CSSS508, Week 2

Plotting with `ggplot2`

Chuck Lanfear

Apr 10, 2019

Updated: Mar 28, 2019

But First...

Some useful stuff

Comments

You may have noticed that sometimes I have written code that looks like this:

```
new.object <- 1:10 # Making vector of 1 to 10
```

`#` is known as the *commenting symbol* in R!

Anything written on the same line *after* `#` will not be run by R.

This is useful for annotating your code to remind you (or others) what you are doing in a section.[1]

[1] In R Markdown documents, comments only work in chunks. Outside of a chunk, `#` creates **headers** like "comments" at the top of this slide.

Saving Files

You can save an R object on your computer as a file to open later:

```
save(new.object, file="new_object.RData")
```

You can open saved files in R as well:

```
load("new_object.RData")
```

But where are these files being saved and loaded from?

Working Directories

R saves files and looks for files to open in your current **working directory**[1]. You can ask R what this is:

```
getwd()
```

```
## [1] "C:/Users/cclan/OneDrive/GitHub/CSSS508/Lectures/Week2"
```

Similarly, we can set a working directory like so:

```
setwd("C:/Users/cclan/Documents")
```

Don't set a working directory in R Markdown documents! They automatically set the directory they are in as the working directory.

[1] For a simple R function to open an Explorer / Finder window at your working directory, see this StackOverflow response.

Managing Files

When managing R projects, it is normally best to give each project (such as a homework assignment) its own folder. I use the following system:

- Every class or project has its own folder
- Each assignment or task has a folder inside that, which is the working directory for that item.
- `.Rmd` and `.R` files are named clearly and completely

For example, this presentation is located and named this:
`GitHub/CSSS508/Lectures/Week2/CSSS508_Week2_ggplot2.Rmd`

You can use whatever system you want, but be consistent so your projects are organized! You don't want to lose work by losing or overwriting files!

For large projects containing many files, I recommend using RStudio's built in project management system found in the top right of the RStudio window.

For journal articles I recommend Ben Marwick's `rrtools` and `huskydown` for UW dissertations and theses. I made an `rrtools` demo presentation here.

File Types

We mainly work with three types of file in this class:

- `.Rmd`: These are **markdown** *syntax* files, where you write code to *make documents*.
- `.R`: These are **R** *syntax* files, where you write code to process and analyze data *without making an output document*.[1]
- `.html` or `.pdf`: These are the output documents created when you *knit* a markdown document.

Make sure you understand the difference between the uses of these file types! Please ask for clarification if needed!

[1] While beyond the scope of this class, you can use the `source()` function to run a `.R` script file inside a `.Rmd` or `.R` file. Using this you can break a large project up into multiple files but still run it all at once!

Data and Subsetting

Gapminder Data

We'll be working with data from Hans Rosling's Gapminder project. An excerpt of these data can be accessed through an R package called `gapminder`, cleaned and assembled by Jenny Bryan at UBC.

In the console: `install.packages("gapminder")`

Load the package and data:

```
library(gapminder)
```

Check Out Gapminder

The data frame we will work with is called `gapminder`, available once you have loaded the package. Let's see its structure:

```
str(gapminder)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 2
##  $ gdpPercap: num  779 821 853 836 740 ...
```

What's Interesting Here?

- **Factor** variables `country` and `continent`
    - Factors are categorical data with an underlying numeric representation
    - We'll spend a lot of time on factors later!
- Many observations: $n = 1704$ rows
- A nested/hierarchical structure: `year` in `country` in `continent`
    - These are panel data!

Subsetting Data

Installing Tidyverse

We'll want to be able to slice up this data frame into subsets (e.g. just the rows for Afghanistan, just the rows for 1997).

We will use a package called `dplyr` to do this neatly.

`dplyr` is part of the tidyverse family of R packages that are the focus of this course.

If you have not already installed the tidyverse, type, in the console: `install.packages("tidyverse")`

This will install a *large* number of R packages we will use throughout the term, including `dplyr`.

`dplyr` is a very useful and powerful package that we will talk more about soon, but today we're just going to use it for "filtering" data.

Loading dplyr

```
library(dplyr)
```

Wait, was that an error?

When you load packages in R that have functions sharing the same name as functions you already have, the more recently loaded functions overwrite the previous ones ("masks them").

This **message** is just letting you know that. To avoid showing this in your R Markdown file, add `message=FALSE` or `include=FALSE` to your chunk options when loading packages.

Sometimes you may get a **warning message** when loading packages---usually because you aren't running the latest version of R:

```
Warning message:
package `gapminder' was built under R version 3.4.1
```

Chunk options `message=FALSE` or `include=FALSE` will hide this. *Update R* to deal with it completely!

`magrittr` and Pipes

`dplyr` allows us to use `magrittr`[1] operators (`%>%`) to "pipe" data between functions. So instead of nesting functions like this:

```
log(mean(gapminder$pop))
```

```
## [1] 17.2
```

We can pipe them like this:

```
gapminder$pop %>% mean() %>% log()
```

```
## [1] 17.2
```

Read this as, "send `gapminder$pop` to `mean()`, then send the output of that to `log()`." In essence, pipes read "left to right" while nested functions read "inside to out." This may be confusing... we'll cover it more later!

[1] Ceci n'est pas un pipe

`filter` Data Frames

```
gapminder %>% filter(country == "Oman")
```

```
## # A tibble: 12 x 6
##    country continent  year lifeExp     pop gdpPercap
##    <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
##  1 Oman    Asia       1952    37.6  507833     1828.
##  2 Oman    Asia       1957    40.1  561977     2243.
##  3 Oman    Asia       1962    43.2  628164     2925.
##  4 Oman    Asia       1967    47.0  714775     4721.
##  5 Oman    Asia       1972    52.1  829050    10618.
##  6 Oman    Asia       1977    57.4 1004533    11848.
##  7 Oman    Asia       1982    62.7 1301048    12955.
##  8 Oman    Asia       1987    67.7 1593882    18115.
##  9 Oman    Asia       1992    71.2 1915208    18617.
## 10 Oman    Asia       1997    72.5 2283635    19702.
## 11 Oman    Asia       2002    74.2 2713462    19775.
## 12 Oman    Asia       2007    75.6 3204897    22316.
```

What is this doing?

Logical Operators

We used `==` for testing "equals": `country == "Oman"`.

There are many other logical operators:

- `!=`: not equal to
- `>`, `>=`, `<`, `<=`: less than, less than or equal to, etc.
- `%in%`: used with checking equal to one of several values

Or we can combine multiple logical conditions:

- `&`: both conditions need to hold (AND)
- `|`: at least one condition needs to hold (OR)
- `!`: inverts a logical condition (`TRUE` becomes `FALSE`, `FALSE` becomes `TRUE`)

We'll use these a lot so don't worry too much right now!

