Giorgos Mintzopoulos <datacamp.com>

Section 1 - Introduction to ggvis

`ggvis` helps you visualize data sets. For example, the last command in the sample code on the right visualizes the `mtcars` data frame that comes with R. It plots the `wt` variable (weight) of `mtcars` on the x axis and the `mpg` variable (miles per gallon) on the y axis, and it uses points to visualize the data.

Instructions

- Load the `ggvis` package using the `library()` function.
- Change the sample code on the right to plot the `disp` variable of `mtcars` on the x axis. If you are not familiar with the `mtcars` database, find out by typing `?mtcars` in the console.

```
# ggvis is already installed for you; now load it and start playing around
library("ggvis")

# change the code below to plot the disp variable of mtcars on the x axis
mtcars%>%ggvis(~disp,~mpg)%>%layer_points()
```

ggvis and its capabilities

The `ggvis` interface is very intuitive. For example, check the first line of code on the right; you can change the fill color of the points in your graph by adding `fill := "blue"`. Or you can change the `layer_points()` function in the second expression to the right to plot your data differently. You can even plot the same data twice to make your graph more informative. The instructions below will give you a first idea of basic plot manipulation; the exercises that follow will go into more detail.

Instructions

- Change the first line of code to the right such that the resulting graph contains red points instead of blue points.
- Change the second line of code to draw *smooths* instead of *points*. A smooth is a smooth line that summarizes the relationship between a set of points.
- Change the third line of code to draw the same information twice: once as points and once as a smoothed summary line. You can do this by adding multiple layers to your plot.

Giorgos Mintzopoulos <datacamp.com>

```
# The ggvis packages is loaded into the workspace already

# Change the code below to make a graph with red points
mtcars %>% ggvis(~wt, ~mpg, fill := "red") %>% layer_points()

# Change the code below draw smooths instead of points
mtcars %>% ggvis(~wt, ~mpg) %>% layer_smooths()

# Change the code below to amke a graph containing both points and a smoothed summary line
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>% layer_smooths()
```

Section 2 - The grammar of ggvis

ggvis grammar ~ graphics grammar

`ggvis` recreates the grammar of graphics. You can combine a set of data, properties and marks with the following format.

```
<data>  %>%
   ggvis(~<x property>,~<y property>,
            fill = ~<fill property>, ...) %>%
   layer_<marks>()
```

The next challenges depend on one another. Make sure you get the first assignment right before proceeding to the next ones!

Instructions

- Make a graph that uses the points mark to plot variables from the `pressure` data frame. The graph should plot the `temperature` variable on the x axis, and the `pressure` variable on the y axis.
- Transform the plot you just created into a bar graph by changing the type of mark it uses.
- Similarly, change the plot into a line graph, again by changing the type of mark it uses.
- Again based on the solution to the first challenge, make an adaptation: map the fill property of your scatterplot to the `temperature` variable
- Extend your solution to the previous challenge: map the size property to the `pressure` variable.

Giorgos Mintzopoulos <datacamp.com>

```
# Make a scatterplot of the pressure dataset
pressure %>% ggvis(~temperature,~pressure)%>%layer_points()

# Adapt the code you wrote for the first challenge: show bars instead of points
pressure %>% ggvis(~temperature,~pressure)%>%layer_bars()

# Adapt the code you wrote for the first challenge: show lines instead of points
pressure %>% ggvis(~temperature,~pressure)%>%layer_lines()

# Adapt the code you wrote for the first challenge: map the fill property to the temperature variable
pressure %>% ggvis(~temperature,~pressure,fill=~temperature)%>%layer_points()

# Extend the code you wrote for the previous challenge: map the size property to the pressure variable
pressure %>% ggvis(~temperature,~pressure,fill=~temperature,size=~pressure)%>%layer_points()
```

4 essential components of a graph

Every `ggvis` graph contains 4 essential components: data, a coordinate system, marks and corresponding properties. By changing the values of each of these components you can create a vast array of unique plots.

Which part in the following call corresponds to the *marks* element of the `ggvis` grammar?

```
faithful %>%
    ggvis(~waiting, ~eruptions, fill := "red") %>%
    layer_points() %>%
    add_axis("y", title = "Duration of eruption (m)",
             values = c(2, 3, 4, 5), subdivide = 9) %>%
    add_axis("x", title = "Time since previous eruption (m)")
```

Try to run the code in the console on the right and play with the parameters to reveal their influence!

Ans: layer_points()

Section 3 - Three new types of syntax

Three operators: %>%, = and :=

Ysou'll notice that `ggvis` uses the pipe operator, `%>%`, from the `magrittr` package. The pipe operator passes the result from its left-hand side into the first argument of the function on its right-hand side. So `f(x) %>% g(y)` is a shortcut for `g(f(x), y)`.

Next, `ggvis` uses both `=` and `:=` to assign properties.

Giorgos Mintzopoulos <datacamp.com>

`=` *maps* a property to a data value or a set of data values. This is how you visualize variation in your data set. `ggvis` will scale the values to a pleasing range of colors (or sizes, widths, etc.) and add a legend that explains how values are mapped to particular instances of the property. `:=` *sets* a property to a specific color (or size, width, etc.). This is how you customize the appearance of your plots. If you set a property to a number, `ggvis` will usually interpret the number as the number of pixels. If you set a location property to a number, `ggvis` will usually interpret the number as the number of pixels from the top left-hand corner of the plot. You can set the fill of points to any common color name. `ggvis` passes your color selection to `vega`, a javascript library, so you can use any color name recognized by `HTML/CSS`.

