Data Science for Public Policy

Alena Stern and Aaron R. Williams - Georgetown University

Exploratory Data Analysis

Reading

- R for Data Science – Chapter 7, skim 14-16

Reading in Data

We've already covered reading in csv files using the `read_csv()` function from the `readr` package, but you may also need to read in data that is stored in a number of other common file formats:

Excel Spreadsheets

readxl is a tidyverse package for reading data from Microsoft Excel files. It is not a core tidyverse package so it needs to be explicitly loaded in each R session.

The tidyverse website has a good tutorial on readxl.

Many excel files can be read with the simple syntax `data <- read_excel(path = "relative/file/path/to/data")`. In cases where the Excel spreadsheet contains multiple sheets, you can use the `sheet` argument to specify the sheet to read as a string (the name of the sheet) or an integer (the position of the sheet).

STATA, SAS, and SPSS files

haven is a tidyverse package for reading data from SAS (`read_sas()`), STATA (`read_dta()`), and SPSS (`read_sav()`) files. Like the `readxl` package, it is not a core tidyverse package and also needs to be explicitly loaded in each R session.

Note that the haven package can only read and write STATA `.dta` files through version 15 of STATA. For files created in more recent versions of STATA, the `readstat13` package's `read.dta13` file can be used.

Zip Files

解压文件存储的地方

You may also want to read in data that is saved in a zip file. In order to do this, you can use the `unzip()` function to unzip the files using the following syntax: `unzip(zipfile = "path/to/zip/file", exdir = "path/to/directory/for/unzipped/files")`.

Often times, you may want to read in a zip file from a website into R. In order to do this, you will need to first download the zip file to your computer using the `download.file()` function, unzip the file using `unzip()` and then read in the data using the appropriate function for the given file type.

To download the week 40 public use file data for the Census Household Pulse Survey, run the following code:

```
base_url <- "https://www2.census.gov/programs-surveys/demo/datasets/hhp/"
week_url <- "2021/wk40/HPS_Week40_PUF_CSV.zip"

pulse_url <- paste0(base_url, week_url)

# For Mac, *.nix systems:
download.file(
  pulse_url,
  destfile = "data/pulse40.zip"
)

# For Windows systems, you need to add the mode = "wb"
# argument to prevent an invalid zip file
download.file(
  pulse_url,
  destfile = "data/pulse40.zip",
  mode = "wb"
)
```

"0" means don't want anything in-between. 两个url是一回事

paste("a","b") = a b

paste("a","b", sep = 0) = ab 也等于paste0("a", "b")

Exercise

Copy and paste the appropriate `download.file` command for your computer into an RScript and edit the `destfile` argument to an appropriate directory.

Write code using the `unzip()` function to unzip the zip file downloaded. Set `exdir` to be the same directory where you just downloaded the zip file. Run both of these commands.

Examine the unzipped files and select the appropriate function to read in the `pulse2021_puf_40` file. Write code to read that file into R, assigning the output to the `pulse` object.

Column Names

As we discussed in week 1, dataframe columns - like other objects - should be given names that are "concise and meaningful". Generally column names should be nouns and only use lowercase letters, numbers, and underscores _ (this is referred to as snake case). You should not include white space in column names (e.g "Birth Month" = bad, "birth_month" = good). It is also best practice for column names to be singular (use "birth_month" instead of "birth_months").

The janitor package is a package that contains a number of useful functions to clean data in accordance with the tidyverse principles. One such function is the `clean_names()` function, which converts column names to snake case according to the tidyverse stype guide (along with some other useful cleaning functions outlined in the link above). The `clean_names()` function works well with the `%>%` operator.

Exercise

这个语法可以存在于没有用library语法，下载了包就可以直接用

Take a look at the column names in the Pulse data file you read in for the exercise earlier.

Then edit the command in the RScript that you wrote to read in the CSV file to pipe the results of that command to the `janitor::clean_names()` function. Note that you may have to install and import the janitor package first.

