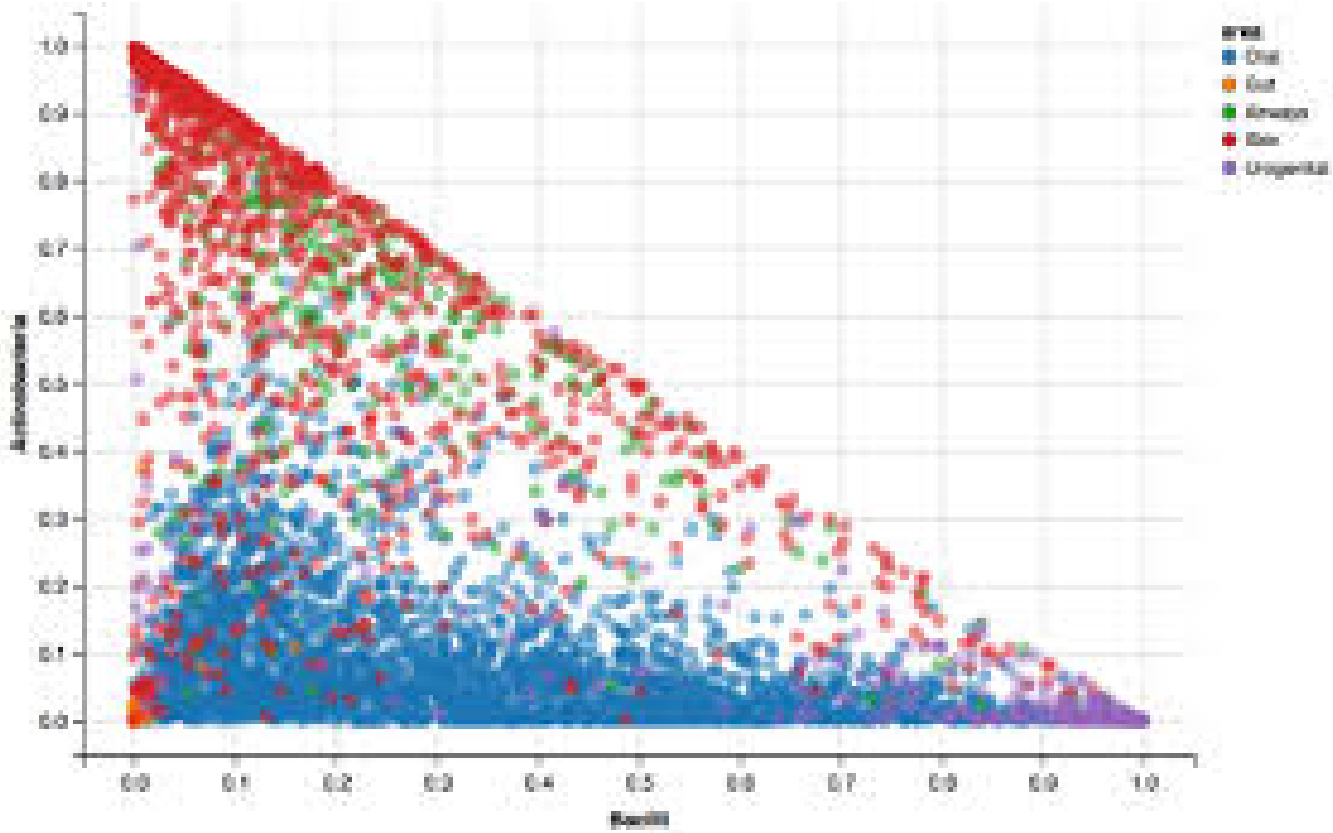
Introduction to ggvis



Overview

- Getting started with ggvis
- The magrittr `R` package and the %>% operator
- Common plot functions and managing aesthetics
- Layers
- Interactivity

Resources

- R (version 3.2)
- RStudio
- ggvis (version 0.4.1)
- Version Particularly Important
- Dataset : nycflights13 (R package)

ggvis: Interactive Grammar of Graphics

An implementation of an interactive grammar of graphics, taking the best parts of ggplot2, combining them with shiny's reactive framework and drawing web graphics using vega.