Instructions

- The line of `ggvis` code on the right is valid, but is rather hard to read. Rewrite the code with the pipe operator, `%>%`.
- Modify the second graph to *map* the size property to the pressure variable.
- Modify the third graph to *set* the size property to 100 pixels. Each point should appear 100 pixels wide.
- The fourth line of code on the right attempts to make the points red, but something goes wrong. `ggvis` *maps* the fill property of each point to the single data value "red". (And adds a legend to explain the mapping). Fix the code to *set* the fill property to "red".

```
# Rewrite the code with the pipe operator
faithful %>%
  ggvis(x=~waiting,y=~eruptions)%>%
  layer_points()

# Modify this graph to map the size property to the pressure variable
pressure %>%
  ggvis(~temperature,~pressure,size=~pressure) %>%
  layer_points()

# Modify this graph by setting the size property
pressure %>%
  ggvis(~temperature,~pressure,size:=100) %>%
  layer_points()


# Fix this code to set the fill property to red
pressure %>% ggvis (~temperature, ~pressure, fill:="red") %>% layer_points()
```

You can refer to three different types of objects in your `ggvis` code: objects, variables, and raw values.

•If you type a string of letters, `ggvis` will treat the string as an *object* name. It will look for an object with that name in your current environment.
•If you place a tilde, `~`, at the start of the string, `ggvis` will treat the string as a *variable* name. It will look for the column with that name in the data set that the graph visualizes.
•If you surround a string of letters with quotation marks, `ggvis` will treat the string as a raw value, e.g., a piece of text.
Which of the commands below will create a graph that has green points? Try to predict the answer before running the code.

```
red <- "green"
pressure$red <- pressure$temperature

# GRAPH A
pressure %>%
   ggvis(~temperature, ~pressure,
         fill = ~red) %>%
   layer_points()

# GRAPH B
pressure %>%
   ggvis(~temperature, ~pressure,
         fill = "red") %>%
   layer_points()

# GRAPH C
pressure %>%
   ggvis(~temperature, ~pressure,
         fill := red) %>%
   layer_points()
```

Ans: Graph C

Referring to different objects (2)
Which of the commands below will create a graph that uses color to reveal the values of the temperature variable in the pressure data set? Try to predict the answer before running the code.

```
red <- "green"
```

```
pressure$red <- pressure$temperature

# GRAPH A
pressure %>%
   ggvis(~temperature, ~pressure,
         fill = ~red) %>%
   layer_points()

# GRAPH B
pressure %>%
   ggvis(~temperature, ~pressure,
         fill = "red") %>%
   layer_points()

# GRAPH C
pressure %>%
   ggvis(~temperature, ~pressure,
         fill := red) %>%
   layer_points()
```

Ans: Graph A

Properties for points

You can manipulate many different properties when using the points mark, including `x`, `y`, `fill`, `fillOpacity`, `opacity`, `shape`, `size`, `stroke`, `strokeOpacity`, and `strokeWidth`.

The `shape` property in turn recognizes several different values: `circle` (default), `square`, `cross`, `diamond`, `triangle-up`, and `triangle-down`.

Instructions

- The first lines of code on the right *map* the fill property of a graph to a default collection of colors and adds a legend to map the default colors to the labels in `pressure$black`. Change the code to directly *set* the fills to the colors saved in `pressure$black`.
- Plot the `faithful` data with the points mark. Put the `waiting` variable on the x axis and the `eruptions` variable on the y axis. Map size to the `eruptions` variable. Set the opacity to 0.5 (50%), the fill to blue, and the stroke to black.

- Plot the `faithful` data with the points mark. Put the `waiting` variable on the x axis and the `eruptions` variable on the y axis. Map fill opacity to the `eruptions` variable. Set size to 100, the fill to red, the stroke to red, and the shape to cross.

```
# Change the second line of code to set the fills using pressure$black
pressure$black <- c("black", "grey80", "grey50",
                    "navyblue", "blue", "springgreen4",
                    "green4", "green", "yellowgreen",
                    "yellow", "orange", "darkorange",
                    "orangered", "red", "magenta", "violet",
                    "purple", "purple4", "slateblue")
pressure %>% ggvis(~temperature,~pressure,fill:=~black) %>% layer_points()

# Plot the faithful data as described in the second instruction
 faithful %>% ggvis(~waiting,~eruptions,size=~eruptions,opacity:=0.5,fill:="blue",stroke:="black") %>%
layer_points()

# Plot the faithful data as described in the third instruction
faithful %>%
ggvis(~waiting,~eruptions,fillOpacity=~eruptions,size:=100,fill:="red",stroke:="red",shape:="cross") %>
% layer_points()
```

Section 4 - the line, a special type of mark

Properties for lines

In the previous section, you learned that you can manipulate many different properties when using the points mark. This mark type responds to, among others, `x`, `y`, `fill`, `fillOpacity`, `opacity`, `shape`, `size`, `stroke`, `strokeOpacity`, and `strokeWidth`.

Similar to points, lines have specific properties; they respond to: `x`, `y`, `fill`, `fillOpacity`, `opacity`, `stroke`, `strokeDash`, `strokeOpacity`, and `strokeWidth`. As you can see, most of them are common to the properties for points, some are missing (e.g., no `size` property) and others are new (e.g., `strokeDash`).

