Barcelona Summer School of Demography

Module 1. Introduction to R

1. Getting started

Tim Riffe
`tim.riffe@ehu.eus`

4 July 2022

Contents

1 Let me introduce myself

Hi, I'm Tim Riffe, a US-American living in the Basque Country, and I did my PhD at this very institute! I've been hooked on `R` since 2009 when I did EDSD, and have since authored several packages, mostly that do demographic things, but sometimes that do plotting things. My own research data prep, analyses, analytic plotting, and presentation-quality plotting are all done in `R`, and in recent years, mostly using the so-called *tidyverse*.

2 About this module

This module will mostly stress data analysis and `R` programming using tidyverse concepts. This will pair better with later modules from Ilya Kashnitsky and Juan Galeano, although Marie Pier's module may need to introduce some more base concepts for matrix algebra especially. Teaching this material within the tidyverse (still to be defined) is a consequential course design decision: Base introductions to `R` get you started being a programmer, whereas `tidyverse` introductions to `R` get you started being a data analyst. Your route to being a better data analyst/visualizer could be via Base programming, as it was for me, but it is a longer road than simply jumping into the `tidyverse`. So this is what we're going to do: We'll spend two days doing the tidyverse approach, and then we'll shift gears to talk about some programming concepts that will augment your abilities and ultimately your experience of `R`. These will most importantly include loops and function-writing. The programming concepts will help you in Marie-Pier's module, while the tidyverse concepts will help you throughout the summer school. This is an absolute intro course, I don't expect you'll have any experience with `R` or have even heard of these concepts. I expect you'll be able to keep up if you type along and take notes as we go, and if you tell me to stop, slow down, or repeat when needed. Real learning will happen when you try to apply these concepts on your own, and for this reason I'll give exercises.

3 `R` and the `Rstudio` environment

`R` is a language. You should download and install the latest version from here: https://cran.r-project.org/ – pick the one that runs on your operating system. If you're using Windows, you might also want to install `Rtools`. The instructions to do this are given on your R download page.

`R` is simple and lightweight. It's possible to work directly in its console, or even from the terminal, but we'll instead use a very user-friendly application to interact with it:

`Rstudio` is an application. You should download and install the latest Desktop (free) version from here: https://www.rstudio.com/products/rstudio/download/ – again, look at the OS specs and pick the one that pairs well with your operating system.

`Rstudio` is many things at once. It is a file manager, a script editor, a document editor, an instance of `R`, a data viewer, even a web browser. This is where we'll do everything. We will be working almost exclusively in the form of *markdown* documents, which allows us to mix code, output, and text in a single file. This very document is written in `R` markdown, and it can generate nice-looking documents in Word, html, or pdf. We'll get started right away with this.

*** Brief intermission for Rstudio tour and demonstration ***

TO-DO list for the demonstration: 1. Rstudio viewers. 2. arithmetic in the console. 3. start a `.project` for this module. 4. make a folder for Monday. 5. make a new `R` markdown file for session. 6. show how to use `R` from markdown. 7. knit. 8. yell hurrah.

Whenever we do `R` stuff from here forward it will be *from* Rmarkdown. This will become second nature, and you can take that with you.

4 `R` basics

`R` is a *functional* programming language. You can think of functions in different ways– they're like the way functions are used in mathematics, $f()$, or you could think of them as little programs.

You give the function input (data, instructions), and then it does something with that and returns output.

Here's a cheap example:

```
sum(1,2,3,4)
```

```
## [1] 10
```

sum is just the name of the function instead of f, and just the same, we enclose its inputs in round parentheses. `R` has many many such functions available to you. They can be combined in many many ways, and you can write your own functions too. If you get the hang of it, you'll find that functions can help you structure thought and analyses. We'll learn a lot about them later this week, but we'll start using them today without thinking about it much. We call `R` a functional language because everything gets done in `R` using functions, and because functions always behave consistently: a given set of inputs (arguments) always returns the same output.