Now look at the column names again after running the modified command. How have they changed?

在install.package()的括号里面，package的名称要加双引号，而library()则不用加。第一次下载这个package，用前者

As discussed in the introduction to the tidyverse, you can also directly rename columns using the `rename()` function from the `dplyr` package as follows: `rename(data, new_col = old_col)`.

Data Overview

Once you've imported your data, a common first step is to get a very high-level summary of your data.

As introduced in the introduction to the tidyverse, the `glimpse()` function provides a quick view of your data, printing the type and first several values of each column in the dataset to the console.

```
glimpse(x = storms)
```

```
## Rows: 10,010
## Columns: 13
## $ name        <chr> "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "A~
## $ year        <dbl> 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975~
```

```
## $ month       <dbl> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7~
## $ day         <int> 27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30~
## $ hour        <dbl> 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0,~
## $ lat         <dbl> 27.5, 28.5, 29.5, 30.5, 31.5, 32.4, 33.3, 34.0, 34.4, 34.0~
## $ long        <dbl> -79.0, -79.0, -79.0, -79.0, -78.8, -78.7, -78.0, -77.0, -7~
## $ status      <chr> "tropical depression", "tropical depression", "tropical de~
## $ category    <ord> -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ wind        <int> 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 45, 50, 50, 55, 60~
## $ pressure    <int> 1013, 1013, 1013, 1013, 1012, 1012, 1011, 1006, 1004, 1002~
## $ ts_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ hu_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
```

The `summary()` function enables you to quickly understand the distribution of a numeric or categorical variable. For a numeric variable, summary() will return the minimum, first quartile, median, mean, third quartile, and max values. For a categorical (or factor) variable, summary() will return the number of observations in each category. If you pass a dataframe to summary() it will summarise every column in the dataframe. You can also call summary on a single variable as shown below:

```r
summary(storms$wind)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.00   30.00   45.00   53.49   65.00  160.00
```

The `str()` function compactly displays the internal structure of any R object. If you pass a dataframe to `str` it will print the column type and first several values for each column in the dataframe, similar to the `glimpse()` function.

Getting a high-level overview of the data can help you identify what questions you need to ask of your data during the exploratory data analysis process. The rest of this lecture will outline several questions that you should always ask when exploring your data - though this list is not exhaustive and will be informed by your specific data and analysis!

Are my columns the right types?

在csv文件中，没有关于变量类型的规定，所以导入的时候可能出错

We'll read in the population-weighted centroids for the District of Columbia exported from the Missouri Census Data Center's geocorr2014 tool.

```r
dc_centroids <- read_csv("data/geocorr2014_dc.csv")
```

```
## Rows: 180 Columns: 7-- Column specification ------------------------------------------------
## Delimiter: ","
## chr (7): county, tract, cntyname, intptlon, intptlat, pop10, afact
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
glimpse(dc_centroids)
```

```
## Rows: 180
## Columns: 7
## $ county   <chr> "County code", "11001", "11001", "11001", "11001", "11001", "~
## $ tract    <chr> "Tract", "0001.00", "0002.01", "0002.02", "0003.00", "0004.00~
## $ cntyname <chr> "County name", "District of Columbia DC", "District of Columb~
## $ intptlon <chr> "Wtd centroid W longitude, degrees", "-77.058857", "-77.07521~
## $ intptlat <chr> "Wtd centroid latitude, degrees", "38.909434", "38.909223", "~
## $ pop10    <chr> "Total population (2010)", "4890", "3916", "5425", "6233", "1~
## $ afact    <chr> "tract to tract allocation factor", "1", "1", "1", "1", "1", ~
```

