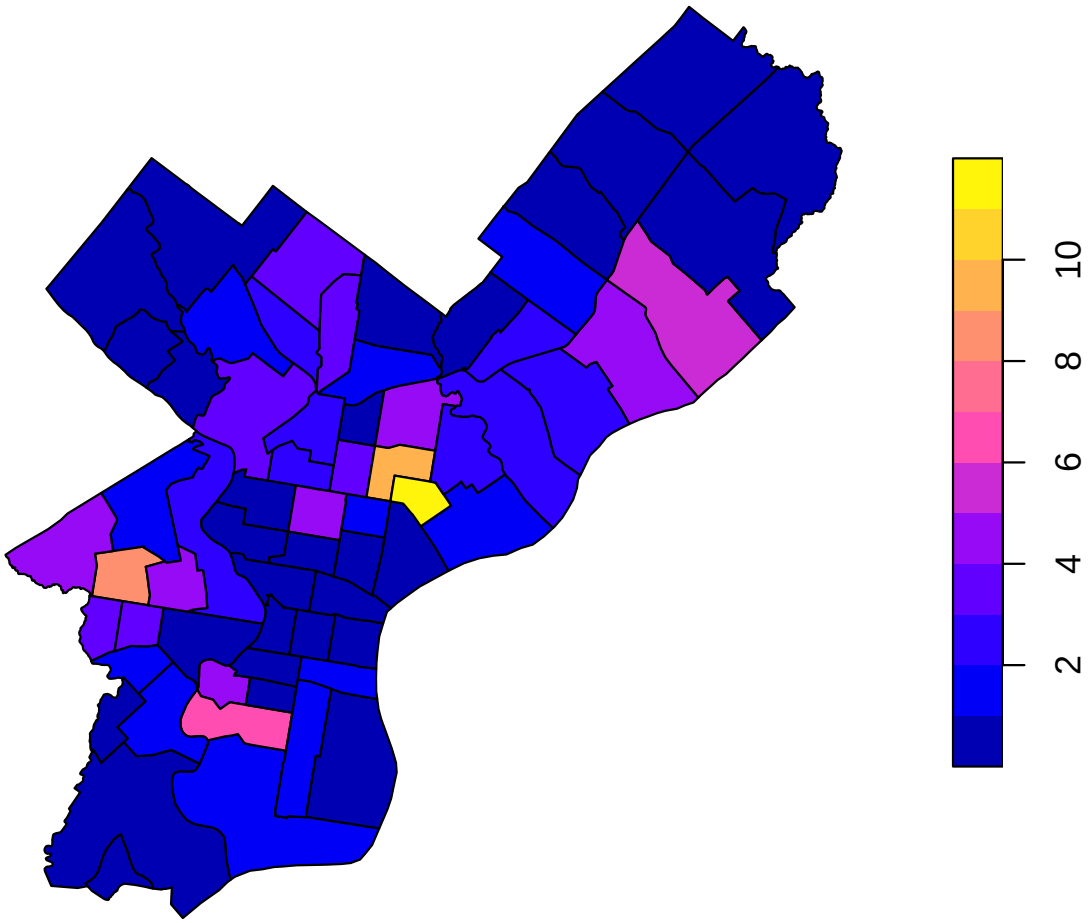

Figure 17: PSAs color-coded by number of OIS incidents, default `sf` colors

You can change the color palette, for example, by using the yellow-orange-red palette from HCL (hue, chroma, luminance) color set.

```
PPDmap |>
  select(nShoot) |>
  plot(main="",
       pal = hcl.colors(6, "YlOrRd", rev = TRUE),
       nbreaks = 6)
```