Multiple Conditions Example

Let's say we want observations from Oman after 1980 and through 2000.

```
gapminder %>%
    filter(country == "Oman" &
           year > 1980 &
           year <= 2000 )
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp     pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Oman    Asia       1982    62.7 1301048    12955.
## 2 Oman    Asia       1987    67.7 1593882    18115.
## 3 Oman    Asia       1992    71.2 1915208    18617.
## 4 Oman    Asia       1997    72.5 2283635    19702.
```

Saving a Subset

If we think a particular subset will be used repeatedly, we can save it and give it a name like any other object:

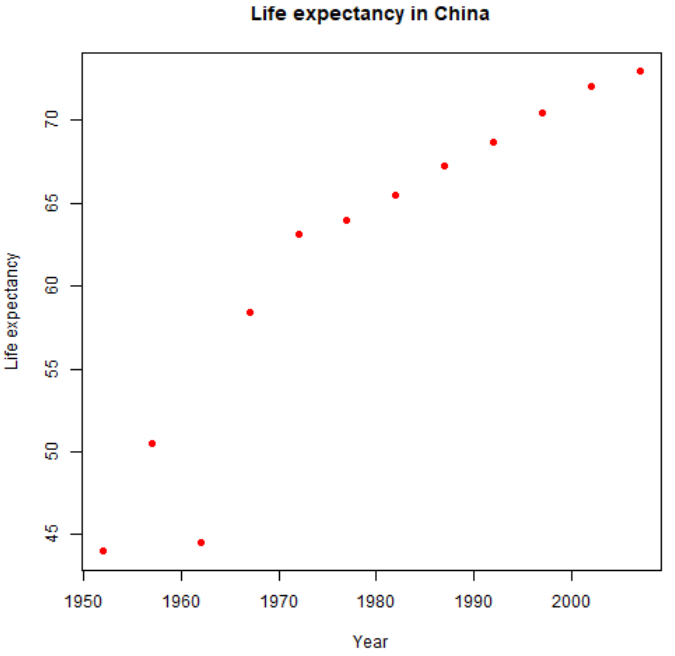
```
China <- gapminder %>% filter(country == "China")
head(China, 4)
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp      pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
## 1 China   Asia       1952    44   556263527      400.
## 2 China   Asia       1957    50.5 637408000      576.
## 3 China   Asia       1962    44.5 665770000      488.
## 4 China   Asia       1967    58.4 754550000      613.
```

ggplot2

Base R Plots from Last Week

```r
plot(lifeExp ~ year,
     data = China,
     xlab = "Year",
     ylab = "Life expectancy",
     main = "Life expectancy in China",
     col = "red",
     cex.lab = 1.5,
     cex.main= 1.5,
     pch = 16)
```



ggplot2

An alternative way of plotting many prefer (myself included)[1] uses the `ggplot2` package in R, which is part of the `tidyverse`.

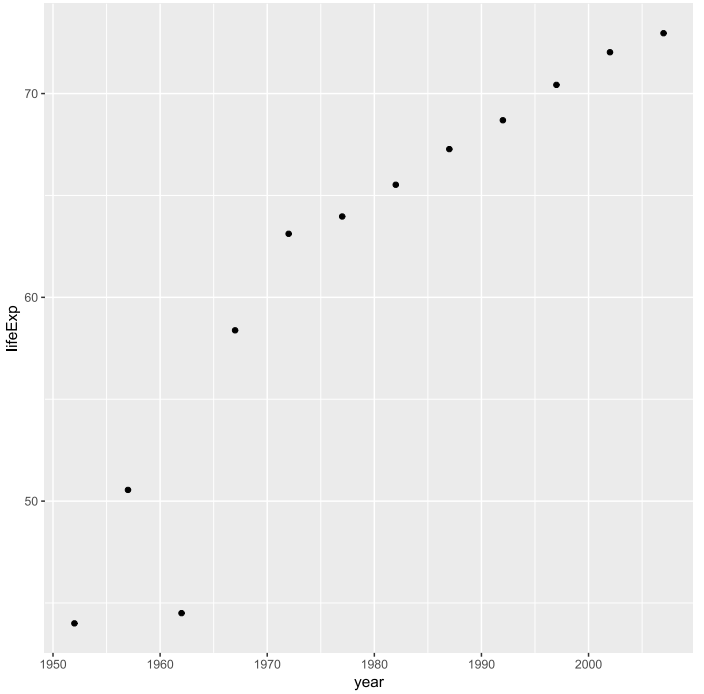
```
library(ggplot2)
```

The core idea underlying this package is the **layered grammar of graphics**: we can break up elements of a plot into pieces and combine them.

[1] Though this is not without debate

Chinese Life Expectancy in `ggplot`

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
    geom_point()
```



Structure of a ggplot

`ggplot2` graphics objects consist of two primary components:

1. **Layers**, the components of a graph.
    - We *add* layers to a `ggplot2` object using `+`.
    - This includes lines, shapes, and text.
2. **Aesthetics**, which determine how the layers appear.
    - We *set* aesthetics using *arguments* (e.g. `color="red"`) inside layer functions.
    - This includes locations, colors, and sizes.
    - Aesthetics also determine how data *map* to appearances.

Layers

Layers are the components of the graph, such as:

- `ggplot()`: initializes `ggplot2` object, specifies input data
- `geom_point()`: layer of scatterplot points
- `geom_line()`: layer of lines
- `ggtitle()`, `xlab()`, `ylab()`: layers of labels
- `facet_wrap()`: layer creating separate panels stratified by some factor wrapping around
- `facet_grid()`: same idea, but can split by two variables along rows and columns (e.g. `facet_grid(gender ~ age_group)`)
- `theme_bw()`: replace default gray background with black-and-white

Layers are separated by a `+` sign. For clarity, I usually put each layer on a new line, unless it takes few or no arguments (e.g. `xlab()`, `ylab()`, `theme_bw()`).

Aesthetics

Aesthetics control the appearance of the layers:

- `x`, `y`: x and y coordinate values to use
- `color`: set color of elements based on some data value
- `group`: describe which points are conceptually grouped together for the plot (often used with lines)
- `size`: set size of points/lines based on some data value
- `alpha`: set transparency based on some data value

Aesthetics: Setting vs. mapping

Layers take arguments to control their appearance, such as point/line colors or transparency (`alpha` between 0 and 1).

- Arguments like `color`, `size`, `linetype`, `shape`, `fill`, and `alpha` can be used directly on the layers (**setting aesthetics**), e.g. `geom_point(color = "red")`. See the `ggplot2` documentation for options. These *don't depend on the data*.
- Arguments inside `aes()` (**mapping aesthetics**) will *depend on the data*, e.g. `geom_point(aes(color = continent))`.
- `aes()` in the `ggplot()` layer gives overall aesthetics to use in other layers, but can be changed on individual layers (including switching `x` or `y` to different variables)

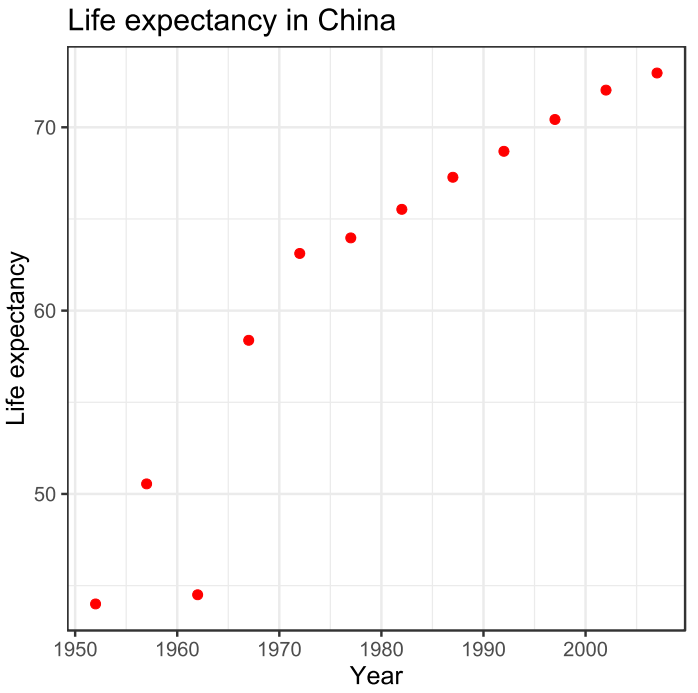
This may seem pedantic, but precise language makes searching for help easier.

Now let's see all this jargon in action.

Axis Labels, Points, No Background

6: Theme

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3) +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy in China") +
  theme_bw(base_size=18)
```