We see that all of the columns have been read in as character vectors because the second line of the csv file has a character description of each column. By default, `read_csv` uses the first 1,000 rows of data to infer the column types of a file. We can avoid this by skipping the first two lines of the csv file and manually setting the column names.

```
#save the column names from the dataframe
col_names <- dc_centroids %>% names()

dc_centroids <- read_csv("data/geocorr2014_dc.csv",
                         col_names = col_names,
                         skip = 2) 跳过两列，这样就可以正确地读入数据类型
```

```
## Rows: 179 Columns: 7-- Column specification ------------------------------------------------
## Delimiter: ","
## chr (2): tract, cntyname
## dbl (5): county, intptlon, intptlat, pop10, afact
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
glimpse(dc_centroids)
```

```
## Rows: 179
## Columns: 7  这是每个地区的FIPS code，应该是字符类型的，被认成了实数
## $ county   <dbl> 11001, 11001, 11001, 11001, 11001, 11001, 11001, 11001, 11001~
## $ tract    <chr> "0001.00", "0002.01", "0002.02", "0003.00", "0004.00", "0005.~
## $ cntyname <chr> "District of Columbia DC", "District of Columbia DC", "Distri~
## $ intptlon <dbl> -77.05886, -77.07522, -77.06813, -77.07583, -77.06670, -77.05~
## $ intptlat <dbl> 38.90943, 38.90922, 38.90803, 38.91848, 38.92316, 38.92551, 3~
## $ pop10    <dbl> 4890, 3916, 5425, 6233, 1455, 3376, 3189, 4539, 4620, 3364, 6~
## $ afact    <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

You can convert column types by using the `as.*` set of functions. For example, we could convert the `county` column to a character vector as follows: `mutate(dc_centroids, county = as.character(county))`. We can also set the column types when reading in the data with `read_csv()` using the `col_types` argument. For example:

```
dc_centroids <- read_csv("data/geocorr2014_dc.csv",
                         col_names = col_names,
                         skip = 2,
                         col_types = c("county" = "character"))
                                          这样解决
glimpse(dc_centroids)
```

```
## Rows: 179
## Columns: 7  把这两列加在一起
## $ county   <chr> "11001", "11001", "11001", "11001", "11001", "11001", "11001"~
## $ tract    <chr> "0001.00", "0002.01", "0002.02", "0003.00", "0004.00", "0005.~
## $ cntyname <chr> "District of Columbia DC", "District of Columbia DC", "Distri~
## $ intptlon <dbl> -77.05886, -77.07522, -77.06813, -77.07583, -77.06670, -77.05~
## $ intptlat <dbl> 38.90943, 38.90922, 38.90803, 38.91848, 38.92316, 38.92551, 3~
## $ pop10    <dbl> 4890, 3916, 5425, 6233, 1455, 3376, 3189, 4539, 4620, 3364, 6~
## $ afact    <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

As you remember from week 1, a vector in R can only contain one data type. If R does not know how to convert a value in the vector to the given type, it may introduce NA values by coercion. For example:

```
as.numeric(c("20", "10", "10+", "25", "~8"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] 20 10 NA 25 NA
```

String manipulation with `stringr` 处理字符串变量（string variables）

Before converting column types, it is critical to clean the column values to ensure that NA values aren't accidentally introduced by coercion. The `stringr` package in the tidyverse offers a number of excellent functions for cleaning character data. This package is part of the core tidyverse and is automatically loaded with `library(tidyverse)`. The stringr cheat sheet offers a great guide to the `stringr` functions.

To demonstrate some of the `stringr` functions, let's create a `state` column with the two digit state FIPS code for DC and a `geoid` column in the `dc_centroid` dataframe which contains the 11-digit census tract FIPS code, which can be useful for joining this dataframe with other dataframes that commonly use the FIPS code as a unique identifier. We will need to first remove the period from the tract column and then concatenate the county and tract columns into a geoid column. We can do that using stringr as follows:

```
dc_centroids <- dc_centroids %>%
  mutate(
    #replace first instance of pattern
    tract = str_replace(tract, "\\.", ""),
    #join multiple strings into single string
    geoid = str_c(county, tract, sep = "")
  )
```