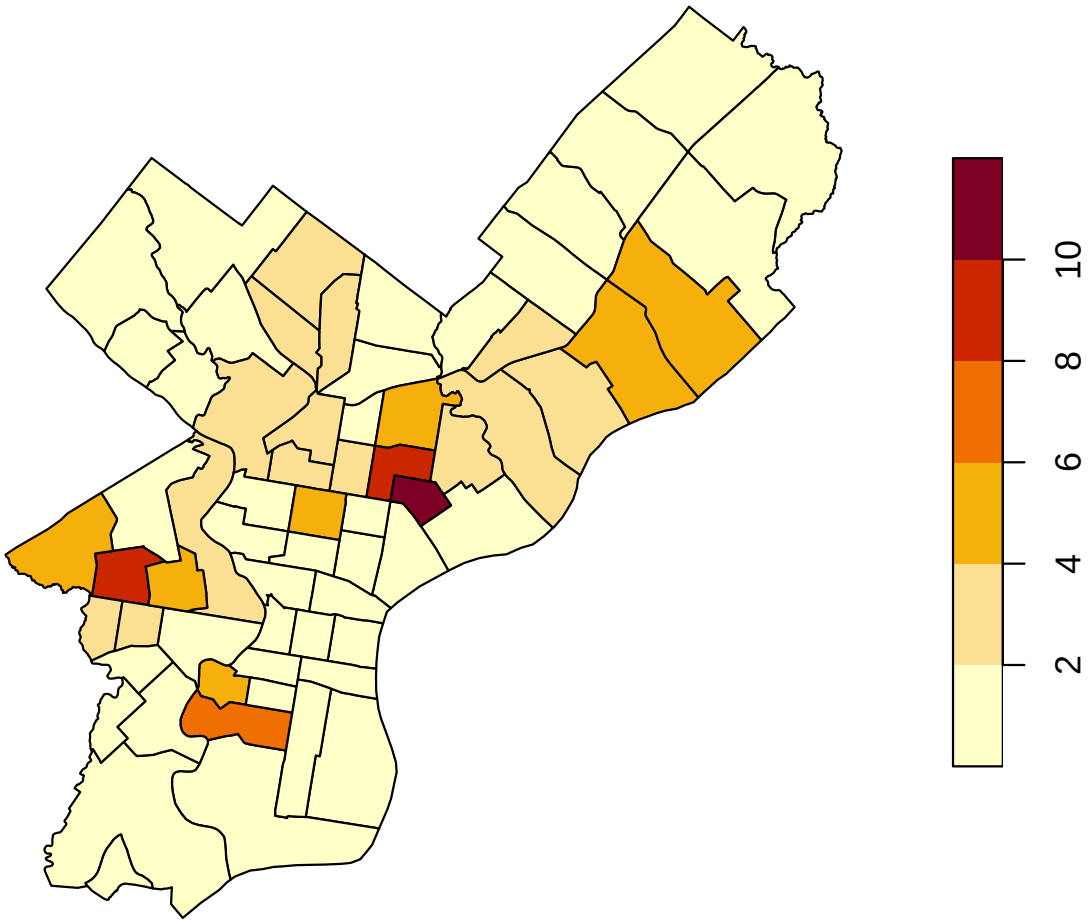

Figure 18: PSAs color-coded by number of OIS incidents, YlOrRd HCL palette

Lastly, I will share the classic *viridis* palette, noted for being color-blind friendly and "perceptually uniform".

```r
# classic viridis palette
library(viridis)
```

```
Warning: package 'viridis' was built under R version 4.6.1
```

```
Loading required package: viridisLite
```

```r
PPDmap |>
  select(nShoot) |>
  plot(main="",
       pal = viridis_pal(option="D"),
       nbreaks = 12)
```