Instructions

- Change the first line of code on the right to use the lines mark.
- Change the code for the second graph on the right: Use the lines mark, *set* the line color to red, the line width to 2 pixels, and the line type to use dashes that are six pixels long.

```
# Change the code below to use the lines mark
pressure %>% ggvis(~temperature, ~pressure) %>% layer_lines()

# Set the properties described in the second instruction in the graph below
pressure %>% ggvis(~temperature, ~pressure,stroke:="red",strokeWidth:=2,strokeDash:=6) %>%
layer_lines()
```

Giorgos Mintzopoulos <datacamp.com>

Path marks and polygons

The lines mark will always connect the points in your plot from the leftmost point to the rightmost point. This can be undesirable if you are trying to plot a specific shape.

For example, the first lines of code on the right would plot a map of Texas if `ggvis` connected the points in the correct order. You can do this with the paths mark, `layer_paths()`. The paths mark is similar to the lines mark except that it connects points *in the order that they appear in the data set*. So the paths mark will connect the point that corresponds to the first row of the data to the point that corresponds to the second row of data, and so on — no matter where those points appear in the graph.

The second challenge will teach how to create a polygon in `ggvis` by setting the `fill` property of a line or path.

Instructions

- The `texas` data frame is arranged such that consequent observations should be connected. Change the code chunk on the right to plot Texas with paths.
- Set the fill property of the texas map to dark orange (`"darkorange"`). You can start from the solution to the first instruction. This additional property will color your map!

```
# change the third line of code to plot a map of Texas
library("maps")
texas <- ggplot2::map_data("state", region = "texas")
texas %>% ggvis(~long, ~lat) %>% layer_paths()

# Same plot, but set the fill property of the texas map to dark orange
texas %>% ggvis(~long,~lat) %>% layer_paths(fill:="darkorange")
```

Display model fits

`compute_smooth()` is a useful function to use with line graphs. It takes a data frame as input and returns a new data frame as output. The new data frame will contain the x and y values of a smooth line fitted to the data in the original data frame.

For example, the code below computes a smooth line that shows the relationship between the `eruptions` and `waiting` variables of the `faithful` data set.

```
faithful %>%
  compute_smooth(eruptions ~ waiting,
                 model = "lm")
```

Giorgos Mintzopoulos <datacamp.com>

Notice that `compute_smooth()` takes a couple of arguments. First we use the pipe operator to pass it the data set `faithful`. Then we provide an R formula, `eruptions ~ waiting`. An R formula contains two variables connected by a tilde, `~`. `compute_smooth()` will use the variable on the left as the y variable for the smooth line, and it will use the variable on the right as the x variable for the smooth line.
Finally, `compute_smooth()` takes a `model` argument. This is the name of the R modelling function that `compute_smooth()` should use to calculate the smooth line. `lm()` is R's function for building linear models. If you do not supply a `model` value, `compute_smooth()` will generate a set of smoothed coordinates with `loess`.

Instructions

Use `compute_smooth()` to generate the x and y coordinates for a loess smooth line that predicts the `mpg` value of `mtcars` with the `wt` value.

```
# Compute the x and y coordinates for a loess smooth line that predicts mpg with the wt
mtcars %>% compute_smooth(mpg~wt)
```

compute_smooths() to simplify model fits

`compute_smooth()` always returns a data set with two columns, one named `pred_` and one named `resp_`. As a result, it is very easy to use `compute_smooth()` to plot a smoothed line of your data. For example, you can extend your code from the last exercise to plot the results of `compute_smooth()` as a line graph.
Calling `compute_smooth()` can be a bit of a hassle, so `ggvis` includes a layer that automatically calls `compute_smooth()` in the background and plots the results as a smoothed line. That layer is `layer_smooths()`.

Instructions

- Extend the code on the right which you created in the last exercise to plot the results of `compute_smooth()` with the lines mark. The plot should place `pred_` on the x axis and `resp_` on the y axis.
- Recreate the same graph by plotting `mtcars` with the smooths mark. Be sure to place `wt` on the x axis and `mpg` on the y axis.
- Often you'll want to place the results of `layer_smooths()` on a points plot that contains the raw data. Extend the code for the second plot on the right to use *both* `layer_points()` and `layer_smooths()`.

```
# Use 'ggvis()' and 'layer_lines()' to plot the results of compute smooth
mtcars %>% compute_smooth(mpg ~ wt) %>% ggvis(~resp_,~pred_) %>% layer_lines()

# Recreate the graph you coded above with 'ggvis()' and 'layer_smooths()'
mtcars %>% ggvis(x=~wt,y=~mpg) %>% layer_smooths()

# Extend the code for the second plot and add 'layer_points()' to the graph
mtcars %>% ggvis(x=~wt,y=~mpg) %>% layer_points() %>% layer_smooths()
```

Section 5 - Compute functions

Histograms (1)

A histogram - plotted using `layer_histograms()` - shows the distribution of a single continuous variable. To do this, a histogram divides the x axis into evenly spaced intervals, known as bins. Above each bin, the histogram plots a rectangle. The height of the rectangle displays how many values of the variable fell within the range of the bin. As a result, the rectangles show how the frequency of values varies over the range of the variable.
You can change the appearance of a histogram by changing the width of the bins in the histogram. In fact, you should explore different binwidths whenever you make a histogram because different binwidths can reveal different types of information. To change the binwidth of a histogram, *map* the `width` argument of `layer_histograms()` to a number.

`width` is an argument of `layer_histograms()`. For best results, you should write the `width` argument in the parentheses that follow `layer_histograms()`. Always *map* `width` to its value; This will ensure that it uses the same units as the variable on the x axis.

