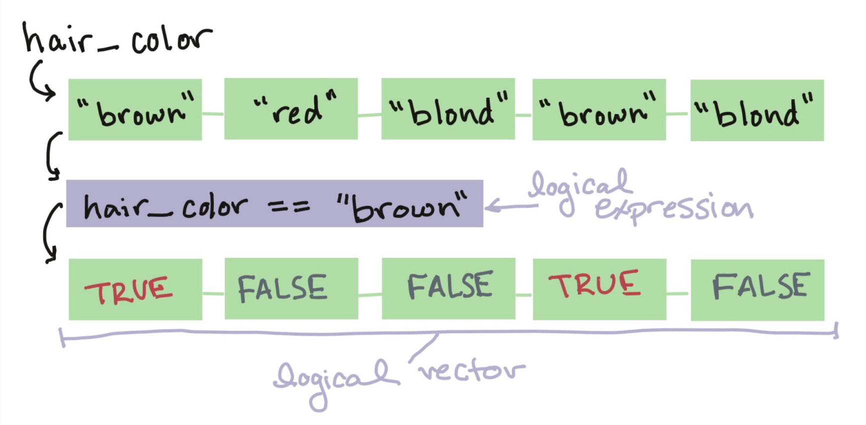
Exploring data #1

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:

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

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

```
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
##    <chr>         <dbl> <lgl>
##  1 1/1/2017 0:00   505 TRUE
##  2 1/1/2017 1:00   485 FALSE
##  3 1/1/2017 2:00   466 FALSE
##  4 1/1/2017 3:00   435 FALSE
##  5 1/1/2017 4:00   405 FALSE
##  6 1/1/2017 5:00   402 FALSE
##  7 1/1/2017 6:00   407 FALSE
##  8 1/1/2017 7:00   435 FALSE
##  9 1/1/2017 8:00   472 FALSE
## 10 1/1/2017 9:00   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:

```
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:

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

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

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

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

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

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

```r
!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 6
##   sample_time value qc    aqi   beyond_index
##   <chr>       <dbl> <chr> <fct> <lgl>
## 1 1/1/2017 0~   505 Valid Beyo~ TRUE
## 2 1/1/2017 1~   485 Valid Haza~ FALSE
## 3 1/1/2017 2~   466 Valid Haza~ FALSE
## # ... with 1 more variable: heating <lgl>
```

Logical vectors

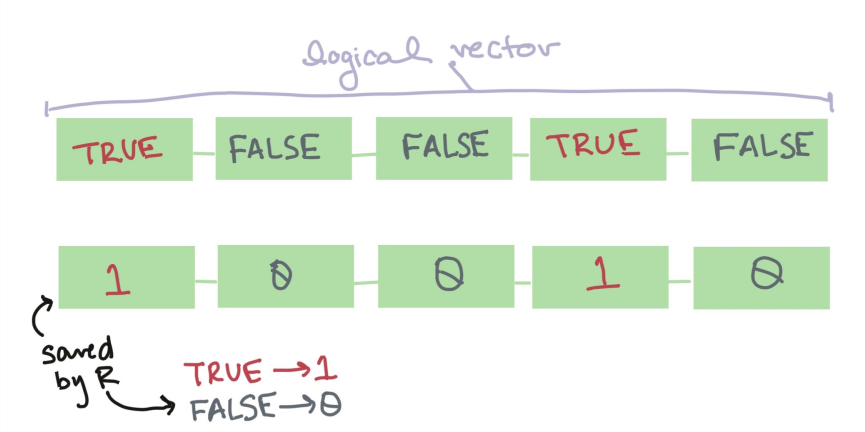
Or, with `!`, just the rows where `heating` is `FALSE`:

