Biological Data Science with R

Stephen D. Turner

2018-12-18

Table of contents

Preface

This book was written as a companion to a series of courses introducing the essentials of biological data science with R. While this book was written with the accompanying live instruction in mind, this book can be used as a self-contained self study guide for quickly learning the essentials need to get started with R. The BDSR book and accompanying course introduces methods, tools, and software for reproducibly managing, manipulating, analyzing, and visualizing large-scale biological data using the R statistical computing environment. This book also covers essential statistical analysis, and advanced topics including survival analysis, predictive modeling, forecasting, and text mining.

This is not a *"Tool X"* or *"Software Y"* book. I want you to take away from this book and accompanying course the ability to use an extremely powerful scientific computing environment (R) to do many of the things that you'll do *across study designs and disciplines* – managing, manipulating, visualizing, and analyzing large, sometimes high-dimensional data. Regardless of your specific discipline you'll need the same computational know-how and data literacy to do the same kinds of basic tasks in each. This book might show you how to use specific tools here and there (e.g., DESeq2 for RNA-seq analysis (Love, Huber, and Anders 2014), ggtree for drawing phylogenetic trees (Yu et al. 2017), etc.), but these are not important – you probably won't be using the same specific software or methods 10 years from now, but you'll still use the same underlying data and computational foundation. That is the point of this series – to arm you with a basic foundation, and more importantly, to enable you to figure out how to use *this tool* or *that tool* on your own, when you need to.

This is not a statistics book. There is a short lesson on essential statistics using R in Chapter 7 but this short chapter offers neither a comprehensive background on underlying theory nor in-depth coverage of implementation strategies using R. Some general knowledge of statistics and study design is helpful, but isn't required for going through this book or taking the accompanying course.

There are no prerequisites to this book or the accompanying course. However, each chapter involves lots of hands-on practice coding, and you'll need to download and install required softwar and download required data. See the setup instructions in Appendix A.

Acknowledgements

This book is partially adapted from material we developed for the University of Virginia BIMS8382 graduate course . The material for this course was adapted from and/or inspired by Jenny Bryan's STAT545 course at UBC (Bryan 2019), Software Carpentry (Wilson 2014) and Data Carpentry (Teal et al. 2015) courses, David Robinson's *Variance Explained* blog (Robinson 2015), the ggtree vignettes (Yu 2022) *Tidy Text Mining with R* (Silge and Robinson 2017), and likely many others.

Part I

Core lessons

1 Basics

This chapter introduces the R environment and some of the most basic functionality aspects of R that are used through the remainder of the book. This section assumes little to no experience with statistical computing with R. This chapter introduces the very basic functionality in R, including variables, functions, and importing/inspecting data frames (tibbles).

1.1 RStudio

Let's start by learning about RStudio. **R** is the underlying statistical computing environment. **RStudio** is a graphical integrated development environment (IDE) that makes using R much easier.

- **Options:** First, let's change a few options. We'll only have to do this once. Under *Tools… Global Options…*:
  - Under *General*: Uncheck "Restore most recently opened project at startup"
  - Under *General*: Uncheck "Restore .RData into workspace at startup"
  - Under *General*: Set "Save workspace to .RData on exit:" to Never.
  - Under *General*: Set "Save workspace to .RData on exit:" to Never.
  - Under *R Markdown*: Uncheck "Show output inline for all R Markdown documents"
- Projects: first, start a new project in a new folder somewhere easy to remember. When we start reading in data it'll be important that the *code and the data are in the same place.* Creating a project creates an Rproj file that opens R running *in that folder.* This way, when you want to read in dataset *whatever.txt*, you just tell it the filename rather than a full path. This is critical for reproducibility, and we'll talk about that more later.
- Code that you type into the console is code that R executes. From here forward we will use the editor window to write a script that we can save to a file and run it again whenever we want to. We usually give it a `.R` extension, but it's just a plain text file. If you want to send commands from your editor to the console, use `CMD+Enter` (`Ctrl+Enter` on Windows).
- Anything after a `#` sign is a comment. Use them liberally to *comment your code.*

1.2 Basic operations

R can be used as a glorified calculator. Try typing this in directly into the console. Make sure you're typing into into the editor, not the console, and save your script. Use the run button, or press `CMD`+`Enter` (`Ctrl`+`Enter` on Windows).

```
2+2
```

```
[1] 4
```

```
5*4
```

```
[1] 20
```

```
2^3
```

```
[1] 8
```

R Knows order of operations and scientific notation.

```
2+3*4/(5+3)*15/2^2+3*4^2
```

```
[1] 55.6
```

```
5e4
```

```
[1] 50000
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Mostly similar to `=` but not always. Learn to

use `<-` as it is good programming practice. Using `=` in place of `<-` can lead to issues down the line. The keyboard shortcut for inserting the `<-` operator is `Alt-dash`.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

```
[1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg).

```
2.2 * weight_kg
```

```
[1] 121
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
[1] 127
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once. You can also use the little broom button in your environment pane to remove everything from your environment.

```
ls()
rm(weight_lb, weight_kg)
ls()
weight_lb # oops! you should get an error because weight_lb no longer exists!
```

Exercise 1

What are the values after each statement in the following?

```
mass <- 50              # mass?
age  <- 30              # age?
mass <- mass * 2        # mass?
age  <- age - 10        # age?
mass_index <- mass/age  # massIndex?
```

1.3 Functions

R has built-in functions.

```
# Notice that this is a comment.
# Anything behind a # is "commented out" and is not run.
sqrt(144)
```

```
[1] 12
```

```
log(1000)
```

```
[1] 6.91
```

Get help by typing a question mark in front of the function's name, or `help(functionname)`:

```
help(log)
?log
```

Note syntax highlighting when typing this into the editor. Also note how we pass *arguments* to functions. The `base=` part inside the parentheses is called an argument, and most functions use arguments. Arguments modify the behavior of the function. Functions some input (e.g., some data, an object) and other options to change what the function will return, or how to treat the data provided. Finally, see how you can *next* one function inside of another (here taking the square root of the log-base-10 of 1000).