We often speak of `R` sessions, a given interactive instance of `R`. In a session, you can have different pieces of data, *objects*, and `R` has different kinds of these. At this point I'd usually start talking about the properties and usage of different kinds of objects, but let's not sweat it. That can come along as we go. You can make an object by *assigning* to its name, like this:

```
my_new_object <- rnorm(n = 10, mean = 0, sd = 1)
my_new_object
```

```
##  [1] -0.04272044 -0.78934012  1.61683023 -0.05305509 -0.29023145  1.63305221
##  [7]  0.21050266  0.88347892  1.46604447 -0.58419265
```

Here we have created a new object `my_new_object` by assigning (`<-`) the output of the function `rnorm()` (ten random draws from a standard normal distribution). See how we use `rnorm()` there? This function has three *arguments* (i.e. parameters), in this case intuitively named if you've ever taken an intro statistics class. Now the object `my_new_object` is simply available to use throughout the remainder of this `R` session, yay.

`R` objects can be many kinds of things. Functions are objects, but so are vectors (that's what `my_new_object` is), and other things, and they can come in all shapes, sizes, and structures. Today, from now one, we'll work with a kind of object called a `data.frame`, and its young cousin the `tibble`. These are like a rectangular spreadsheet, or like a dataset in `Stata`: `data.frames` have rows and columns, and there can be different kinds of data in different columns.

Let's get a `data.frame` to play with: Hans Rosling's famous data, which you can get as follows: First we download and install an `R` package (library) called `gapminder` using the function `install.packages()`. You only need to do this once to get the package.

```
install.packages("gapminder")
```

`R` has thousands of packages, built by developers and users alike. Many packages are even written by researchers to make research easier. `R` has a big community of users and developers: let's find it in different places online…

*** Time out to find the 'R' community ***

So some people created `gapminder` somewhere, uploaded it to `R`'s central repository, and then we installed it on our machines straight from an `R` session. Now you can load the package using the `library()` function. This loads the contents of the package into memory, i.e. makes them objects in your `R` session. The main object in this package is a `data.frame` called `gapminder`. There are different ways to take a look at it.

```
library(gapminder)
```

*** Clicky demonstration of viewing data ***

Or you could get metadata about the object from `str()` (structure). This tells us that we have 1704 observations (rows) of 6 variables, and then it tells us the variable (column) names, what kind of data it is (`Factor` = categorical, `int` = integer, `num` = numeric), and a sample of the first few observations in each.

```
str(gapminder)
```

```
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
##  $ 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 3 ...
##  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
##  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 1
##  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

We will mostly work with `data.frames` or data-frame-like objects. To do so, we'll use the *tidyverse* toolkit, which is a collection of packages, so let's make sure we get that installed and loaded.