Version:	0.4.1
Depends:	R (≥ 3.0)
Imports:	assertthat, jsonlite (≥ 0.9.11), shiny (≥ 0.11.1), magrittr, dplyr (≥ 0.3), lazyeval, htmltools (≥ 0.2.4)
Suggests:	MASS, mgcv, lubridate, testthat (≥ 0.8.1), knitr (≥ 1.6), rmarkdown
Published:	2015-03-12
Author:	Winston Chang [aut, cre], Hadley Wickham [aut], RStudio

Using ggvis - a word of warning!

```
> require(ggvis)
Loading required package: ggvis
The ggvis API is currently rapidly evolving. We strongly
recommend that you do not rely on this for production,
but feel free to explore. If you encounter a clear bug,
please file a minimal reproducible example at
https://github.com/rstudio/ggvis/issues. For questions
and other discussion, please use
https://groups.google.com/group/ggvis.
```

Figure:

Downloads:

Reference manual:	ggvis.pdf
Vignettes:	Axes and legends
	ggvis cookbook
	Data hierarchy
	ggvis vs ggplot2
	ggvis basics
	Interactivity
	Marks
	ggvis basics
	Properties and scales
	ggvis vs vega/d3

The Data

- All examples will be using tubeData
- London Tube performance Data from the TFL website
- The original data can be found on

`http://data.london.gov.uk/dataset/tube-networkperformance-c`

Main features of ggplot2

- Create graphics using `qplot` or `ggplot`
- Add layers to an existing plot using "+"
- Change aesthetics by variables in the data
- Control the type of plot using `geoms`
- Panel by variables using the `facet_*` functions

Tube Data with ggplot2

```
library(ggplot2)
qplot(Month, Excess, data = tubeData) +
   geom_smooth(method = "lm",
     col = "red") +
  facet_wrap(~Line) +
  theme_bw()
```

The geoms

ggplot2 includes a variety of geoms for controlling the type of plot we create

```
> grep("^geom", objects("package:ggplot2"), value = TRUE)
 [1] "geom_abline"     "geom_area"       "geom_bar"        "geom_bin2d"
 [5] "geom_blank"      "geom_boxplot"    "geom_contour"    geom_crossbar"
 [9] "geom_density"    "geom_density2d"  "geom_dotplot"    "geom_errorbar"
[13] "geom_errorbarh"  "geom_freqpoly"   "geom_hex"        "geom_histogram"
[17] "geom_hline"      "geom_jitter"     "geom_line"       "geom_linerange"
[21] "geom_map"        "geom_path"       "geom_point"      "geom_pointrange"
[25] "geom_polygon"    "geom_quantile"   "geom_raster"     "geom_rect"
[29] "geom_ribbon"     "geom_rug"        "geom_segment"    "geom_smooth"
[33] "geom_step"       "geom_text"       "geom_tile"       "geom_violin"
[37] "geom_vline"
```

Facetting

- We can panel graphics based on variables in the data using facets
- `facet_wrap` and `facet_grid` add panels as layers

Scales and Themes

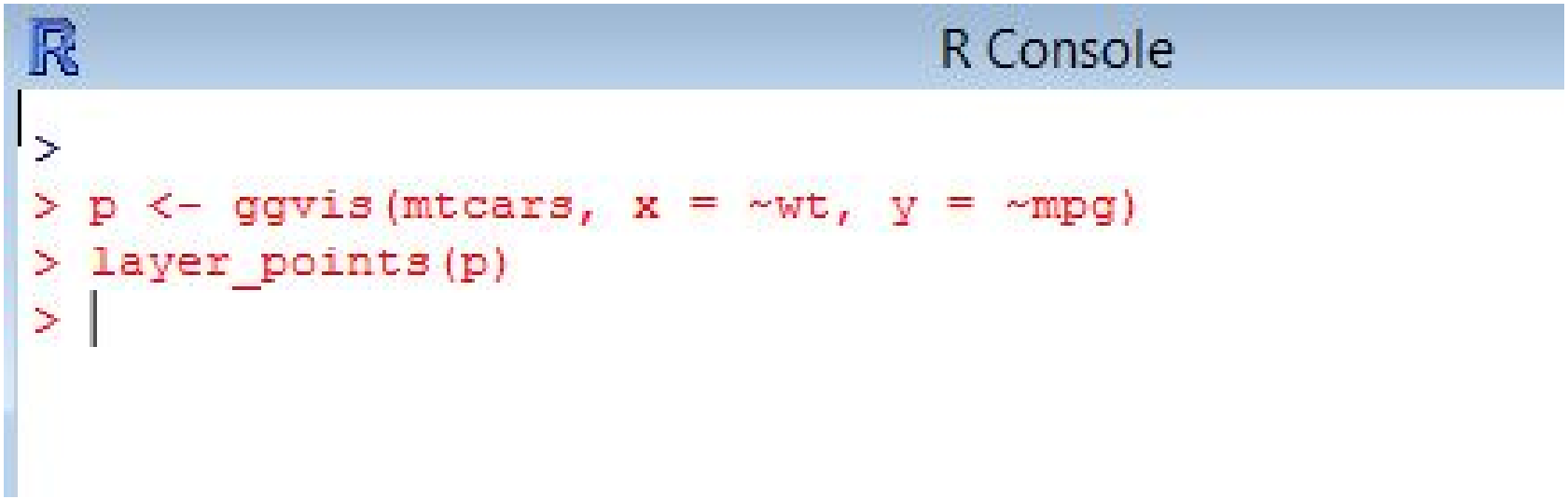
- ggplot2 provides a large number of scale functions to control aspects of a graphic including axes and legends
- theme functions allow us to control the overall style of the graphic