Instructions

- Make a histogram that shows the distribution of the `waiting` variable of the `faithful` data set. Look at the plot to familiarize yourself with the structure of a histogram.
- Starting from the solution to the first challenge, *map* the binwidth of this histogram to 5 units.

```
# Build a histogram of the waiting variable of the faithful data set.
faithful %>%
  ggvis(~waiting) %>% layer_histograms()

# Build the same histogram, but with a binwidth (width argument) of 5 units
faithful %>%
  ggvis(~waiting) %>% layer_histograms(width=5)
```

Giorgos Mintzopoulos <datacamp.com>

Histograms (2)

Have you noticed that the histogram plots data that does not appear directly in your data set? Specifically, it plots the counts of each bin on the y axis.

Behind the scenes, `layer_histograms()` calls `compute_bin()` to calculate these counts. You can calculate the same values by calling `compute_bin()` manually. `compute_bin()` requires at least two arguments: a data set (which you will provide with the `%>%` syntax), and a variable name to bin on. You can also pass `compute_bin()` a binwidth argument, just as you pass `layer_histograms()` a binwidth argument.

`compute_bin()` returns a data frame that provides everything you need to build a histogram from scratch. Notice the similarity with previous cases:

combining `compute_smooth()` and `layer_points()` had the exact same result as using `layer_smooths()` directly! Can you spot the analogy?

Instructions

- Use `compute_bin()` to calculate the counts displayed in the graph on the right. What does your output look like?
- Combine the output of the solution to the first instruction with `ggvis()` and `layer_rects()` to plot a histogram. `layer_rects()` plots simple rectangles. To use it, you need to pass `ggvis()` four properties: `x`, `x2`, `y`, and `y2`. These should correspond to the minimum and maximum `x` values for each rectangle and the minimum and maximum `y` values for each rectangle, respectively. Make sure to keep the binwidth (specified using `width`) equal to 5.

```
# Transform the code below: just compute the bins instead of plotting a histogram
faithful %>% compute_bin(~waiting,width=5)

# Combine the solution to the first challenge with layer_rects() to build a histogram
faithful %>% compute_bin(~waiting,width=5) %>%
             ggvis(x=~xmin_,x2=~xmax_,y=0,y2=~count_) %>%
               layer_rects()
```

Density plots

Density plots provide another way to display the distribution of a single variable. A density plot uses a line to display the density of a variable at each point in its range. You can think of a density plot as a continuous version of a histogram with a different y scale (although this is not exactly accurate).

`compute_density()` takes two arguments, a data set and a variable name. It returns a data

frame with two columns: `pred_`, the x values of the variable's density line, and `resp_`, the y values of the variable's density line.
You can use `layer_densities()` to create density plots. Like `layer_histograms()` it calls the compute function that it needs in the background, so you do not need to worry about calling `compute_density()`.

Instructions

- Use `compute_density()` and `layer_lines()` to make a density plot of the `waiting` variable in the `faithful` data set.
- Use `layer_densities()` to create a density plot of the `waiting` variable of the `faithful` data set. Set the `fill` of the density to green.

```
# Combine compute_density() with layer_lines() to make a density plot of the waiting variable.
faithful %>% compute_density(~waiting) %>% ggvis(~pred_,~resp_) %>% layer_lines()

# Build a density plot directly using layer_densities. Use the correct variables and properties.
faithful %>% ggvis(~waiting) %>% layer_densities(fill:="green")
```

Shortcuts

You do not need to use a compute function to transform the variables in your data set. You can directly plot transformations of the variables. To do this, use the `~` syntax to pass a transformed variable to `ggvis`. For example, the first line of code in the editor on the right will plot a version of `cyl` that has been transformed into a factor (R's version of a categorical variable).

Instructions

- Complete the first line of code to plot a bar graph of the `cyl` factor.
- `layer_bars()` will automatically plot count values on the y axis when you do not provide a y variable. To do this, it uses `compute_count()`. Use `compute_count()` to calculate the count values used in the graph that you coded to solve the first challenge. Do not plot anything for this part of the exercise.

```
# Complete the code to plot a bar graph of the cyl factor.
mtcars %>% ggvis(~factor(cyl)) %>% layer_bars()

# Adapt the solution to the first challenge to just calculate the count values. No plotting!
mtcars %>% compute_count(~factor(cyl))
```

Section 6 - ggvis and dplyr

ggvis and group_by

The first line of code on the right uses the `group_by()` function from the `dplyr` package to plot two grouped smooth lines. `ggvis` plots a separate smooth line for each unique value of the `am` variable (0 = automatic transmission; 1 = manual transmission).

Since `group_by()` does not come with the `ggvis` package, it does not use the `~` syntax (although this may change in the future). You should just pass `group_by()` a variable name without quotes.

`group_by()` uses a grouping variable to organize a data set into several groups of observations. It places each observation into a group with other observations that have the same value of the grouping variable. In other words `group_by()` will create a separate group

for each unique value of the grouping variable. When `ggvis` encounters grouped data, it will draw a separate mark for each group of observations. But which mark corresponds to which group?
The challenges below will show you how to plot grouped data and how to make grouped graphs clearer by additionally mapping a property to the grouping variable.