```
install.packages("tidyverse")
```

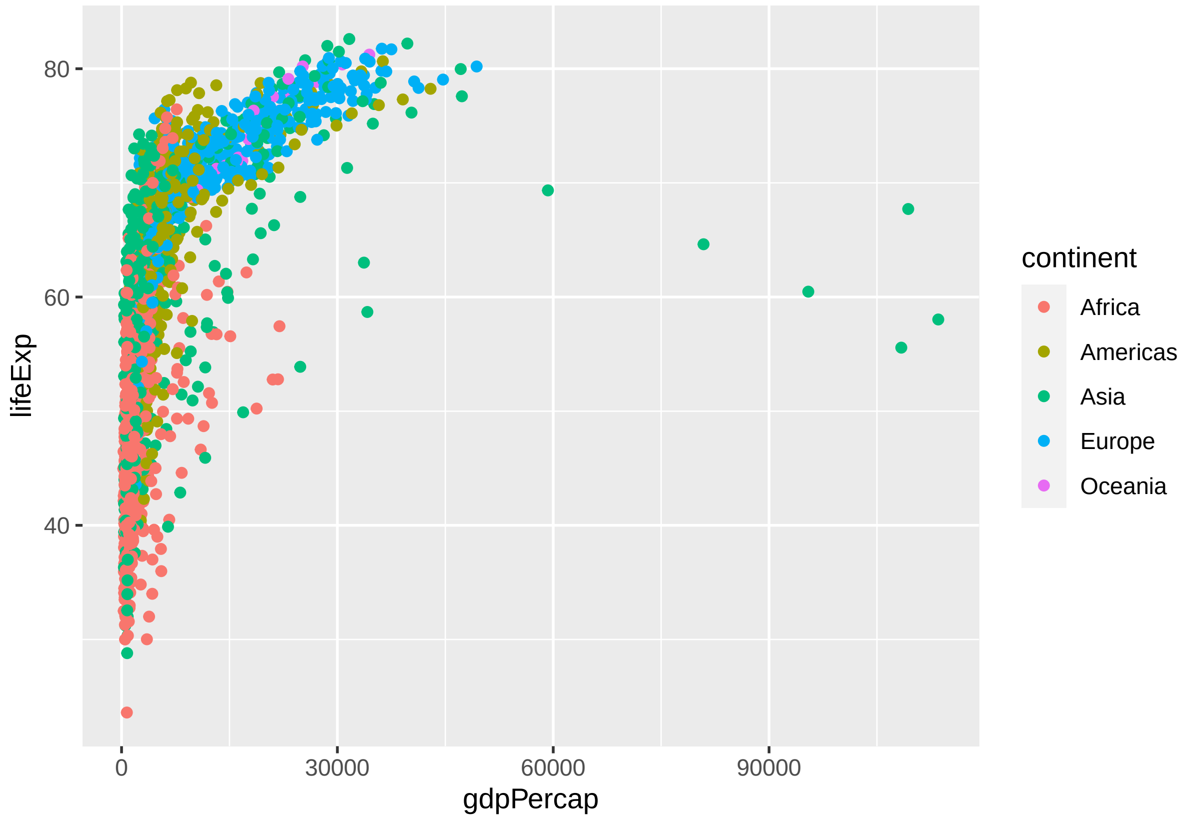
```
library(tidyverse)
```

5 *tidy* data

Tidyverse packages work well together because they share a standard approach to formatting and working with datasets. A dataset is called tidy if each row contains one *observation* and each column one *variable*. The gapminder dataset is tidy. Some other common formats are not tidy. Tidy datasets processed using tidyverse tools allow for fast and understandable analyses that in many cases require no programming, whereas it often takes a certain amount of head-scratching (programming) to analyze not-tidy datasets. I'll give a hand-wavy set of examples in person right about here.

Tidy datasets can also be visualized without further ado using a systematic grammar (Wilkinson 2012) implemented in the `ggplot2` package (Wickham (2016), this loads automatically with `tidyverse`). The `gapminder` examples I'll give today and tomorrow are either gratuitously lifted or modified from Healy (2018), which you can either purchase as the book (I showed it to you), or refer to the free version online: www.socviz.co. It's a fabulous book. Let's take a first look:

```
ggplot(gapminder, mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
    geom_point()
```



Aside: this pattern gives the so-called Preston-curve (S. H. Preston 1975), as-in the same Preston with the popular intro to demography book (S. Preston, Heuveline, and Guillot 2000), which I hear he's busy re-editing.

6 basic ggplot2

The `ggplot()` function *maps* variables in the `gapminder` dataset to either coordinates (x,y) or aesthetics (for example color). This sets up the plot metadata, but it does not plot the data because we didn't tell it how to do so: this final step is done by adding a point geometry `geom_point()`

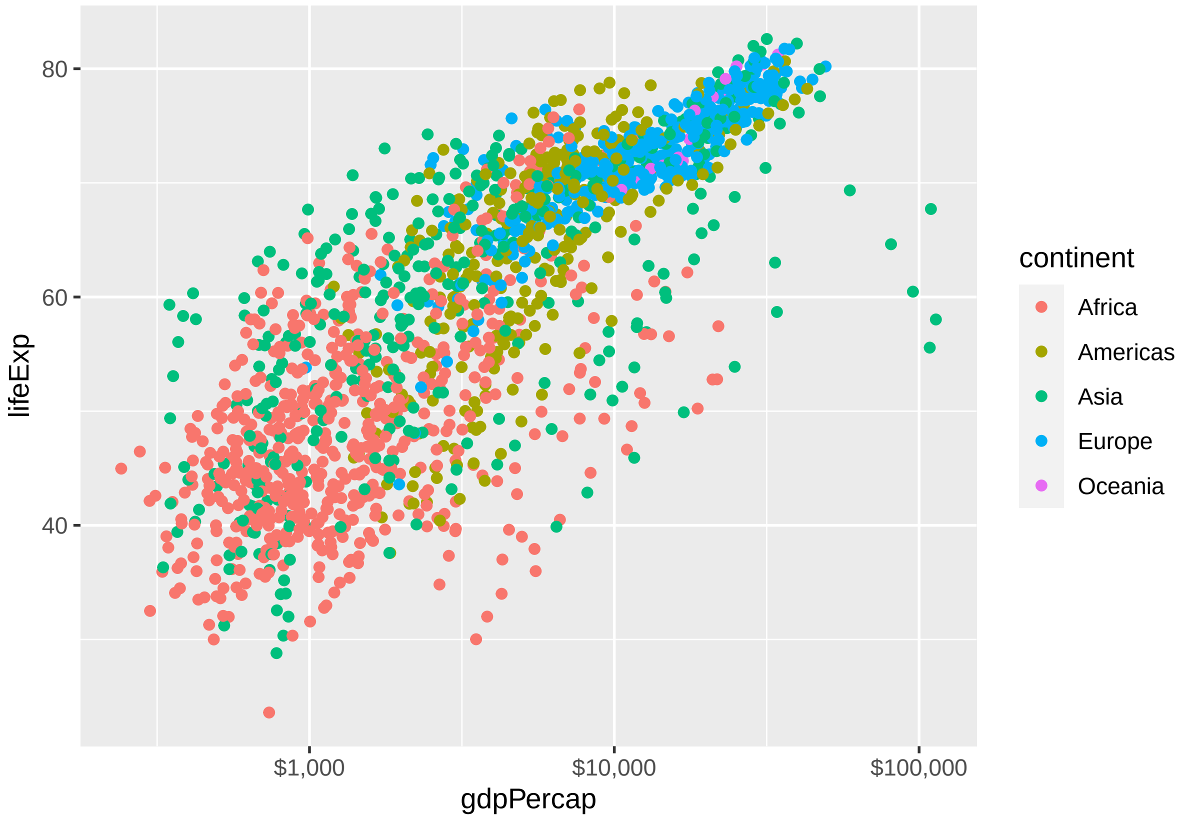
```
library(scales)
```

