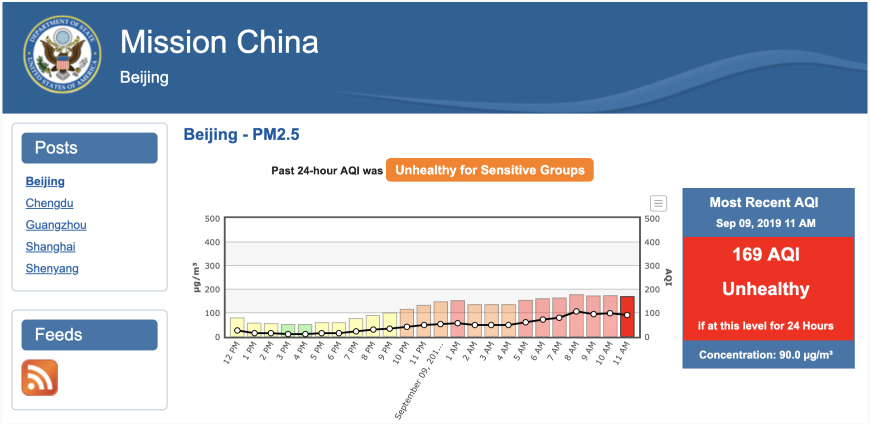
Exploring data #1

Exploring data

- How to explore depends on data type / class
- Data exploration includes simple statistics (max, mean, min, standard deviation)
- Data exploration include plots

Example data—Beijing air quality


Mission China

Beijing

Posts

Beijing

Chengdu

Guangzhou

Shanghai

Shenyang

Feeds

Beijing - PM2.5

Past 24-hour AQI was **Unhealthy for Sensitive Groups**

Most Recent AQI

Sep 09, 2019 11 AM

169 AQI

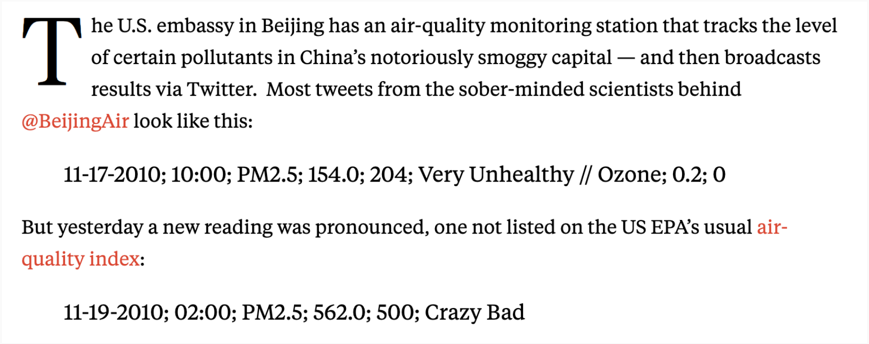
Unhealthy

if at this level for 24 Hours

Concentration: 90.0 µg/m³

Source: http://www.stateair.net/web/post/1/1.HTML

Example data—Beijing air quality



The U.S. embassy in Beijing has an air-quality monitoring station that tracks the level of certain pollutants in China's notoriously smoggy capital — and then broadcasts results via Twitter. Most tweets from the sober-minded scientists behind @BeijingAir look like this:

11-17-2010; 10:00; PM2.5; 154.0; 204; Very Unhealthy // Ozone; 0.2; 0

But yesterday a new reading was pronounced, one not listed on the US EPA's usual air-quality index:

11-19-2010; 02:00; PM2.5; 562.0; 500; Crazy Bad

Source: https://foreignpolicy.com/2010/11/19/beijing-air-crazy-bad/

Example data—Beijing air quality

Find out more:

https://www.wired.com/2015/03/opinion-us-embassy-beijing-tweeted-clear-air/
https://www.theguardian.com/environment/blog/2010/nov/19/crazy-bad-beijing-air-pollution
https://www.sciencemag.org/news/2018/04/rooftop-sensors-us-embassies-are-warning-world-about-crazy-bad-air-pollution

Example data—Beijing air quality

Download the data here.

Then you can read this data into your R session:

```r
library("readr")
```

```r
beijing_pm_raw <- read_csv("data/Beijing_2017_HourlyPM25.csv",
                           skip = 3)
```

Example data—Beijing air quality

```r
head(beijing_pm_raw, n = 3)
```

```
## # A tibble: 3 x 11
##   Site  Parameter `Date (LST)`  Year Month   Day
##   <chr> <chr>     <chr>        <dbl> <dbl> <dbl>
## 1 Beij~ PM2.5     1/1/2017 0:~  2017     1     1
## 2 Beij~ PM2.5     1/1/2017 1:~  2017     1     1
## 3 Beij~ PM2.5     1/1/2017 2:~  2017     1     1
## # ... with 5 more variables: Hour <dbl>,
## #   Value <dbl>, Unit <chr>, Duration <chr>, `QC
## #   Name` <chr>
```

Example data—Beijing air quality

Let's clean this up a bit:

```r
library("dplyr")
beijing_pm <- beijing_pm_raw %>%
  rename(sample_time = `Date (LST)`,
         value = Value,
         qc = `QC Name`) %>%
  select(sample_time, value, qc)
head(beijing_pm, n = 3)
```

```
## # A tibble: 3 x 3
##   sample_time   value qc
##   <chr>         <dbl> <chr>
## 1 1/1/2017 0:00   505 Valid
## 2 1/1/2017 1:00   485 Valid
## 3 1/1/2017 2:00   466 Valid
```

Example data—Beijing air quality

This code will add the AQI categories:

```r
beijing_pm <- beijing_pm %>%
  mutate(aqi = cut(value,
                   breaks = c(0, 50, 100, 150, 200,
                              300, 500, Inf),
                   labels = c("Good", "Moderate",
                              "Unhealthy for Sensitive Groups",
                              "Unhealthy", "Very Unhealthy",
                              "Hazardous", "Beyond Index")))
head(beijing_pm, n = 2)
```

```
## # A tibble: 2 x 4
##   sample_time   value qc    aqi
##   <chr>         <dbl> <chr> <fct>
## 1 1/1/2017 0:00   505 Valid Beyond Index
## 2 1/1/2017 1:00   485 Valid Hazardous
```

Data types and vector classes

Data types and vector classes

Here are a few common vector classes in R:

Class	Example
`character`	“Chemistry”, “Physics”, “Mathematics”
`numeric`	10, 20, 30, 40
`factor`	Male [underlying number: 1], Female [2]
`Date`	“2010-01-01” [underlying number: 14,610]
`logical`	TRUE, FALSE [underlying numbers: 1, 0]

Numeric vectors

To explore numeric vectors, there are a few base R functions that are very helpful. For example:

Function	Description
`min()`	Minimum of values in the vector
`max()`	Maximum of values in the vector
`mean()`	Mean of values in the vector
`median()`	Median of values in the vector

Simple statistic examples

All of these take, as the main argument, the vector(s) for which you want the statistic.

```r
mean(x = beijing_pm$value)
```

```
## [1] 63.18646
```

```r
min(x = beijing_pm$value)
```

```
## [1] -999
```

If there are missing values in the vector, you'll need to add an option to say what to do when them (e.g., `na.rm` or `use="complete.obs"`—see help files).