Exercise

Copy the code above into an R script and edit it to add the creation of a variable called state that is equal to the first two characters of the county variable using the str_sub() function.

mutate(state = str_sub(county, 1, 2)) 第一个variable是string的名称，后面两个是开始计算的数位以及停止计算的数位

Note that the str_replace() function uses regular expressions to match the pattern that gets replaced. Regular expressions is a concise and flexible tool for describing patterns in strings - but it's syntax can be complex and not particularly intuitive. This vignette provides a useful introfuction to regular expressions, and when in doubt - there are plentiful Stack Overflow posts to help when you search your specific case.

在这个case里面，前面两位indicates state，后面三位indicates county

Date manipulation with lubridate

The lubridate package makes it much easier to work with dates and times in R. While also part of the tidyverse, it is not a core tidyverse package and must be explicitly loaded in each session with library(lubridate).

We'll use the flights dataset from the nycflights13 library to illustrate the power of lubridate. For example, the make_datetime() function creates a datetime object by providing the date (year, month, day) and time (hour, minute, second) values. Note that all arguments have default values if not specified. Here we do not specify the sec argument and it defaults to 00.

这个顺序是一定的，年在前面

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'

## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

```
library(nycflights13)

flight_times <- flights %>%
  mutate(departure = make_datetime(year,
                                   month,
                                   day,
                                   hour,
                                   minute),
         arrival = make_datetime(year,
                                 month,
                                 day,    这个语法是求整除的结果
                                 arr_time %/% 100,
                                 arr_time %% 100)
         ) %>%                          这个语法是求余数
  select(departure, arrival)
```

Creating datetime columns enables the use of a number of different operations, such as filtering based on date:

```
flight_times %>%      这样来过滤时间
  filter(departure > ymd("20130601")) #flights that departed after 2013-06-01
```

```
## # A tibble: 198,861 x 2
##    departure           arrival
##    <dttm>              <dttm>
##  1 2013-10-01 05:00:00 2013-10-01 06:14:00
##  2 2013-10-01 05:17:00 2013-10-01 07:35:00
##  3 2013-10-01 05:45:00 2013-10-01 08:09:00
##  4 2013-10-01 05:45:00 2013-10-01 08:01:00
##  5 2013-10-01 05:45:00 2013-10-01 09:17:00
##  6 2013-10-01 05:50:00 2013-10-01 09:12:00
##  7 2013-10-01 06:00:00 2013-10-01 06:53:00
##  8 2013-10-01 06:00:00 2013-10-01 06:48:00
##  9 2013-10-01 06:00:00 2013-10-01 06:49:00
## 10 2013-10-01 06:00:00 2013-10-01 07:27:00
## # ... with 198,851 more rows
```

Or calculating durations:

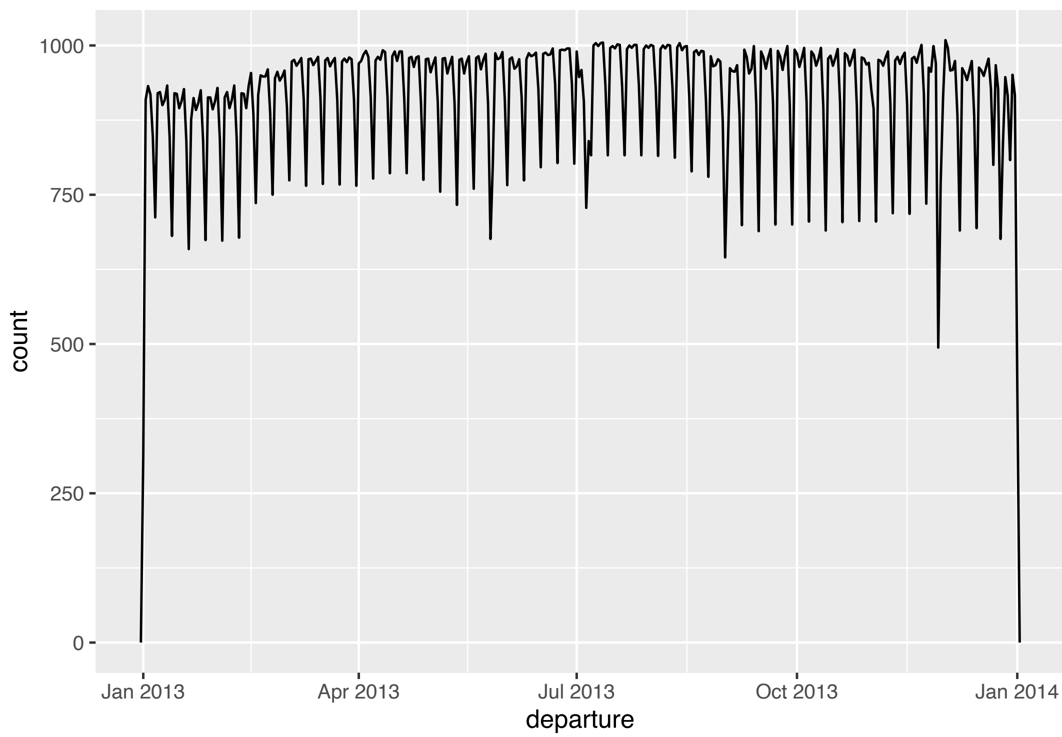
```
flight_times %>%
  mutate(flight_duration = arrival - departure)
```

```
## # A tibble: 336,776 x 3
##    departure           arrival             flight_duration
##    <dttm>              <dttm>              <drtn>
##  1 2013-01-01 05:15:00 2013-01-01 08:30:00 195 mins
##  2 2013-01-01 05:29:00 2013-01-01 08:50:00 201 mins
##  3 2013-01-01 05:40:00 2013-01-01 09:23:00 223 mins
```

```
## 4 2013-01-01 05:45:00 2013-01-01 10:04:00 259 mins
## 5 2013-01-01 06:00:00 2013-01-01 08:12:00 132 mins
## 6 2013-01-01 05:58:00 2013-01-01 07:40:00 102 mins
## 7 2013-01-01 06:00:00 2013-01-01 09:13:00 193 mins
## 8 2013-01-01 06:00:00 2013-01-01 07:09:00  69 mins
## 9 2013-01-01 06:00:00 2013-01-01 08:38:00 158 mins
## 10 2013-01-01 06:00:00 2013-01-01 07:53:00 113 mins
## # ... with 336,766 more rows
```

Datetimes can also much more easily be plotted using ggplot2. For example, it is easy to visualize the distribution of departure times across the year:

```
flight_times %>%    ggplot也可以识别时间变量
  ggplot(aes(departure)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day
```



For more information on lubridate, see the lubridate cheat sheet.

Source: R for Data Science, Ch 16   这些都可以在date-time cheat sheet里面看到

Categorical and Factor Variables

Categorical variables can be stored as characters in R. The `case_when()` function makes it very easy to create categorical variables based on other columns. For example:

```
pulse <- pulse %>%
  mutate(hisp_rrace = case_when(
        rrace == 1 ~ "White alone, not Hispanic",
        rrace == 2 ~ "Black alone, not Hispanic",
        rrace == 3 ~ "Asian alone, not Hispanic",
        rrace == 4 ~ "Two or more races + Other races, not Hispanic",
        TRUE ~ NA_character_)
        )
```

这个是告诉它，NA的数据类型是character（而不是输入“NA”）
在if_else语法中，输入的数据类型也是要一致的

Factors are a data type specifically made to work with categorical variables. The `forcats` library in the core tidyverse is made to work with factors. Factors are particularly valueable if the values have a ordering that is not alphanumeric.

```
x1 <- c("Dec", "Apr", "Jan", "Mar")

month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)

y1 <- factor(x1, levels = month_levels)

sort(x1)
```

月份这里没太跟上。。。
今天就讲到这里。

```
## [1] "Apr" "Dec" "Jan" "Mar"
```

```
sort(y1)
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Factors are also valuable if you want to show all possible values of the categorical variable, even when they have no observations.