```
##
## Attaching package: 'scales'

## The following object is masked from 'package:purrr':
##
##     discard

## The following object is masked from 'package:readr':
##
##     col_factor
```

```
ggplot(gapminder, mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
    geom_point() +
    scale_x_log10(labels = dollar)
```



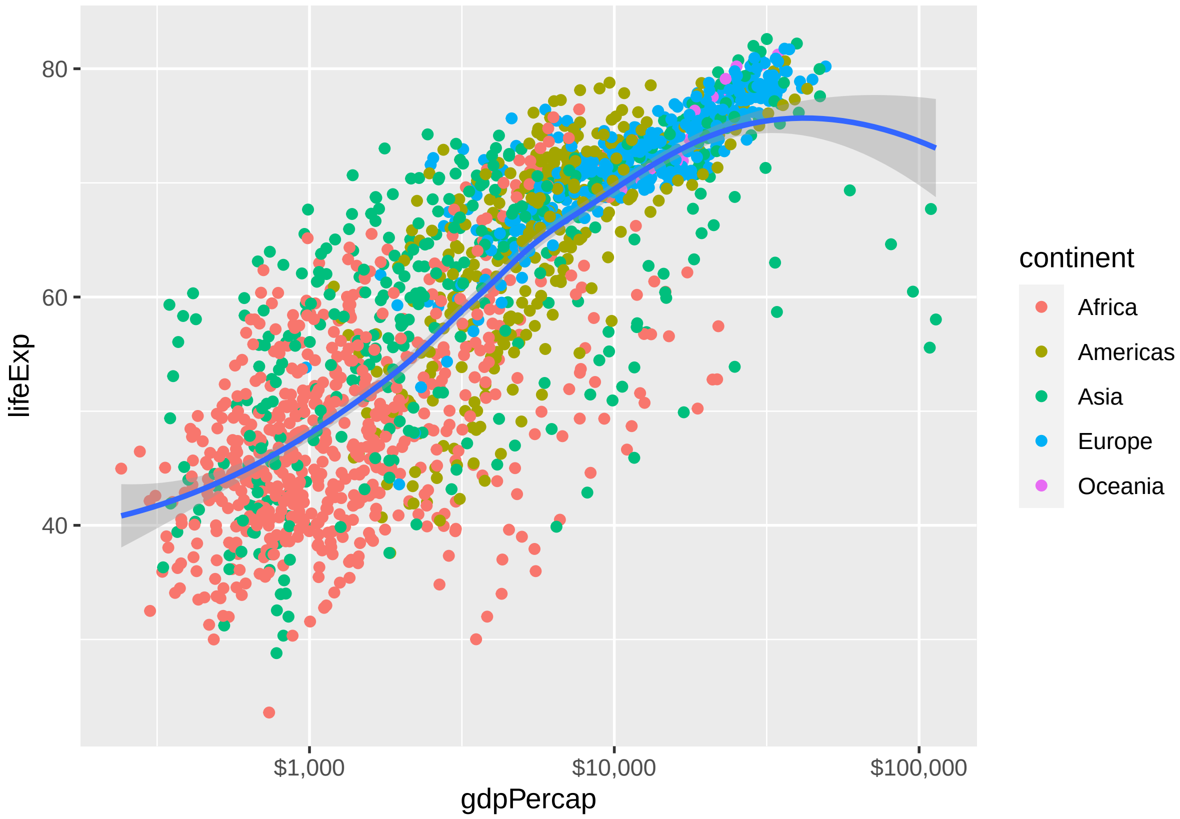
It turns out that by specifying a nice large set of possible mappings, coordinate systems, geoms, and discrete and continous options for scaling mappings, that you can create just about any plot. For other tricky ones, there are usually packages available. Here's a nice overview of `ggplot2` functionality: https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

My objective now is to show you a few different geoms and ways of doing panels in `ggplot2`.

For example, we can also add in a smoother to see the global trend:

```
ggplot(gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
    geom_point(mapping = aes(color = continent)) +
    geom_smooth(method = "loess") +
    scale_x_log10(labels = dollar)
```

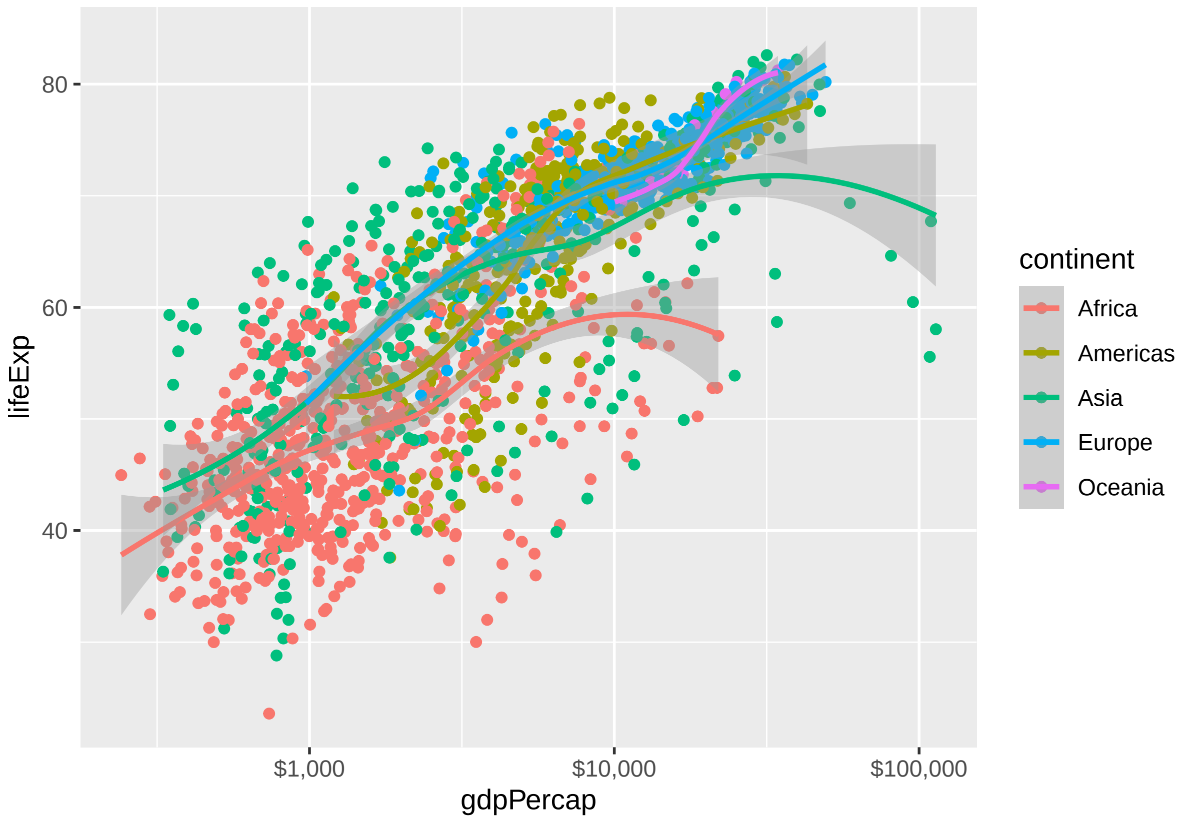
```
## `geom_smooth()` using formula 'y ~ x'
```



See how geoms can also have their own special mapping? Had we left color in the first mapping, then everything that follows would have been split on continent. See:

```
ggplot(gapminder, mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
    geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10(labels = dollar)
```

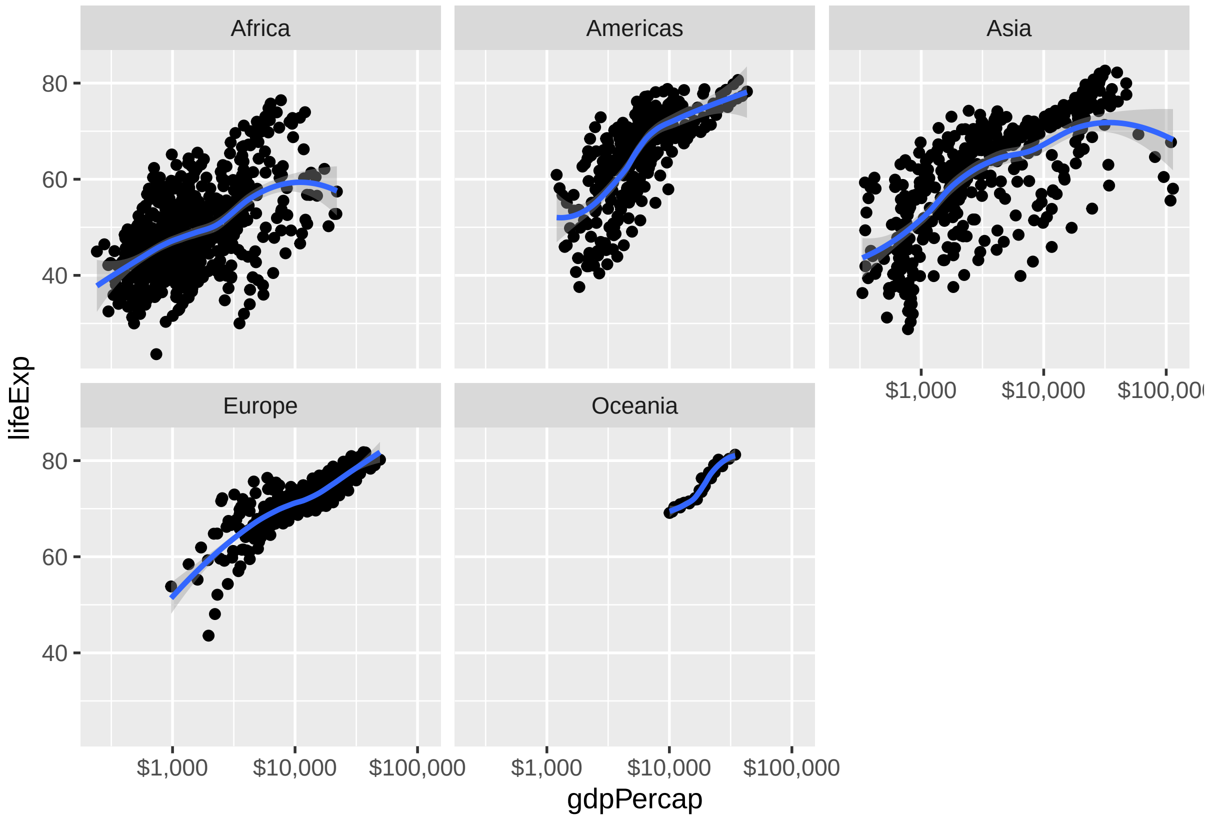
```
## `geom_smooth()` using formula 'y ~ x'
```



Wow, that's a noisy plot! What if instead of color we just split it into subplots by continent?