Simple statistic examples

These functions require a **numeric vector** as input.

Remember that you can pull a column from a dataframe as a vector using either `$` or the `pluck` function from `purrr`. Therefore, you can use either of these calls to get the mean weight of the children in the dataset:

```
mean(beijing_pm$value)
```

```
## [1] 63.18646
```

```
library("purrr")
beijing_pm %>%
  pluck("value") %>%
  mean()
```

```
## [1] 63.18646
```

The `summarize` function

Within a "tidy" workflow, you can use the `summarize` function from the `dplyr` package to create summary statistics for a dataframe. This function inputs a dataframe and outputs a dataframe with the specified summary measures.

The `summarize` function

The basic format for using `summarize` is:

```
## Generic code
summarize(dataframe,
          summary_column_1 = function(existing_columns),
          summary_column_2 = function(existing_columns))
```

The summarize function

As an example, to summarize the beijing_pm dataset to get the minimum, mean, and maximum $PM_{2.5}$ concentrations, you could run:

```
summarize(beijing_pm,
          min_pm = min(value),
          mean_pm = mean(value),
          max_pm = max(value))
```

```
## # A tibble: 1 x 3
##   min_pm mean_pm max_pm
##    <dbl>   <dbl>  <dbl>
## 1   -999    63.2    684
```

Notice that the output is one row (since the summary was on ungrouped data), with three columns (since we defined three summaries in the summarize function).

The summarize function

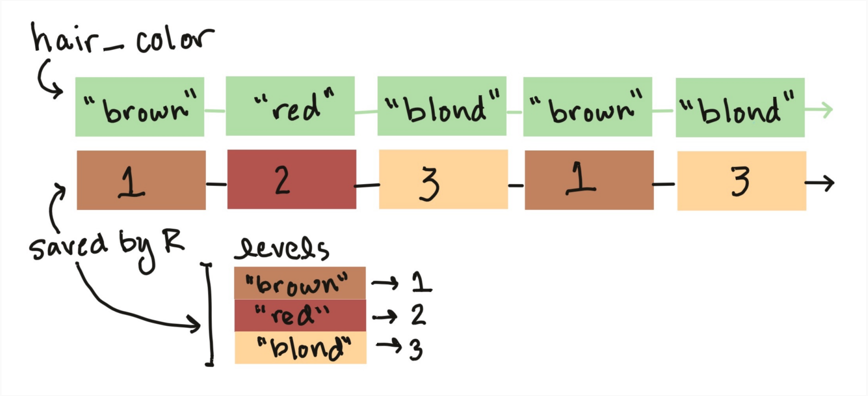
Because the first input to the summarize function is a dataframe, you can "pipe into" a summarize call. For example, we could have written the code on the previous slide as:

```
beijing_pm %>%
  summarize(min_pm = min(value),
            mean_pm = mean(value),
            max_pm = max(value))
```

As another note, because the output from summarize is also a dataframe, we could also "pipe into" another tidyverse function after running summarize.

Factors in R

Factor vectors are used in R for **categorical variables**, where more than one observation can have the same category.



Factor variables have one or more **levels**. While you will always see a factor printed with its factor level labels, R "remembers" the variable with each level assigned a number.

Factors in R

In tibbles, factors will be noted with “fctr” under the column name. For example, look at the aqi column in the beijing_pm data:

```r
head(beijing_pm, n = 3)
```

```
## # A tibble: 3 x 4
##   sample_time   value qc    aqi
##   <chr>         <dbl> <chr> <fct>
## 1 1/1/2017 0:00   505 Valid Beyond Index
## 2 1/1/2017 1:00   485 Valid Hazardous
## 3 1/1/2017 2:00   466 Valid Hazardous
```

Factors in R

You can use the `levels` function to see the levels of a factor vector, as well as the order those levels are recorded in R.

```r
levels(beijing_pm$aqi)
```

```
## [1] "Good"
## [2] "Moderate"
## [3] "Unhealthy for Sensitive Groups"
## [4] "Unhealthy"
## [5] "Very Unhealthy"
## [6] "Hazardous"
## [7] "Beyond Index"
```

Factors in R

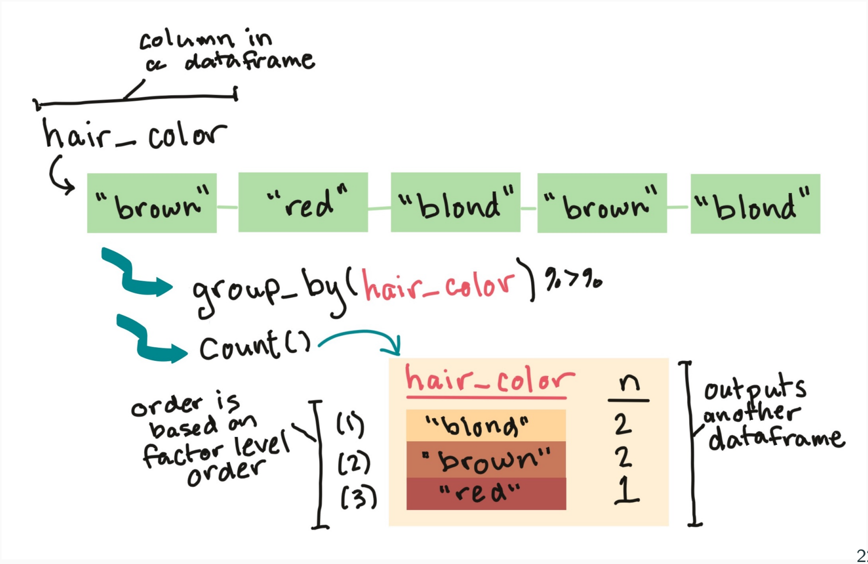
To explore a factor vector, you'll often want to **count** the number of observations in each category. You can do that with two functions in the `dplyr` package, `group_by` and `count`.

Start with a dataframe that includes the factor variable as a column. First `group_by` the factor, then pipe the output of that into the `count` function.

This will create a new summary dataframe, with a row for each level of the factor. A column called `n` will give the number of observations in the original data that had that level of the factor.

Factors in R

You can **count** how many observations have each level of a factor.



Factors in R

```r
beijing_pm %>%
  group_by(aqi) %>%
  count()
```

```
## # A tibble: 8 x 2
## # Groups:   aqi [8]
##   aqi                                n
##   <fct>                          <int>
## 1 Good                            2438
## 2 Moderate                        1021
## 3 Unhealthy for Sensitive Groups   374
## 4 Unhealthy                        167
## 5 Very Unhealthy                   179
## 6 Hazardous                        107
## 7 Beyond Index                      27
## 8 <NA>                              31
```

Factors in R

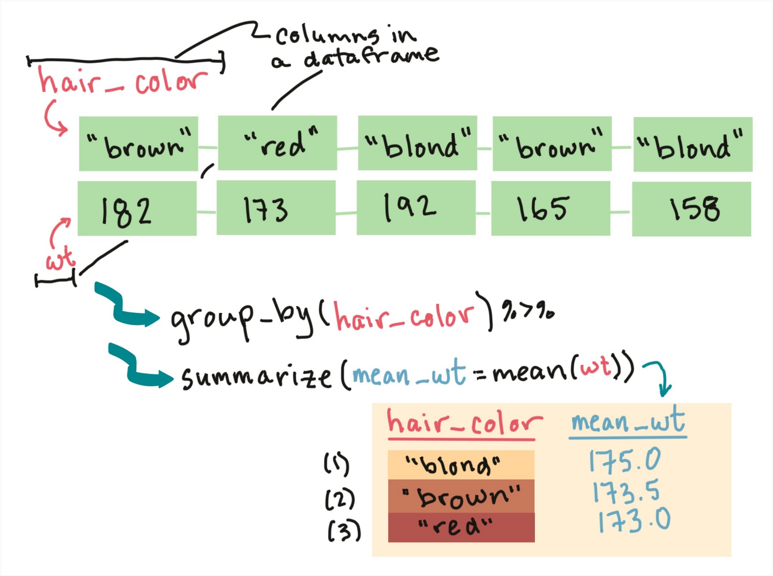
You can jointly explore multiple columns in a dataframe.