```
table(x1)
```

```
## x1
## Apr Dec Jan Mar
##   1   1   1   1
```

```
table(y1)
```

```
## y1
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

```
## 1 0 1 1 0 0 0 0 0 0 0 1
```

Is there missing data?

Before you work with any dataset, you should understand how missing values are encoded. The best place to find this information is the data dictionary - which you should always read before working with any new dataset!

This is particularly important because while R automatically recognizes standard missing values as NA, it doesn't recognize non-standard encodings like numbers, "missing", "na", "N/A", etc.

Non-standard missing values should be converted to NA before conducting analysis. One way of doing this is with `mutate` and the `ifelse` or `case_when` functions.

Exercise

Go to the folder where you unzipped the Pulse data from earlier and open the data dictionary file. How does this dataset represent missing values for the `RECVDVACC` variable?

Using `mutate` and `if_else` or `case_when`, replace the missing values in the `recvdvacc` column with NA.

Once you have converted all missing value encodings to NA, the next question you need to ask is how you want to handle missing values in your analysis. The right approach will depend on what the missing value represents and the goals of your analysis.

- *Leave as NA*: This can be the best choice when the missing value truly represents a case when the true value is unknown. You will need to handle NAs by setting `na.rm = TRUE` in functions or filtering using `is.na()`. One drawback of this approach is that if the values aren't missing at random (e.g. smokers may be less likely to answer survey questions about smoking habits), your results may be biased. Additionally, this can cause you to lose observations and reduce power of analyses.
- *Replace with 0*: This can be the best choice if a missing value represents a count of zero for a given entity. For example, a dataset on the number of Housing Choice Voucher tenants by zip code and quarter may have a value of NA if there were no HCV tenants in the given zip code and quarter.
- *Impute missing data*: Another approach is imputing the missing data with a reasonable value. There are a number of different imputation approaches:
  - *Mean/median/mode imputation*: Fills the missing values with the column mean or median. This approach is very easy to implement, but can artifically reduce variance in your data and be sensitive to outliers in the case of mean imputation.
  - *Predictive imputation*: Fills the missing values with a predicted value based on a model that has been fit to the data or calculated probabilities based on other

columns in the data. This is a more complex approach but is likely more accurate (for example, it can take into account variable correlation).

The `replace_na()` function in `dplyr` is very useful for replacing NA values in one or more columns.

```
df <- tibble(x = c(1, 2, NA), y = c("a", NA, "b"))

# Using replace_na to replace one column
df %>% mutate(x = replace_na(x, 0))
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 <NA>
## 3     0 b
```

```
# Using replace_na to replace multiple columns with different values
df %>% replace_na(list(x = 0, y = "unknown"))
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 unknown
## 3     0 b
```

```
# Using if_else to perform mean imputation
df %>% mutate(x = if_else(is.na(x), mean(x, na.rm = TRUE), x))
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1   1   a
## 2   2   <NA>
## 3   1.5 b
```

Do I have outliers or unexpected values?