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

```
## # A tibble: 0 x 6
## # ... with 6 variables: sample_time <chr>,
## #   value <dbl>, qc <chr>, aqi <fct>,
## #   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/}

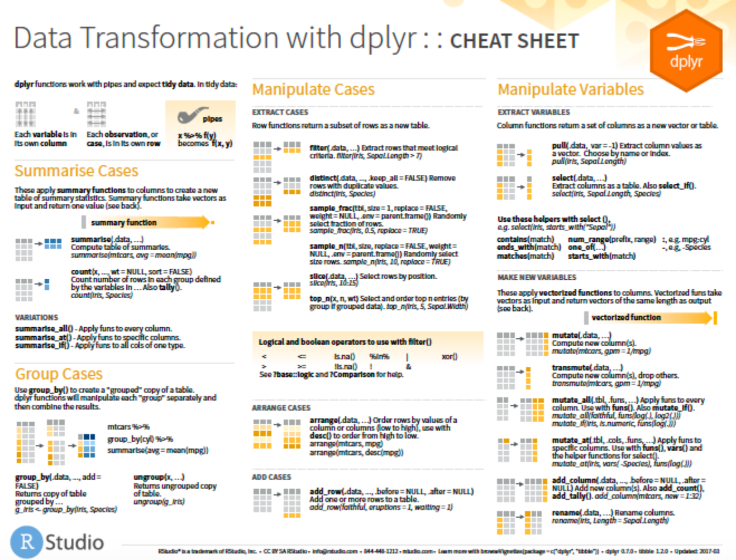
Cheatsheets

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:

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

```r
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:

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

```
## Warning: package 'ggplot2' was built under R
## version 3.5.2
```

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)
```

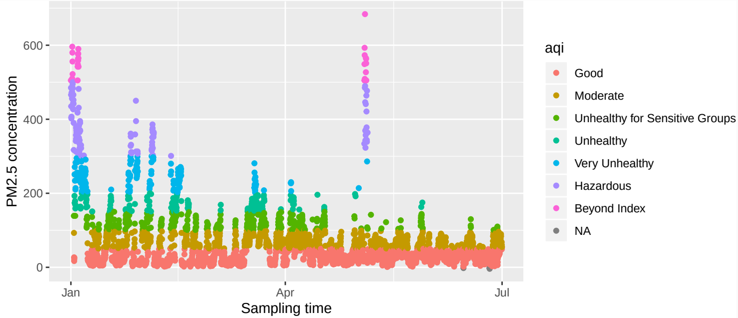
```
## # A tibble: 3 x 5
##   sample_time         value qc    aqi       heating
##   <dttm>              <dbl> <chr> <fct>     <lgl>
## 1 2017-01-01 00:00:00   505 Valid Beyond ~ TRUE
## 2 2017-01-01 01:00:00   485 Valid Hazardo~ TRUE
## 3 2017-01-01 02:00:00   466 Valid Hazardo~ TRUE
```

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))
```

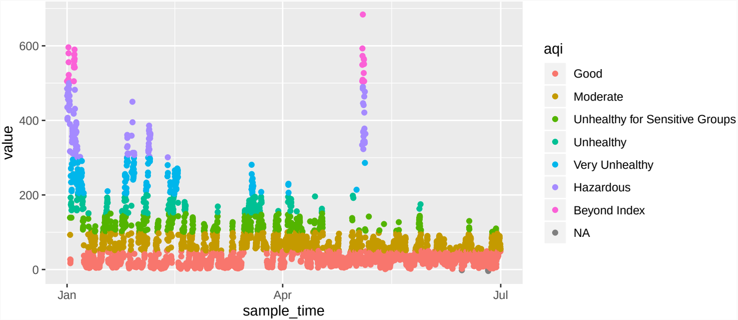
Notice that:

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.