For example, if one column is a factor and one is numeric, it can be useful to explore values of the numeric column within each level of the factor column.

For the Beijing data, you may want to find out the mean comcentration of $PM_{2.5}$ within each AQI level.

Factors in R

You can **summarize** a numeric column within levels of a factor column:



Factors in R

To do this, pipe the dataframe into `group_by` (where you can group by the factor column) and then into `summarize`, where you can calculate summaries.

```r
beijing_pm %>%
  group_by(aqi) %>%
  summarize(mean_pm = mean(value))
```

```
## # A tibble: 8 x 2
##   aqi                            mean_pm
##   <fct>                            <dbl>
## 1 Good                              23.5
## 2 Moderate                          70.7
## 3 Unhealthy for Sensitive Groups   122.
## 4 Unhealthy                        172.
## 5 Very Unhealthy                   243.
## 6 Hazardous                        378.
## 7 Beyond Index                     554.
```

Factors in R

You can create several summaries at once:

```
beijing_pm %>%
  group_by(aqi) %>%
  summarize(min_pm = min(value),
            max_pm = max(value))
```

```
## # A tibble: 8 x 3
##   aqi                            min_pm max_pm
##   <fct>                           <dbl>  <dbl>
## 1 Good                                1     50
## 2 Moderate                           51    100
## 3 Unhealthy for Sensitive Groups    101    150
## 4 Unhealthy                         151    200
## 5 Very Unhealthy                    202    300
## 6 Hazardous                         301    500
## 7 Beyond Index                      505    684
## 8 <NA>                             -999     -2
```

Factors in R

As a note, there's a function called `n()` that you can use inside summarize to replace `count`. For example, these two expressions give the same output:

```r
beijing_pm %>%
  group_by(aqi) %>%
  count()

beijing_pm %>%
  group_by(aqi) %>%
  summarize(n = n())
```

Factors in R

If a column is in a character class, but you'd like it to be a factor, you can use `as.factor`:

```
beijing_pm %>%
  mutate(qc = as.factor(qc))
```

```
## # A tibble: 4,344 x 4
##    sample_time   value qc    aqi
##    <chr>         <dbl> <fct> <fct>
##  1 1/1/2017 0:00   505 Valid Beyond Index
##  2 1/1/2017 1:00   485 Valid Hazardous
##  3 1/1/2017 2:00   466 Valid Hazardous
##  4 1/1/2017 3:00   435 Valid Hazardous
##  5 1/1/2017 4:00   405 Valid Hazardous
##  6 1/1/2017 5:00   402 Valid Hazardous
##  7 1/1/2017 6:00   407 Valid Hazardous
##  8 1/1/2017 7:00   435 Valid Hazardous
##  9 1/1/2017 8:00   472 Valid Hazardous
```

In-course exercise

We'll take a break now to start the in-course exercise for this week (Sections 1 and 2 for Week 3).

Data from R packages

Data from R packages

So far we've covered two ways to get data into R:

1. From flat files (either on your computer or online)
2. From files like SAS and Excel

Many R packages come with their own data, which is very easy to load and use.

Data from R packages

For example, the `faraway` package has a dataset called `worldcup` that you'll use today. To load it, use the `data()` function once you've loaded the package with the name of the dataset as its argument:

```r
library("faraway")
data("worldcup")
```

Data from R packages

Unlike most data objects you'll work with, the data that comes with an R package will often have its own help file. You can access this using the `?` operator:

```
?worldcup
```

Data from R packages

To find out all the datasets that are available in the packages you currently have loaded, run `data()` without an option inside the parentheses:

```r
data()
```

To find out all of the datasets available within a certain package, run `data` with the argument `package`:

```r
data(package = "faraway")
```

As a note, you can similarly use `library()`, without the name of a package, to list all of the packages you have installed that you could call with `library()`:

```r
library()
```

Dates in R

`Date` class

A common task when changing or adding columns is to change the class of some of the columns. This is especially common for dates, which will often be read in as a character vector when reading data into R.

`lubridate` package

The `lubridate` package is helpful for working with vectors of dates or date-times.

You will see dates represented in many different ways. For example, October might be included in data as "October", "Oct", or "10". Further, the way the elements are separated can vary.

Computers are very literal, so this ambiguity can be confusing for them.

`lubridate` package

The `lubridate` package has a number of functions for converting character strings into dates (or date-times). To decide which one to use, you just need to know the order of the elements of the date in the character string.

For example, here are some commonly-used `lubridate` functions:

`lubridate` function	Order of date elements
`ymd`	year-month-day
`dmy`	day-month-year
`mdy_hm`	month-day-year-hour-minute
`ymd_hms`	year-month-day-hour-minute-second

(Remember, you can use `vignette("lubridate")` and `?lubridate` to get help with the `lubridate` package.)

lubridate package

In many cases you can use functions from the lubridate package to parse dates pretty easily.

For example, if you have a character string with the date in the order of *year-month-day*, you can use the ymd function from lubridate to convert the character string to the Date class. For example:

```r
library("lubridate")
my_date <- ymd("2008-10-13")
class(my_date)
```

```
## [1] "Date"
```

lubridate package

The functions in `lubridate` are pretty good at working with different ways of expressing date and time elements intelligently:

```r
mdy("10-31-2017")
```

```
## [1] "2017-10-31"
```

```r
dmy("31 October 2017")
```

```
## [1] "2017-10-31"
```

lubridate package

There are lubridate functions that can parse date-times, too:

```
ymd_hms("2017/10/31--17:33:10")
```

```
## [1] "2017-10-31 17:33:10 UTC"
```

```
mdy_hm("Oct. 31, 2017 5:33PM", tz = "MST")
```

```
## [1] "2017-10-31 17:33:00 MST"
```

Converting to Date class

We can use the mdy_hms function from lubridate to convert the sample_time column in the beijing_pm dataset to a date-time class ("POSIXct"):

```
beijing_pm <- beijing_pm %>%
  mutate(sample_time = mdy_hm(sample_time))

head(beijing_pm, 3)
```

```
## # A tibble: 3 x 4
##   sample_time         value qc    aqi
##   <dttm>              <dbl> <chr> <fct>
## 1 2017-01-01 00:00:00   505 Valid Beyond Index
## 2 2017-01-01 01:00:00   485 Valid Hazardous
## 3 2017-01-01 02:00:00   466 Valid Hazardous
```