```
log(1000)
```

```
[1] 6.91
```

```
log(1000, base=10)
```

```
[1] 3
```

```
log(1000, 10)
```

```
[1] 3
```

```
sqrt(log(1000, base=10))
```

```
[1] 1.73
```

Exercise 2

See `?abs` and calculate the square root of the log-base-10 of the absolute value of `-4*(2550-50)`. Answer should be 2.

1.4 Tibbles (data frames)

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. We are going to skip straight to the data structure you'll probably use most – the **tibble** (also known as the data frame). We use tibbles to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

We'll learn more about tibbles in Chapter 2.

2 Tibbles

There are *lots* of different basic data structures in R. If you take any kind of longer introduction to R you'll probably learn about arrays, lists, matrices, etc. Let's skip straight to the data structure you'll probably use most – the **data frame**. We use data frames to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

This lesson assumes a basic familiarity with R (see Chapter 1).

Recommended reading: Review the *Introduction* (10.1) and *Tibbles vs. data.frame* (10.3) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

2.1 Our data

The data we're going to look at is cleaned up version of a gene expression dataset from Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast (2008) *Mol Biol Cell* 19:352-367. This data is from a gene expression microarray, and in this paper the authors are examining the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a *single* nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. **Raise or lower their expression in response to growth rate**. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.

2. **Respond differently when different nutrients are being limited**. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

You can download the cleaned up version of the data here. The file is called **brauer2007_tidy.csv**. Later on we'll actually start with the original raw data (minimally processed) and manipulate it so that we can make it more amenable for analysis.

2.2 Reading in data

2.2.1 dplyr and readr

There are some built-in functions for reading in data in text files. These functions are *read-dot-something* – for example, `read.csv()` reads in comma-delimited text data; `read.delim()` reads in tab-delimited text, etc. We're going to read in data a little bit differently here using the readr package. When you load the readr package, you'll have access to very similar looking functions, named *read-underscore-something* – e.g., `read_csv()`. You have to have the readr package installed to access these functions. Compared to the base functions, they're *much* faster, they're good at guessing the types of data in the columns, they don't do some of the other silly things that the base functions do. We're going to use another package later on called dplyr, and if you have the dplyr package loaded as well, and you read in the data with readr, the data will display nicely.

First let's load those packages.

```
library(readr)
library(dplyr)
```

If you see a warning that looks like this: `Error in library(packageName) : there is no package called 'packageName'`, then you don't have the package installed correctly. See the setup chapter (Appendix A).

2.2.2 `read_csv()`

Now, let's actually load the data. You can get help for the import function with `?read_csv`. When we load data we assign it to a variable just like any other, and we can choose a name for that data. Since we're going to be referring to this data a lot, let's give it a short easy name to type. I'm going to call it `ydat`. Once we've loaded it we can type the name of the object itself (`ydat`) to see it printed to the screen.

```
ydat <- read_csv(file="data/brauer2007_tidy.csv")
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

Take a look at that output. The nice thing about loading dplyr and reading in data with readr is that data frames are displayed in a much more friendly way. This dataset has nearly 200,000 rows and 7 columns. When you import data this way and try to display the object in the console, instead of trying to display all 200,000 rows, you'll only see about 10 by default. Also, if you have so many columns that the data would wrap off the edge of your screen, those columns will not be displayed, but you'll see at the bottom of the output which, if any, columns were hidden from view. If you want to see the whole dataset, there are two ways to do this. First, you can click on the name of the data.frame in the **Environment** panel in RStudio. Or you could use the `View()` function (*with a capital V*).

```
View(ydat)
```

2.3 Inspecting data.frame objects

2.3.1 Built-in functions

There are several built-in functions that are useful for working with data frames.

- Content:
  - `head()`: shows the first few rows
  - `tail()`: shows the last few rows

- Size:
  - `dim()`: returns a 2-element vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()`: returns the number of rows
  - `ncol()`: returns the number of columns
- Summary:
  - `colnames()` (or just `names()`): returns the column names
  - `str()`: structure of the object and information about the class, length and content of each column
  - `summary()`: works differently depending on what kind of object you pass to it. Passing a data frame to the `summary()` function prints out useful summary statistics about numeric column (min, max, median, mean, etc.)

```
head(ydat)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport   mole~
2 <NA>   YNL095C         Glucose   0.05       0.28 biological process unk~ mole~
3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and peptid~ meta~
4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylation* RNA ~
5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*         t-SN~
6 PSP2   YML017W         Glucose   0.05      -0.69 biological process unk~ mole~
```

```
tail(ydat)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 DOA1   YKL213C         Uracil     0.3       0.14 ubiquitin-dependent pr~ mole~
2 KRE1   YNL322C         Uracil     0.3       0.28 cell wall organization~ stru~
3 MTL1   YGR023W         Uracil     0.3       0.27 cell wall organization~ mole~
4 KRE9   YJL174W         Uracil     0.3       0.43 cell wall organization~ mole~
5 UTH1   YKR042W         Uracil     0.3       0.19 mitochondrion organiza~ mole~
6 <NA>   YOL111C         Uracil     0.3       0.04 biological process unk~ mole~
```

```
dim(ydat)
```

```
[1] 198430      7
```

```
names(ydat)
```

```
[1] "symbol"          "systematic_name" "nutrient"        "rate"
[5] "expression"      "bp"              "mf"
```

```
str(ydat)
```

```
spc_tbl_ [198,430 x 7] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ symbol         : chr [1:198430] "SFB2" NA "QRI7" "CFT2" ...
 $ systematic_name: chr [1:198430] "YNL049C" "YNL095C" "YDL104C" "YLR115W" ...
 $ nutrient       : chr [1:198430] "Glucose" "Glucose" "Glucose" "Glucose" ...
 $ rate           : num [1:198430] 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 ...
 $ expression     : num [1:198430] -0.24 0.28 -0.02 -0.33 0.05 -0.69 -0.55 -0.75 -0.24 -0.16
 $ bp             : chr [1:198430] "ER to Golgi transport" "biological process unknown" "prot
 $ mf             : chr [1:198430] "molecular function unknown" "molecular function unknown"
 - attr(*, "spec")=
  .. cols(
  ..   symbol = col_character(),
  ..   systematic_name = col_character(),
  ..   nutrient = col_character(),
  ..   rate = col_double(),
  ..   expression = col_double(),
  ..   bp = col_character(),
  ..   mf = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

```
summary(ydat)
```

```
   symbol          systematic_name      nutrient             rate
Length:198430      Length:198430      Length:198430      Min.   :0.050
Class :character   Class :character   Class :character   1st Qu.:0.100
Mode  :character   Mode  :character   Mode  :character   Median :0.200
                                                         Mean   :0.175
                                                         3rd Qu.:0.250
                                                         Max.   :0.300
```

```
  expression          bp                 mf
Min.   :-6.50   Length:198430      Length:198430
1st Qu.:-0.29   Class :character   Class :character
Median : 0.00   Mode  :character   Mode  :character
Mean   : 0.00
3rd Qu.: 0.29
Max.   : 6.64
```

2.3.2 Other packages

The `glimpse()` function is available once you load the **dplyr** library, and it's like `str()` but its display is a little bit better.