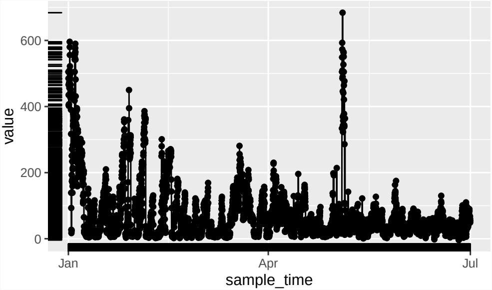
Some of the most common direct-mapping geoms are:

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

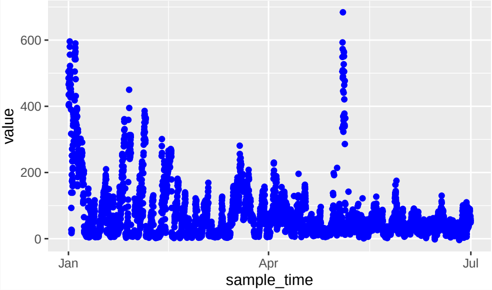
Which aesthetics you must specify in the `aes` call depend on which geom you are adding to the plot.

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:

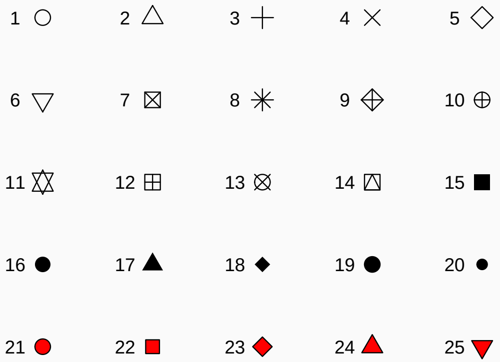


In this case, you can define that aesthetic as a constant for the geom, **outside** of an `aes` statement.

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.

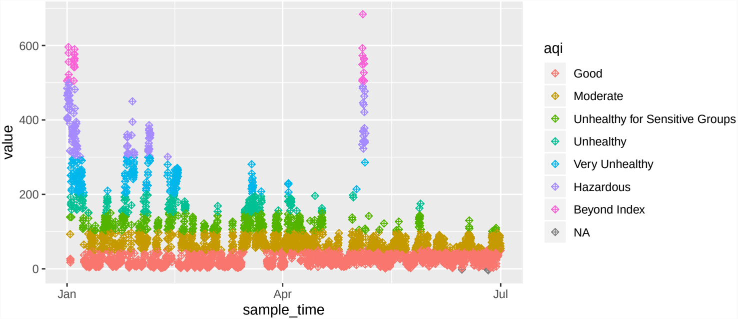
In R, you can specify point shape with a number. Here are the shapes that correspond to the numbers 1 to 25:



Constant aesthetics

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

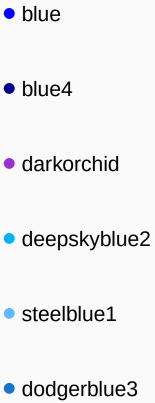
```
ggplot(data = beijing_pm) +
  geom_point(mapping = aes(x = sample_time, y = value,
                           color = aqi),
             shape = 9)
```



Constant aesthetics

R has character names for different colors. For example:

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

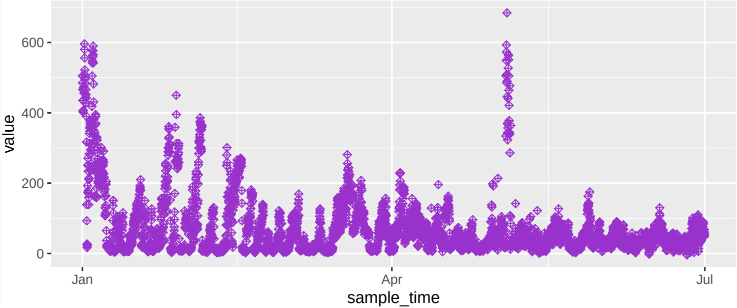


Google "R colors" and search the images to find links to listings of different R colors.