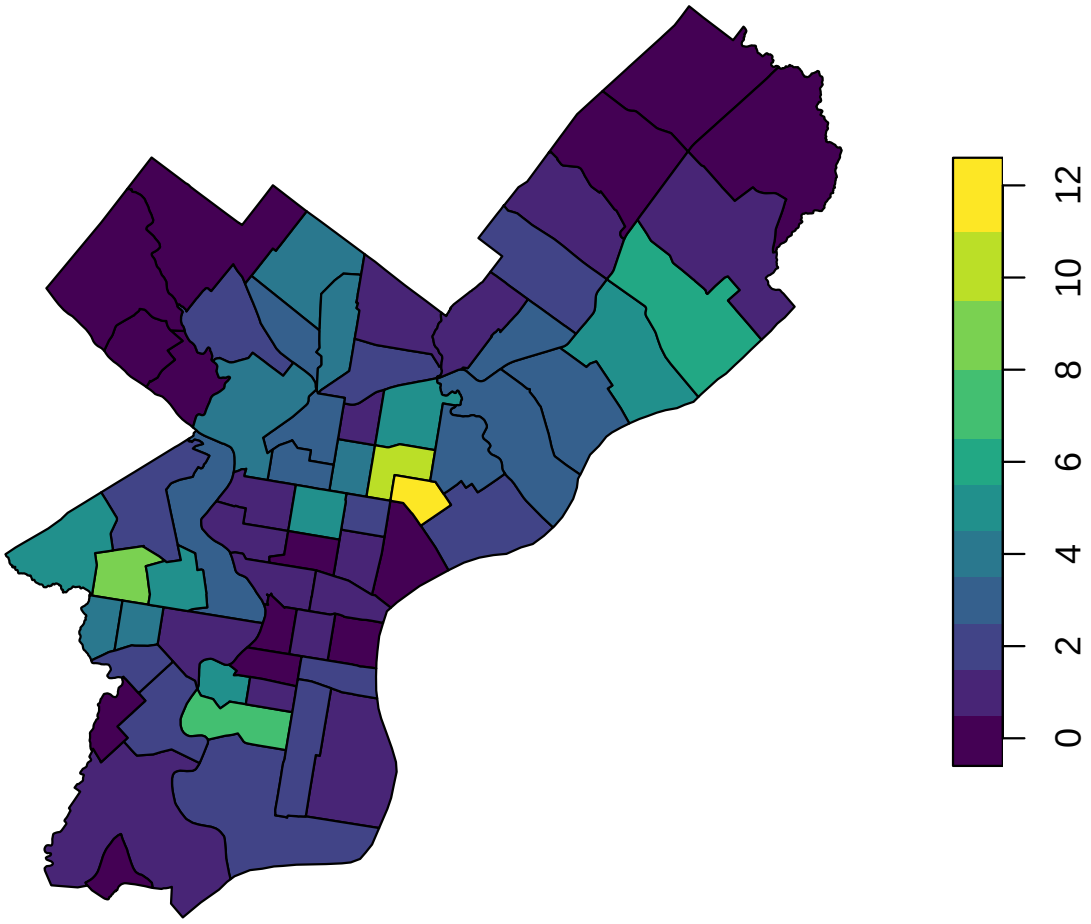

Figure 19: PSAs color-coded by number of OIS incidents, viridis palette

And a leaflet version to end on.

```
PPDmap <- PPDmap |>
  mutate(label = paste("PSA:",psa_num, "Count:",nShoot))

PPDmap |>
   leaflet(width = 1200, height = 800) |>
   addPolygons(weight=1, col=~col, label=~label) |>
   addTiles(
      urlTemplate = "https://api.maptiler.com/maps/streets/{z}/{x}/{y}.png?key={key}",
```

```
      options = tileOptions(key = Sys.getenv("MAPTILER_API_KEY"),
                            tileSize = 512,
                            zoomOffset = -1),
      attribution = '© MapTiler © OpenStreetMap contributors')
```

Figure 20: Map of OIS counts by PSA

7 Using a large language model to extract information from text

We have studied all the readily available structured data about the OIS incidents. However, we barely made use of the text of the incident reports… aside from using some regular expressions to extract the dates. For example, we might want to know

1. Was the suspect armed?
2. Had the suspect harmed or threatened to harm someone?
3. Did police gunfire strike a person, a dog, or no one?
4. If a person was shot, was that person transported to the hospital for treatment of the shooting?
5. To which hospital were they transported?

We could try to use regular expressions to extract this information from the text, but the descriptions are not standardized and there are many ways to express the same fact. The standard practice for decades has been to read (or hire people to read) all the reports and code the information into structured data fields. This is expensive and time-consuming. A large language model (LLM) can read the text and extract the information for us.

