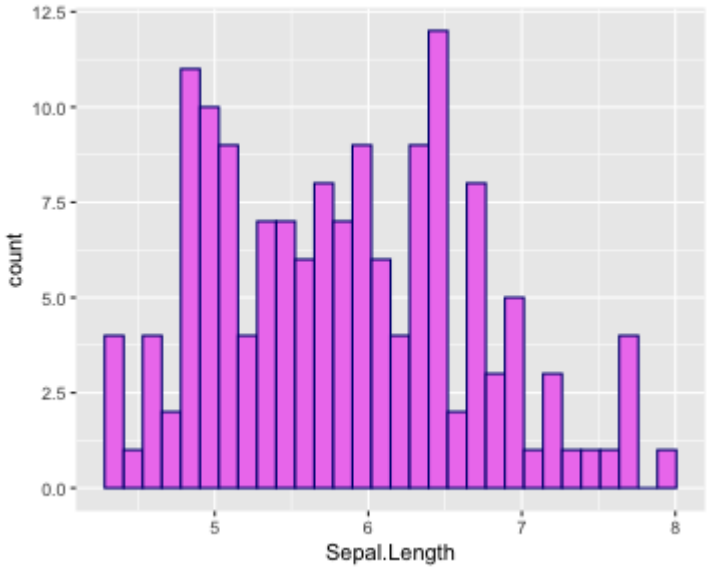
Stats and geoms, and some best coding practices

Stephanie J. Spielman

Data Science for Biologists, Spring 2020

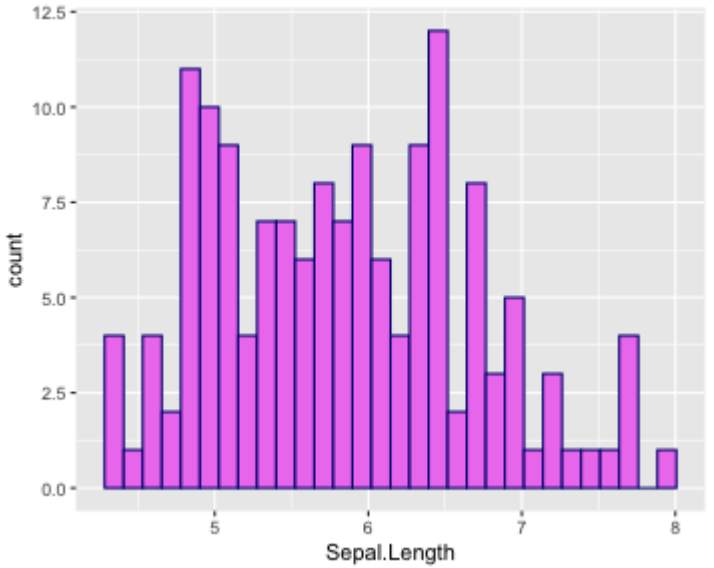
Where **geom** and **stat** meet

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram(fill = "violet",
                 color = "navy")
## `stat_bin()` using `bins = 30`. Pick better value with
`binwidth`.
```



Where **geom** and **stat** meet

```
ggplot(iris, aes(x = Sepal.Length)) +
  stat_bin(fill = "violet",
           color = "navy")
## `stat_bin()` using `bins = 30`. Pick better value with
`binwidth`.
```



?geom_histogram

geom_freqpoly {ggplot2} R Documentation

Histograms and frequency polygons

Description

Visualise the distribution of a single continuous variable by dividing the x axis into bins and counting the number of observations in each bin. Histograms (`geom_histogram()`) display the counts with bars; frequency polygons (`geom_freqpoly()`) display the counts with lines. Frequency polygons are more suitable when you want to compare the distribution across the levels of a categorical variable.

Usage