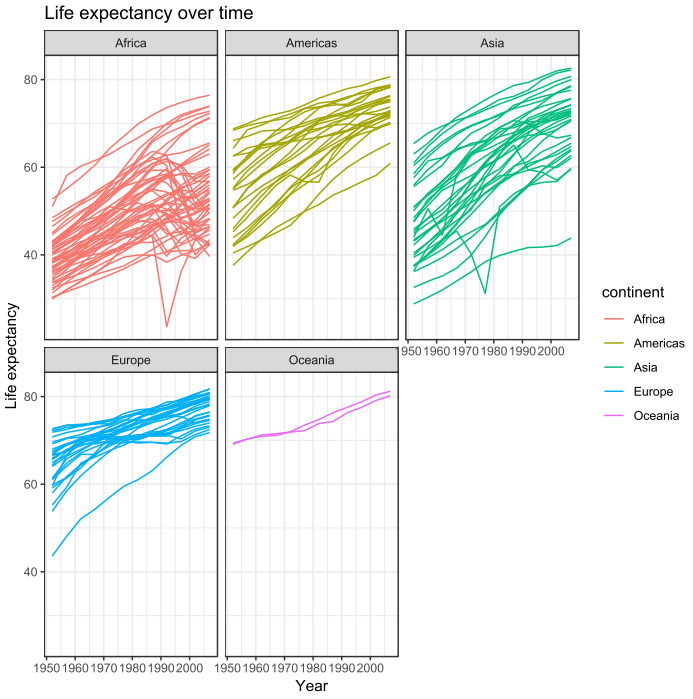


Pick a theme and increase the text size.

Plotting All Countries

7: Facets

```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp,
           group = country,
           color = continent)) +
  geom_line() +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy over time") +
  theme_bw() +
  facet_wrap(~ continent)
```



Looking good!

Storing Plots

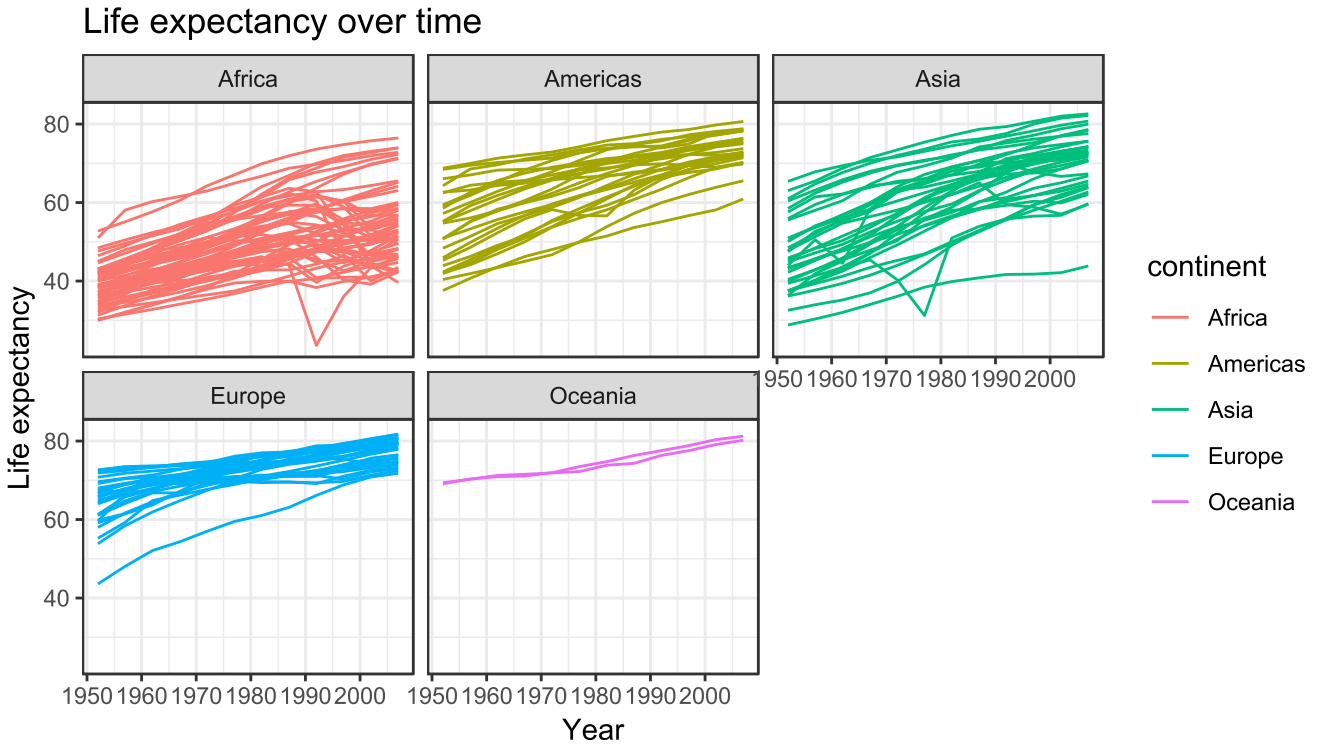
We can assign a `ggplot` object to a name:

```
lifeExp_by_year <- ggplot(data = gapminder,
                          aes(x = year, y = lifeExp,
                              group = country, color = continent)) +
  geom_line() +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy over time") +
  theme_bw() +
  facet_wrap(~ continent)
```

The graph won't be displayed when you do this. You can show the graph using a single line of code with just the object name, *or take the object and add more layers*.