In this section, we are going to use the OpenAI API to send the text of each OIS description to a large language model and ask it to extract structured information. The model will return a JSON object with the requested fields. We will then combine all of these JSON objects into a data frame for analysis. It can make mistakes… and humans do too. I read all of these reports and recorded my answers and compared them to the LLM. I found a couple that the LLM got wrong, but it also found one that I got wrong. Depending on your tolerance for errors, you may want to have a human review the LLM's output. The LLM can also provide a short excerpt from the text that supports its coding decision. This is useful for quality control and for training a human reviewer.

The method I will show you here is not free. You need to create an account on the OpenAI Developer Platform and obtain an API key. You need to give a credit card and you will be billed for each request you make to the API. The cost is based on the model you use and the number of tokens in your request and response (tokens are units of text, often a word or part of a word). You can find the current pricing on the OpenAI website. You should also be aware that the API is rate-limited, meaning you can only make a certain number of requests per minute. You can find the current rate limits on the OpenAI website. That said, the cost of this exercise for me was a few cents. If you decide to get an API key, be sure to set a spend limit. There are some bugs that will end up repeatedly sending requests to the API and you can end up with a large bill if you do not set a limit.

> **Expect the API to change**
>
> LLM APIs, available models, prices, rate limits, request fields, and response formats change frequently. The code below follows the OpenAI Responses API at the time these notes were written. Check the current OpenAI API documentation and current model list before using it. You may need to change the model name or modify the request and response-processing code as standards and protocols develop. Never assume that an old classroom example is still the least expensive or currently recommended approach.

Do not send sensitive or protected data to this OpenAI API. If you have sensitive data, then you can install a local LLM on your own computer to process the data, but they do require a lot of computing power to run locally. Some organizations (like Penn) do have local LLMs that are HIPAA compliant that do not send any data outside of the organization's computers.

7.1 Acquire and protect an API key

OpenAI API use requires an account on the OpenAI Developer Platform. API billing is separate from a ChatGPT subscription. After creating an account, create a secret key on the API keys page. If the platform offers key permissions, give the key only the permissions needed for this exercise. A key is shown in full only when it is created, so copy it to a secure location at that time. Treat an API key like a password. Do not paste it into an R script, a Quarto file, an email, a chat message, or a file tracked by Git. If a key is exposed, delete it on the API keys page and create a replacement.

While you can paste your API key directly into your script, I recommend you store the key in your user-level `.Renviron` file. This file is outside the class project and R reads it when a new session starts. Run the following command in the R console.

Not sure where your .Renviron file is? Run `path.expand("~/.Renviron")` in the R console.

```
path.expand("~/.Renviron")
```

```
[1] "C:/Users/greg_/OneDrive/Documents/.Renviron"
```

Now edit the file.

```
file.edit(path.expand("~/.Renviron"))
```

Add one line to that file, replacing the example text with the key you copied.

```
OPENAI_API_KEY=replace_with_your_secret_key
```

Save the file and restart R. Load the key into an object when the script needs it. Assigning the result does not display the key in the console, but it will show up in your Environment pane or if you run `openaiKey` at the console. You can also check that the key is present with `Sys.getenv("OPENAI_API_KEY")`. If you see an empty string, check your `.Renviron` file and restart R.

```
openaiKey <- Sys.getenv("OPENAI_API_KEY")

if (!nzchar(openaiKey))
   stop("OPENAI_API_KEY was not found. Check .Renviron and restart R")
```

You can also set your API key directly in the console with

```
Sys.setenv(OPENAI_API_KEY="your_very_secret_API_key")
```