```
glimpse(ydat)
```

```
Rows: 198,430
Columns: 7
$ symbol          <chr> "SFB2", NA, "QRI7", "CFT2", "SSO2", "PSP2", "RIB2", "V~
$ systematic_name <chr> "YNL049C", "YNL095C", "YDL104C", "YLR115W", "YMR183C",~
$ nutrient        <chr> "Glucose", "Glucose", "Glucose", "Glucose", "Glucose",~
$ rate            <dbl> 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, ~
$ expression      <dbl> -0.24, 0.28, -0.02, -0.33, 0.05, -0.69, -0.55, -0.75, ~
$ bp              <chr> "ER to Golgi transport", "biological process unknown",~
$ mf              <chr> "molecular function unknown", "molecular function unkn~
```

The **skimr** package has a nice function, skim, that provides summary statistics the user can skim quickly to understand your data. You can install it with `install.packages("skimr")` if you don't have it already.

```
library(skimr)
skim(ydat)
```

Table 2.1: Data summary

Name	ydat
Number of rows	198430
Number of columns	7
Column type frequency:	
character	5

numeric	2
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
symbol	47250	0.76	2	9	0	4210	0
systematic_name	0	1.00	5	9	0	5536	0
nutrient	0	1.00	6	9	0	6	0
bp	7663	0.96	7	82	0	880	0
mf	7663	0.96	11	125	0	1085	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
rate	0	1	0.18	0.09	0.05	0.10	0.2	0.25	0.30	
expression	0	1	0.00	0.67	-6.50	-0.29	0.0	0.29	6.64	

2.4 Accessing variables & subsetting data frames

We can access individual variables within a data frame using the `$` operator, e.g., `mydataframe$specificVariable`. Let's print out all the gene names in the data. Then let's calculate the average expression across all conditions, all genes (using the built-in `mean()` function).

```
# display all gene symbols
ydat$symbol
```

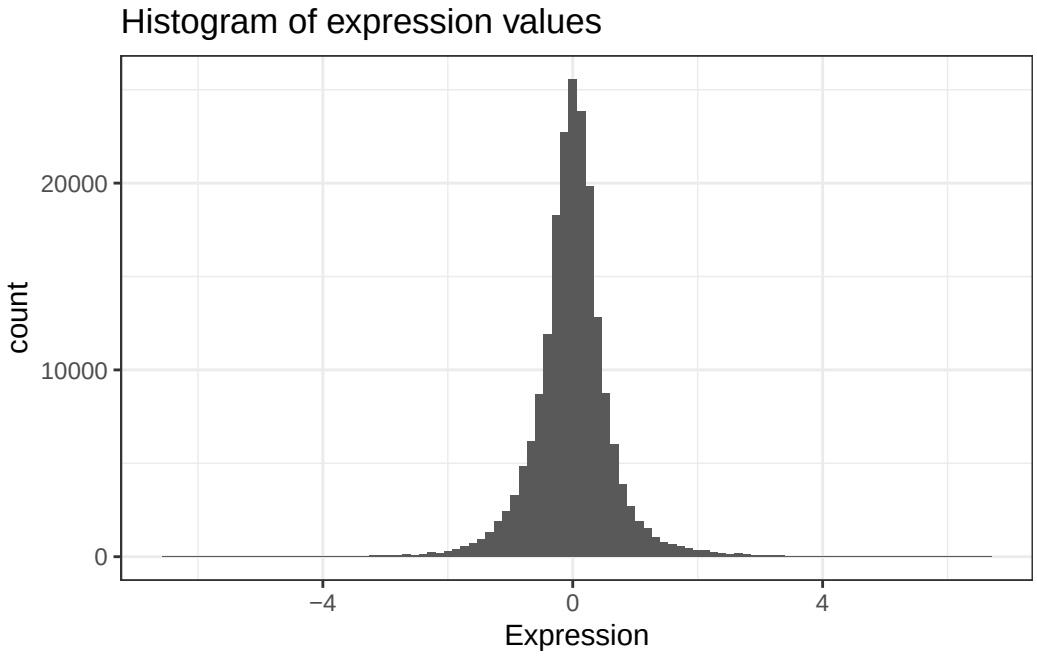
```
 [1] "SFB2"     NA         "QRI7"     "CFT2"     "SSO2"     "PSP2"
 [7] "RIB2"     "VMA13"    "EDC3"     "VPS5"     NA         "AMN1"
[13] "SCW11"    "DSE2"     "COX15"    "SPE1"     "MTF1"     "KSS1"
[19] NA         NA         "YAP7"     NA         "YVC1"     "CDC40"
[25] NA         "RMD1"     "PCL6"     "AI4"      "GGC1"     "SUL1"
[31] "RAD57"    NA         "PER1"     "YHC3"     "SGE1"     "HNM1"
[37] "SWI1"     "NAM8"     NA         "BGL2"     "ACT1"     NA
[43] "SFL1"     "OYE3"     "MMP1"     "MHT1"     "SUL2"     "IPP1"
```