Introduction to ggvis

- Installing ggvis
- Our first plot
- Useful things to know (magrittr)

Creating a Basic Plot

- To create a plot object we use the function `ggvis()`
- When we refer to variables in the data we use the '$\sim$' symbol before the name, i.e. $\sim$ `Ozone`
- We need to use a layer function, such as `layer_points`, to plot the object.

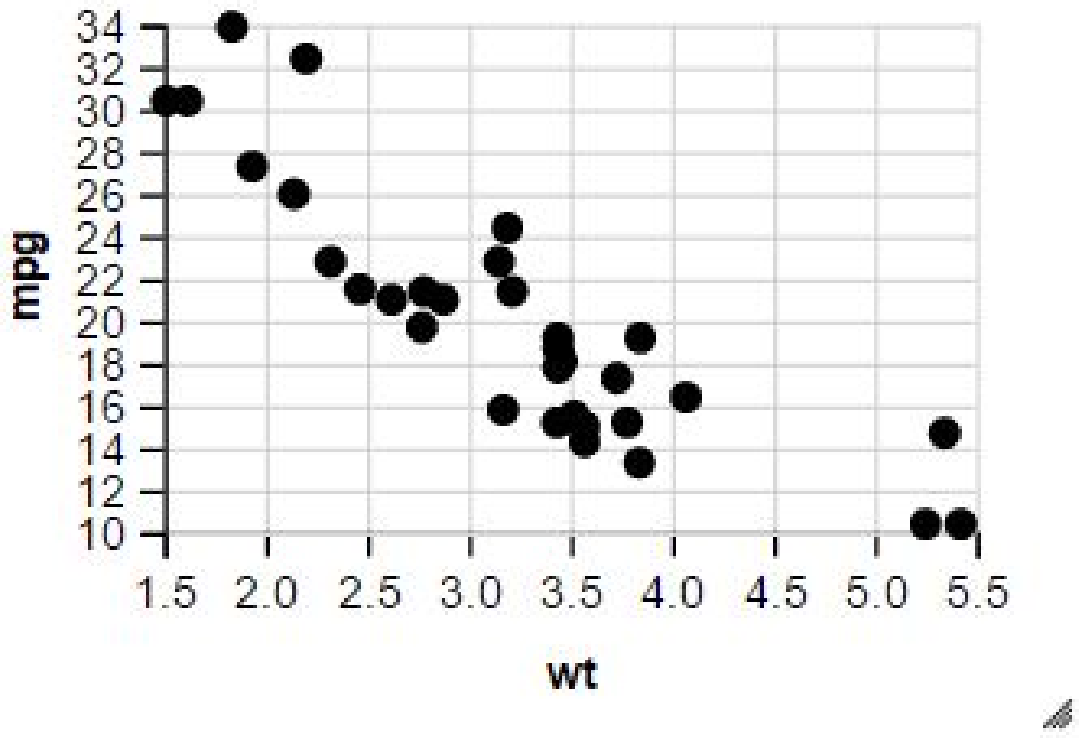



```
p <- ggvis(mtcars, x = ~wt, y = ~mpg)
layer_points(p)
```

(deprecated code?- Watch out for this)

A basic scatter plot:

```
# qvis(mtcars, ~wt, ~mpg)
ggvis(mtcars, props(x = ~wt, y = ~mpg)) + mark_point()
```



Web Graphics

- You will notice that this plot opens in your **web browser** (unless youre using RStudio).
- Thats because all **ggvis** graphics are web graphics, and need to be shown in the web browser.