7.2 Ask the model for structured data

We will use the Responses API through `httr2`. The request sends one OIS description to the LLM and will return a structured response. The `who` field distinguishes incidents in which police gunfire struck a person, struck a dog, or struck no one. This helps prevent us from coding a person as transported because of a police shooting when officers missed and the person was transported for an unrelated reason.

```
library(httr2)

extractOIS <- function(description,
                       model = "gpt-5.6-luna")
{
   outputSchema <- list(
      type = "object",
      properties = list(
         who = list(
            type = "string",
            enum = c("person", "dog", "no one"),
            description = paste(
               "Who was struck by police gunfire. Code person if at least",
               "one person was struck, dog if a dog but no person was struck,",
               "and no one if neither a person nor a dog was struck")),
         transported = list(
            type = "string",
            enum = c("yes", "no", "unknown"),
```

```
            description = paste(
               "Whether a person struck by police gunfire was transported",
               "for treatment of that shooting")),
         hospital = list(
            type = "string",
            description = paste(
               "Hospital name stated in the description,",
               "not transported, or unknown")),
         evidence = list(
            type = "string",
            description = paste(
               "A short excerpt supporting the coding,",
               "or an empty string when there is no evidence"))
      ),
      required = c("who", "transported", "hospital", "evidence"),
      additionalProperties = FALSE)

   prompt <- paste(
      "Code this police narrative for research.",
      "Use only information explicitly stated in the narrative.",
      "Do not infer a hospital from geography.",
      "For who, code person if police gunfire struck at least one person,",
      "dog if it struck a dog but no person, and no one if officers missed",
      "or neither a person nor a dog was struck.",
      "Code transported as yes only when a person struck by police gunfire",
      "was transported for treatment of that shooting.",
      "Code transported as no when who is dog or no one, or when the narrative",
      "explicitly says the person who was shot was not transported.",
      "Code transported as unknown when a person was struck but the narrative",
      "does not establish whether that person was transported for treatment.",
      "Do not comment on other victims or police officers injured",
      "or transported to the hospital",
      "Narrative:", description)

   response <- request("https://api.openai.com/v1/responses") |>
      req_auth_bearer_token(openaiKey) |>
      req_body_json(list(
         model = model,
         input = prompt,
         store = FALSE,
         text = list(
            format = list(
               type = "json_schema",
```

```
                name = "ois_text_extraction",
                strict = TRUE,
                schema = outputSchema)))) |>
      req_perform() |>
      resp_body_json(simplifyVector = FALSE)

   messageItem <- response$output |>
      keep(\(x) identical(x$type, "message")) |>
      pluck(1)

   outputText <- messageItem$content |>
      keep(\(x) identical(x$type, "output_text")) |>
      pluck(1, "text")

   fromJSON(outputText)
}
```

The API key is placed in the HTTP authorization header by `req_auth_bearer_token()`. It is never stored in the script or included in the request body. `store = FALSE` asks the API not to retain the response for later retrieval through the API.

Let's test the function on one incident before processing many records.

```
testDescription <- ois |>
   filter(id == "23-23") |>
   pull(text)

extractOIS(testDescription)
```

```
$who
[1] "person"

$transported
[1] "yes"

$hospital
[1] "Temple University Hospital"

$evidence
[1] "Officer #1 returned fire, striking the suspect in the left ankle. He was
 ↪  placed into custody and transported to Temple University Hospital in
 ↪  stable condition."
```

Next, run a small pilot. The list column `llmResult` holds one structured response per description, and `unnest_wider()` expands those responses into ordinary columns.

```
oisPilot <- ois |>
   filter(year(date) == 2026) |>
   slice_head(n=3) |>
   # map applies the extractOIS() function to each value of text
   mutate(llmResult = map(text, extractOIS)) |>
   # map packs four columns in one... separate them
   unnest_wider(llmResult)

oisPilot |>
   select(id, who, transported, hospital, evidence)
```

```
# A tibble: 3 x 5
  id    who    transported hospital                  evidence
  <chr> <chr>  <chr>       <chr>                     <chr>
1 26-16 person yes         local hospital            The male was struck by
 ↪  gu~
2 26-14 person yes         Temple University Hospital Sergeant #1 discharged
 ↪  on~
3 26-12 no one no          not transported           Officer #1 discharged
 ↪  his~
```