Instructions

- Change the code to plot a unique smooth line for each value of the `cyl` variable. Note that you will also need to change the stroke property from `am` to `cyl`. Make sure to use categorical versions of the variables where needed.
- Refactor the code of the second graph such that it contains a separate density for each value of `cyl`.
- Copy and alter the solution to the second challenge to map the `fill` property to a categorical version of `cyl`. This addition clarifies which density corresponds to which group of observations.

```
# Both ggvis and dplyr are loaded into the workspace

# Change the code to plot a unique smooth line for each value of the cyl variable.
mtcars %>% group_by(cyl) %>% ggvis(~mpg, ~wt, stroke = ~cyl) %>% layer_smooths()

# Adapt the graph to contain a separate density for each value of cyl.
mtcars %>% group_by(cyl) %>% ggvis(~mpg, stroke=~cyl) %>% layer_densities()

# Copy and alter the solution to the second challenge to map the fill property to a categorical version of cyl.
mtcars %>% group_by(cyl) %>% ggvis(~mpg, stroke=~cyl) %>% layer_densities(fill=~factor(cyl))
```

group_by() versus interaction()

`group_by()` can also group data based on the interaction of two or more variables. To group based on the interaction of multiple variables, give `group_by()` multiple variable names, like this:

```
<data> %>% group_by(<var1>, <var2>, <var3>, ...)
```

`group_by` will create a separate group for each distinct combination of values within the grouping variables. `group_by()` does not change the raw values of the data set. The grouping information is saved as an attribute (e.g., metadata). You can remove the grouping information from a data set with `ungroup()` (e.g., `mtcars %>% ungroup()`).
You can also map *properties* to unique combinations of variables. To do this, use

the `interaction()` function. For example,

```
stroke = ~interaction(<var1>, <var2>, <var3>)
```

will map stroke to the unique combinations of `<var1>`, `<var2>`, and `<var3>`.

Instructions

- Alter the first graph in the editor so that it displays a separate density for each unique combination of `cyl` and `am`.
- Update the second graph to map `fill` to the unique combinations of the grouping variables.

```
# Alter the graph below: separate density for each unique combination of 'cyl' and 'am'.
mtcars %>% group_by(cyl, am) %>% ggvis(~mpg, fill = ~factor(cyl)) %>% layer_densities()

# Update the graph below to map `fill` to the unique combinations of the grouping variables.
mtcars %>% group_by(cyl, am) %>% ggvis(~mpg, fill = ~interaction(cyl, am)) %>% layer_densities()
```

Chaining is a virtue

You might have noticed that by now, the size of our of chain of operations is becoming considerable. Using the pipe becomes even more useful! Let's retake the solution to the previous exercise:

```
mtcars %>%
      group_by(cyl, am) %>%
      ggvis(~mpg, fill = ~interaction(cyl, am)) %>%
      layer_densities()
```

This call is exactly equivalent to the following piece of code that is very hard to read:

```
layer_densities(ggvis(group_by(mtcars, cyl, am), ~mpg, fill = ~interaction(cyl, am)))
```

The pipe efficiently solves the so-called *Dagwood sandwich problem* that drives your functions and arguments further and further apart when building richer plots.
However, you should always remember that the pipe is just a way to restructure your command rather than an actual operation. Which of the following statements is correct concerning the following line of code:

```
mtcars %>%
   group_by(am) %>%
   ggvis(~mpg, ~hp) %>%
   layer_smooths(stroke = ~factor(am)) %>%
```

```
layer_points(fill = ~factor(am))
```

- A) The first argument specified in the `ggvis()` function is `~mpg`.
- B) `layer_smooths(...)` and `layer_points(...)` can be safely interchanged.
- C) `ggvis` plots `hp` (gross horse power) and `mpg` (miles per gallon) on the x and y axis, respectively.
- D) `group_by(mtcars,am)` is the first argument of the `ggvis()` function.

Ans: B and D are ccorrect

Section 7 - Interactive plots

The basics of interactive plots

The first lines of code on the right make a basic interactive plot. The plot includes a select box that you can use to change the shape of the points in the plot. If you ran this code inside the RStudio IDE, you'd get an interactive plot, with visualizations that change on the fly. For the moment, the DataCamp editor only supports static plots, with the interactions removed. The dynamic versions of this code - as they would appear in RStudio - can be found on DataCamp's Shiny Server.

You can make your plots interactive by setting a property to the output of an input widget. `ggvis` comes with seven input widgets: `input_checkbox()`, `input_checkboxgroup()`, `input_numeric()`, `input_radiobuttons()`, `input_select()`, `input_slider()`, and `input_text()`. By default, each returns their current value as a number or character string.

Instructions

- Inspect the output of the first code chunk on the right. Visit the shiny app hosted at shiny.datacamp.com to see and experiment with the interactive plot by changing the point shape.
- The first chunk of code sets `shape` to a select box created with `input_select()`. Copy this code and add a `fill` property; set it equal to a select box that has the label `"Choose color:"` and provides the choices black, red, blue, and green. The resulting interactive plot should look like this.
- Add a group of radiobuttons to the last code chunk that controls the `fill` of the plot. Give the group the label `"Choose color:"`, and provide the choices black, red, blue, and green. You want to end up with this plot.

```
# Run this code and inspect the output. Follow the link in the instructions for the interactive version
faithful %>%
 ggvis(~waiting, ~eruptions, fillOpacity := 0.5,
      shape := input_select(label = "Choose shape:",
                        choices = c("circle", "square", "cross", "diamond",
                                    "triangle-up", "triangle-down"))) %>%
 layer_points()

# Copy the first code chunk and alter the code to make the fill property interactive using a select box
faithful %>%
 ggvis(~waiting, ~eruptions, fillOpacity := 0.5,
      fill:=input_select(label="Choose color:",
                         choices=c("black","red","blue","green")),
      shape := input_select(label = "Choose shape:",
                        choices = c("circle", "square", "cross", "diamond",
                                    "triangle-up", "triangle-down"))) %>%
   layer_points()

# Add radio buttons to control the fill of the plot
mtcars %>%
 ggvis(~mpg, ~wt,fill:=input_radiobuttons(choices=c("black","red","blue","green"),label="Choose
color:")) %>%
 layer_points()
```

Input widgets in more detail

Some input widgets provide choices for the user to select from. Others allow the user to provide their own input. For example, `input_text()` provides a text field for users to type input into. Instead of assigning `input_text()` a `choices` argument, you assign it a `value` argument: a character string to display when the plot first loads.