Converting to `Date` class

Once you have an object in a date or date-time class, you can do things like plot by date, calculate the range of dates, and calculate the total number of days the dataset covers:

```r
range(beijing_pm$sample_time)
```

```
## [1] "2017-01-01 00:00:00 UTC"
## [2] "2017-06-30 23:00:00 UTC"
```

```r
diff(range(beijing_pm$sample_time))
```

```
## Time difference of 180.9583 days
```

lubridate package

The lubridate package also includes functions to pull out certain elements of a date. For example, we could use wday to create a new column with the weekday of each show:

```
beijing_pm <- mutate(beijing_pm,
                     sample_weekday = wday(sample_time,
                                           label = TRUE))
```

lubridate package

```
beijing_pm %>%
  select(sample_time, sample_weekday) %>%
  sample_n(size = 3)
```

```
## # A tibble: 3 x 2
##   sample_time         sample_weekday
##   <dttm>              <ord>
## 1 2017-06-01 03:00:00 Thu
## 2 2017-01-30 16:00:00 Mon
## 3 2017-05-25 22:00:00 Thu
```

`lubridate` package

The `wday` function created an **ordered factor** (“ord” below the column name in the tibble print-out). You can use this like other factors.

lubridate package

```r
beijing_pm %>%
  group_by(sample_weekday) %>%
  summarize(mean_pm = mean(value))
```

```
## # A tibble: 7 x 2
##   sample_weekday mean_pm
##   <ord>            <dbl>
## 1 Sun               67.6
## 2 Mon               52.2
## 3 Tue               64.3
## 4 Wed               76.1
## 5 Thu               75.4
## 6 Fri               61.0
## 7 Sat               45.1
```

`lubridate` package

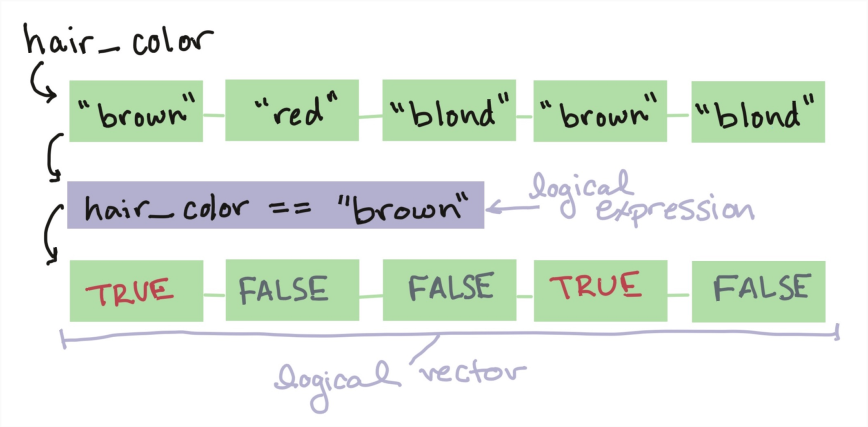
Other functions in `lubridate` for pulling elements from a date include:

- `mday`: Day of the month
- `yday`: Day of the year
- `month`: Month
- `quarter`: Fiscal quarter
- `year`: Year

Logical operators, vectors, and expressions

Logical operators, vectors, and expressions

Logical expressions are operators that conduct a logical test based on one or more vectors, while logical expressions are the full R expressions that use these operators to conduct the test. The output is a **logical vector**.



Logical expressions

Last week, you learned some about logical expressions and how to use them with the `filter` function.

You can use *logical vectors*, created with these expressions, for a lot data exploration tasks. We'll review them and add some more details this week.

Logical vectors

A logical expression outputs a *logical vector*. This logical vector will be the same length as the original vector tested by the logical statement:

```
length(beijing_pm$value)
```

```
## [1] 4344
```

```
length(beijing_pm$value > 500)
```

```
## [1] 4344
```

Logical vectors

Each element of the logical vector can only have one of three values (`TRUE`, `FALSE`, `NA`). The logical vector will have the value `TRUE` at any position where the original vector met the logical condition you tested, and `FALSE` anywhere else:

```r
head(beijing_pm$value)
```

```
## [1] 505 485 466 435 405 402
```

```r
head(beijing_pm$value > 500)
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

Logical vectors

Because the logical vector is the same length as the vector it's testing, you can add logical vectors to dataframes with `mutate`:

```
beijing_pm <- beijing_pm %>%
  mutate(beyond_index = value > 500)
```

Logical vectors

```r
beijing_pm %>%
  select(sample_time, value, beyond_index)
```

```
## # A tibble: 4,344 x 3
##    sample_time         value beyond_index
##    <dttm>              <dbl> <lgl>
##  1 2017-01-01 00:00:00   505 TRUE
##  2 2017-01-01 01:00:00   485 FALSE
##  3 2017-01-01 02:00:00   466 FALSE
##  4 2017-01-01 03:00:00   435 FALSE
##  5 2017-01-01 04:00:00   405 FALSE
##  6 2017-01-01 05:00:00   402 FALSE
##  7 2017-01-01 06:00:00   407 FALSE
##  8 2017-01-01 07:00:00   435 FALSE
##  9 2017-01-01 08:00:00   472 FALSE
## 10 2017-01-01 09:00:00   465 FALSE
## # ... with 4,334 more rows
```

Logical vectors

As another example, you could add a column that is a logical vector of whether a day was in the “heating season”, which usually ends on March 15 each year:

```r
beijing_pm <- beijing_pm %>%
  mutate(heating = sample_time < ymd("2017-03-15"))
```

Common logical and relational operators in R

The **bang operator** (`!`) negates (flips) a logical expression:

```
c(1, 2, 3) == c(1, 2, 5)
```

```
## [1]  TRUE  TRUE FALSE
```

```
!(c(1, 2, 3) == c(1, 2, 5))
```

```
## [1] FALSE FALSE  TRUE
```

```
is.na(c(1, 2,  NA))
```

```
## [1] FALSE FALSE  TRUE
```

```
!is.na(c(1, 2, NA))
```

```
## [1]  TRUE  TRUE FALSE
```

Common logical and relational operators in R

The `%in%` operator will check each element of a vector to see if it's a value that is included in a second vector.

In this case, the two vectors don't have to have the same length:

```
c(1, 2, 3) %in% c(1, 5)
```

```
## [1]  TRUE FALSE FALSE
```

This logical expressions is asking *Is the first element of the first vector, 1, in the set given by the second vector, 1 and 5? Is the second element of the first vector, 2, in the set given by the second vector? Etc.*

Logical vectors

You can do a few cool things now with this vector. For example, you can use it with the `filter` function to pull out just the rows where `heating` is `TRUE`:

```
beijing_pm %>%
  filter(heating) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 7
##   sample_time         value qc    aqi
##   <dttm>              <dbl> <chr> <fct>
## 1 2017-01-01 00:00:00   505 Valid Beyo~
## 2 2017-01-01 01:00:00   485 Valid Haza~
## 3 2017-01-01 02:00:00   466 Valid Haza~
## # ... with 3 more variables:
## #   sample_weekday <ord>, beyond_index <lgl>,
## #   heating <lgl>
```

Logical vectors

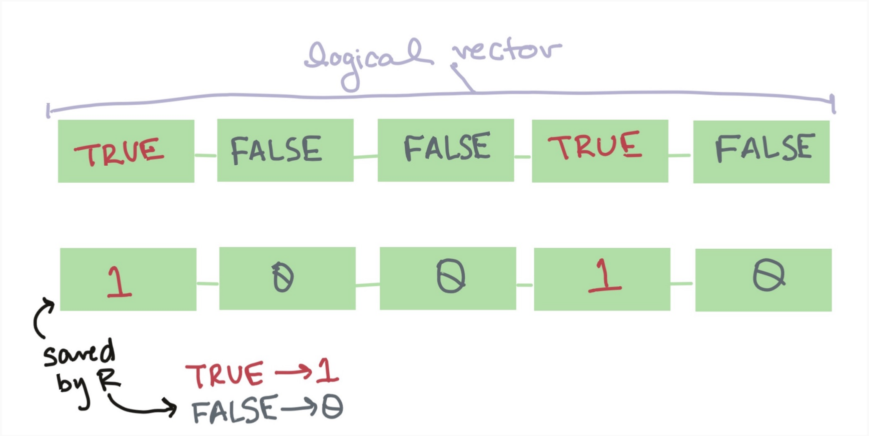
Or, with `!`, just the rows where `heating` is `FALSE`:

```r
beijing_pm %>%
  filter(!heating) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 7
##   sample_time         value qc    aqi
##   <dttm>              <dbl> <chr> <fct>
## 1 2017-03-15 00:00:00    54 Valid Mode~
## 2 2017-03-15 01:00:00    43 Valid Good
## 3 2017-03-15 02:00:00    39 Valid Good
## # ... with 3 more variables:
## #   sample_weekday <ord>, beyond_index <lgl>,
## #   heating <lgl>
```

Logical vectors

All of the values in a logical vector are saved, at a deeper level, with a number. Values of `TRUE` are saved as 1 and values of `FALSE` are saved as 0.



Logical vectors

```r
head(beijing_pm$beyond_index)
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

```r
head(as.numeric(beijing_pm$beyond_index))
```

```
## [1] 1 0 0 0 0 0
```

Logical vectors

Therefore, you can use `sum()` to get the sum of all values in a vector. Because logical vector values are linked with numerical values of 0 or 1, you can use `sum()` to find out how many males and females are in the dataset:

```
sum(beijing_pm$beyond_index)
```

```
## [1] 27
```

```
sum(!beijing_pm$beyond_index)
```

```
## [1] 4317
```

In-course exercise

We'll take a break now to start the in-course exercise for this week (Section 3 for Week 3).

Tidyverse and cheatsheets

The "tidyverse"

So far, we have used a number of packages that are part of the *tidyverse*. The tidyverse is a collection of recent and developing packages for R, many written by Hadley Wickham.

The “tidyverse”

"A giant among data nerds"

{ https://priceonomics.com/hadley-wickham-the-man-who-revolutionized-r/}

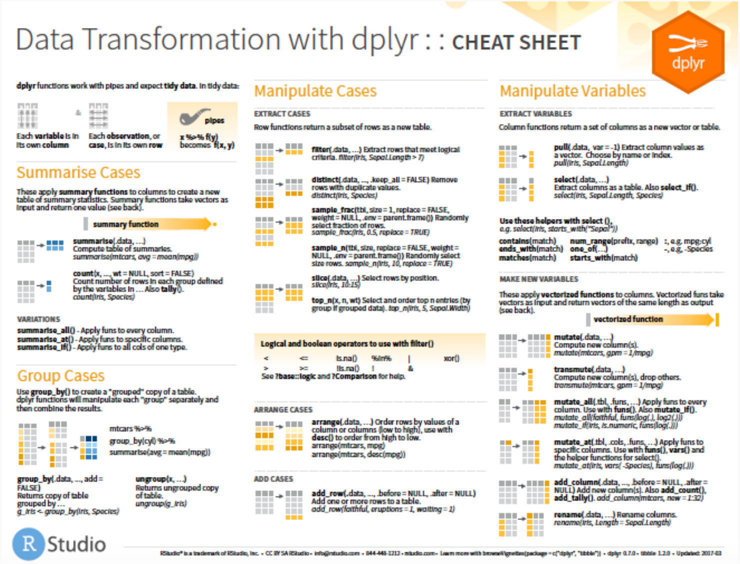
Cheatsheets

RStudio has several very helpful **cheatsheets**. These are one-page sheets (front and back) that cover many of the main functions for a certain topic or task in R. These cheatsheets cover a lot of the main "tidyverse" functions.

You can access these directly from RStudio. Go to "Help" -> "Cheatsheets" and select the cheatsheet on the topic of interest.

You can find even more of these cheatsheets at https://www.rstudio.com/resources/cheatsheets/.

Cheatsheets



Data Transformation with dplyr : : CHEAT SHEET

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes
x %>% f(y) becomes **f(x, y)**

Summarise Cases

These apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function

summarise(.data, ...)
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in ... Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
summarise_if() - Apply funs to all cols of one type.

Group Cases

Use **group_by**() to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))

group_by(.data, ..., add = FALSE)
Returns copy of table grouped by ...
g_iris <- group_by(iris, Species)

ungroup(x, ...)
Returns ungrouped copy of table.
ungroup(g_iris)

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, ...) Extract rows that meet logical criteria. *filter(iris, Sepal.Length > 7)*

distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
distinct(iris, Species)

sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*

slice(.data, ...) Select rows by position.
slice(iris, 10:15)

top_n(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, ...) Order rows by values of a column or columns (low to high), use with **desc**() to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES

add_row(.data, ..., .before = NULL, .after = NULL)
Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1) Extract column values as a vector. Choose by name or index.
pull(iris, Sepal.Length)

select(.data, ...)
Extract columns as a table. Also **select_if**().
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match)	**num_range**(prefix, range)	:, e.g. mpg:cyl
ends_with(match)	**one_of**(...)	-, e.g. -Species
matches(match)	**starts_with**(match)	

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).

vectorized function

mutate(.data, ...)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, ...)
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

mutate_all(.tbl, .funs, ...) Apply funs to every column. Use with **funs**(). Also **mutate_if**().
mutate_all(faithful, funs(log(.), log2(.)))
mutate_if(iris, is.numeric, funs(log(.)))

mutate_at(.tbl, .cols, .funs, ...) Apply funs to specific columns. Use with **funs**(), **vars**() and the helper functions for select().
mutate_at(iris, vars(-Species), funs(log(.)))

add_column(.data, ..., .before = NULL, .after = NULL) Add new column(s). Also **add_count**(), **add_tally**(). *add_column(mtcars, new = 1:32)*

rename(.data, ...) Rename columns.
rename(iris, Length = Sepal.Length)

RStudio

More reading / practice

If you would like more reading and practice on what we've covered so far on transforming data, see chapter 5 of the "R for Data Science" book suggested at the start of the course.