Let's read the three original descriptions and compare them with the proposed codes and evidence. Refine the prompt if the coding rule is ambiguous or the results are inconsistent.

```
ois |>
   st_drop_geometry() |>
   filter(id %in% oisPilot$id) |>
   select(id, text)
```

```
     id
1 26-16
2 26-14
3 26-12
```

```
1 On 6/13/2026 at approximately 10:30 p.m., officers assigned to the 19th
  ↪ District responded to a report of a person with a weapon and shots fired
  ↪ in the area of 54th and Arlington Streets. Upon arrival, officers began
  ↪ investigating the incident and canvassing the area for witnesses and
  ↪ evidence. During the course of that investigation, an adult male
  ↪ approached an area on 2000 N. 54th Street which was being investigated by
  ↪ officers where a vehicle had been struck by gunfire. A confrontation
  ↪ developed between the male and police personnel. For a period of time,
  ↪ police personnel are trying to explain the situation to the male in
  ↪ regard to establishing a crime scene, and at that point, the male becomes
  ↪ increasingly agitated. As the verbal disagreement escalates, the male
  ↪ shoves sergeant #1 with both hands. At that point personnel, Officer #1
  ↪ and Officer #2, attempt to detain the male. As officers attempted to take
  ↪ the individual into custody, the situation escalated rapidly. According
  ↪ to preliminary information from witness statements, surveillance footage,
  ↪ and body-worn camera footage reviewed thus far, the male produced a
  ↪ firearm during the encounter. Officers repeatedly issued commands that
  ↪ the male not to draw the weapon. The male then discharged the firearm
  ↪ directly toward the police officers. During the ensuing exchange of
  ↪ gunfire, three Philadelphia police officers [Sergeant #1, and Officers #1
  ↪ and #2] were struck by gunfire. A fourth officer [Officer #3] was present
  ↪ and returned fire but was not injured. All three injured officers were
  ↪ transported to a local hospital and are expected to recover. The male was
  ↪ struck by gunfire in the chest and rear right leg during the exchange. He
  ↪ was transported to a local hospital, where he was pronounced deceased.
  ↪ Investigators recovered a 9mm handgun from the scene that was possessed
  ↪ by the male. The male possessed a valid PA LTCF permit. The Philadelphia
  ↪ District Attorney's Office (DAO) was notified and responded to the scene.
  ↪ This investigation remains active and ongoing and is being conducted by
  ↪ the Officer-Involved Shooting Investigation Unit (OISI), The Philadelphia
  ↪ Police Department Internal Affairs Bureau (IAB), and the Philadelphia
  ↪ District Attorney's Office. Discharging officers: Sergeant #1: 39/W/M, 8
  ↪ years of service, discharged multiple rounds and sustained a gunshot
  ↪ wound. Officer #1: 43/W/M, 1 year of service, discharged multiple rounds
  ↪ and sustained a gunshot wound. Officer #2: 30/W/M, 7 years of service,
  ↪ discharged multiple rounds and sustained gunshot wounds. Officer #3:
  ↪ 21/W/M, 1 year of service, discharged multiple rounds and was not
  ↪ injured. Decedent Info: 57/B/M
```

```
2
↪ On Saturday, May 23, 2026, at approximately 10:22 a.m., Sergeant #1 was
↪ in full uniform operating a marked police vehicle in the area of
↪ Kensington Ave and Hart Lane.\nThe preliminary investigation indicates
↪ that Sergeant #1, who was assigned as the street supervisor for the
↪ Kensington Police District, was flagged down by an individual who pointed
↪ out a male that he had been involved in a disturbance with and indicated
↪ that the male (defendant) was armed with a firearm. Sergeant #1
↪ approached the defendant regarding the disturbance. During the encounter,
↪ a physical altercation briefly occurred between the defendant and the
↪ complainant before the defendant fled on foot southbound on Kensington
↪ Avenue. Sergeant #1 pursued the defendant on foot into the rear area of a
↪ property on the 2800 block of Kensington Avenue.\nDuring the foot
↪ pursuit, the defendant fell while in possession of a firearm. A struggle
↪ then ensued between Sergeant #1 and the defendant over control of the
↪ weapon, with Sergeant #1 repeatedly ordering the defendant to drop the
↪ weapon. The preliminary investigation indicates that as the defendant
↪ began raising the firearm during the struggle, Sergeant #1 discharged one
↪ round, striking the defendant in the wrist and chest.\nResponding
↪ officers rendered aid and transported the defendant to Temple University
↪ Hospital, where he was placed in stable condition. No officers or other
↪ civilians were injured during the incident.\nA firearm was recovered at
↪ the scene. The preliminary investigation determined the firearm was
↪ loaded with eleven live rounds in the magazine and one live round in the
↪ chamber.\nBody-Worn Camera footage was activated during the incident. In
↪ addition, video recovered from the area depicts the defendant falling
↪ during the pursuit, dropping the firearm, retrieving it, and subsequently
↪ struggling with Sergeant #1 over control of the weapon.\nThe
↪ investigation remains active and ongoing with the Officer-Involved
↪ Shooting Investigations Unit (OISI), PPD Internal Affairs (IAB), and the
↪ Philadelphia District Attorney's Office (DAO).\nPer PPD Policy, Sergeant
↪ #1 has been placed on administrative duty pending the outcome of the
↪ investigations.\nDefendant Info: 39/B/M - Charges: Aggravated Assault,
↪ VUFA, and related offenses\nSergeant #1: 36/W/M assigned to the
↪ Kensington Police District
```

```
3
 ↪ On Friday, April 10, 2026, at 10:50 a.m., PPD officers responded to a
 ↪ radio call for person with a gun at 69th Avenue and Lawnton Avenue. Upon
 ↪ arrival, officers made contact with an off-duty police officer (Officer
 ↪ #1), who was out for a walk with his family and dog on the 6900 block of
 ↪ Lawnton Avenue. Officer #1 saw a grey pitbull exit out of a yard and
 ↪ charge towards him and his family prompting Officer #1 to discharge his
 ↪ personal weapon, Sig Sauer 9 mm, striking the pitbull in the leg.\nNo
 ↪ injuries to the Officer and his family.\nThe owner of grey pitbull took
 ↪ control of dog inside her house.\nThis incident is under active
 ↪ investigation by the Philadelphia Police Department (PPD) Officer
 ↪ Involved Shooting Investigation Unit (OISI), the PPD's Internal Affairs
 ↪ Division (IAD), and the Philadelphia District Attorney's Office
 ↪ (DAO).\nPer department policy, Officer #1 has been placed on
 ↪ administrative duty pending the outcome of the investigations.
```