- RStudio includes a built-in browser so it can show you the plots directly.

Code Legibility

Quoting Hadley Wickham

- All ggvis functions take the visualisation as the first argument and return a modified visualisation.
- This seems a little bit awkward.
- Either you have to create temporary variables and modify them, or you have to use a lot of parentheses:

```
layer_points(ggvis(mtcars, x = ~wt, y = ~mpg))
```

Viewing ggvis graphics

- ggvis uses **Vega** to render graphics in a web browser
- In RStudio the default it to use the "Viewer" pane
- From the web browser we can download SVG or png version of our graphics

ggvis and Vega/D3

- While ggvis is built on top of **vega**, which in turn borrows many ideas from **d3**, it is designed more for data exploration than data presentation.
- This means that ggvis makes many more assumptions about what youre trying to do: this allows it to be much more concise, at some cost of generality.

ggvis and Vega/D3

The main difference to vega is that ggvis provides a tree like structure allowing properties and data to be specified once and then inherited by children.

ggvis and Vega/D3

- Vega plays a similar role to ggvis that grid does to ggplot2. That means that you shouldnt have to know anything about vega to use ggvis effectively, and you shouldnt have to refer to the vega docs to solve common problems.
- However, some knowledge of how vega works is likely to be necessary when you start doing more complex layouts or when you start pushing the limits of the ggvis DSL.

THE %>% OPERATOR


%>%
magrittr
Ceci n'est pas un pipe.

The %>% operator

- From **magrittr** package.
- Used extensively in **dplyr**.
- %>% is a piping operator, and can be verbalised as "*then*".
- It takes the output of the left side, and uses it as the first argument of the function on the right side.

magrittr : the %>% operator

```
subset(mtcars, cyl == 6, c(mpg, wt))

mtcars %>% subset(cyl == 6, c(mpg, wt))
```

magrittr : the %>% operator

```
summary(subset(mtcars, cyl == 6,
               c(mpg, wt)), digits=2)

mtcars %>%
    subset(cyl == 6, c(mpg, wt)) %>%
    summary(digits=2)
```

magrittr : the %>% operator

```
mtcars %>%
    subset(cyl == 6, c(mpg, wt)) %>%
        summary(digits=2)
```

- Get the mtcars data set
- **Then** subset it like this
- **Then** get the summary, with this setting

magrittr : the %>% operator

- You can use the %>% operator with any `R` functions.
- The rules are simple: the object on the left hand side is passed as the first argument to the function on the right hand side. So:

my.data %>% `my.function` is the same as
`my.function(my.data)` my.data %>%
`my.function(arg=value)` is the same as
`my.function(my.data, arg=value)`

The % > % Operator

- ggvis makes use of the % > % operator from the package magrittr
- This allows us to layer up graphics in the same way we would with ggplot2

The %>% Operator

Tube Data Example
(*Dr. Aimee Gott, Mango Solutions*)

```
> tubeData$Excess %>%  tapply(tubeData$Line, mean)

# Bakerloo                5.047714
# Central                 5.998667
# Circle & HamDistrict    7.166095
```

magrittr : the %>% operator

%>% in ggvis

- With ggvis we pass "`ggvis`" objects
- We create the initial object by passing data to `ggvis()`
- All other functions expect a ggvis object as the first argument and return a ggvis object

Recall:

The **magrittr** package allows you to rewrite the previous function call as:

```
mtcars %>%
        ggvis(x = ~wt, y = ~mpg) %>%
        layer_points()
```

Pipe operator must be at the end of line, if using multiple lines

```
mtcars %>%
ggvis(x = ~wt, y = ~mpg) %>%
layer_points()
```

This following code LOOKS neat, but doesn't work.

```
mtcars
    %>% ggvis(x = ~wt, y = ~mpg)
    %>% layer_points()
```

Data Visualization with ggvis

- This style of programming (i.e. using the pipe operator) also allows gives you a lot of power when you start creating a lot of power.
- Also it allows you to seemlessly intermingle **ggvis** and **dplyr** code (*Next Slide*).