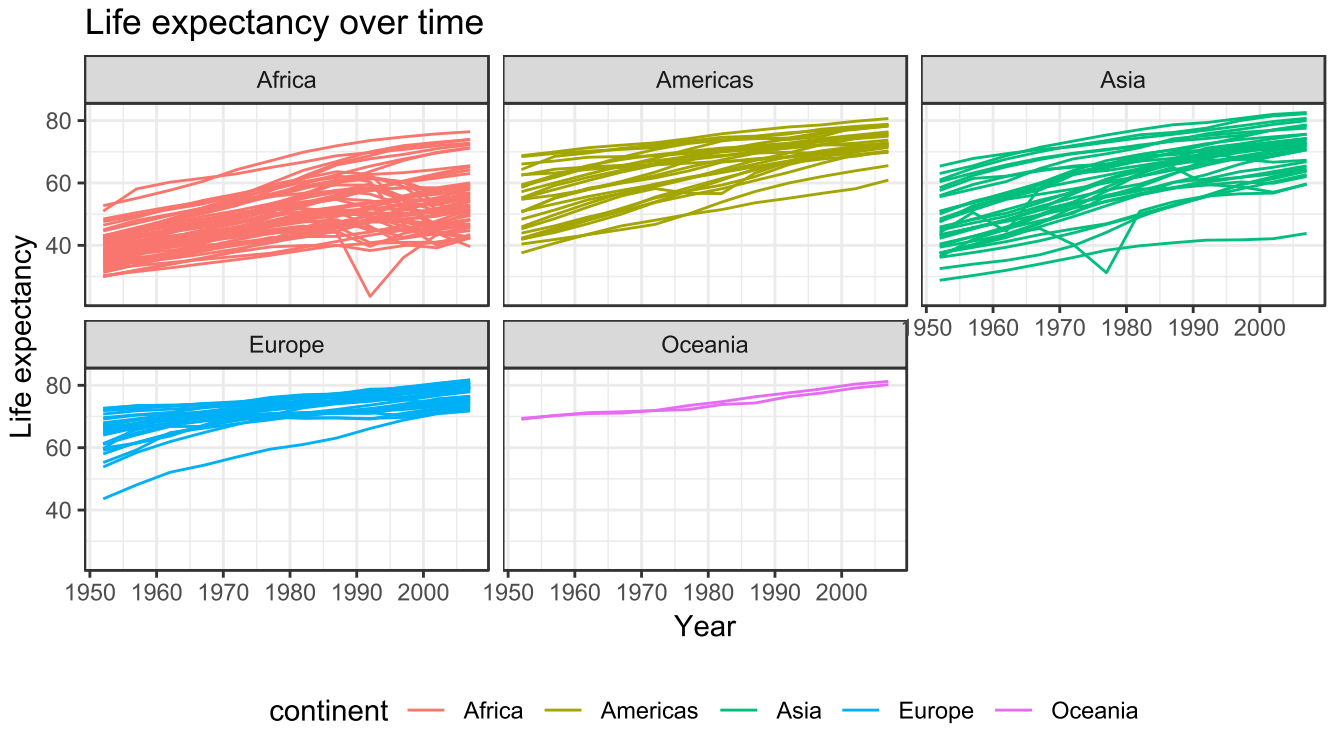
Showing a Stored Graph

```
lifeExp_by_year
```



Adding a Layer

```
lifeExp_by_year +
    theme(legend.position = "bottom")
```



Tinkering Suggestions

Start experimenting with making some graphs in `ggplot2` of the Gapminder data. You can look at a subset of the data using `filter()` to limit rows, plot different x and y variables, facet by a factor, etc.

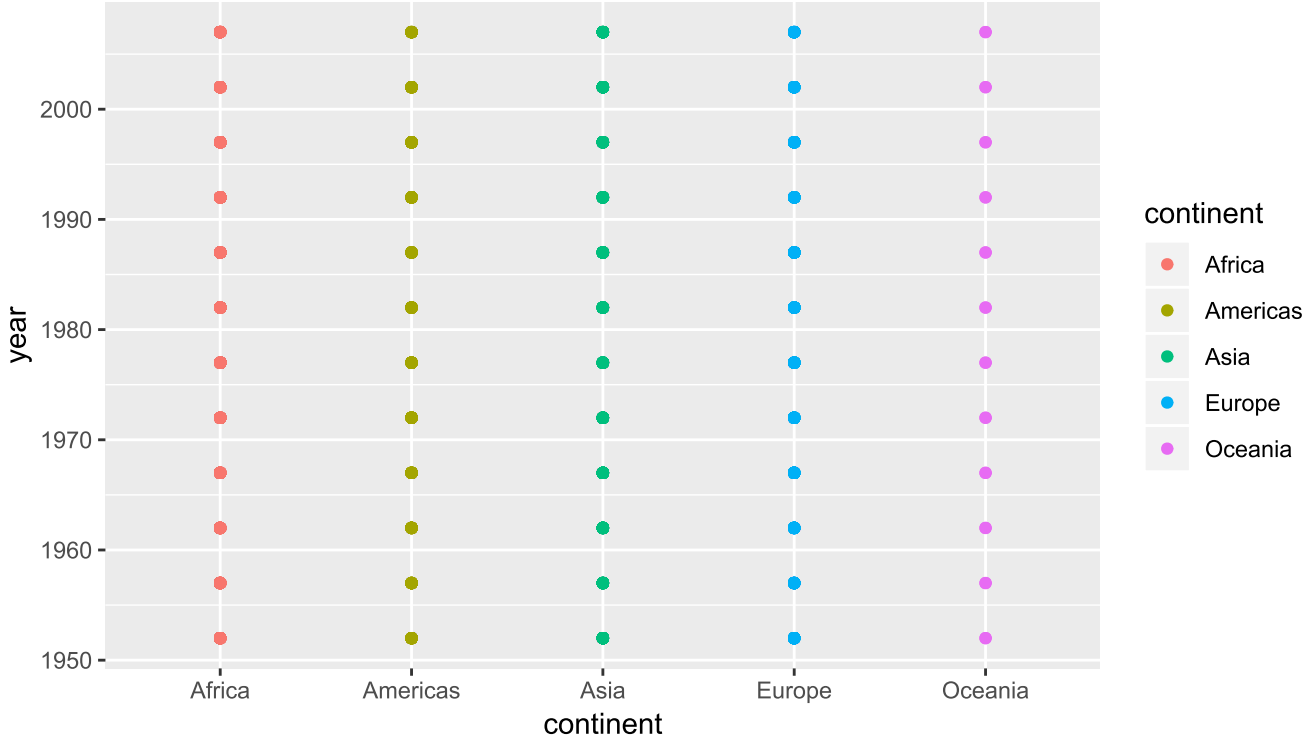
Some other options:

- `geom_histogram()`, `geom_density()`, `geom_boxplot()` (see the Cookbook for R site for a reference)
- `geom_smooth()` for adding loess or regression lines (see the `ggplot2` documentation)
- Install Jeff Arnold's `ggthemes` package, load it, and try `theme_economist()`, `theme_stata()`, `theme_excel()` instead of no theme or `theme_bw()`

Common Problem: Overplotting

Often we want a scatterplot of things that have discrete units. All those dots plot over each other!