```
 [49] "CWP1"     "SNF11"    "PEX25"    "ELO1"     NA         "CDC13"
 [55] "FKH1"     "SWD1"     NA         "HOF1"     "HOC1"     "BNI5"
 [61] "CSN12"    "PGS1"     "MLP2"     "HRP1"     NA         "SEC39"
 [67] "ECM31"    NA         NA         "ADE4"     "ABC1"     "DLD2"
 [73] "PHA2"     NA         "HAP3"     "MRPL23"   NA         NA
 [79] "MRPL16"   NA         NA         NA         NA         "AI3"
 [85] "COX1"     NA         "VAR1"     "COX3"     "COX2"     "AI5_BETA"
 [91] "AI2"      NA         NA         "GPI18"    "COS9"     NA
 [97] NA         "PRP46"    "XDJ1"     "SLG1"     "MAM3"     "AEP1"
[103] "UGO1"     NA         "RSC2"     "YAP1801"  "ZPR1"     "BCD1"
[109] "UBP10"    "SLD3"     "RLF2"     "LRO1"     NA         "ITR2"
[115] "ABP140"   "STT3"     "PTC2"     "STE20"    "HRD3"     "CWH43"
[121] "ASK10"    "MPE1"     "SWC3"     "TSA1"     "ADE17"    "GFD2"
[127] "PXR1"     NA         "BUD14"    "AUS1"     "NHX1"     "NTE1"
[133] NA         "KIN3"     "BUD4"     "SLI15"    "PMT4"     "AVT5"
[139] "CHS2"     "GPI13"    "KAP95"    "EFT2"     "EFT1"     "GAS1"
[145] "CYK3"     "COQ2"     "PSD1"     NA         "PAC1"     "SUR7"
[151] "RAX1"     "DFM1"     "RBD2"     NA         "YIP4"     "SRB2"
[157] "HOL1"     "MEP3"     NA         "FEN2"     NA         "RFT1"
[163] NA         "MCK1"     "GPI10"    "APT1"     NA         NA
[169] "CPT1"     "ERV29"    "SFK1"     NA         "SEC20"    "TIR4"
[175] NA         NA         "ARC35"    "SOL1"     "BIO2"     "ASC1"
[181] "RBG1"     "PTC4"     NA         "OXA1"     "SIT4"     "PUB1"
[187] "FPR4"     "FUN12"    "DPH2"     "DPS1"     "DLD1"     "ASN2"
[193] "TRM9"     "DED81"    "SRM1"     "SAM50"    "POP2"     "FAA4"
[199] NA         "CEM1"
 [ reached getOption("max.print") -- omitted 198230 entries ]
```

```r
#mean expression
mean(ydat$expression)
```

```
[1] 0.00337
```

Now that's not too interesting. This is the average gene expression across all genes, across all conditions. The data is actually scaled/centered around zero:



We might be interested in the average expression of genes with a particular biological function, and how that changes over different growth rates restricted by particular nutrients. This is the kind of thing we're going to do in the next section.

Exercise 1

1. What's the standard deviation expression (hint: get help on the `sd` function with `?sd`).
2. What's the range of rate represented in the data? (hint: `range()`).

2.5 BONUS: Preview to advanced manipulation

What if we wanted show the mean expression, standard deviation, and correlation between growth rate and expression, separately for each limiting nutrient, separately for each gene, for all genes involved in the leucine biosynthesis pathway?

```
ydat |>
  filter(bp=="leucine biosynthesis") |>
  group_by(nutrient, symbol) |>
  summarize(mean=mean(expression), sd=sd(expression), r=cor(rate, expression))
```

nutrient	symbol	mean	sd	r
Ammonia	LEU1	-0.82	0.39	0.66
Ammonia	LEU2	-0.54	0.38	-0.19
Ammonia	LEU4	-0.37	0.56	-0.67
Ammonia	LEU9	-1.01	0.64	0.87
Glucose	LEU1	-0.55	0.41	0.98
Glucose	LEU2	-0.39	0.33	0.90
Glucose	LEU4	1.09	1.01	-0.97
Glucose	LEU9	-0.17	0.35	0.35
Leucine	LEU1	2.70	1.08	-0.95
Leucine	LEU2	0.28	1.16	-0.97
Leucine	LEU4	0.80	1.06	-0.97
Leucine	LEU9	0.39	0.18	-0.77
Phosphate	LEU1	-0.43	0.27	0.95
Phosphate	LEU2	-0.26	0.19	0.70
Phosphate	LEU4	-0.99	0.11	0.24
Phosphate	LEU9	-1.12	0.53	0.90
Sulfate	LEU1	-1.17	0.34	0.98
Sulfate	LEU2	-0.96	0.30	0.57
Sulfate	LEU4	-0.24	0.43	-0.60
Sulfate	LEU9	-1.24	0.55	0.99
Uracil	LEU1	-0.74	0.73	0.89
Uracil	LEU2	0.18	0.13	-0.07
Uracil	LEU4	-0.65	0.44	0.77
Uracil	LEU9	-1.02	0.91	0.94

Neat eh? We'll learn how to do that in the advanced manipulation with dplyr lesson.

3 Data Manipulation

Data analysis involves a large amount of janitor work – munging and cleaning data to facilitate downstream data analysis. This lesson demonstrates techniques for advanced data manipulation and analysis with the split-apply-combine strategy. We will use the dplyr package in R to effectively manipulate and conditionally compute summary statistics over subsets of a "big" dataset containing many observations.

This lesson assumes a basic familiarity with R (Chapter 1) and data frames (Chapter 2).

Recommended reading: Review the *Introduction* (10.1) and *Tibbles vs. data.frame* (10.3) sections of the ***R for Data Science* book**. We will initially be using the `read_*` functions from the **readr** package. These functions load data into a *tibble* instead of R's traditional data.frame. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. These sections explain the few key small differences between traditional data.frames and tibbles.

3.1 Review

3.1.1 Our data

We're going to use the yeast gene expression dataset described on the data frames lesson in Chapter 2. This is a cleaned up version of a gene expression dataset from Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast (2008) *Mol Biol Cell* 19:352-367. This data is from a gene expression microarray, and in this paper the authors are examining the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a *single* nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. **Raise or lower their expression in response to growth rate**. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also

found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.

2. **Respond differently when different nutrients are being limited**. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

You can download the cleaned up version of the data here. The file is called **brauer2007_tidy.csv**. Later on we'll actually start with the original raw data (minimally processed) and manipulate it so that we can make it more amenable for analysis.

3.1.2 Reading in data

We need to load both the dplyr and readr packages for efficiently reading in and displaying this data. We're also going to use many other functions from the dplyr package. Make sure you have these packages installed as described on the setup chapter (Appendix A).