As a reminder, that is available at:

http://r4ds.had.co.nz

Basic plotting

Example data—Beijing air quality

Let's read the Beijing data in and clean up the "-999" values:

```
library("readr")
```

```
beijing_pm_raw <- read_csv("data/Beijing_2017_HourlyPM25.csv",
                           skip = 3, na = "-999")
```

Example data—Beijing air quality

Clean up as before:

```
beijing_pm <- beijing_pm_raw %>%
  rename(sample_time = `Date (LST)`,
         value = Value,
         qc = `QC Name`) %>%
  select(sample_time, value, qc) %>%
  mutate(aqi = cut(value,
                   breaks = c(0, 50, 100, 150, 200,
                              300, 500, Inf),
                   labels = c("Good", "Moderate",
                              "Unhealthy for Sensitive Groups",
                              "Unhealthy", "Very Unhealthy",
                              "Hazardous", "Beyond Index"))) %>%
  mutate(sample_time = mdy_hm(sample_time)) %>%
  mutate(heating = sample_time < mdy("03/15/2017"))
```

Plots

Plots to explore data

Plots can be invaluable in exploring your data.

Today, we will focus on **useful**, rather than **attractive** graphs, since we are focusing on exploring rather than presenting data.

Next lecture, we will talk more about customization, to help you make more attractive plots that would go into final reports.

ggplot conventions

Here, we'll be using functions from the ggplot2 library, so you'll need to install that package:

```r
library("ggplot2")
```

The basic steps behind creating a plot with ggplot2 are:

1. Create an object of the ggplot class, typically specifying the **data** to be shown in the plot;
2. Add on (using +) one or more **geoms**, specifying the **aesthetics** for each; and
3. Add on (using +) other elements to create and customize the plot (e.g., add layers to customize scales or themes or to add facets).

Note: To avoid errors, end lines with +, don't start lines with it.

Plot data

The `ggplot` function requires you to input a dataframe with the data you will plot. All the columns in that dataframe can be mapped to specific aesthetics within the plot.

```
beijing_pm %>%
  slice(1:3)
```

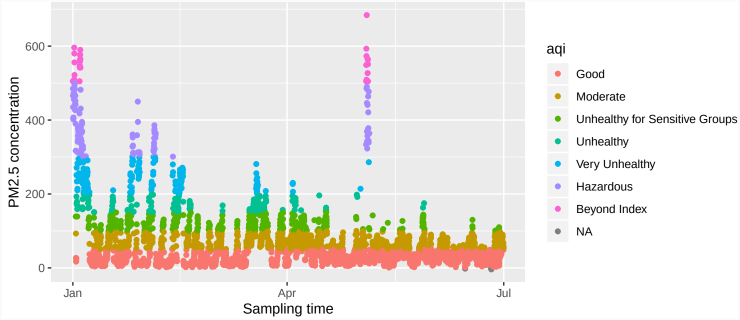
```
## # A tibble: 3 x 5
##   sample_time         value qc    aqi       heating
##   <dttm>              <dbl> <chr> <fct>     <lgl>
## 1 2017-01-01 00:00:00   505 Valid Beyond ~ TRUE
## 2 2017-01-01 01:00:00   485 Valid Hazardo~ TRUE
## 3 2017-01-01 02:00:00   466 Valid Hazardo~ TRUE
```

For example, if we input the `beijing_pm` dataframe, we would be able to create a plot that shows each sample's sampling time on the x-axis, $PM_{2.5}$ concentration on the y-axis, and AQI by the color of the point.

Plot aesthetics

Aesthetics are plotting elements that can show certain elements of the data.

For example, you may want to create a scatterplot where color shows AQI, x-position shows sampling time, and y-position shows $PM_{2.5}$ concentration.



Plot aesthetics

In the previous graph, the mapped aesthetics are color, x, and y. In the ggplot code, all of these aesthetic mappings will be specified within an aes call, which will be nested in another call in the ggplot pipeline.

Aesthetic	ggplot abbreviation	beijing_pm column
x-axis position	x =	sample_time
y-axis position	y =	value
color	color =	aqi

This is how these mappings will be specified in an aes call:

```
# Note: This code should not be run by itself.
# It will eventually be nested in a ggplot call.
aes(x = sample_time, y = value, color = aqi)
```

Plot aesthetics

Here are some common plot aesthetics you might want to specify:

Code	Description
`x`	Position on x-axis
`y`	Position on y-axis
`shape`	Shape
`color`	Color of border of elements
`fill`	Color of inside of elements
`size`	Size
`alpha`	Transparency (1: opaque; 0: transparent)
`linetype`	Type of line (e.g., solid, dashed)

Geoms

You will add **geoms** that create the actual geometric objects on the plot. For example, a scatterplot has “points” geoms, since each observation is displayed as a point.

There are `geom_*` functions that can be used to add a variety of geoms. The function to add a “points” geom is `geom_point`.

We just covered three plotting elements:

- Data
- Aesthetics
- Geoms

These are three elements that you will almost always specify when using `ggplot`, and they are sufficient to create a number of basic plots.

Creating a ggplot object

You can create a scatterplot using ggplot using the following code format:

```
## Generic code
ggplot(data = dataframe) +
  geom_point(mapping = aes(x = column_1, y = column_2,
                           color = column_3))
```

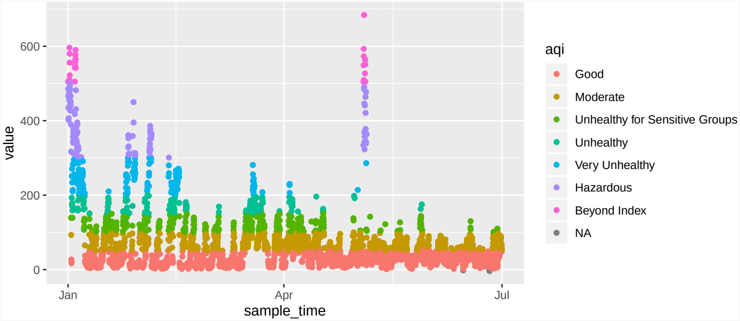
Notice that:

1. The ggplot call specifies the **dataframe** with the data you want to plot
2. A **geom** is added using the appropriate geom_* function for a scatterplot (geom_point).
3. The mappings between columns in the dataframe and **aesthetics** of the geom is specified within an aes call in the mapping argument of the geom_* function call.
4. The aes call includes mappings to two aesthetics that are required from the geom_point geom (x and y) and one that is optional (color).

Creating a ggplot object

Let's put these ideas together to write the code to create a plot for our example data:

```
ggplot(data = beijing_pm) +
geom_point(mapping = aes(x = sample_time, y = value,
                         color = aqi))
```



Adding geoms

There are a number of different `geom_*` functions you can use to add geoms to a plot. They are divided between geoms that directly map the data to an aesthetic and those that show some summary or statistic of the data.