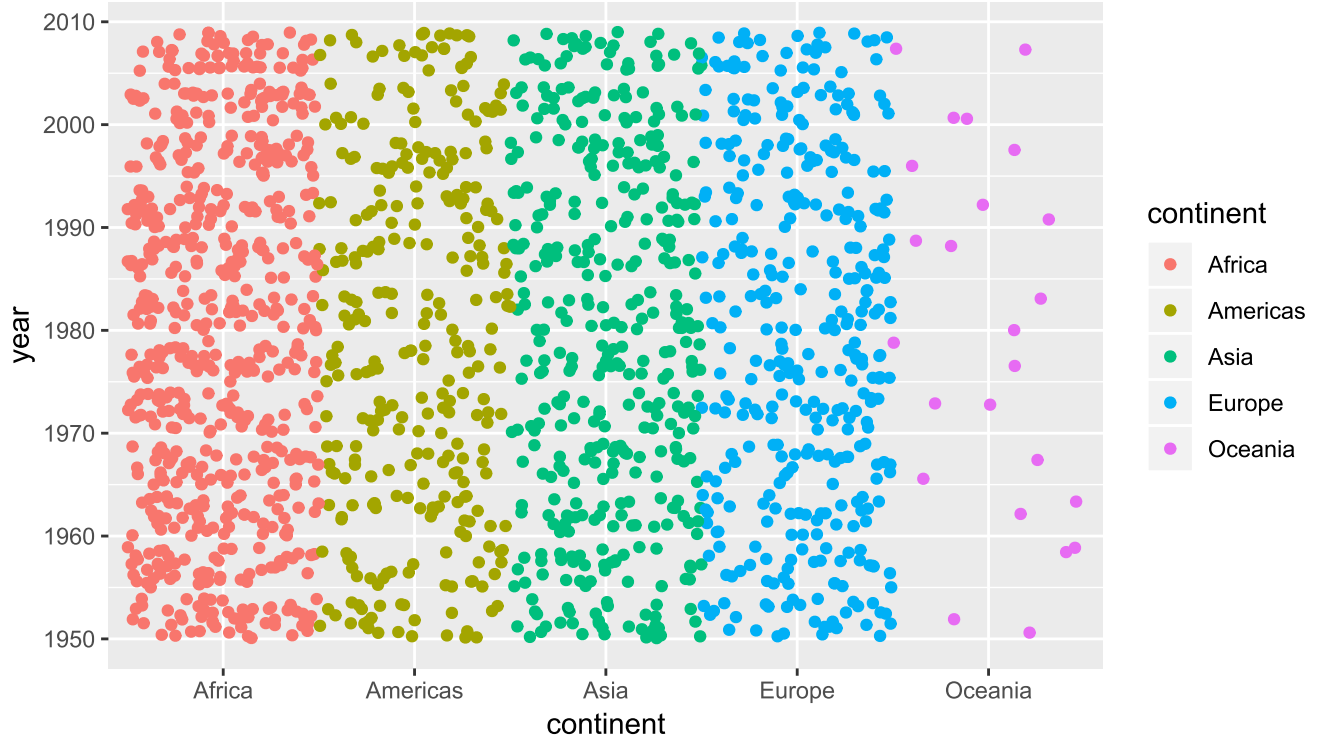
```
ggplot(data = gapminder, aes(x = continent, y = year, color = continent)) +
    geom_point()
```



Fixing Overplotting with Jitter

Inside `geom_point()`, `position = position_jitter(width=a, height=b)` shifts points up to `a` units horizontally and `b` units vertically.

```
ggplot(data = gapminder, aes(x = continent, y = year, color = continent)) +
    geom_point(position = position_jitter(width = 0.5, height = 2))
```



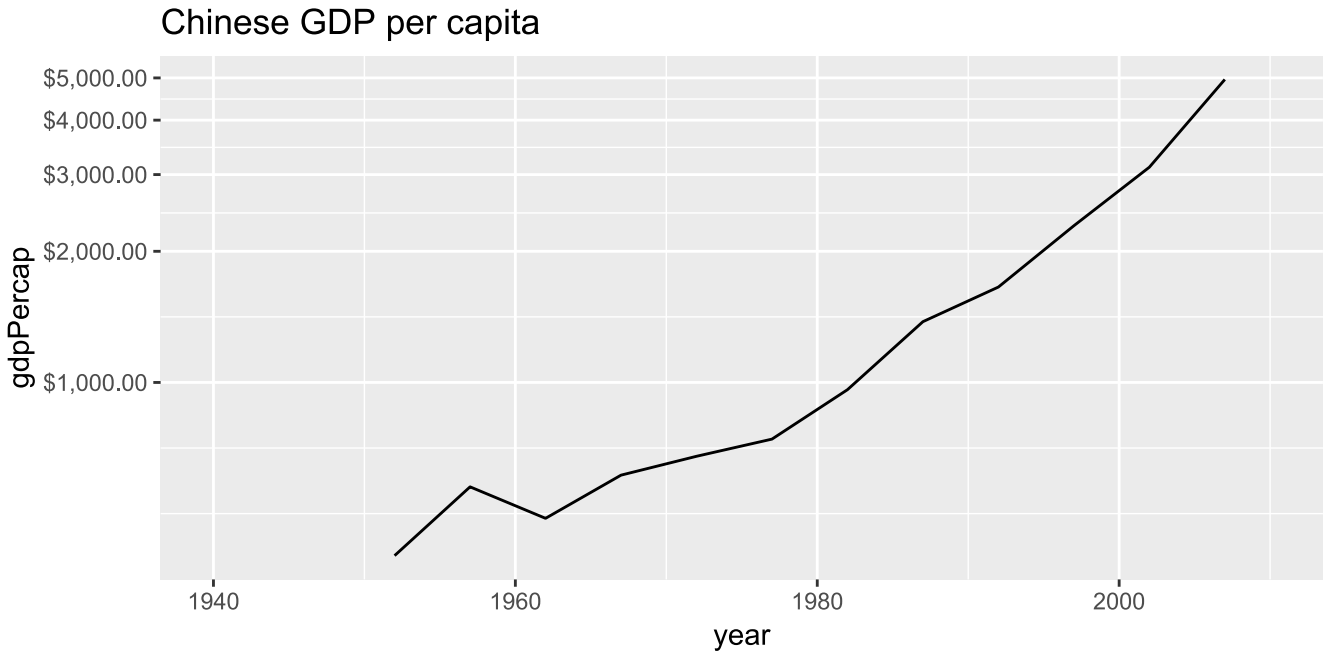
Changing the Axes

We can modify the axes in a variety of ways, such as:

- Change the x or y range using `xlim()` or `ylim()` layers
- Change to a logarithmic or square-root scale on either axis: `scale_x_log10()`, `scale_y_sqrt()`
- Change where the major/minor breaks are: `scale_x_continuous(breaks =, minor_breaks = )`

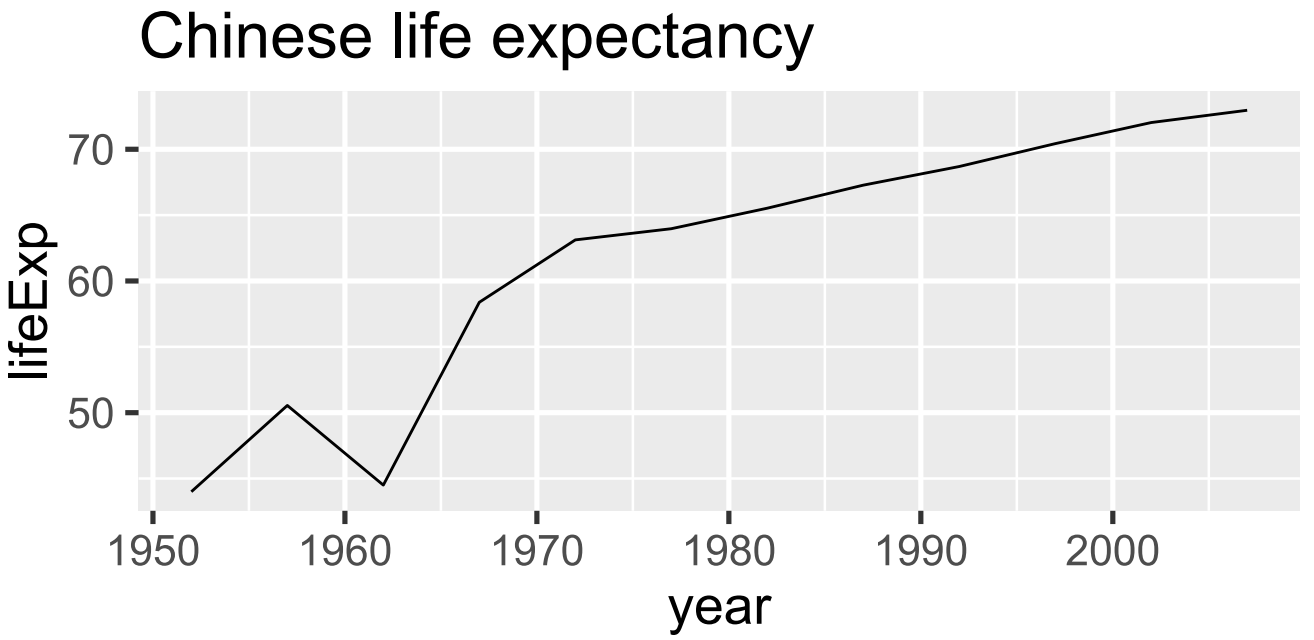
Axis Changes

```
ggplot(data = China, aes(x = year, y = gdpPercap)) +
    geom_line() +
    scale_y_log10(breaks = c(1000, 2000, 3000, 4000, 5000),
                  labels = scales::dollar) +
    xlim(1940, 2010) + ggtitle("Chinese GDP per capita")
```