Data Visualization with ggvis

```
library(dplyr)
# convert engine displacment to litres

mtcars %>%
  ggvis(x = ~mpg, y = ~disp) %>%
      mutate(disp = disp / 61.0237) %>%
         layer_points()
```

Calling Formulas

- The format of the visual properties needs a little explanation.
- We use $\sim$ before the variable name to indicate that we dont want to literally use the value of the **mpg** variable (which doesnt exist), but instead we want we want to use the **mpg** variable inside in the dataset.
- This is a common pattern in **ggvis**: well always use formulas to refer to variables inside the dataset.

Data Visualization with ggvis

The first two arguments to `ggvis()` are usually the position, so by convention you can drop `x` and `y`:

```
mtcars   %>%
  ggvis(~mpg, ~disp) %>%
  layer_points()
```

(x for mpg, y for displacement)

All the mtcars variables

```
> names(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat"
 [6] "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
>
```

Data Visualization with ggvis

You can add more variables to the plot by mapping them to other visual properties like **fill, stroke, size** and **shape**.

```
mtcars %>%
  ggvis(~mpg, ~disp, stroke = ~vs) %>%
     layer_points()
```

The "fill" property

```
mtcars %>%
   ggvis(~mpg, ~disp, fill = ~vs) %>%
      layer_points()
```

The "size" property

```
mtcars %>%
    ggvis(~mpg, ~disp, size = ~vs) %>%
        layer_points()
```

The "shape" property

```
mtcars %>%
    ggvis(~mpg, ~disp,
          shape = ~factor(cyl)) %>%
    layer_points()
```

The ":=" operator

- If you want to make the points a fixed colour or size, you need to use := instead of =.
- The := operator means to use a raw, unscaled value.
- This seems like something that `ggvis()` should be able to figure out by itself, but making it explicit allows you to create some useful plots that you couldnt otherwise.

Data Visualization with ggvis

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

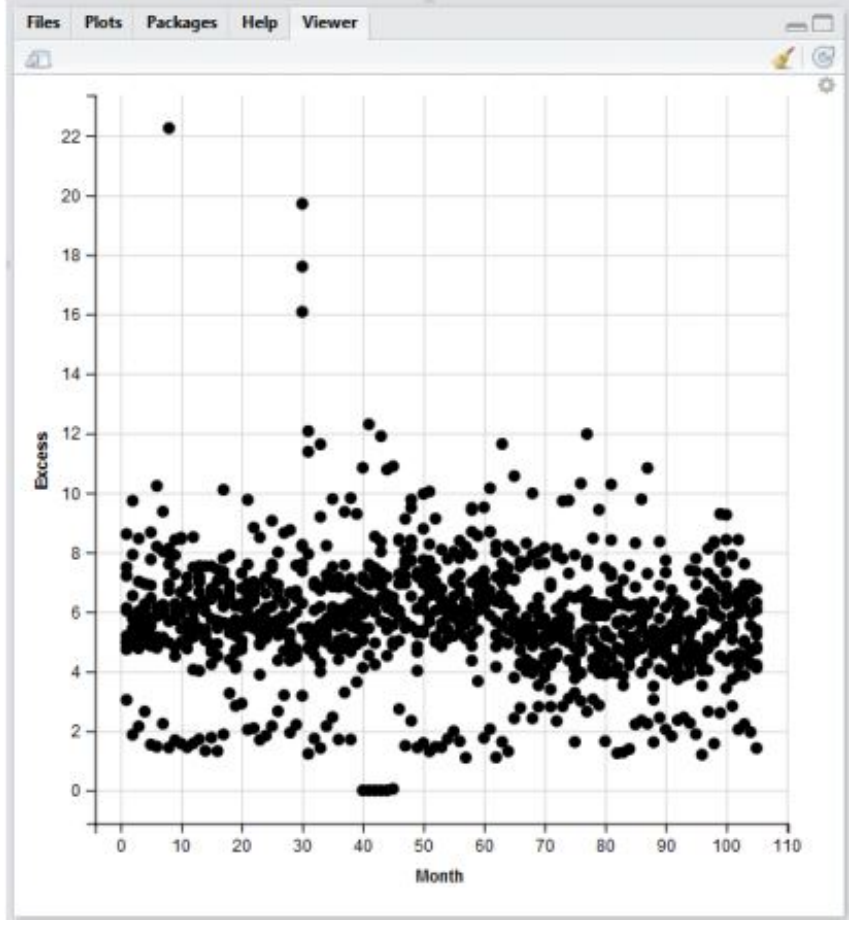
Data Visualization with ggvis

```
mtcars   %>%
     ggvis(~wt, ~mpg,
           size := 300,
           opacity := 0.4)    %>%
     layer_points()
```