Now let's run it on all of the OIS descriptions.

```
ois <- ois |>
   mutate(llmResult = map(text, extractOIS)) |>
   unnest_wider(llmResult)
```

OpenAI tells me that analyzing all of these descriptions cost $0.04. Let's check the results.

```
ois |>
   select(id, who, transported, hospital, evidence) |>
   head(5) |>
   data.frame()
```

```
     id    who transported                        hospital
1 26-16 person         yes                         unknown
2 26-14 person         yes       Temple University Hospital
3 26-12    dog          no                  not transported
4 26-11 person         yes Penn Presbyterian Medical Center
5 26-08 no one          no                  not transported
```

```
1                 The male was struck by gunfire in the chest and rear right
 ↪ leg during the exchange. He was transported to a local hospital, where he
 ↪ was pronounced deceased.
2 Sergeant #1 discharged one round, striking the defendant in the wrist and
 ↪ chest. Responding officers rendered aid and transported the defendant to
 ↪ Temple University Hospital.
3
 ↪ Officer #1 ... discharged his personal weapon ... striking the pitbull in
 ↪ the leg.
4                           Officer #1 discharged her service weapon one
 ↪ time, striking the male. Officers ... immediately transported him to Penn
 ↪ Presbyterian Medical Center.
5
 ↪ Officer #1 drew his weapon and fired twice, missing the dog.
```

And save our work so we do not need to run that again.

```
save(ois, PSAlookup, file="PPDOIS.RData")
```

Some tips:

1. Start with a small pilot
2. Check that it is answering your questions correctly
3. Inspect usage in the OpenAI developer dashboard
4. Save completed results locally to avoid rerunning calls unnecessarily

8 Summary

We started with just a web page linking to a collection of text descriptions. We used a variety of web scraping and regular expressions to extract everything we could from the web page tables. We had R "read" the text descriptions to extract the dates and used a large language model to propose structured codes from narrative text. We geocoded the incidents so that we could put them on a map. Finally, we tabulated by PSA the number of OISs and mapped those as well.