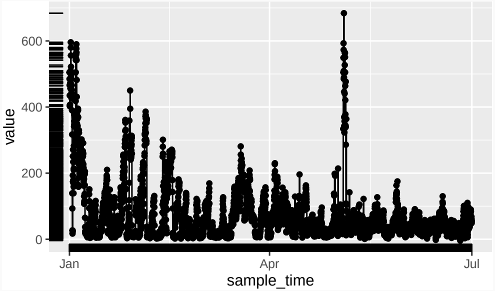
Some of the most common direct-mapping geoms are:

Geom(s)	Description
`geom_point`	Points in 2-D (e.g. scatterplot)
`geom_line`, `geom_path`	Connect observations with a line
`geom_abline`	A line with a certain intercept and slope
`geom_hline`, `geom_vline`	A horizontal or vertical line
`geom_rug`	A rug plot
`geom_label`, `geom_text`	Text labels

Creating a ggplot object

You can add several geoms to the same plot as layers:

```
ggplot(data = beijing_pm) +
geom_point(mapping = aes(x = sample_time, y = value)) +
geom_line(mapping = aes(x = sample_time, y = value)) +
geom_rug(mapping = aes(x = sample_time, y = value))
```



Creating a ggplot object

You may have noticed that all of these geoms use the same aesthetic mappings (height to x-axis position, weight to y-axis position, and sex to color). To save time, you can specify the aesthetic mappings in the first ggplot call. These mappings will then be the default for any of the added geoms.

```
ggplot(data = beijing_pm,
       mapping = aes(x = sample_time, y = value)) +
  geom_point() +
  geom_line() +
  geom_rug()
```

Creating a ggplot object

Because the first argument of the ggplot call is a dataframe, you can also “pipe into” a ggplot call:

```
beijing_pm %>%
  ggplot(aes(x = sample_time, y = value)) +
  geom_point() +
  geom_line() +
  geom_rug()
```

In-course exercise

We'll take a break now to continue the in-course exercise for this week (Section 4).

Plot aesthetics

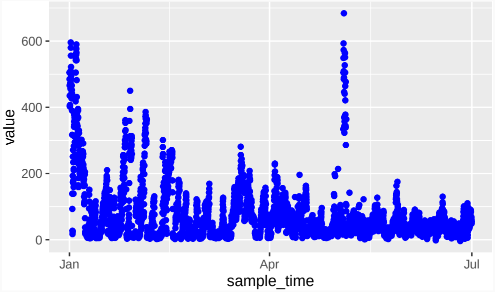
Which aesthetics you must specify in the `aes` call depend on which geom you are adding to the plot.

You can find out the aesthetics you can use for a geom in the "Aesthetics" section of the geom's help file (e.g., `?geom_point`).

Required aesthetics are in bold in this section of the help file and optional ones are not.

Constant aesthetics

Instead of mapping an aesthetic to an element of your data, you can use a constant value for the aesthetic. For example, you may want to make all the points blue, rather than having color map to AQI:



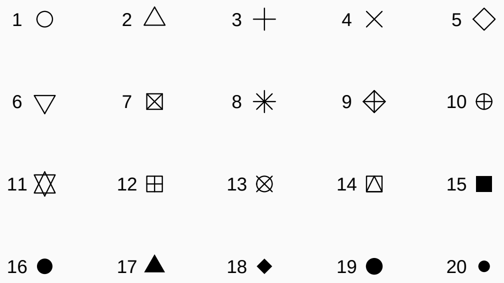
In this case, you can define that aesthetic as a constant for the geom, **outside** of an `aes` statement.

Constant aesthetics

For example, you may want to change the shape of the points in a scatterplot from their default shape, but not map them to a particular element of the data.

In R, you can specify point shape with a number. Here are the shapes that correspond to the numbers 1 to 25:

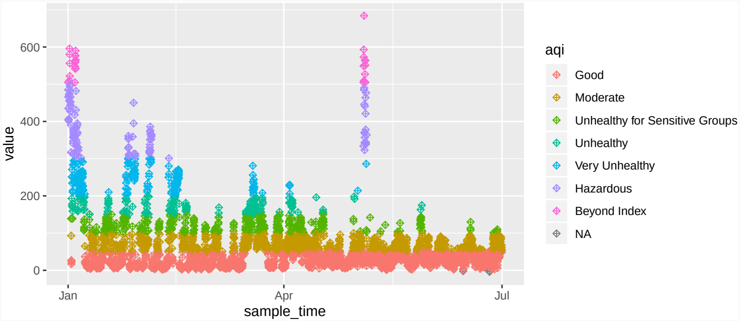
```
## Warning: `data_frame()` is deprecated, use `tibble()`.
## This warning is displayed once per session.
```



Constant aesthetics

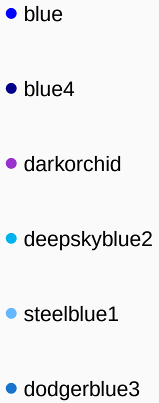
Here is an example of mapping point shape to a constant value other than the default:

```
ggplot(data = beijing_pm) +
  geom_point(mapping = aes(x = sample_time, y = value,
                                        color = aqi),
             shape = 9)
```



Constant aesthetics

R has character names for different colors. For example:

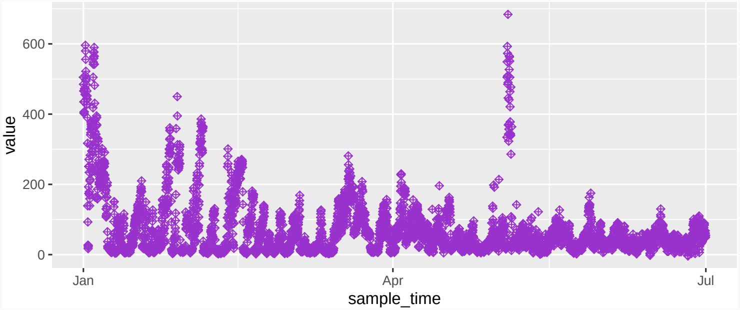


Google “R colors” and search the images to find links to listings of different R colors.

Constant aesthetics

Here is an example of mapping point shape and color to constant values other than the defaults:

```
ggplot(data = beijing_pm) +
  geom_point(mapping = aes(x = sample_time, y = value),
             shape = 9,
             color = "darkorchid")
```



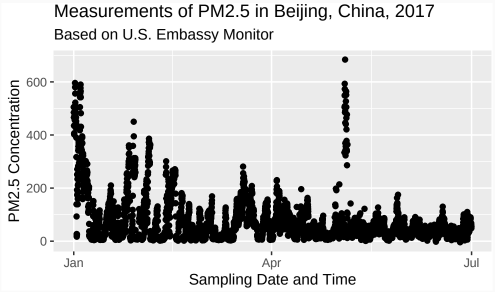
Useful plot additions

There are also a number of elements that you can add onto a `ggplot` object using `+`. A few very frequently used ones are:

Element	Description
`ggtitle`	Plot title
`xlab`, `ylab`, `labs`	x- and y-axis labels
`xlim`, `ylim`	Limits of x- and y-axis
`expand_limits`	Include a value in a range

Useful plot additions

```r
ggplot(data = beijing_pm) +
  geom_point(mapping = aes(x = sample_time, y = value)) +
  labs(x = "Sampling Date and Time",
       y = "PM2.5 Concentration") +
  ggtitle("Measurements of PM2.5 in Beijing, China, 2017",
          subtitle = "Based on U.S. Embassy Monitor")
```



In-course exercise

We'll take a break now to continue the in-course exercise for this week (Section 5 of Week 3).

Adding geoms

There are a number of different `geom_*` functions you can use to add geoms to a plot. They are divided between geoms that directly map the data to an aesthetic and those that show some summary or statistic of the data.