Data Visualization with ggvis

```
mtcars  %>%
    ggvis(~wt, ~mpg,
        shape := "cross")   %>%
    layer_points()
```

```
tubeData %>%
    ggvis(x = ~Month, y = ~Excess) %>%
        layer_points()
```



Aesthetics for ggvis

As with all graphics there are a number of aesthetics we can set

- stroke
- fill
- size
- opacity

Changing Aesthetics based on variables

- In ggvis we map a variable to a property using =
- We have to remember to use the "$\sim$" with all variable names
- `fill = ~Line` would set the fill based on the Line variable

```
tubeData %>%
    ggvis(x = ~Month, y = ~Excess,
          fill = ~Type) %>%
       layer_points()
```

Setting property values

- When we set a property based on a value we use `":="`
- `fill := "red"` would set the fill to red

```
tubeData %>%
    ggvis(x = ~Month, y = ~Excess,
          fill := "orange",
          opacity := 0.6) %>%
        layer_points()
```

Exercise

- Create a plot of mpg against wt using the mtcars data
- Update the plot to colour by the cylinder variable, ensure that the points are coloured by distinct colours rather than on a scale
- Update the plotting symbol to be triangles.

Controlling axis and legends

- We can control the axes using the `add_axis` function
- This controls axis labels, tick marks and even grid lines

```
add_axis("x", title = "Month")
```

Controlling axis and legends

The `add_legend` and `hide_legend` functions allow us to control if we see a legend and where it appears

```
hide_legend("fill")
add_legend(c("fill", shape))
```

Scales

ggvis has fewer scale functions than in ggplot2 but control much more

```
> grep("^scale", objects("package:ggvis"), v
[1] "scale_datetime" "scale_logical" "scale_
[5] "scale_ordinal" "scale_singular" "scaled
```

ggvis VS ggplot2 : How are they similar?

- We can layer graphics in a similar fashion
- Aesthetics can be set based on variables in the data
- We can control the type of plot with specific functions

ggvis vs ggplot2 : How are they different?

From point of view of **ggvis**

- Only one main plot function to work with as opposed to two
- Layering is done using $\%>\%$ rather than $+$
- Fewer scale functions
- Much functionality is not yet available in **ggvis** e.g. facetting

Which should I use?

- For static graphics: **ggplot2**
- For interactive graphics: **ggvis**
- **WARNING:** If you are using ggvis remember it's still being actively developed and may change in structure and functionality

Changing the plot type

ggvis Layers

- In ggplot2 you use **geom** functions to determine the type of plot that you create
- In ggvis you can use `layer` functions
- N.B. Not all geoms are currently available as layers

Layers

So far, you seen two layer functions: `layer_points()` and `layer_histograms()`. There are many other layers, and they can be roughly categorised into two types:

- Simple, which include primitives like points, lines and rectangles.
- Compound, which combine data transformations with one or more simple layers.

All layer functions use the plural, not the singular.

> *Think the verb, not the noun: Im going to layer some points onto my plot.*

Layers

Function	Description
layer_points	Adds data as points
layer_histograms	Adds data as a histogram
layer_boxplots	Draws as a boxplot
layer_lines	Adds data as lines
layer_smooths	Adds a smoothing line
layer_paths	Joins data as a single path
layer_text	Adds text
layer_model_predictions	Adds lines for model predictions, such as lm

Layers

Simple layers

There are five simple layers:

1. **Points** -`layer_points`
2. **Paths** and **polygons**, - `layer_paths()`.
3. **Filled Areas** - `layer_ribbons()`
4. **Rectangles** - `layer_rects()`
5. **Text** - `layer_text()`