```
# Load packages
library(readr)
library(dplyr)

# Read in data
ydat <- read_csv(file="data/brauer2007_tidy.csv")

# Display the data
ydat

# Optionally, bring up the data in a viewer window
# View(ydat)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
```

```
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

3.2 The dplyr package

The dplyr package is a relatively new R package that makes data manipulation fast and easy. It imports functionality from another package called magrittr that allows you to chain commands together into a pipeline that will completely change the way you write R code such that you're writing code the way you're thinking about the problem.

When you read in data with the readr package (`read_csv()`) and you had the dplyr package loaded already, the data frame takes on this "special" class of data frames called a `tbl` (pronounced "tibble"), which you can see with `class(ydat)`. If you have other "regular" data frames in your workspace, the `as_tibble()` function will convert it into the special dplyr `tbl` that displays nicely (e.g.: `iris <- as_tibble(iris)`). You don't have to turn all your data frame objects into tibbles, but it does make working with large datasets a bit easier.

You can read more about tibbles in Tibbles chapter in *R for Data Science* or in the tibbles vignette. They keep most of the features of data frames, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors). You can read more about the differences between data frames and tibbles in this section of the tibbles vignette, but the major convenience for us concerns **printing** (aka displaying) a tibble to the screen. When you print (i.e., display) a tibble, it only shows the first 10 rows and all the columns that fit on one screen. It also prints an abbreviated description of the column type. You can control the default appearance with options:

- `options(tibble.print_max = n, tibble.print_min = m)`: if there are more than n rows, print only the first m rows. Use `options(tibble.print_max = Inf)` to always show all rows.
- `options(tibble.width = Inf)` will always print all columns, regardless of the width of the screen.

3.3 dplyr verbs

The dplyr package gives you a handful of useful **verbs** for managing data. On their own they don't do anything that base R can't do. Here are some of the *single-table* verbs we'll be working with in this lesson (single-table meaning that they only work on a single table – contrast that to *two-table* verbs used for joining data together, which we'll cover in a later lesson).

1. `filter()`

2. `select()`
3. `mutate()`
4. `arrange()`
5. `summarize()`
6. `group_by()`

They all take a data frame or tibble as their input for the first argument, and they all return a data frame or tibble as output.

3.3.1 filter()

If you want to filter **rows** of the data where some condition is true, use the `filter()` function.

1. The first argument is the data frame you want to filter, e.g. `filter(mydata, ....`
2. The second argument is a condition you must satisfy, e.g. `filter(ydat, symbol == "LEU1")`. If you want to satisfy *all* of multiple conditions, you can use the "and" operator, `&`. The "or" operator `|` (the pipe character, usually shift-backslash) will return a subset that meet *any* of the conditions.

- `==`: Equal to
- `!=`: Not equal to
- `>`, `>=`: Greater than, greater than or equal to
- `<`, `<=`: Less than, less than or equal to

Let's try it out. For this to work you have to have already loaded the dplyr package. Let's take a look at LEU1, a gene involved in leucine synthesis.

```
# First, make sure you've loaded the dplyr package
library(dplyr)

# Look at a single gene involved in leucine synthesis pathway
filter(ydat, symbol == "LEU1")
```

```
# A tibble: 36 x 7
   symbol systematic_name nutrient  rate expression bp                  mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>               <chr>
 1 LEU1   YGL009C         Glucose   0.05      -1.12 leucine biosynthesis 3-isop~
 2 LEU1   YGL009C         Glucose   0.1       -0.77 leucine biosynthesis 3-isop~
 3 LEU1   YGL009C         Glucose   0.15      -0.67 leucine biosynthesis 3-isop~
 4 LEU1   YGL009C         Glucose   0.2       -0.59 leucine biosynthesis 3-isop~
 5 LEU1   YGL009C         Glucose   0.25      -0.2  leucine biosynthesis 3-isop~
```

```
 6 LEU1   YGL009C         Glucose   0.3        0.03 leucine biosynthesis 3-isop~
 7 LEU1   YGL009C         Ammonia   0.05      -0.76 leucine biosynthesis 3-isop~
 8 LEU1   YGL009C         Ammonia   0.1       -1.17 leucine biosynthesis 3-isop~
 9 LEU1   YGL009C         Ammonia   0.15      -1.2  leucine biosynthesis 3-isop~
10 LEU1   YGL009C         Ammonia   0.2       -1.02 leucine biosynthesis 3-isop~
# i 26 more rows
```

```
  # Optionally, bring that result up in a View window
  # View(filter(ydat, symbol == "LEU1"))

  # Look at multiple genes
  filter(ydat, symbol=="LEU1" | symbol=="ADH2")
```

```
# A tibble: 72 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 LEU1   YGL009C         Glucose   0.05      -1.12 leucine biosynthesis 3-isop~
 2 ADH2   YMR303C         Glucose   0.05       6.28 fermentation*        alcoho~
 3 LEU1   YGL009C         Glucose   0.1       -0.77 leucine biosynthesis 3-isop~
 4 ADH2   YMR303C         Glucose   0.1        5.81 fermentation*        alcoho~
 5 LEU1   YGL009C         Glucose   0.15      -0.67 leucine biosynthesis 3-isop~
 6 ADH2   YMR303C         Glucose   0.15       5.64 fermentation*        alcoho~
 7 LEU1   YGL009C         Glucose   0.2       -0.59 leucine biosynthesis 3-isop~
 8 ADH2   YMR303C         Glucose   0.2        5.1  fermentation*        alcoho~
 9 LEU1   YGL009C         Glucose   0.25      -0.2  leucine biosynthesis 3-isop~
10 ADH2   YMR303C         Glucose   0.25       1.89 fermentation*        alcoho~
# i 62 more rows
```

```
  # Look at LEU1 expression at a low growth rate due to nutrient depletion
  # Notice how LEU1 is highly upregulated when leucine is depleted!
  filter(ydat, symbol=="LEU1" & rate==.05)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient   rate expression bp                    mf
  <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
1 LEU1   YGL009C         Glucose    0.05      -1.12 leucine biosynthesis 3-isop~
2 LEU1   YGL009C         Ammonia    0.05      -0.76 leucine biosynthesis 3-isop~
3 LEU1   YGL009C         Phosphate  0.05      -0.81 leucine biosynthesis 3-isop~
4 LEU1   YGL009C         Sulfate    0.05      -1.57 leucine biosynthesis 3-isop~
5 LEU1   YGL009C         Leucine    0.05       3.84 leucine biosynthesis 3-isop~
6 LEU1   YGL009C         Uracil     0.05      -2.07 leucine biosynthesis 3-isop~
```

```
# But expression goes back down when the growth/nutrient restriction is relaxed
filter(ydat, symbol=="LEU1" & rate==.3)
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient   rate expression bp                   mf
  <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                <chr>
1 LEU1   YGL009C         Glucose     0.3       0.03 leucine biosynthesis 3-isop~
2 LEU1   YGL009C         Ammonia     0.3      -0.22 leucine biosynthesis 3-isop~
3 LEU1   YGL009C         Phosphate   0.3      -0.07 leucine biosynthesis 3-isop~
4 LEU1   YGL009C         Sulfate     0.3      -0.76 leucine biosynthesis 3-isop~
5 LEU1   YGL009C         Leucine     0.3       0.87 leucine biosynthesis 3-isop~
6 LEU1   YGL009C         Uracil      0.3      -0.16 leucine biosynthesis 3-isop~
```

```
# Show only stats for LEU1 and Leucine depletion.
# LEU1 expression starts off high and drops
filter(ydat, symbol=="LEU1" & nutrient=="Leucine")
```