```
geom_freqpoly(mapping = NULL, data = NULL, stat = "bin",
  position = "identity", ..., na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)

geom_histogram(mapping = NULL, data = NULL, stat = "bin",
  position = "stack", ..., binwidth = NULL, bins = NULL,
  na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)

stat_bin(mapping = NULL, data = NULL, geom = "bar",
  position = "stack", ..., binwidth = NULL, bins = NULL,
  center = NULL, boundary = NULL, breaks = NULL,
  closed = c("right", "left"), pad = FALSE, na.rm = FALSE,
  show.legend = NA, inherit.aes = TRUE)
```

Using `geom_freqpoly`

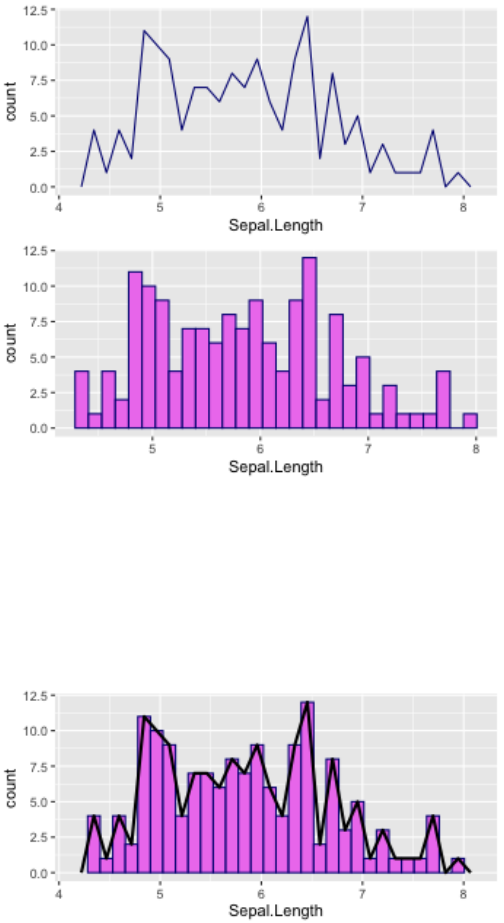
```
ggplot(iris,
       aes(x = Sepal.Length)) ->
p_shared

p1 <- p_shared +
       geom_freqpoly(color = "navy")

p2 <- p_shared +
       geom_histogram(fill = "violet",
                      color = "navy")

p3 <- p_shared +
       geom_histogram(fill = "violet",
                      color = "navy") +
       geom_freqpoly(color = "black",
                     size = 1)

p1 + p2 + plot_spacer() + p3 +
plot_layout(ncol=1)
```



?geom_histogram

geom_freqpoly {ggplot2} R Documentation

Histograms and frequency polygons

Description

Visualise the distribution of a single continuous variable by dividing the x axis into bins and counting the number of observations in each bin. Histograms (`geom_histogram()`) display the counts with bars; frequency polygons (`geom_freqpoly()`) display the counts with lines. Frequency polygons are more suitable when you want to compare the distribution across the levels of a categorical variable.

Usage

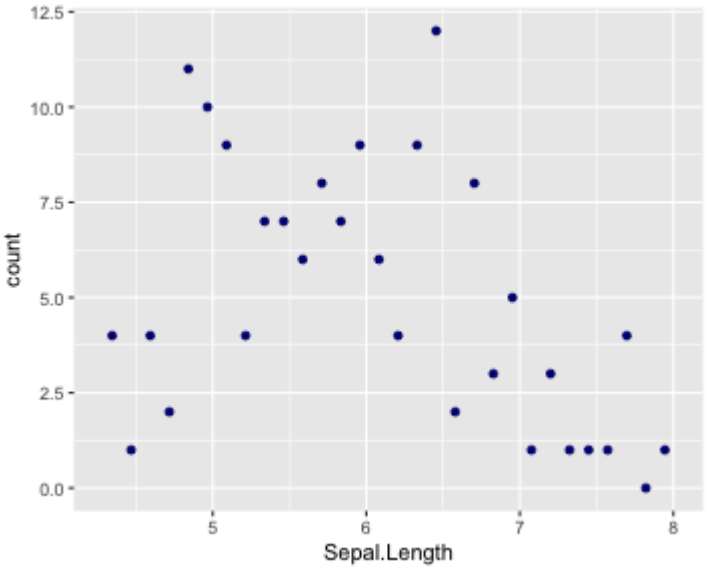
```
geom_freqpoly(mapping = NULL, data = NULL, stat = "bin",
  position = "identity", ..., na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)

geom_histogram(mapping = NULL, data = NULL, stat = "bin",
  position = "stack", ..., binwidth = NULL, bins = NULL,
  na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)