1. **Points,** `layer_points()`

properties: `x`, `y`, `shape`, `stroke`, `fill`, `strokeOpacity`, `fillOpacity`, and `opacity`.

```
mtcars  %>%
   ggvis(~wt, ~mpg) %>%
      layer_points()
```

2. Paths and polygons, layer_paths().

```
df <- data.frame(x = 1:10,
        y = runif(10))

df %>%
    ggvis(~x, ~y) %>%
    layer_paths()
```

Layers

If you supply a `fill`, youll get a polygon

```
t <- seq(0, 2 * pi, length = 100)
df <- data.frame(x = sin(t), y = cos(t))

df %>%
    ggvis(~x, ~y) %>%
    layer_paths(fill := "red")
```

3. **Filled areas,** `layer_ribbons()`

Use properties y and y2 to control the extent of the area.

```
df <- data.frame(x = 1:10,
     y = runif(10))
df %>%
   ggvis(~x, ~y) %>%
   layer_ribbons()
```

Layers

```
df %>% ggvis(~x, ~y + 0.1,
             y2 = ~y - 0.1) %>%
       layer_ribbons()
```

Layers

4. Rectangles, `layer_rects()`.
The location and size of the rectangle is controlled by the x, x2, y and y2 properties.

```
set.seed(1014)
df <- data.frame(x1 = runif(5), x2 = runif(5),
         y1 = runif(5), y2 = runif(5))

df %>% ggvis(~x1, ~y1,
           x2 = ~x2, y2 = ~y2,
           fillOpacity := 0.1)  %>%
       layer_rects()
```

Layers

5. Text, layer_text()..

The `text` layer has many new options to control the apperance of the text:

- `text` (the label),
- `dx` and `dy` (margin in pixels between text and anchor point),
- `angle` (rotate the text),
- `font` (font name) and fontSize (size in pixels),
- `fontWeight` (e.g. bold or normal),
- `fontStyle` (e.g. italic or normal.)

Layers

```
df <- data.frame(x = 3:1,
                 y = c(1, 3, 2),
                 label = c("a", "b", "c"))

df %>% ggvis(~x, ~y, text := ~label)
   %>% layer_text()
```

Layers

```
df %>%
  ggvis(~x, ~y, text := ~label) %>%
  layer_text(fontSize := 50)
```

Layers

```
df %>%
    ggvis(~x, ~y, text := ~label) %>%
    layer_text(angle := 45)
```

Compound layers

The four most common compound layers are:

1. `layer_paths()`
2. `layer_histograms()`
3. `layer_polygons()`
4. `layer_smooths()`

Layers

`layer_lines()` which automatically orders by the x variable:

```
t <- seq(0, 2 * pi, length = 20)
df <- data.frame(x = sin(t), y = cos(t))

df %>%
   ggvis(~x, ~y) %>%
   layer_paths()
```

Compound layers

```
df %>%
   ggvis(~x, ~y) %>%
   layer_lines()
```

Layers

`layer_lines()` is equivalent to `arrange()` + `layer_paths()`:

```
df %>%
   ggvis(~x, ~y) %>%
   arrange(x) %>%
   layer_paths()
```

Layers

`layer_histograms()` and `layer_freqpolys()`

- `layer_histograms()` and `layer_freqpolys()` which allows you to explore the distribution of continuous.
- Both layers first bin the data with `compute_bin()` then display the results with either rects or lines.

Layers

```
mtcars  %>%
    ggvis(~mpg) %>%
    layer_histograms()

# Guessing width = 1
# range / 24
```

```
# Or equivalently
binned <- mtcars %>% compute_bin(~mpg)

#  Guessing width = 1
#  range / 24
binned   %>% ggvis(x = ~xmin_,
         x2 = ~xmax_,
         y2 = 0, y = ~count_,
         fill := "black") %>%
  layer_rects()
```

Compound Layers

`layer_smooths()` fits a smooth model to the data, and displays predictions with a line.
Its used to highlight the trend in noisy data:

```
mtcars   %>%
   ggvis(~wt, ~mpg) %>%
   layer_smooths()
```

You can control the degree of *wiggliness* with the span parameter:

```
span <- input_slider(0.2, 1, value = 0.75)
mtcars  %>%
  ggvis(~wt, ~mpg) %>%
  layer_smooths(span = span)
```

Vignette

- You can learn more about layers in the **layers** vignette.