By default, input widgets return their values as character strings and numbers. To have a widget return its value as a variable name, you need to add the extra argument `map = as.name`.

For example, the text widget in the first challenge will pass the character string `"black"` to the `fill` argument, which is useful for *setting*. If we add `map = as.name` to the arguments of `input_text()`, the widget would return `~black` which is useful for *mapping* (or would be if black were a real variable in `mtcars`):

Instructions

- Change the radiobuttons widget in the first command on the right to a text widget that displays the initial value `"black"`. The wanted resulting plot can be found here.
- In the second graph on the right, *map* fill to a select box that returns variable names. The

select box should use the label `"Choose fill variable:"` and should offer the choices created by `names(mtcars)`, as shownhere. Use the `map` argument inside `input_select()` to *map* the selection instead of *setting* it.

```
# Change the radiobuttons widget to a text widget
mtcars %>%
 ggvis(~mpg, ~wt,
      fill := input_text("black",label = "Choose color:",
                         )) %>%
 layer_points()

# Map the fill property to a select box that returns variable names
mtcars %>%
 ggvis(~mpg, ~wt, fill=input_select(choices=names(mtcars),label="Choose fill variable:",map=as.name))
%>%
 layer_points()
```

Control parameters and values

The previous exercises all manipulated properties of the `ggvis` plots, such as the `shape` and `fill` of points in scatterplots. As you will recall from earlier exercises, `ggvis` often needs additional parameters to build the correct graphs. You can also use widgets to control these parameters. Typically, you want to use the `input_numeric()` and `input_slider()` widgets to set numerical parameters.

Instructions

- Change the first graph coded on the right to map the binwidth to a numeric field that uses the label `"Choose a binwidth:"` and has a starting value of 1. The resulting plot should look like this.
- Change the second graph to map binwidth to a slider bar that uses the label `"Choose a binwidth:"` and has a `min` value of 1 and a `max` value of 20. `input_slider` will place the initial value at `(min + max) / 2`. The plot you want to create can be found here.

```
# Map the bindwidth to a numeric field ("Choose a binwidth:")
mtcars %>%
 ggvis(~mpg) %>%
 layer_histograms(width = input_numeric(label="Choose a binwidth:",value=1))

# Map the binwidth to a slider bar ("Choose a binwidth:") with the correct specifications
mtcars %>%
```

```
ggvis(~mpg) %>%
layer_histograms(width = input_slider(label="Choose a binwidth:",min=1, max=20))
```

Section 8 - Multi-layered plots

Multi-layered plots and their properties

You can create multi-layered plots by adding additional layers to a graph with the `%>%` syntax.

If you set or map a property inside `ggvis()` it will be applied *globally* , every layer in the graph will use the property. If you set or map a property inside a `layer_<marks>()` function it will be applied *locally*: only the layer created by the function will use the property. Where applicable, local properties will override global properties.

Instructions

- Add a layer of points to the first graph on the right.
- Starting from the solution to the first instruction, build a second graph in which only the lines layer uses a skyblue stroke.
- Global properties can cause trouble when you use multiple layers. For example, the code for the third graph on the right causes an error since it applies the `shape` property to `layer_lines`, which does not use the `shape` property. Rewrite the code so that only the points layer uses the `shape` property.
- If you like, you can define every property at the local level, including the x and y mappings. However, your code would not be very concise. As an example, consider the last graph that is coded on the right. Rewrite it to use global properties where possible. Make the code as concise as possible, but make sure that it still plots the same graph.

```
# Add a layer of points to the graph below.
pressure %>%
  ggvis(~temperature, ~pressure, stroke := "skyblue") %>%
  layer_lines() %>%
  layer_points()

# Copy and adapt the solution to the first instruction below so that only the lines layer uses a skyblue
stroke.
pressure %>%
    ggvis(~temperature, ~pressure) %>%
    layer_lines(stroke:="skyblue") %>%
    layer_points()

# Rewrite the code below so that only the points layer uses the shape property.
pressure %>%
```

```
 ggvis(~temperature, ~pressure) %>%
 layer_lines(stroke := "skyblue") %>%
 layer_points(shape:="triangle-up")

# Refactor the code for the graph below to make it as concise as possible
pressure %>%
   ggvis(x=~temperature,y=~pressure, stroke:="skyblue", strokeOpacity:=0.5, strokeWidth:=5) %>%
layer_lines() %>% layer_points(fill=~temperature, shape:="triangle-up", size:=300)
```

There is no limit on the number of layers!

`layer_model_predictions()` plots the prediction line of a model fitted to the data. It is similar to `layer_smooths()`, but you can extend it to more models than just the `"loess"` or `"gam"` model.