Fonts Too Small?

```
ggplot(data = China, aes(x = year, y = lifeExp)) +
    geom_line() +
    ggtitle("Chinese life expectancy") +
    theme_gray(base_size = 20)
```



Text and Tick Adjustments

Text size, labels, tick marks, etc. can be messed with more precisely using arguments to the `theme()` layer.

Examples:

- `plot.title = element_text(size = rel(2), hjust = 0)` makes the title twice as big as usual and left-aligns it
- `axis.text.x = element_text(angle = 45)` rotates x axis labels
- `axis.text = element_text(colour = "blue")` makes the x and y axis labels blue
- `axis.ticks.length = unit(.5, "cm")` makes the axis ticks longer

Note: `theme()` is a different layer than `theme_gray()` or `theme_bw()`, which you might also be using in a previous layer. See the `ggplot2` documentation for details.

I recommend using `theme()` *after* `theme_bw()` or other *global themes*.

Scales for Color, Shape, etc.

Scales are layers that control how the mapped aesthetics appear. You can modify these with a `scale_[aesthetic]_[option]()` layer where `[aesthetic]` is `color`, `shape`, `linetype`, `alpha`, `size`, `fill`, etc. and `[option]` is something like `manual`, `continuous` or `discrete` (depending on nature of the variable).

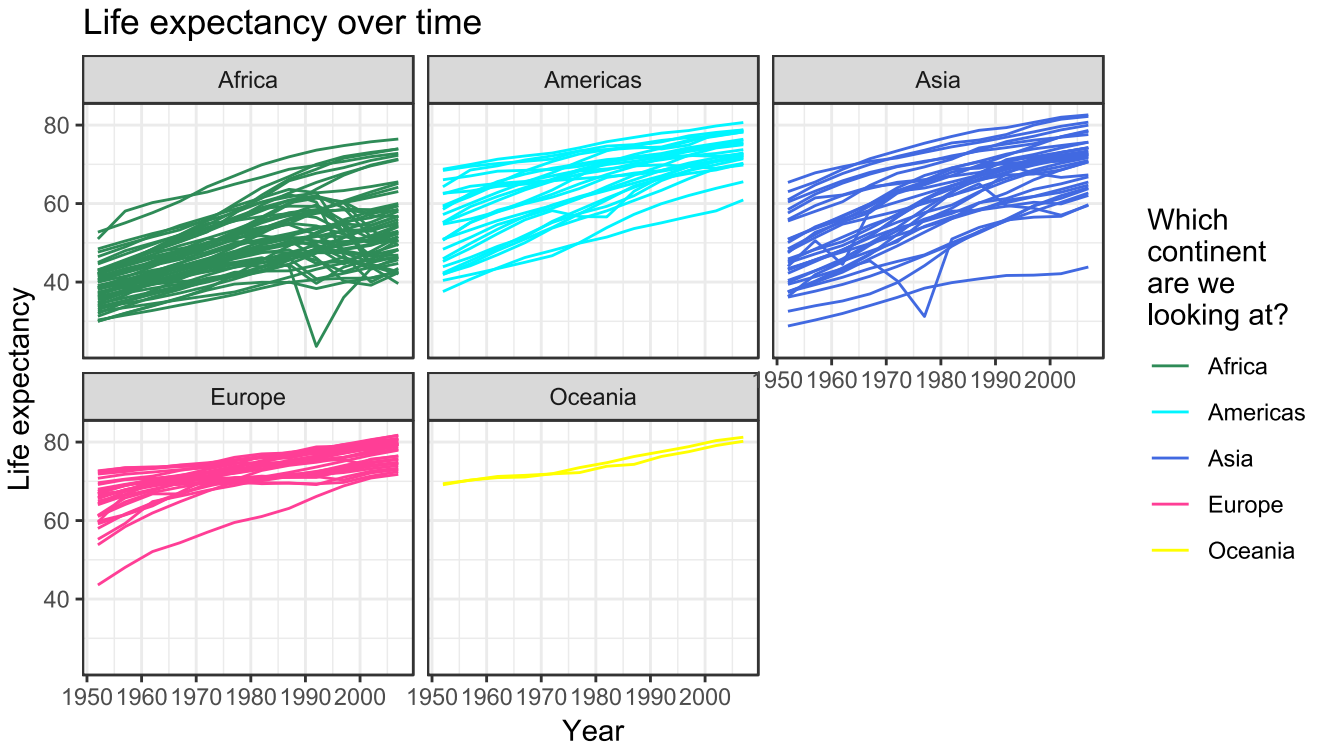
Examples:

- `scale_linetype_manual()`: manually specify the linetype for each different value
- `scale_alpha_continuous()`: varies transparency over a continuous range
- `scale_color_brewer(palette = "Spectral")`: uses a palette from http://colorbrewer2.org (great site for picking nice plot colors!)

When confused... Google or StackOverflow it!

Legend Name and Manual Colors

```
lifeExp_by_year +
  scale_color_manual(
    name = "Which\ncontinent\nare we\nlooking at?", # \n adds a line break
    values = c("Africa" = "seagreen", "Americas" = "turquoise1",
               "Asia" = "royalblue", "Europe" = "violetred1", "Oceania" = "yellow"))
```