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")
```



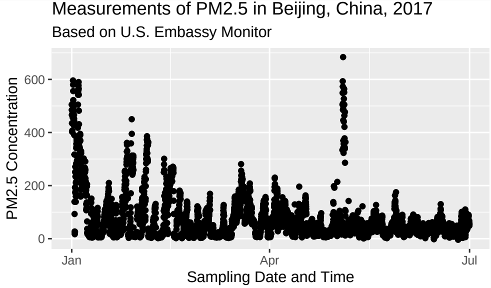
Useful plot additions

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

```
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:

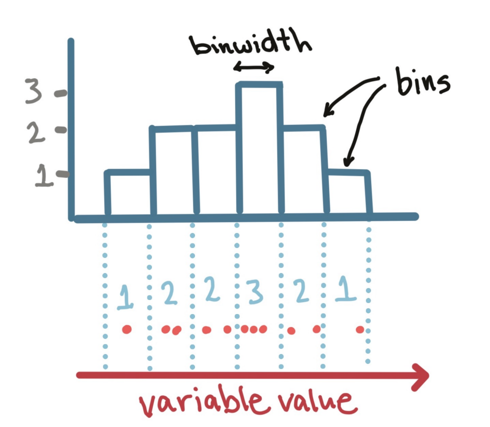
Geom(s)	Description
`geom_histogram`	Show distribution in 1-D
`geom_hex`, `geom_density`	Show distribution in 2-D
`geom_col`, `geom_bar`	Create bar charts
`geom_boxplot`, `geom_dotplot`	Create boxplots and related plots
`geom_smooth`	Add a fitted line to a scatterplot

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.

Adding geoms

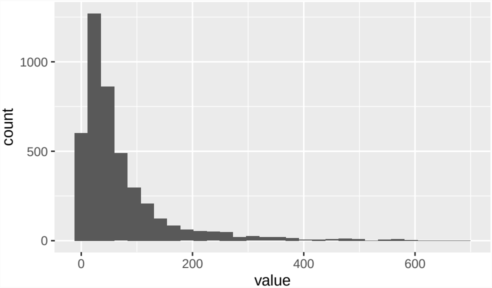
For example, the geom for a histogram (`geom_hist`) divides the data into an evenly-sized set of “bins” and then calculates the number of points in each bin to provide a visualization of how the data is distributed.



Adding geoms

To plot a histogram of PM_{2.5} concentrations in the Beijing data, run:

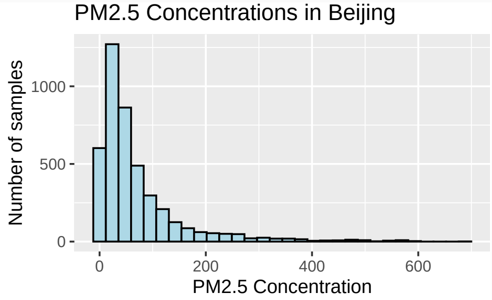
```r
ggplot(data = beijing_pm) +
  geom_histogram(aes(x = value))
```



Histogram example

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

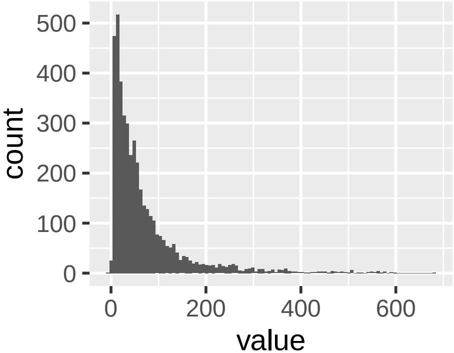
```
ggplot(beijing_pm, aes(x = value)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("PM2.5 Concentrations in Beijing") +
  labs(x = "PM2.5 Concentration", y = "Number of samples")
```



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:

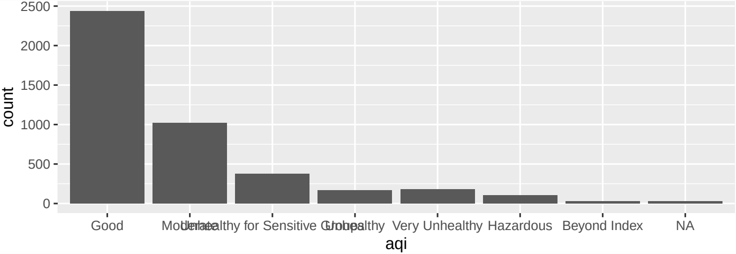
```
ggplot(beijing_pm, aes(x = value)) +
  geom_histogram(bins = 100)
```



Bar chart

You can use the geom_bar geom to create a barchart:

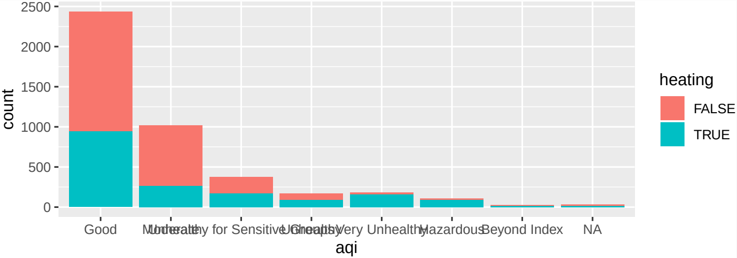
```r
ggplot(beijing_pm, aes(x = aqi)) +
  geom_bar()
```



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:

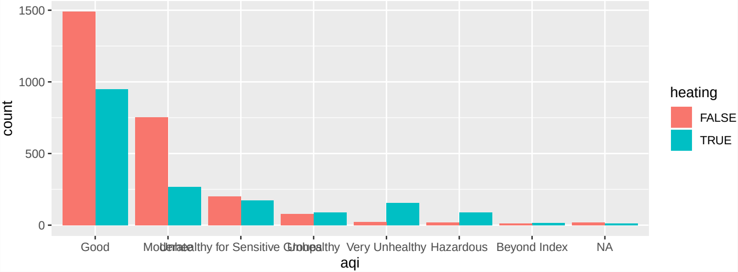
```r
ggplot(beijing_pm, aes(x = aqi, fill = heating)) +
  geom_bar()
```



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"):

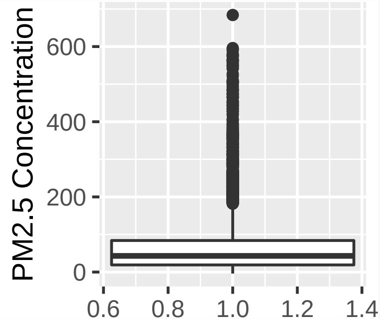
```
ggplot(beijing_pm, aes(x = aqi, fill = heating)) +
  geom_bar(position = "dodge")
```



Boxplot example

To create a boxplot, you can use `geom_boxplot`:

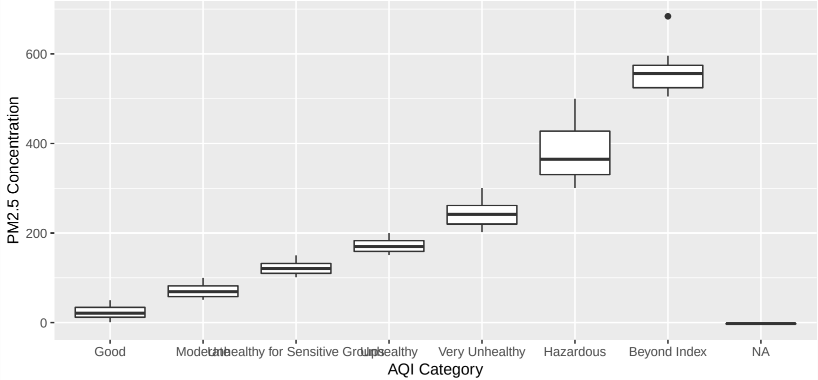
```r
ggplot(beijing_pm, aes(x = 1, y = value)) +
  geom_boxplot() +
  labs(x = "", y = "PM2.5 Concentration")
```



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.

```
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).