```
ggplot(gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
    geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10(labels = dollar) +
    facet_wrap(~continent)
```

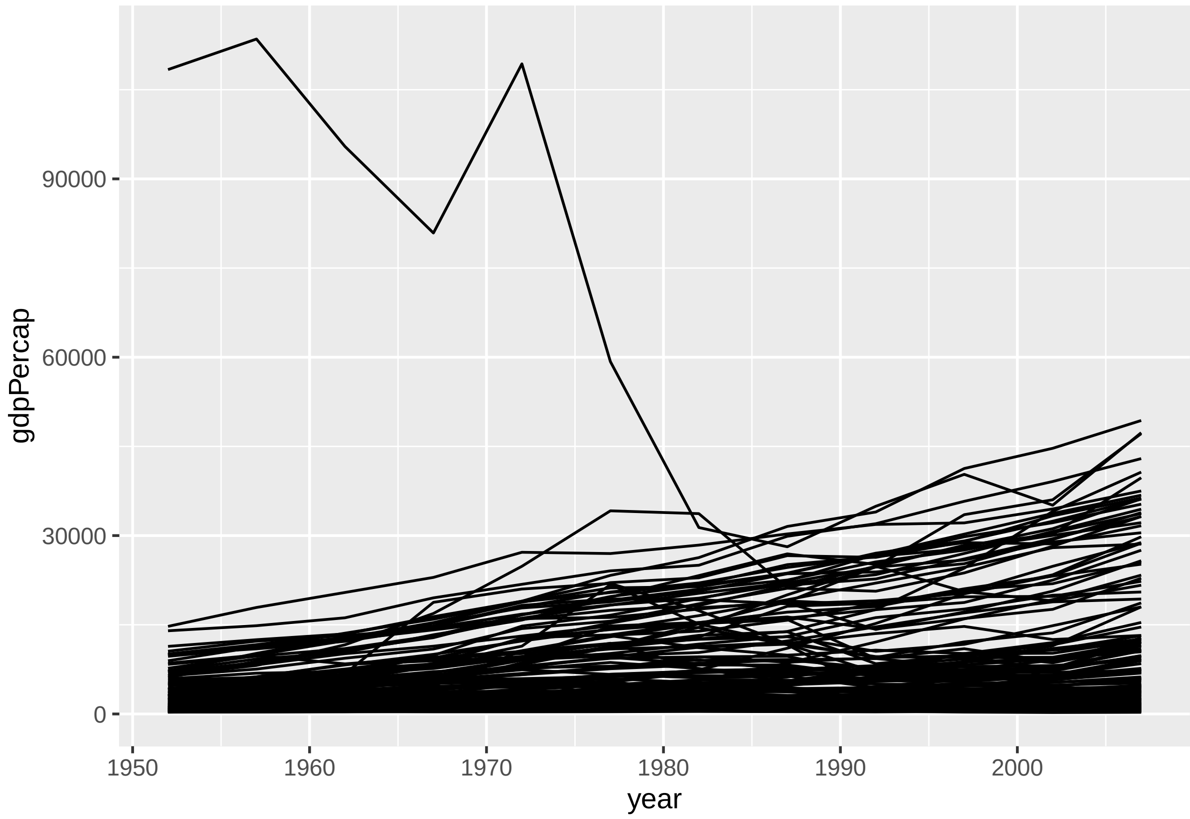
```
## `geom_smooth()` using formula 'y ~ x'
```



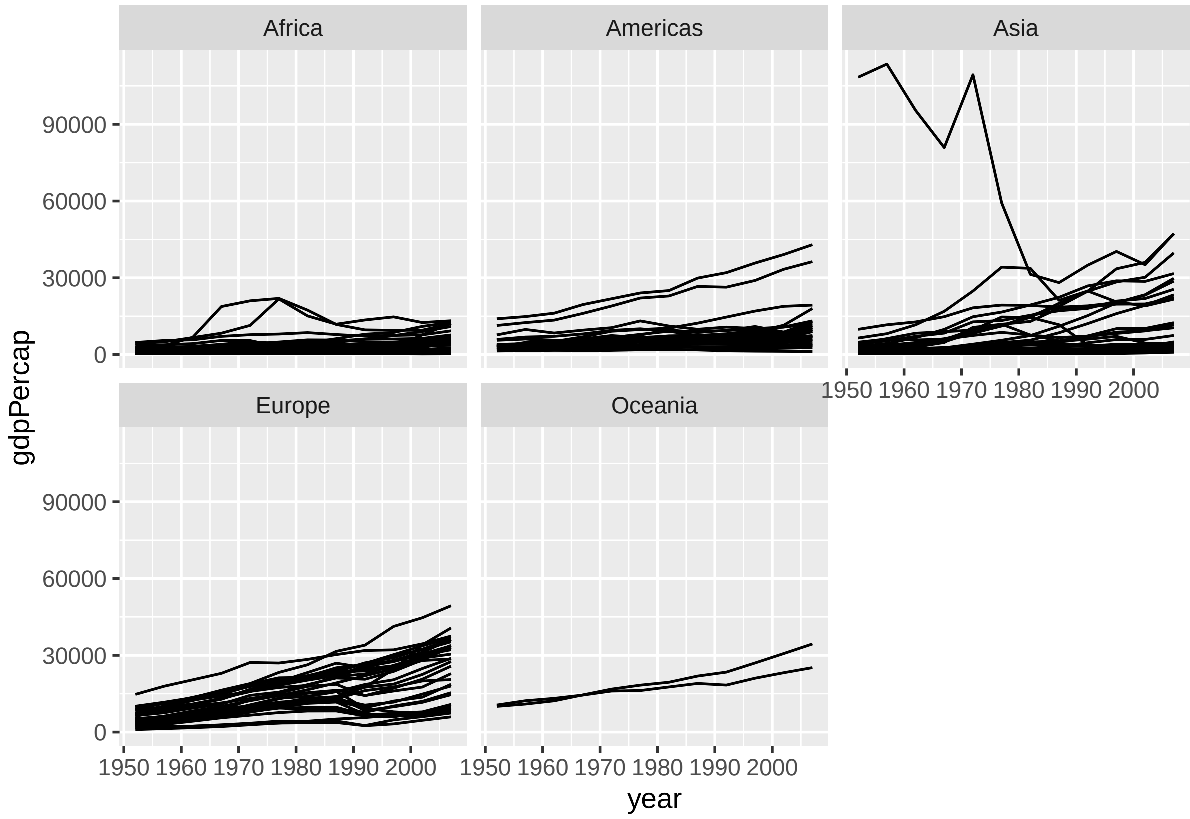
We get panel plots like this by specifying a layout formula in `facet_wrap()`, where ~ separates left and right sides of the fomula, where left usually means rows in a panel layout and right means columns. Since there's nothing on the left it just orders the continents. Color is no longer needed since the groups are separated.

There is a time variable in this dataset that we've basically been ignoring. How about a `gdpPercap` time series?