If you have worked through all of this, then I would recommend that you save your objects, using `save(ois, PSAlookup, file="PPDOIS.RData")`. That way you will not have to scrape everything off the web again or redo any geocoding or rerun everything through the LLM.

9 Exercises

1. Revisit the geocoding section discussing geocoding errors. Examine the OISs that have not been geocoded to specific locations. Fix their addresses and redo the geocoding of these OISs to improve the accuracy of the data.
2. Identify officer-involved shootings that resulted in the suspect being transported to the Hospital at the University of Pennsylvania. Create a map marking the location of HUP, the location of officer-involved shootings resulting in the suspect being transported to HUP, and the locations of all other shootings.
3. For each shooting determine which hospital treated the suspect. Use `st_distance()` to determine what percentage of those shot in an OIS went to the closest hospital.
4. Use the LLM extraction function on ten OIS descriptions. Manually code the same descriptions, compare your codes with the model outputs, and describe every disagreement.

10 An alternative way to obtain the PPD data

At the beginning of these notes, we used `chromote` to work with the live DOM, move through the pages of the OIS table, and collect the incident data. Learning that approach is important. Many websites use JavaScript to request data only after a page has loaded or after a user clicks a button. In those cases, the complete dataset is not present in the static HTML returned by the server, so `scan()` or `read_html()` cannot retrieve it by itself. A browser automation tool such as `chromote` lets R interact with the page after JavaScript has run.

It turns out that the current version of the PPD website does include all of the OIS records in the static HTML. JavaScript uses the DataTables library to display only 25 rows at a time in the browser, but the server has already sent every row. This means that, for this particular website, we can obtain the entire table with one call to `read_html()` without opening a hidden browser or clicking the Next button.

```
oisTableDirect <-
   read_html("https://www.phillypolice.com/accountability/ois/") |>
   html_element("table#data-table-ois")

oisDirect <- oisTableDirect |>
   html_table() |>
   mutate(url = oisTableDirect |>
             html_elements("tbody a") |>
             html_attr("href")) |>
   select(-Year) |>
   rename(id        = Title,
```

```
          location  = Location,
          subInjury = `Subject Injury`,
          subArrest = `Subject Arrested`,
          offInjury = `Officer Injury`,
          text      = `OIS Content`)

dim(oisDirect)
```

```
[1] 160   9
```

```
head(oisDirect)
```

```
# A tibble: 6 x 9
  id    location  subInjury subArrest offInjury `DA Action` `Use of Force`
   ↪  text
  <chr> <chr>     <chr>     <chr>     <chr>     <chr>       <chr>
   ↪  <chr>
1 26-16 2000 blo~ Killed    N/A       Yes       Pending     ""             On
 ↪  6~
2 26-14 2800 blo~ Wounded   Yes       No        Pending     ""             On
 ↪  S~
3 26-12 6900 blo~ N/A       N/A       No        Pending     ""             On
 ↪  F~
4 26-11 5400 blo~ Killed    N/A       No        Pending     ""             On
 ↪  T~
5 26-08 3400 blo~ N/A       N/A       No        Pending     ""             On
 ↪  T~
6 26-06 1400 blo~ N/A       N/A       No        Pending     ""             On
 ↪  T~
# i 1 more variable: url <chr>
```

The `OIS Content` column already contains the incident narratives, so this approach also avoids visiting every individual incident page. The URLs remain useful if we want to inspect those pages or verify particular records.

This shortcut depends entirely on how the PPD has implemented its website. Another site might show a similar paginated table while loading each page of records from a server only when the user requests it. The PPD could also redesign this page in the future. A useful first step in any web scraping project is therefore to inspect the static HTML with `read_html()`. If the desired data are present, extract them directly. If they are created or retrieved only after JavaScript runs, use `chromote` or another browser automation tool to work with the live DOM.