stat_bin(mapping = NULL, data = NULL, geom = "bar",
  position = "stack", ..., binwidth = NULL, bins = NULL,
  center = NULL, boundary = NULL, breaks = NULL,
  closed = c("right", "left"), pad = FALSE, na.rm = FALSE,
  show.legend = NA, inherit.aes = TRUE)
```

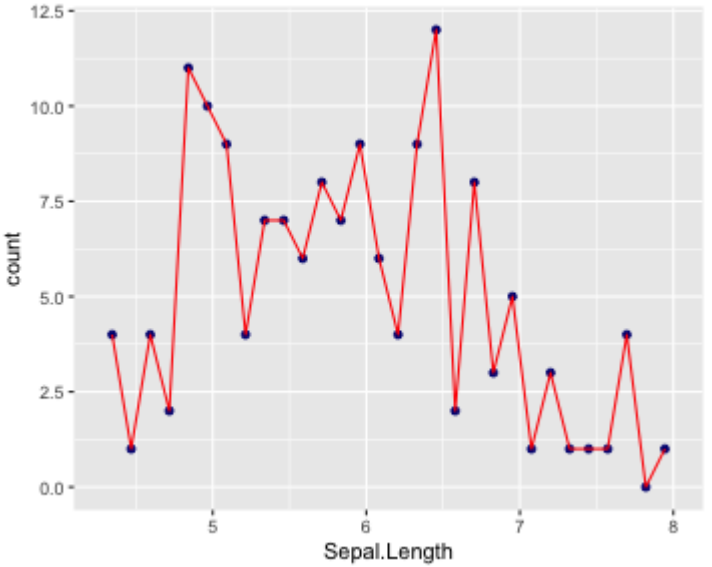
Examining stat and geom interplay

```
ggplot(iris, aes(x = Sepal.Length)) +
  stat_bin(geom = "point",
           color = "navy")
```

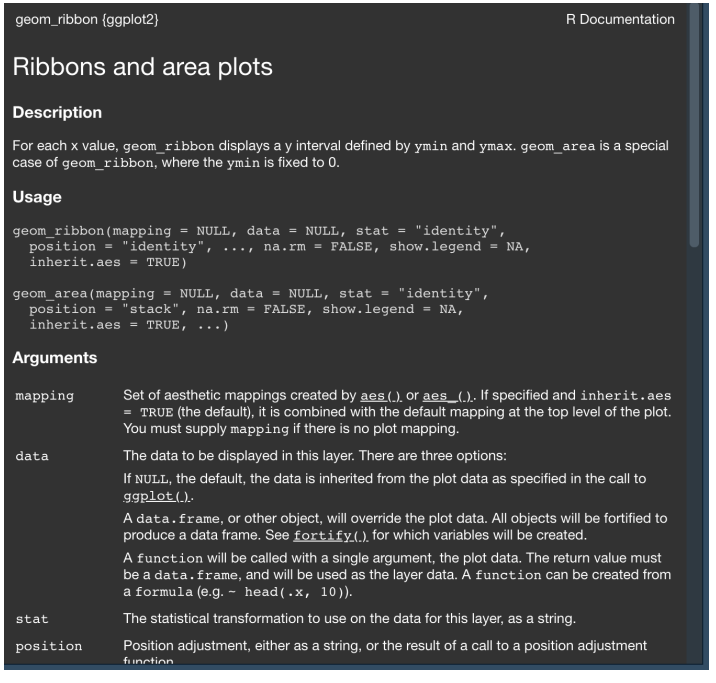


Examining stat and geom interplay

```
ggplot(iris, aes(x = Sepal.Length)) +
  stat_bin(geom = "point",
           color = "navy") +
  stat_bin(geom = "line",
           color = "red")
```



?geom_area



geom_ribbon {ggplot2} R Documentation

Ribbons and area plots

Description

For each x value, `geom_ribbon` displays a y interval defined by `ymin` and `ymax`. `geom_area` is a special case of `geom_ribbon`, where the `ymin` is fixed to 0.

Usage