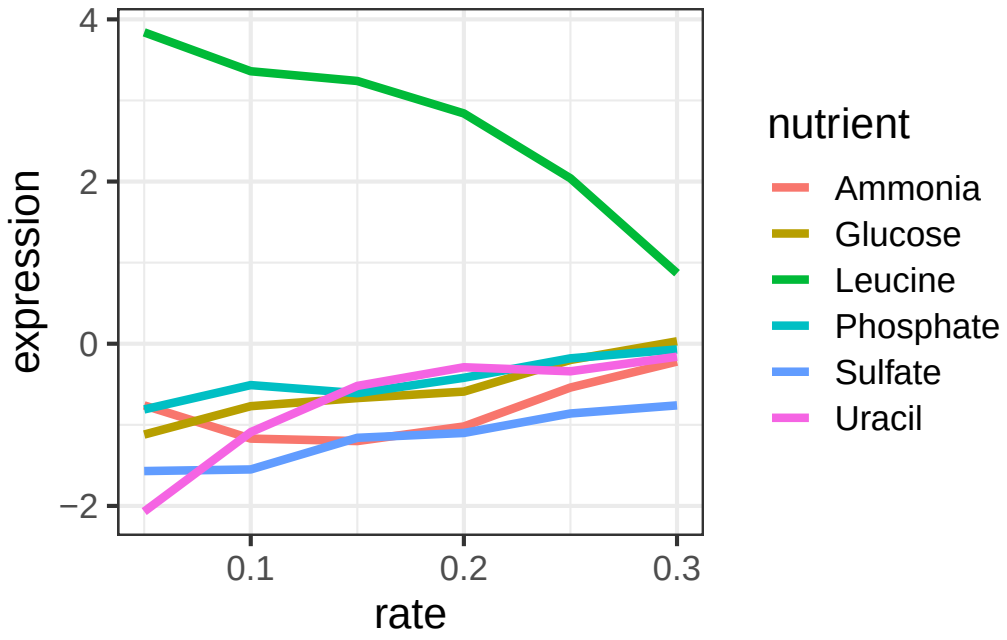
```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                   mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                <chr>
1 LEU1   YGL009C         Leucine   0.05       3.84 leucine biosynthesis 3-isopr~
2 LEU1   YGL009C         Leucine   0.1        3.36 leucine biosynthesis 3-isopr~
3 LEU1   YGL009C         Leucine   0.15       3.24 leucine biosynthesis 3-isopr~
4 LEU1   YGL009C         Leucine   0.2        2.84 leucine biosynthesis 3-isopr~
5 LEU1   YGL009C         Leucine   0.25       2.04 leucine biosynthesis 3-isopr~
6 LEU1   YGL009C         Leucine   0.3        0.87 leucine biosynthesis 3-isopr~
```

```
# What about LEU1 expression with other nutrients being depleted?
filter(ydat, symbol=="LEU1" & nutrient=="Glucose")
```

```
# A tibble: 6 x 7
  symbol systematic_name nutrient  rate expression bp                   mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                <chr>
1 LEU1   YGL009C         Glucose   0.05      -1.12 leucine biosynthesis 3-isopr~
2 LEU1   YGL009C         Glucose   0.1       -0.77 leucine biosynthesis 3-isopr~
3 LEU1   YGL009C         Glucose   0.15      -0.67 leucine biosynthesis 3-isopr~
4 LEU1   YGL009C         Glucose   0.2       -0.59 leucine biosynthesis 3-isopr~
5 LEU1   YGL009C         Glucose   0.25      -0.2  leucine biosynthesis 3-isopr~
6 LEU1   YGL009C         Glucose   0.3        0.03 leucine biosynthesis 3-isopr~
```

Let's look at this graphically. Don't worry about what these commands are doing just yet - we'll cover that later on when we talk about ggplot2. Here's I'm taking the filtered dataset containing just expression estimates for LEU1 where I have 36 rows (one for each of 6 nutrients × 6 growth rates), and I'm *piping* that dataset to the plotting function, where I'm plotting rate on the x-axis, expression on the y-axis, mapping the value of nutrient to the color, and using a line plot to display the data.

```
library(ggplot2)
filter(ydat, symbol=="LEU1") |>
  ggplot(aes(rate, expression, colour=nutrient)) + geom_line(lwd=1.5)
```



Look closely at that! LEU1 is *highly expressed* when starved of leucine because the cell has to synthesize its own! And as the amount of leucine in the environment (the growth *rate*) increases, the cell can worry less about synthesizing leucine, so LEU1 expression goes back down. Consequently the cell can devote more energy into other functions, and we see other genes' expression very slightly raising.