```
ggplot(gapminder, mapping = aes(x = year, y = gdpPercap, by = country)) +
    geom_line()
```




```
ggplot(gapminder, mapping = aes(x = year, y = gdpPercap, by = country)) +
    geom_line() +
    facet_wrap(~continent)
```



7 Exercises

Exercise 1.1: Try making the gapminder scatterplot with different smoothing methods: i) on the whole dataset and ii) by continent. You can find out the other methods by reading the help file, typing `?geom_smooth`

Exercise 1.2

Here's a way to remove some rows from a dataset:

```
gaps <- filter(gapminder, continent != "Oceania")
```

Try subsetting `gapminder` to just the first and last years (and also removing Oceania). Plot the density of log gdpPercap by continents (tip: `fill = continent`). They will overlap (tip: try `alpha = .5` to make them transparent). Make a 2-panel plot showing how these distributions changed between 1952 and 2007.

References

Healy, Kieran. 2018. *Data Visualization: A Practical Introduction*. Princeton University Press.

Preston, Samuel H. 1975. "The Changing Relation Between Mortality and Level of Economic Development." *Population Studies* 29 (2): 231–48.

Preston, S, Patrick Heuveline, and Michael Guillot. 2000. "Demography: Measuring and Modeling Population Processes. 2001." *Malden, MA: Blackwell Publishers*.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer.

Wilkinson, Leland. 2012. "The Grammar of Graphics." In *Handbook of Computational Statistics*, 375–414. Springer.