```
geom_ribbon(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", ..., na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)

geom_area(mapping = NULL, data = NULL, stat = "identity",
  position = "stack", na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE, ...)
```

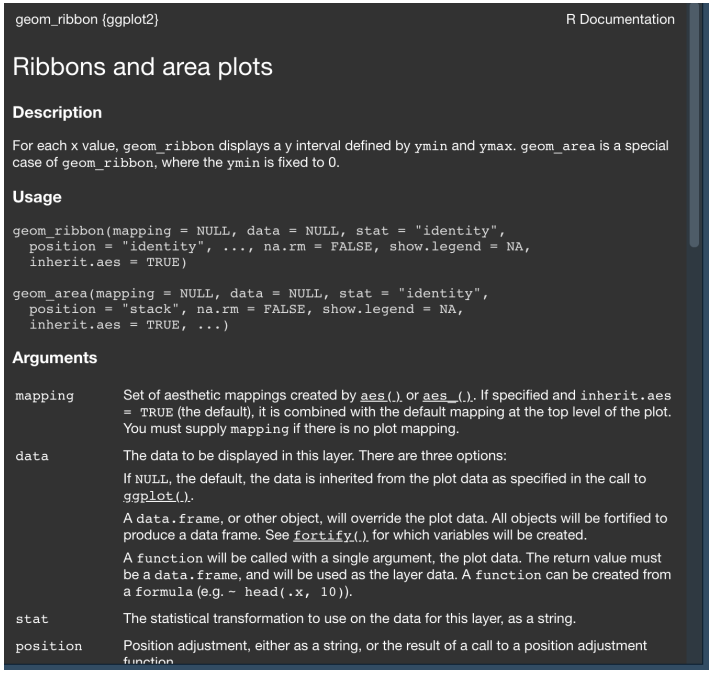
Arguments

`mapping`	Set of aesthetic mappings created by `aes()` or `aes_()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer, as a string.
`position`	Position adjustment, either as a string, or the result of a call to a position adjustment function.

Examining stat and geom interplay

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_area(fill = "violet",
            color = "navy")
## Error: geom_area requires the following missing aesthetics: y
```

?geom_area



geom_ribbon {ggplot2} R Documentation

Ribbons and area plots

Description

For each x value, `geom_ribbon` displays a y interval defined by `ymin` and `ymax`. `geom_area` is a special case of `geom_ribbon`, where the `ymin` is fixed to 0.