`layer_model_predictions()` takes a parameter named `model`; it should be set to a character string that contains the name of an R model function. `layer_model_predictions()` will use this function to generate the model predictions. So for example, you could draw the model line of a linear model with

```
layer_model_predictions(model = "lm").
```

Notice that `model` is a parameter, not a property. This means that you do not need to worry about setting vs. mapping. You can always set parameters with the equal sign, `=`.

Instructions

Create a scatterplot of the `pressure` data set that has the `temperature` variable on the x axis and the `pressure` variable on the y axis. Connect the points with a black line that has 50% opacity. Then add a linear model line to the data that is navy in color. Then add a smooth line that is skyblue in color.

```
# Create a graph containing a scatterplot, a linear model and a smooth line.
pressure %>%
 ggvis(~temperature, ~pressure) %>%
 layer_lines(opacity := 0.5) %>%
 layer_points() %>%
 layer_model_predictions(model = "lm", stroke := "navy") %>%
 layer_smooths(stroke := "skyblue")
```

Taking local and global to the next level

In the first exercise set of this section, we mentioned that you can define local properties and

global properties. Also, local properties override global properties when applicable. When you copy the chunk of code below to the console and run it, you will see that it generates an error. Can you see why?

```
pressure %>%
    ggvis(~temperature, ~pressure, shape := "circle") %>%
    layer_lines(stroke := "orange", strokeDash := 5, strokeWidth := 5) %>%
    layer_points(size := 100, fill := "lightgreen") %>%
    layer_smooths(stroke := "darkred")
```

What has to be done to
(1) remove the error from the code and
(2) set the `stroke` of the point marks to `"darkred"` like the smooth lines mark in a way that your code is as concise as possible?
Feel free to experiment in the console!

Ans: (1) Move the `shape := circle` property inside `layer_points()` and (2) move `stroke := "darkred"` from inside `layer_smooths()` to `ggvis()`.

Section 9 - Axes and Legends

Axes

As Garrett explained in the video, axes help you to customize the plots you create with `ggvis`. `add_axis()` allows you to change the titles, tick schemes and positions of your axes. The example code below clarifies:

```
add_axis("x",
         title = "x axis title",
         values = c(1,2,3),
         subdivide = 5,
         orient = "top")
```

The `title` argument is rather straightforward, as it simply sets the title of the axis you specified in the first argument.
You can use the `values` argument of `add_axis` to determine where labelled tick marks will appear on each axis. You can use the `subdivide` argument to insert unlabelled tick marks between the labelled tick marks on an axis.
To control where an axis appears, use the `orient` argument. For example, the above code makes the x axis appear on the `"top"` (and not on the `"bottom"`) side of the graph. Similarly, you can have the y axis appear on the `"left"` or `"right"` side of the graph.

Giorgos Mintzopoulos <datacamp.com>

Instructions

- The code for the first graph on the right changes the title of the Y axis to “Duration of eruption (m)”. Add to the code to change the title of the x axis to “Time since previous eruption (m)”.
- The second code chunk places a labelled tick mark at each integer on the y axis and then inserts nine subdividing tickmarks between each integer. Add to the code to place a labelled tick mark at 50, 60, 70, 80, 90 on the x axis. Place nine subdividing tick marks between each labelled tick mark.
- Change the sample code for the last graph: move the y axis to the right side of the plot. Move the x axis to the top of the plot.

```
# add the title of the x axis: "Time since previous eruption (m)"
faithful %>%
  ggvis(~waiting, ~eruptions) %>%
  layer_points() %>%
  add_axis("y", title = "Duration of eruption (m)") %>%
  add_axis("x",title="Time since previous eruption (m)")

# Add to the code to place a labelled tick mark at 50, 60, 70, 80, 90 on the x axis.
faithful %>%
  ggvis(~waiting, ~eruptions) %>%
  layer_points() %>%
  add_axis("y", title = "Duration of eruption (m)",
         values = c(2, 3, 4, 5), subdivide = 9) %>%
  add_axis("x", title = "Time since previous eruption (m)",
           values=c(50,60,70,80,90),subdivide=9)

# Change the code below to change the axes' locations
faithful %>%
  ggvis(~waiting, ~eruptions) %>%
  layer_points() %>%
  add_axis("y", title = "Duration of eruption (m)",
         values = c(2, 3, 4, 5), subdivide = 9,orient="right") %>%
  add_axis("x", title = "Time since previous eruption (m)",
           values=c(50,60,70,80,90),subdivide=9,orient="top")
```

Giorgos Mintzopoulos <datacamp.com>

Legends

`add_legend()` works similarly to `add_axis()`, except that it alters the legend of a plot. Instead of specifying which axis to change, you have to specify the property you want to add to the legend. For example

```
pressure %>%
  ggvis(~temperature, ~pressure, fill = ~pressure) %>%
  layer_points() %>%
  add_legend("fill", title = "~ pressure")
```

adds a legend to the graph that gives more information about the color of the points in the scatterplot.

`ggvis` will create a separate legend for each property that you use. Often the results can be confusing. For example, the second chunk of code on the right creates three separate legends. One for fill, one for shape, and one for size - and it draws them each on top of each other.
You can use `add_legend()` to combine multiple legends into a single legend. To do this, give `add_legend()` a vector of property names as its first argument. For example, to combine a stroke legend with an opacity legend, call `add_legend(c("stroke", "opacity"))`. Similarly, you can specify the `values` property inside `add_legend()` to explicitly set the visible legend values.