MAKING PLOTS INTERACTIVE

As well as mapping visual properties to variables or setting them to specific values, you can also connect them to interactive controls.

Basic interactivity

Basic Interactivity

- The most basic interactivity we can add is "hover over" changes
- We can change properties by using `property.hover` arguments `fill.hover := "red"`

Basic interactivity

```
tubeData %>%
  ggvis(~Excess)  %>%
     layer_histograms(fill.hover = "red")
```

Interactive Input

The := operator

- We can also set properties to be the output of an interactive control
- We use the setting ":=" for this input
- We can optionally set labels next to the control

```
opacity := input_slider(0, 1, label = "Opaci
```

Interactive Input Functions

Functionality

- As well as `input_slider()`, ggvis provides `input_checkbox()`, `input_checkboxgroup()` and more (*See next slide*).
- See the examples in the documentation for how you might use each one.
- You can also use keyboard controls with `left_right()` and `up_down()`.

Interactive Input Functions

Function	**Description**
`input_slider`	Slider to select values or ranges of values
`input_checkbox`	A single check box
`input_checkboxgroup`	A group of check boxes
`input_numeric`	A spin box
`input_radiobuttons`	Selection of a single value from a set of options
`input_select`	A drop down text selection
`input_text`	Text input

Tooltips

- `add_tooltip` allows us to include other behaviour when we hover or click on a point
- We can provide a single function that takes as input a list of the data stored in a given point

Interactivity Exercise

Exercise

- Update the previous plot of *mpg* against *wt* so points change colour when they hover over
- Add a tooltip that shows the value of *mpg* when the point is hovered over
- Add a slider for the span of the smooth line so that values can be set between 0 and 1

Controlling with Sliders

The following example allows you to control the size and opacity of points with two sliders:

```
mtcars  %>%
   ggvis(~wt, ~mpg,
      size := input_slider(10, 100),
      opacity := input_slider(0, 1)) %>%
   layer_points()
```

You can also connect interactive components to other plot parameters like the width and centers of histogram bins:

```
mtcars %>%
  ggvis(~wt) %>%
  layer_histograms(width =  input_slider(0, 2,
                           step = 0.10,
                           label = "width"),
                   center = input_slider(0, 2,
                           step = 0.05,
                           label = "center"))
```

```
keys_s <- left_right(10,1000,step = 50)

mtcars  %>%
  ggvis(~wt, ~mpg, size := keys_s,
        opacity := 0.5) %>%
    layer_points()
```

Interactvity : Tooltips

You can also add on more complex types of interaction like tooltips:

```
mtcars  %>%
  ggvis(~wt, ~mpg) %>%
     layer_points() %>%
     add_tooltip(function(df) df$wt)
```

Vignette : Youll learn more about complex interaction in the **Interactivity** vignette.

Shiny

- Behind the scenes, interactive plots are built with shiny, and you can currently only have one running at a time in a given R session.
- To finish with a plot, press the stop button in Rstudio, or close the browser window and then press Escape or Ctrl + C in R.

Multiple layers

- Rich graphics can be created by combining multiple layers on the same plot.

Starting Off

```
mtcars %>%
    ggvis(~wt, ~mpg) %>%
      layer_smooths() %>%
      layer_points()
```

(Indentation not abssolutely necessary in R, but helpful for readers)

Mutliple Smoothers

You could use this approach to add two smoothers with varying degrees of *wiggliness*:

```
mtcars  %>%
    ggvis(~wt, ~mpg) %>%
       layer_smooths(span = 1)    %>%
       layer_smooths(span = 0.3,
            stroke := "red")
```

Exercises

- Try out the previous code with different setting for `span` and `stroke`.
- Try out a similar exercise for the *iris* data set, using pairings of the following variables
  - `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`
- Try out the code on the next slide.

Multiple Layers

Slider to specify the smoothing parameter and point size:

```
mtcars %>%
    ggvis(~wt, ~mpg) %>%
        layer_smooths(span =
            input_slider(0.5, 1,
            value = 1)) %>%
        layer_points(size :=
            input_slider(100, 1000,
            value = 100))
```

Multiple Layers

- You can learn more about building up rich hierarchical graphics in Hadley Wickham's "**Data Hierarchy**" vignette.