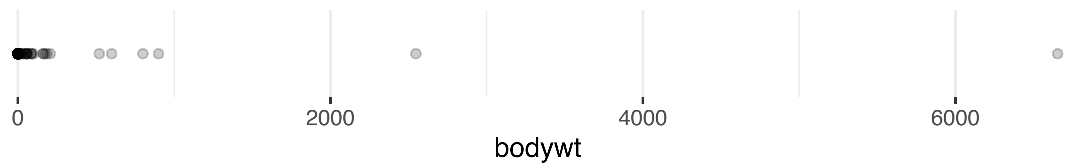
Identifying Outliers/Unexpected Values

Using R to examine the distribution of your data is one way to identify outliers or unexpected values. For example, we can examine the distribution of the `bodywt` variable in the `msleep` dataset both by examining the mathematical distribution using the `summary()` function and visually using `ggplot`.

```r
summary(msleep$bodywt)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    0.005    0.174    1.670  166.136   41.750 6654.000
```

```r
msleep %>%
  ggplot(aes(bodywt, 1)) +
  geom_point(alpha = 0.2) +
  scale_y_continuous(breaks = 0) +
  labs(y = NULL) +
  theme_bw() +
  theme(panel.border = ggplot2::element_blank())
```



Unit Tests

Writing tests in R is a great way to test that your data does not have unexpected/incorrect values. These tests can also be used to catch mistakes that can be introduced by errors in the data cleaning process. There are a number of R packages that have tools for writing tests, including:

- testthat
- assertthat
- assertr

```r
library(assertr)

df <- tibble(age = c(20, 150, 47, 88),
             height = c(60, 2, 72, 66))

df %>%
  assertr::verify(age < 120) %>%
  summarise(mean(age, na.rm = TRUE))
```

```
## verification [age < 120] failed! (1 failure)
##
##     verb redux_fn predicate column index value
## 1 verify       NA age < 120     NA     2    NA
```

```
## Error: assertr stopped execution
```

Critically, adding the test caused the code to return an error **before** calculating the mean of the age variable. This is a feature, not a bug! It can prevent you from introducing errors into your analyses. Moreover, by writing a set of tests in your analysis code, you can run the same checks everytime you perform the analysis which can help you catch errors caused by changes in the input data.

Handling Outliers/Unexpected Values

When you identify outliers or unexpected values, you will have to decide how you want to handle those values in your data. The proper way to handle those values will depend on the reason for the outlier value and the objectives of your analysis.

- If the outlier is caused by data errors, such as an implausible age or population value, you can replace the outlier values with `NA` using `mutate` and `if_else` or `case_when` as described above.
- If the outlier represents a different population than the one you are studying - e.g. the different consumption behaviors of indvidual consumers versus wholesale business orders - you can remove it from the data.
- You can transform the data, such as taking the natural log to reduce the variation caused by outliers.
- You can select a different analysis method that is less sensitive to outliers, such as using the median rather than the mean to measure central tendency.

Exercise

1. Read the column descriptions in the csv file for the DC centroids `data/geocorr2014_dc.csv`.
2. Use one of the methods above to identify whether the `pop10` column contains any outliers.
3. Calculate the mean of the `pop10` column in the `dc_centroids` dataframe, but first write one test using `assertr::verify()` to test for invalid values based on the column definition.

Data Quality Assurance

Data quality assurance is the foundation of high quality analysis. Four key questions that you should always ask when considering using a dataset for analysis are:

1. Does the data suit the research question? Examine the data quality (missingness, accuracy) of key columns, the number of observations in subgroups of interest, etc.
2. Does the data accurately represent the population of interest? Think about the data generation process (e.g. using 311 calls or Uber ride data to measure potholes) - are any populations likely to be over or underrepresented? Use tools like Urban's Spatial Equity Data Tool to test data for unrepresentativeness.
3. Is the data gathering reproducible? Wherever possible, eliminate manual steps to gather or process the data. This includes using reproducible processes for data ingest

such as APIs or reading data directly from the website rather than manually downloading files. All edits to the data should be made programattically (e.g. skipping rows when reading data rather than deleting extraneous rows manually). Document the source of the data including the URL, the date of the access, and specific metadata about the vintage.

4. How can you verify that you are accessing and using the data correctly? This may include writing tests to ensure that raw and calculated data values are plausible, comparing summary statistics against those provided in the data documentation (if applicable), published tables/statistics from the data provider, or published tables/statistics from a trusted third party.