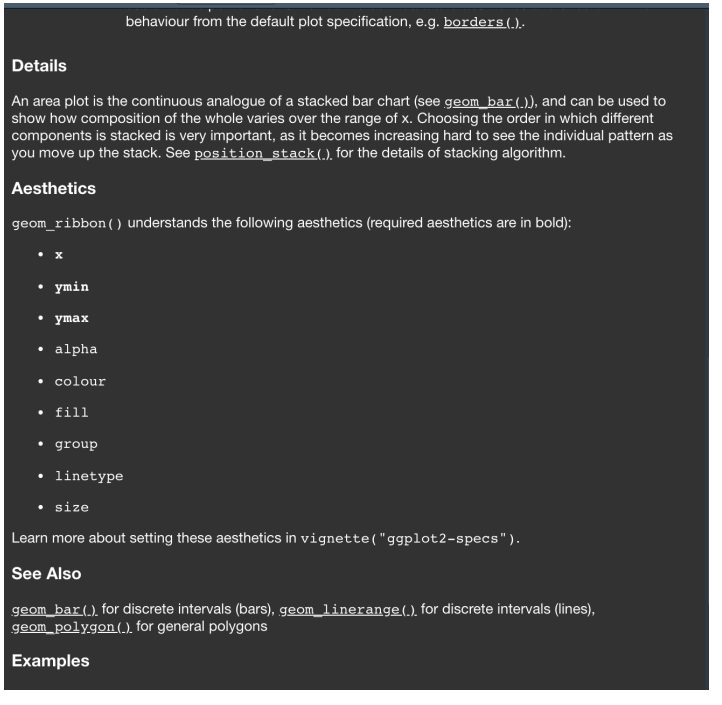
Usage

```
geom_ribbon(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", ..., na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)

geom_area(mapping = NULL, data = NULL, stat = "identity",
  position = "stack", na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE, ...)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()` or `aes_()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer, as a string.
`position`	Position adjustment, either as a string, or the result of a call to a position adjustment function.



behaviour from the default plot specification, e.g. `borders()`.

Details

An area plot is the continuous analogue of a stacked bar chart (see `geom_bar()`), and can be used to show how composition of the whole varies over the range of x. Choosing the order in which different components is stacked is very important, as it becomes increasing hard to see the individual pattern as you move up the stack. See `position_stack()` for the details of stacking algorithm.

Aesthetics

`geom_ribbon()` understands the following aesthetics (required aesthetics are in bold):

- **`x`**
- **`ymin`**
- **`ymax`**
- `alpha`
- `colour`
- `fill`
- `group`
- `linetype`
- `size`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`geom_bar()` for discrete intervals (bars), `geom_linerange()` for discrete intervals (lines), `geom_polygon()` for general polygons

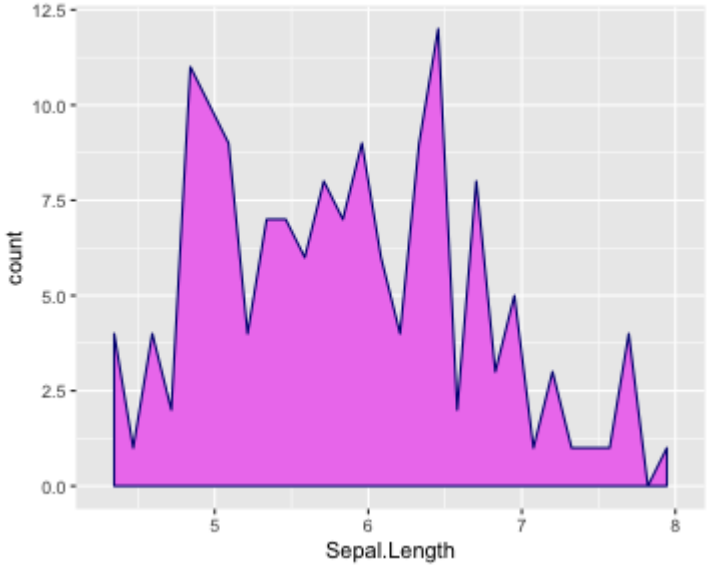
Examples

Examining stat and geom interplay

```
ggplot(iris, aes(x = Sepal.Length,
                 y = ..count..)) +
  geom_area(fill = "violet",
            color = "navy")
## Error: Aesthetics must be valid computed stats. Problematic
aesthetic(s): y = ..count...
## Did you map your stat in the wrong layer?
```

Examining stat and geom interplay