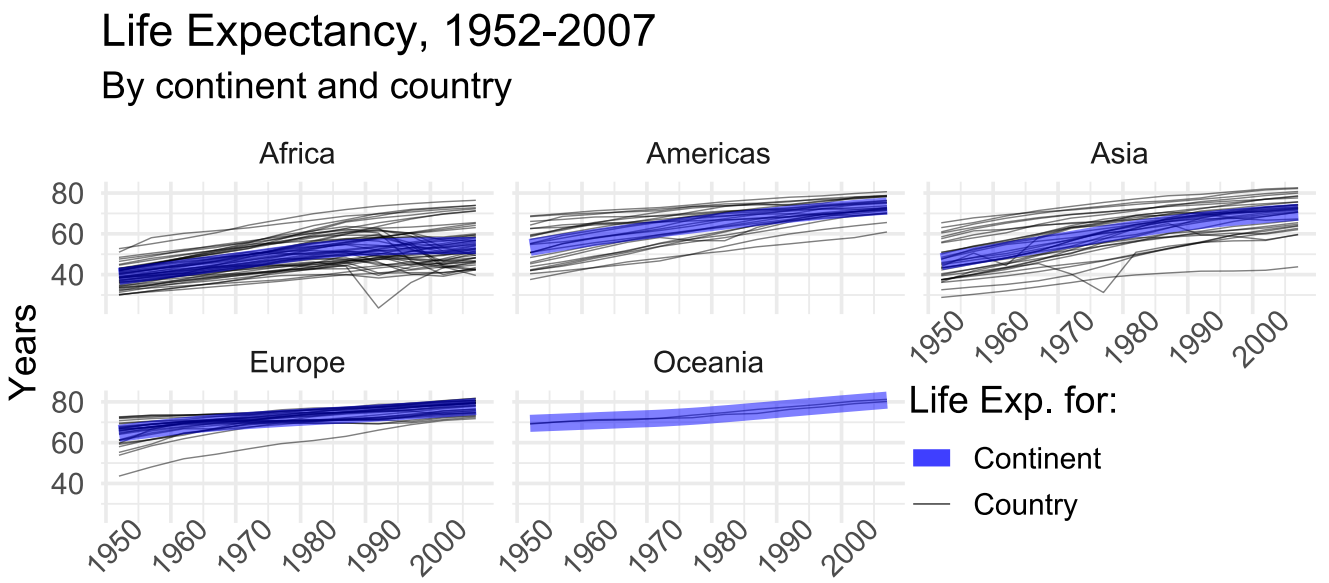
Fussy Manual Legend Example Code

```
ggplot(data = gapminder, aes(x = year, y = lifeExp, group = country)) +
    geom_line(alpha = 0.5, aes(color = "Country", size = "Country")) +
    geom_line(stat = "smooth", method = "loess",
          aes(group = continent, color = "Continent", size = "Continent"), alpha = 0.5) +
    facet_wrap(~ continent, nrow = 2) +
    scale_color_manual(name = "Life Exp. for:",
                       values = c("Country" = "black", "Continent" = "blue")) +
    scale_size_manual(name = "Life Exp. for:",
                      values = c("Country" = 0.25, "Continent" = 3)) +
    theme_minimal(base_size = 14) +
    ylab("Years") + xlab("") +
    ggtitle("Life Expectancy, 1952-2007", subtitle = "By continent and country") +
    theme(legend.position=c(0.75, 0.2), axis.text.x = element_text(angle = 45))
```

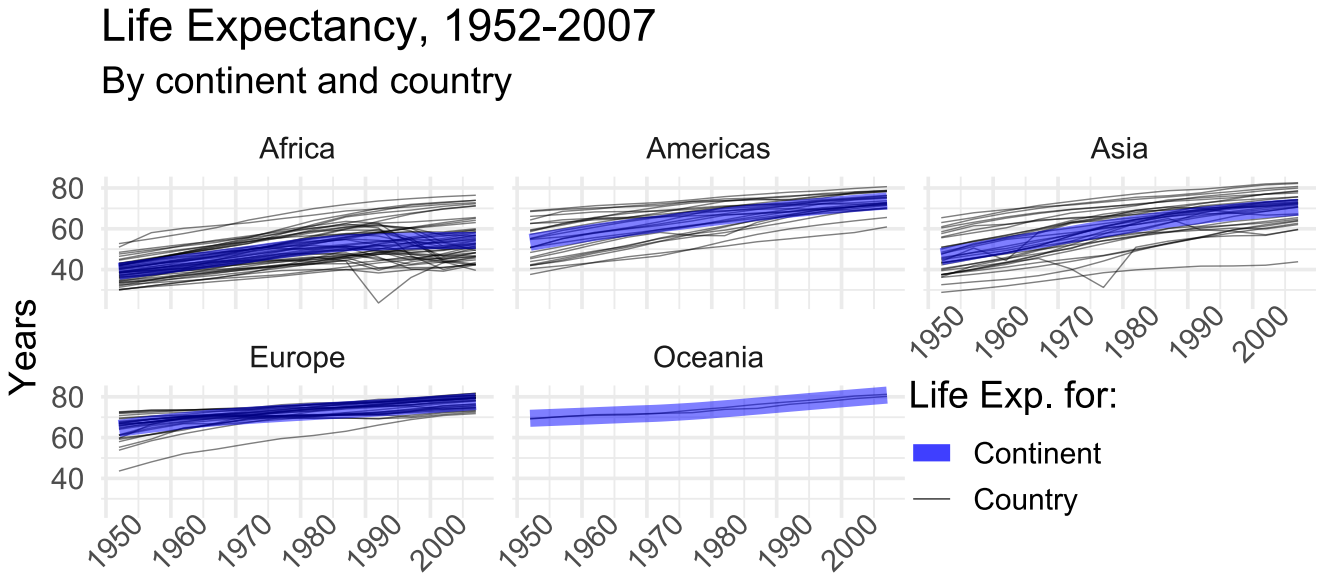
Wow, there's a lot going on here!

- Two different `geom_line()` calls
  - One of them draws a *loess* curve
- `facet_wrap()` to make a plot for each level of `continent`
- Manual scales for size and color
- Custom labels, titles, and rotated x axis text

```
ggplot(data = gapminder, aes(x = year, y = lifeExp, group = country)) +
  geom_line(alpha = 0.5, aes(color = "Country", size = "Country")) +
  geom_line(stat = "smooth", method = "loess",
            aes(group = continent, color = "Continent", size = "Continent"), alpha = 0.5) +
  facet_wrap(~ continent, nrow = 2) +
  scale_color_manual(name = "Life Exp. for:", values = c("Country" = "black", "Continent" = "blue")) +
  scale_size_manual(name = "Life Exp. for:", values = c("Country" = 0.25, "Continent" = 3)) +
  theme_minimal(base_size = 14) + ylab("Years") + xlab("") +
  ggtitle("Life Expectancy, 1952-2007", subtitle = "By continent and country") +
  theme(legend.position=c(0.75, 0.2), axis.text.x = element_text(angle = 45))
```



Fussy Manual Legend



Observation: One could use `filter()` to identify the countries with dips in life expectancy and investigate.

Know Your History: What happened in Africa in the early 1990s and Asia in the mid-1970s that might reduce life expectancy suddenly *for one country*?

More on Customizing Legends

You can move the legends around, flip their orientation, remove them altogether, etc. The Cookbook for R website is my go-to for burning questions such as how to change the legend labels.

Saving `ggplot` Plots

When you knit an R Markdown file, any plots you make are automatically saved in the "figure" folder in `.png` format. If you want to save another copy (perhaps of a different file type for use in a manuscript), use `ggsave()`:

```
ggsave("I_saved_a_file.pdf", plot = lifeExp_by_year,
       height = 3, width = 5, units = "in")
```

If you didn't manually set font sizes, these will usually come out at a reasonable size given the dimensions of your output file.

Bad/non-reproducible way[1]: choose *Export* on the plot preview or take a screenshot / snip.

[1] I still do this for quick emails of simple plots. Bad me!