Exercise 1

1. Display the data where the gene ontology biological process (the `bp` variable) is "leucine biosynthesis" (case-sensitive) *and* the limiting nutrient was Leucine. (Answer should return a 24-by-7 data frame – 4 genes × 6 growth rates).
2. Gene/rate combinations had high expression (in the top 1% of expressed genes)?

> *Hint:* see `?quantile` and try `quantile(ydat$expression, probs=.99)` to see the expression value which is higher than 99% of all the data, then `filter()` based on that. Try wrapping your answer with a `View()` function so you can see the whole thing. What does it look like those genes are doing? Answer should return a 1971-by-7 data frame.

3.3.1.1 Aside: Writing Data to File

What we've done up to this point is read in data from a file (`read_csv(...)`), and assigning that to an object in our *workspace* (`ydat <- ...`). When we run operations like `filter()` on our data, consider two things:

1. The `ydat` object in our workspace is not being modified directly. That is, we can `filter(ydat, ...)`, and a result is returned to the screen, but `ydat` remains the same. This effect is similar to what we demonstrated in our first session.

```
# Assign the value '50' to the weight object.
weight <- 50

# Print out weight to the screen (50)
weight

# What's the value of weight plus 10?
weight + 10

# Weight is still 50
weight

# Weight is only modified if we *reassign* weight to the modified value
weight <- weight+10
# Weight is now 60
weight
```

2. More importantly, the *data file on disk* (`data/brauer2007_tidy.csv`) is *never* modified. No matter what we do to ydat, the file is never modified. If we want to *save* the result of an operation to a file on disk, we can assign the result of an operation to an object, and `write_csv` that object to disk. See the help for `?write_csv` (note, `write_csv()` with an underscore is part of the **readr** package – not to be confused with the built-in `write.csv()` function).

```
# What's the result of this filter operation?
filter(ydat, nutrient=="Leucine" & bp=="leucine biosynthesis")

# Assign the result to a new object
leudat <- filter(ydat, nutrient=="Leucine" & bp=="leucine biosynthesis")

# Write that out to disk
write_csv(leudat, "leucinedata.csv")
```

Note that this is different than saving your *entire workspace to an Rdata file*, which would contain all the objects we've created (weight, ydat, leudat, etc).

3.3.2 select()

The `filter()` function allows you to return only certain *rows* matching a condition. The `select()` function returns only certain *columns*. The first argument is the data, and subsequent arguments are the columns you want.

```
# Select just the symbol and systematic_name
select(ydat, symbol, systematic_name)
```

```
# A tibble: 198,430 x 2
   symbol systematic_name
   <chr>  <chr>
 1 SFB2   YNL049C
 2 <NA>   YNL095C
 3 QRI7   YDL104C
 4 CFT2   YLR115W
 5 SSO2   YMR183C
 6 PSP2   YML017W
 7 RIB2   YOL066C
 8 VMA13  YPR036W
 9 EDC3   YEL015W
10 VPS5   YOR069W
# i 198,420 more rows
```

```
# Alternatively, just remove columns. Remove the bp and mf columns.
select(ydat, -bp, -mf)
```

```
# A tibble: 198,430 x 5
   symbol systematic_name nutrient  rate expression
   <chr>  <chr>           <chr>    <dbl>      <dbl>
 1 SFB2   YNL049C         Glucose   0.05      -0.24
 2 <NA>   YNL095C         Glucose   0.05       0.28
 3 QRI7   YDL104C         Glucose   0.05      -0.02
 4 CFT2   YLR115W         Glucose   0.05      -0.33
 5 SSO2   YMR183C         Glucose   0.05       0.05
 6 PSP2   YML017W         Glucose   0.05      -0.69
 7 RIB2   YOL066C         Glucose   0.05      -0.55
 8 VMA13  YPR036W         Glucose   0.05      -0.75
 9 EDC3   YEL015W         Glucose   0.05      -0.24
10 VPS5   YOR069W         Glucose   0.05      -0.16
# i 198,420 more rows
```

```
# Notice that the original data doesn't change!
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

Notice above how the original data doesn't change. We're selecting out only certain columns of interest and throwing away columns we don't care about. If we wanted to *keep* this data, we would need to *reassign* the result of the `select()` operation to a new object. Let's make a new object called `nogo` that does not contain the GO annotations. Notice again how the original data is unchanged.

```
# create a new dataset without the go annotations.
nogo <- select(ydat, -bp, -mf)
```

```
nogo
```

```
# A tibble: 198,430 x 5
   symbol systematic_name nutrient  rate expression
   <chr>  <chr>           <chr>    <dbl>      <dbl>
 1 SFB2   YNL049C         Glucose   0.05      -0.24
 2 <NA>   YNL095C         Glucose   0.05       0.28
 3 QRI7   YDL104C         Glucose   0.05      -0.02
 4 CFT2   YLR115W         Glucose   0.05      -0.33
 5 SSO2   YMR183C         Glucose   0.05       0.05
 6 PSP2   YML017W         Glucose   0.05      -0.69
 7 RIB2   YOL066C         Glucose   0.05      -0.55
 8 VMA13  YPR036W         Glucose   0.05      -0.75
 9 EDC3   YEL015W         Glucose   0.05      -0.24
10 VPS5   YOR069W         Glucose   0.05      -0.16
# i 198,420 more rows
```

```
# we could filter this new dataset
filter(nogo, symbol=="LEU1" & rate==.05)
```

```
# A tibble: 6 x 5
  symbol systematic_name nutrient   rate expression
  <chr>  <chr>           <chr>     <dbl>      <dbl>
1 LEU1   YGL009C         Glucose    0.05      -1.12
2 LEU1   YGL009C         Ammonia    0.05      -0.76
3 LEU1   YGL009C         Phosphate  0.05      -0.81
4 LEU1   YGL009C         Sulfate    0.05      -1.57
5 LEU1   YGL009C         Leucine    0.05       3.84
6 LEU1   YGL009C         Uracil     0.05      -2.07
```

```
# Notice how the original data is unchanged - still have all 7 columns
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
```

```
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

3.3.3 mutate()

The `mutate()` function adds new columns to the data. Remember, it doesn't actually modify the data frame you're operating on, and the result is transient unless you assign it to a new object or reassign it back to itself (generally, not always a good practice).