```
ggplot(iris, aes(x = Sepal.Length,
                 y = ..count..)) +
  geom_area(fill = "violet",
            color = "navy",
            stat = "bin")
```



`geom_bar` vs `geom_col`

Look it up and practice the same concepts!

Best practices (a limited set)

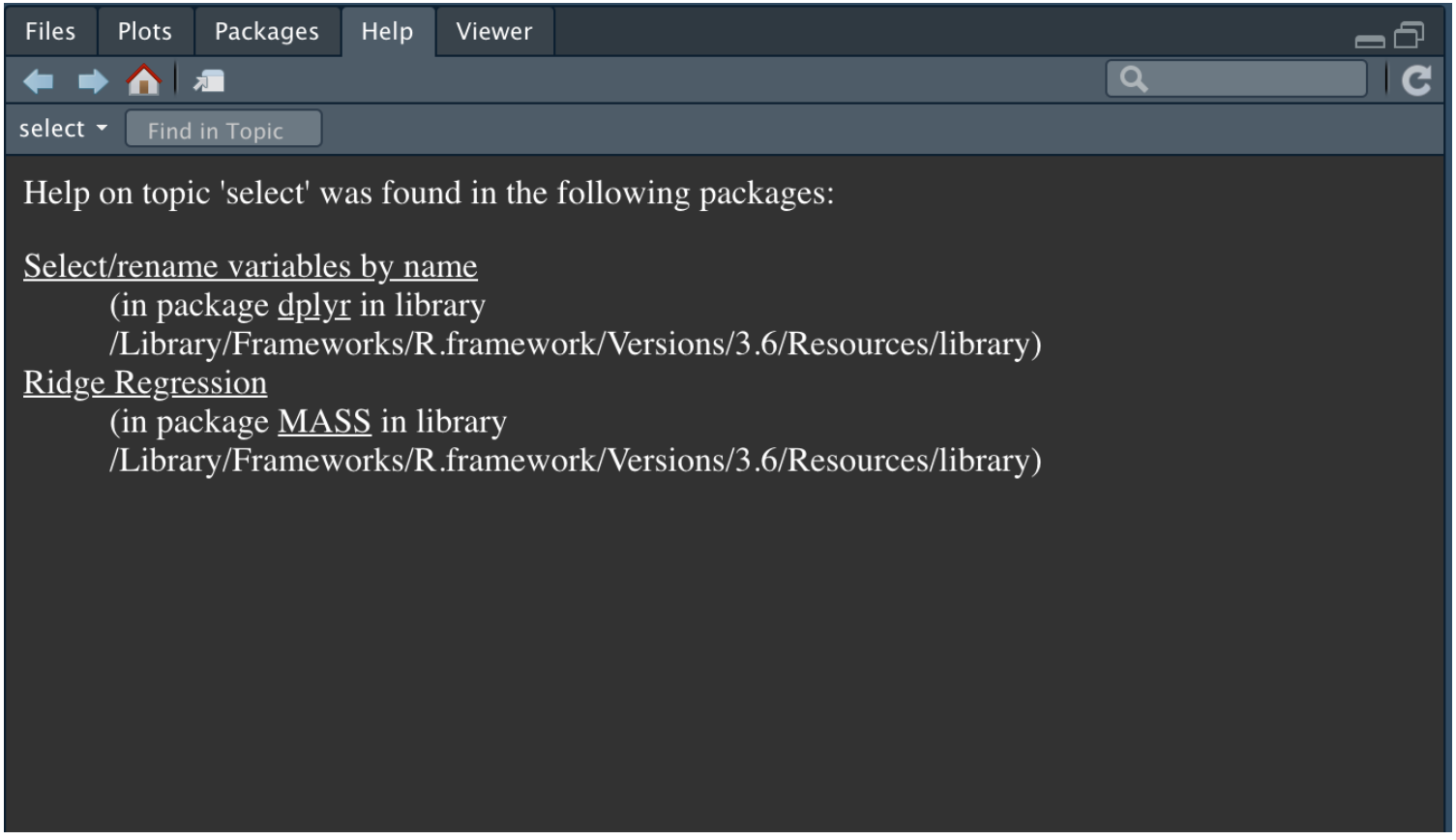
- Know your namespace
- Use meaningful variable names
- Avoid hardcoding
- Set checkpoints ("assertions")
- Whenever you copy/paste more than twice, write a function

Know your namespace

```
diamonds %>%
  select(cut)
## # A tibble: 53,940 x 1
##    cut
##    <ord>
##  1 Ideal
##  2 Premium
##  3 Good
##  4 Premium
##  5 Good
##  6 Very Good
##  7 Very Good
##  8 Very Good
##  9 Fair
## 10 Very Good
## # … with 53,930 more rows
```

```
diamonds %>%
  select(cut)
## Error in select(., cut):
unused argument (cut)
```

?select


Files
Plots
Packages
Help
Viewer
select
Find in Topic
Help on topic 'select' was found in the following packages:
Select/rename variables by name
(in package dplyr in library
/Library/Frameworks/R.framework/Versions/3.6/Resources/library)
Ridge Regression
(in package MASS in library
/Library/Frameworks/R.framework/Versions/3.6/Resources/library)

For safety, be explicit with your namespace

```
diamonds %>%
  dplyr::select(cut)
## # A tibble: 53,940 x 1
##    cut
##    <ord>
##  1 Ideal
##  2 Premium
##  3 Good
##  4 Premium
##  5 Good
##  6 Very Good
##  7 Very Good
##  8 Very Good
##  9 Fair
## 10 Very Good
## # … with 53,930 more rows
```

Know exactly the order R searches for commands/object