Instructions

- Use `add_legend()` to change the title of the legend in the first plot on the right to "~ duration (m)" and to orient the legend to the left side of the graph.
- Use `add_legend()` to combine the legends in the second plot. Also set the legend title to "~ duration (m)" and specify that only the values 2, 3, 4, and 5 should receive a labelled symbol.

```
# Add a legend to the plot below: use the correct title and orientation
faithful %>%
  ggvis(~waiting, ~eruptions, opacity := 0.6,
        fill = ~factor(round(eruptions))) %>%
  layer_points() %>%
  add_legend("fill",title="~duration (m)", orient="left")

# Use add_legend() to combine the legends in the plot below. Adjust its properties as instructed.
faithful %>%
  ggvis(~waiting, ~eruptions, opacity := 0.6,
        fill = ~factor(round(eruptions)), shape = ~factor(round(eruptions)),
        size = ~round(eruptions))  %>%
  layer_points() %>%
  add_legend(c("fill","shape","size"),title="~duration (m)", values=c(2,3,4,5))
```

Giorgos Mintzopoulos <datacamp.com>

Section 10 - Customize property mappings

Scale types

You can change the color scheme of a `ggvis` plot by adding a new scale to map a data set variable to fill colors. The first chunk of code on the right creates a new scale that will map the numeric `disp` variable to the `fill` property. The scale will create color output that ranges from red to yellow. Copy the code to console, run it and be amazed.

`ggvis` provides several different functions for creating scales: `scale_datetime()`, `scale_logical()`, `scale_nominal()`, `scale_numeric()`, `scale_singular()`. Each maps a different type of data input to the visual properties that `ggvis` uses. For example, the first two challenges below require `scale_numeric()` because the code maps a numeric variable to a visual property.

Instructions

- Add to the first chunk of code to make the stroke color range from “darkred” to “orange”.
- Change the second graph to make the fill colors range from green to beige.
- The third code chunk on the right maps a categorical (e.g., nominal variable) to `fill`. Create a scale that will map `factor(cyl)` to a new range of colors: purple, blue, and green.
  Since `factor(cyl)` has three unique values, the range of your scale will need three unique color names.

```
# Add to the code below to make the stroke color range from "darkred" to "orange".
mtcars %>%
  ggvis(~wt, ~mpg, fill = ~disp, stroke = ~disp, strokeWidth := 2) %>%
  layer_points() %>%
  scale_numeric("fill", range = c("red", "yellow")) %>%
  scale_numeric("stroke",range=c("darkred","orange"))

# Change the graph below to make the fill colors range from green to beige.
mtcars %>% ggvis(~wt, ~mpg, fill = ~hp) %>%
  layer_points() %>%
  scale_numeric("fill",range=c("green","beige"))

# Create a scale that will map `factor(cyl)` to a new range of colors: purple, blue, and green.
mtcars %>% ggvis(~wt, ~mpg, fill = ~factor(cyl)) %>%
  layer_points() %>%
  scale_nominal("fill",range=c("purple","blue","green"))
```

Adjust any visual property

You can adjust any visual property in your graph with a scale (not just color). Let's look at

another property that you may frequently want to adjust.

Often when you map a variable to `opacity` some data points will end up so transparent that they are hard to see, as in the first plot in the editor on the right.

Just as you can change the range of visual values that your scales produce, you can also change the domain of data values that they consider. For example, you can expand the domain of the x and y scales to zoom out on your plot. The second plot on the right will expand the y axis to cover data values from 0 to the largest y value in the data set.

Instructions

- Consider the first code chunk in the editor. Add a scale that limits the range of opacity from 0.2 to 1. Be sure to consider whether `hp` is a datetime, logical, nominal, numeric, or singular value.
- Read the code for the second graph carefully. Add a second scale that will expand the x axis to cover data values from 0 to 6.

```
# Add a scale that limits the range of opacity from 0.2 to 1.
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~factor(cyl), opacity = ~hp) %>%
  layer_points() %>%
  scale_numeric("opacity", range=c(0.2,1))

# Add a second scale that will expand the x axis to cover data values from 0 to 6.
mtcars %>% ggvis(~wt, ~mpg, fill = ~disp) %>%
  layer_points() %>%
  scale_numeric("y", domain = c(0, NA)) %>%
  scale_numeric("x", domain=c(0,6))
```

"=" versus ":="

Scales help explain the difference between `=` and `:=`. Variables tend to contain values in the data space, things such as numbers measured in various units, datetimes, and so on. But properties need visual values, things such as numbers measured in pixels, colors, opacity levels, and so on.

Whenever you use `=` to *map* a variable to a property, `ggvis` will use a scale to transform the variable values into visual values. Whenever you *set* a value (or variable) to a property with `:=`, `ggvis` will pass the value on as is, without transforming it. For example, the code below passes `"red"` straight through to the visual space to create a red fill:

```
mtcars %>%
  ggvis(x = ~wt, y = ~mpg,
        fill := "red") %>%
  layer_points()
```

This can work nicely if the value passed through makes sense in the visual space, but it can

have unfortunate consequences if the value does not.

Instructions

The code on the right adds a new column to `mtcars` that contains valid color names. If you map `fill` to the color column, `ggvis` transforms the color names into a new set of colors in the visual space. What would happen if you *set* the fill value to the color variable? Make a prediction and then change and run the code to find out.

```
# Set the fill value to the color variable instead of mapping it, and see what happens
mtcars$color <- c("red", "teal", "#cccccc", "tan")
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill := ~color) %>% layer_points()
```