Some of the most common "statistical" geoms are:

Geom(s)	Description
`geom_histogram`	Show distribution in 1-D
`geom_hex`, `geom_density`	Show distribution in 2-D
`geom_col`, `geom_bar`	Create bar charts
`geom_boxplot`, `geom_dotplot`	Create boxplots and related plots
`geom_smooth`	Add a fitted line to a scatterplot

Adding geoms

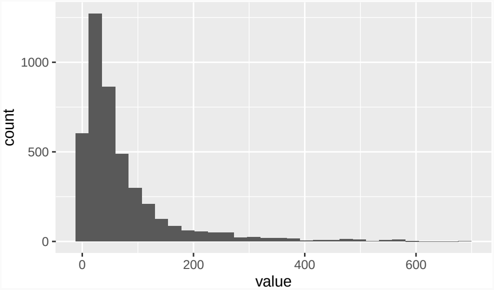
These “statistical” geoms all input the original data and perform some calculations on that data to determine how to plot the final geom. Often, this calculation involves some kind of summarization.

For example, the geom for a histogram (`geom_hex`) divides the data into an evenly-sized set of “bins” and then calculates the number of points in each bin to provide a visualization of how the data is distributed.

Adding geoms

A histogram geom is a similar idea, but only gives the distribution across one variable:

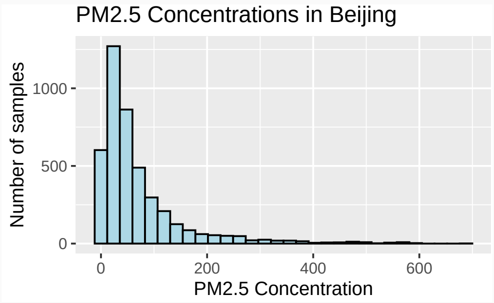
```r
ggplot(data = beijing_pm) +
  geom_histogram(aes(x = value))
```



Histogram example

You can add some elements to the histogram, like `ggtitle`, and `labs`:

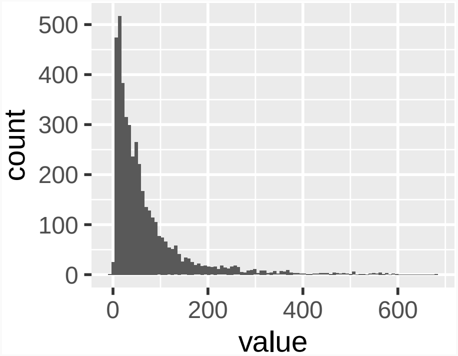
```
ggplot(beijing_pm, aes(x = value)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("PM2.5 Concentrations in Beijing") +
  labs(x = "PM2.5 Concentration", y = "Number of samples")
```



Histogram example

geom_histogram also has its own special argument, bins. You can use this to change the number of bins that are used to make the histogram:

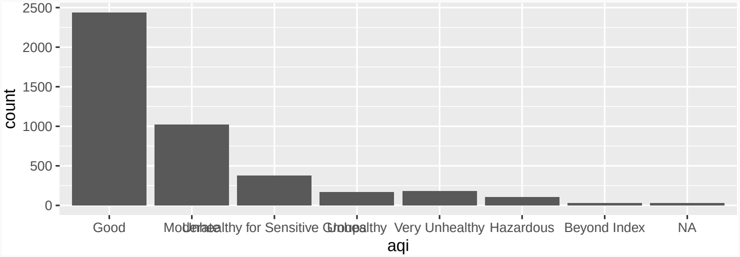
```r
ggplot(beijing_pm, aes(x = value)) +
  geom_histogram(bins = 100)
```



Bar chart

You can use the geom_bar geom to create a barchart:

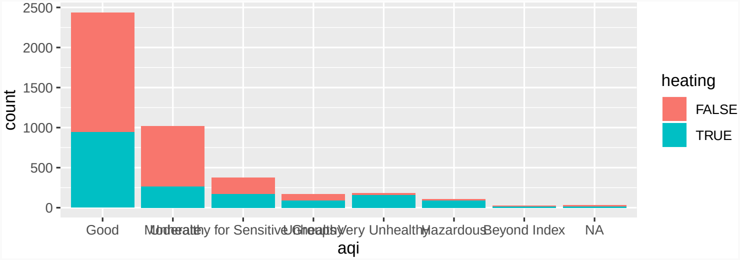
```
ggplot(beijing_pm, aes(x = aqi)) +
  geom_bar()
```



Bar chart

You can use the geom_bar geom to show counts for two factors by using x for one and fill for the other:

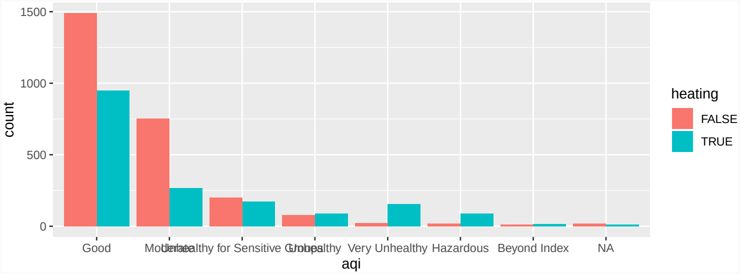
```r
ggplot(beijing_pm, aes(x = aqi, fill = heating)) +
  geom_bar()
```



Bar chart

With the geom_bar geom, you can use the position argument to change how the bars for different groups are shown ("stack", "dodge", "fill"):

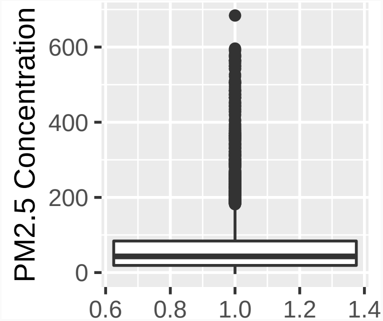
```r
ggplot(beijing_pm, aes(x = aqi, fill = heating)) +
  geom_bar(position = "dodge")
```



Boxplot example

To create a boxplot, you can use geom_boxplot:

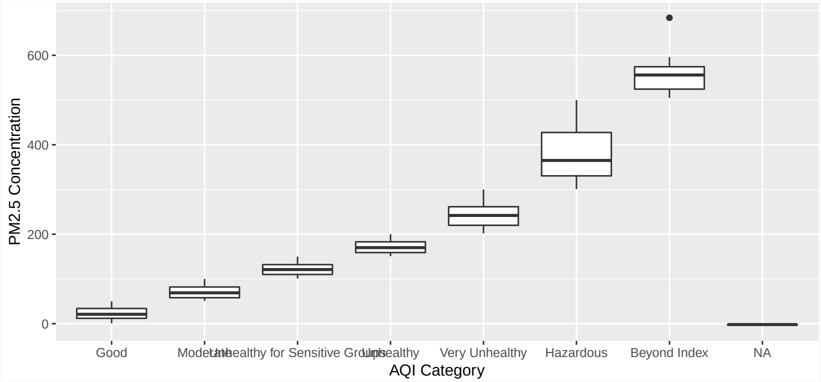
```
ggplot(beijing_pm, aes(x = 1, y = value)) +
  geom_boxplot() +
  labs(x = "", y = "PM2.5 Concentration")
```



Boxplot example

You can also do separate boxplots by a factor. In this case, you'll need to include two aesthetics (`x` and `y`) when you initialize the ggplot object.

```r
ggplot(beijing_pm, aes(x = aqi, y = value, group = aqi)) +
  geom_boxplot() +
  labs(x = "AQI Category", y = "PM2.5 Concentration")
```



In-course exercise

We'll take a break now to finish the in-course exercise for this week (Section 6 of Week 3).