```
search()
##  [1] ".GlobalEnv"        "package:MASS"      "package:magrittr"
##  [4] "package:patchwork" "package:xaringan"  "package:forcats"
##  [7] "package:stringr"   "package:purrr"     "package:readr"
## [10] "package:tidyr"     "package:tibble"    "package:ggplot2"
## [13] "package:tidyverse" "package:stats"     "package:graphics"
## [16] "package:grDevices" "package:utils"     "package:datasets"
## [19] "package:methods"   "Autoloads"         "package:base"
```

```
detach(package:MASS)
search()
##  [1] ".GlobalEnv"        "package:magrittr"  "package:patchwork"
##  [4] "package:xaringan"  "package:forcats"   "package:stringr"
##  [7] "package:purrr"     "package:readr"     "package:tidyr"
## [10] "package:tibble"    "package:ggplot2"   "package:tidyverse"
## [13] "package:stats"     "package:graphics"  "package:grDevices"
## [16] "package:utils"     "package:datasets"  "package:methods"
## [19] "Autoloads"         "package:base"
```

Know where your conflicts are

```
conflicts(detail = TRUE)
## $`package:MASS`
## [1] "area" "npk"
##
## $`package:magrittr`
## [1] "%>%"       "%>%"       "%>%"       "set_names" "%>%"
"extract"
##
## $`package:patchwork`
## [1] "area"
##
## $`package:forcats`
## [1] "%>%" "%>%" "%>%" "%>%"
##
## $`package:stringr`
## [1] "%>%" "%>%" "%>%" "%>%"
##
## $`package:purrr`
## [1] "%>%"       "%>%"       "%>%"       "set_names" "%>%"
##
## $`package:tidyr`
## [1] "%>%"       "%>%"       "%>%"       "%>%"       "extract"
"as_tibble"
```

Know where your conflicts are

```
tidyverse_conflicts()
## ── Conflicts ──────────────────────────────────────────────────────────────
 tidyverse_conflicts() ──
## x magrittr::extract()   masks tidyr::extract()
## x magrittr::set_names() masks purrr::set_names()
```

Namespace and variables

```
diamonds <- 25
```

```
head(diamonds)
## [1] 25
```

```
head(ggplot2::diamonds)
## # A tibble: 6 x 10
##   carat cut       color
clarity depth table price
x     y     z
##   <dbl> <ord>     <ord>
<ord>   <dbl> <dbl> <int>
<dbl> <dbl> <dbl>
## 1 0.23  Ideal     E     SI2
61.5    55   326  3.95  3.98
2.43
## 2 0.21  Premium   E     SI1
59.8    61   326  3.89  3.84
2.31
## 3 0.23  Good      E     VS1
56.9    65   327  4.05  4.07
2.31
## 4 0.290 Premium   I     VS2
62.4    58   334  4.2   4.23
```

Use meaningful variable names

- Reserve single letters for integers
  - `x <- 10`
  - NOT `x <- ggplot(...)`
  - NOT `x <- iris %>% filter(...)`
- Use underscores for multiple words
  - `my_plot <- ggplot(...)`
  - NOT `myPlot`, NOT `myplot`, NOT `my-plot`, NOT `my.plot`
- You should always be able to have a *sense* of what a variable is from its name. It's fun to be fun, but too fun leads to bugs. Be KIND to:
  - **You**, returning to code after a week (or 2 minutes.)
  - **Others**, trying to interpret your code
- What do you think `sepal_histogram` is? **A histogram of sepals, somehow**
- What do you think `sillypopculturereference` is? **haha funny! no clue.**

Official styleguide

http://stat405.had.co.nz/r-style.html

Avoid Hardcoding

You **love** seagreen and navy, so you use them everywhere!

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram(fill = "seagreen",
                 color = "navy") -> p1

ggplot(iris, aes(x = Petal.Length)) +
  geom_histogram(fill = "seagreen",
                 color = "navy") -> p2

ggplot(iris, aes(x = Petal.Width)) +
  geom_histogram(fill = "seagreen",
                 color = "navy") -> p3
```

...But what happens if you change your mind?