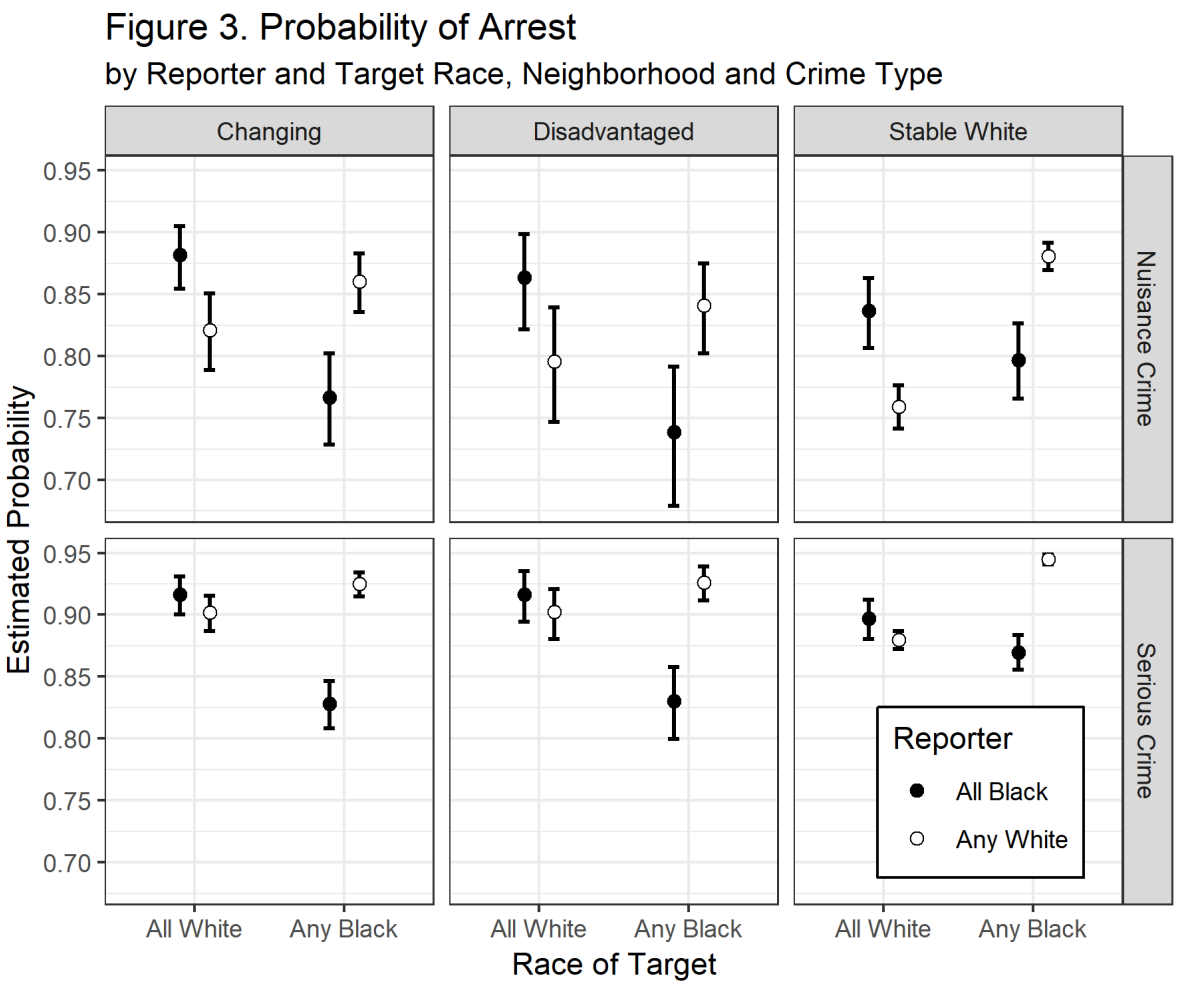
Bonus Plot

`ggplot2` is well suited to making complex, publication ready plots.

This is the complete syntax for one plot from a recent article of mine.[1]

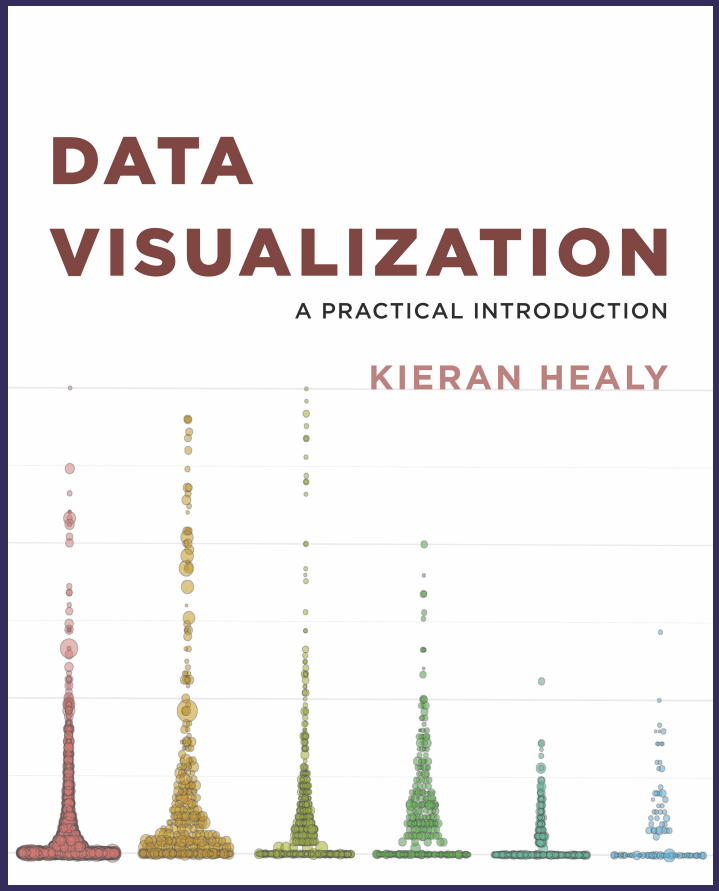
```
ggplot(estimated_pes, aes(x = Target, y = PE, group = Reporter)) +
  facet_grid(`Crime Type` ~ Neighborhood) +
  geom_errorbar(aes(ymin = LB, ymax = UB),
                position = position_dodge(width = .4), size = 0.75, width = 0.15) +
  geom_point(shape = 21, position = position_dodge(width = .4),
             size = 2, aes(fill = Reporter)) +
  scale_fill_manual("Reporter",
                    values = c("Any White" = "white", "All Black" = "black")) +
  ggtitle("Figure 3. Probability of Arrest",
          subtitle = "by Reporter and Target Race, Neighborhood and Crime Type") +
  xlab("Race of Target") + ylab("Estimated Probability") +
  theme_bw() + theme(legend.position = c(0.86, 0.15),
                     legend.background = element_rect(color = 1))
```

[1] Lanfear, Charles C., Lindsey R. Beach, Timothy A. Thomas. 2018. "Formal Social Control in Changing Neighborhoods: Racial Implications of Neighborhood Context on Reactive Policing." *City & Community* 17(4):1075-1099



We'll build this plot in the lecture on model results!

Book Recommendation



- Targeted at Social Scientists without technical backgrounds
- Teaches good visualization principles
- Uses R, `ggplot2`, and `tidyverse`
- Free online version!
- Affordable in print

Homework

Pick some relationship to look at in the Gapminder data and write up a .Rmd file investigating that question graphically. You might work with a subset of the data (e.g. just Africa). Upload both the `.Rmd` file and the `.html` file to Canvas.

- Include 4 to 8 plots.
- All titles, axes, and legends should be labelled clearly (no raw variable names).
- You must have at least one graph with `facet_wrap()` or `facet_grid()`.
- You must include at least one manually specified legend.
- You can use other `geoms` like histograms, bar charts, add vertical or horizontal lines, etc. You may find this data visualization cheat sheet helpful.

Your document should be pleasant for a peer to look at, with some organization. You must write up your observations in words as well as showing the graphs. Use chunk options `echo` and `results` to limit the code/output you show in the `.html`.