Avoid Hardcoding

```
favorite_fill  <- "seagreen"
favorite_color <- "navy"

ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram(fill = favorite_fill,
                 color = favorite_color) -> p1

ggplot(iris, aes(x = Petal.Length)) +
  geom_histogram(fill = favorite_fill,
                 color = favorite_color) -> p2

ggplot(iris, aes(x = Petal.Width)) +
  geom_histogram(fill = favorite_fill,
                 color = favorite_color) -> p3
```

Avoid Hardcoding

- If you use a **value** more than ONE TIME, make it a VARIABLE
- Define at the **top** of your script or Rmarkdown (e.g. in setup chunk?)
    - Let's talk about scope real quick.

Set checkpoints ("assertions")

- Assertion are small checks you put in your code that make sure "is this thing that should be true actually true?"
  - If true, keep chugging along
  - If false, stop all program execution because something is *really* wrong

```
## What NEEDS to be true for this code to work?
mean(iris$Sepal.Length)
## [1] 5.843333
```

```
typeof(iris$Sepal.Length)
## [1] "double"
```

```
length(iris)
## [1] 5
```

Adding an assertion

If statements check if a logical condition is `TRUE` or `FALSE`

```
if (THING IS TRUE)
{
      <run this code only if it's true>
}
```

```
if (nrow(iris) == 150)
{
  print("there are 150 rows in iris")
}
## [1] "there are 150 rows in iris"
```

```
if ("definitely-not-a-column" %in% names(iris))
{
  print("this is totally a column in iris")
}
```

Adding assertions with `stop()`

- `stop()` stops R from reading the code, and you get to add your own error message

```
if ( typeof(iris$Sepal.Length) != "double")
{
  stop("Don't try to take mean of non-numbers!!")
}
# Run code after we've confirmed everything is ok
mean(iris$Sepal.Length)
## [1] 5.843333
```

```
if ( typeof(iris$Sepal.Length) != "double"  &
length(iris$Sepal.Length) > 0 )
{
  stop("Don't try to take mean of non-numbers!!")
}
# Run code after we've confirmed everything is ok
mean(iris$Sepal.Length)
## [1] 5.843333
```

Assertions should be sufficiently general

What if I want the mean of diamond `price` and/or `carat`?

```
diamonds %>%
  dplyr::select(price, carat)
## # A tibble: 53,940 x 2
##    price carat
##    <int> <dbl>
##  1   326 0.23
##  2   326 0.21
##  3   327 0.23
##  4   334 0.290
##  5   335 0.31
##  6   336 0.24
##  7   336 0.24
##  8   337 0.26
##  9   337 0.22
## 10   338 0.23
## # … with 53,930 more rows
```

```r
if ( typeof(diamonds$price) != "integer")
{
  stop("Don't try to take mean of non-numbers!!")
}
mean(diamonds$price)
## [1] 3932.8
```

```r
if ( typeof(diamonds$carat) != "integer")
{
  stop("Don't try to take mean of non-numbers!!")
}
## Error in eval(expr, envir, enclos): Don't try to take mean of
non-numbers!!
```

```r
if ( !( typeof(diamonds$price) %in% c("integer", "double") ) )
{
  stop("Don't try to take mean of non-numbers!!")
}
mean(diamonds$price)
## [1] 3932.8
```

We can add the **else** construct to do something if **FALSE**. If does NOT check specific T/F.

```
if (nrow(iris) == 150)
{
  print("there are 150 rows in iris")
} else
{
  print("there are NOT 150 rows in iris")
}
## [1] "there are 150 rows in iris"
```

We can add the **else if** construct as well. It checks another logical condition.

```
if (nrow(iris) == 150)
{
  print("there are 150 rows in iris")
} else if (nrow(iris) > 150)
{
  print("there are MORE than 150 rows in iris")
} else
{
  print("there are FEWER than 150 rows in iris")
}
## [1] "there are 150 rows in iris"
```

At home, write some:

READ THIS: https://www.datamentor.io/r-programming/if-else-statement/

- Is the sum of array `c(3,5,7,22,13)` greater than 10? **Do not calculate the sum except as part of your `if` statement.**
  - If TRUE, define `x<-10`. If FALSE, define `x<-20`
  - **After the if/else**, print out `x`
  - HINT: use the `sum()` function
- Now check: is the sum of array greater than 50? (This one will be TRICKY!!! Why, I wonder?)
- For Wednesday, we will NEED to understand if/else logic!!