The expression level reported here is the log_2 of the sample signal divided by the signal in the reference channel, where the reference RNA for all samples was taken from the glucose-limited chemostat grown at a dilution rate of 0.25 h^{-1}. Let's mutate this data to add a new variable called "signal" that's the actual raw signal ratio instead of the log-transformed signal.

```
mutate(nogo, signal=2^expression)
```

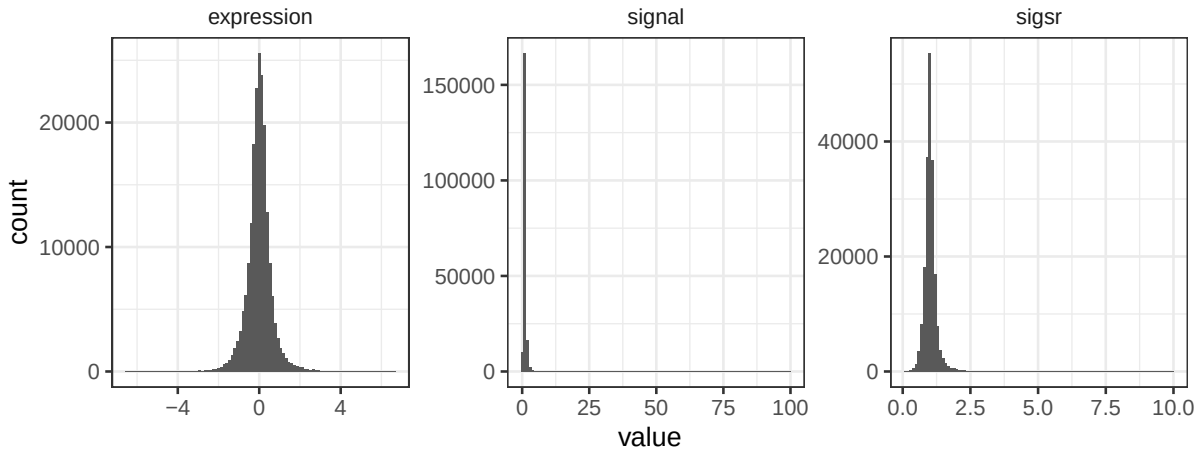
Mutate has a nice little feature too in that it's "lazy." You can mutate and add one variable, then continue mutating to add more variables based on that variable. Let's make another column that's the square root of the signal ratio.

```
mutate(nogo, signal=2^expression, sigsr=sqrt(signal))
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression signal sigsr
   <chr>  <chr>           <chr>    <dbl>      <dbl>  <dbl> <dbl>
 1 SFB2   YNL049C         Glucose   0.05      -0.24  0.847 0.920
 2 <NA>   YNL095C         Glucose   0.05       0.28  1.21  1.10
 3 QRI7   YDL104C         Glucose   0.05      -0.02  0.986 0.993
 4 CFT2   YLR115W         Glucose   0.05      -0.33  0.796 0.892
 5 SSO2   YMR183C         Glucose   0.05       0.05  1.04  1.02
 6 PSP2   YML017W         Glucose   0.05      -0.69  0.620 0.787
 7 RIB2   YOL066C         Glucose   0.05      -0.55  0.683 0.826
 8 VMA13  YPR036W         Glucose   0.05      -0.75  0.595 0.771
 9 EDC3   YEL015W         Glucose   0.05      -0.24  0.847 0.920
10 VPS5   YOR069W         Glucose   0.05      -0.16  0.895 0.946
# i 198,420 more rows
```

Again, don't worry about the code here to make the plot – we'll learn about this later. Why do you think we log-transform the data prior to analysis?

```
library(tidyr)
mutate(nogo, signal=2^expression, sigsr=sqrt(signal)) |>
  gather(unit, value, expression:sigsr) |>
  ggplot(aes(value)) + geom_histogram(bins=100) + facet_wrap(~unit, scales="free")
```



3.3.4 arrange()

The `arrange()` function does what it sounds like. It takes a data frame or tbl and arranges (or sorts) by column(s) of interest. The first argument is the data, and subsequent arguments are columns to sort on. Use the `desc()` function to arrange by descending.

```
# arrange by gene symbol
arrange(ydat, symbol)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                    mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                 <chr>
 1 AAC1   YMR056C         Glucose   0.05        1.5  aerobic respiration* ATP:AD~
 2 AAC1   YMR056C         Glucose   0.1         1.54 aerobic respiration* ATP:AD~
 3 AAC1   YMR056C         Glucose   0.15        1.16 aerobic respiration* ATP:AD~
 4 AAC1   YMR056C         Glucose   0.2         1.04 aerobic respiration* ATP:AD~
 5 AAC1   YMR056C         Glucose   0.25        0.84 aerobic respiration* ATP:AD~
 6 AAC1   YMR056C         Glucose   0.3         0.01 aerobic respiration* ATP:AD~
 7 AAC1   YMR056C         Ammonia   0.05        0.8  aerobic respiration* ATP:AD~
```

```
 8 AAC1   YMR056C         Ammonia   0.1        1.47 aerobic respiration* ATP:AD~
 9 AAC1   YMR056C         Ammonia   0.15       0.97 aerobic respiration* ATP:AD~
10 AAC1   YMR056C         Ammonia   0.2        0.76 aerobic respiration* ATP:AD~
# i 198,420 more rows
```

```
# arrange by expression (default: increasing)
arrange(ydat, expression)
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient   rate expression bp                    mf
   <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
 1 SUL1   YBR294W         Phosphate  0.05      -6.5  sulfate transport     sulf~
 2 SUL1   YBR294W         Phosphate  0.1       -6.34 sulfate transport     sulf~
 3 ADH2   YMR303C         Phosphate  0.1       -6.15 fermentation*         alco~
 4 ADH2   YMR303C         Phosphate  0.3       -6.04 fermentation*         alco~
 5 ADH2   YMR303C         Phosphate  0.25      -5.89 fermentation*         alco~
 6 SUL1   YBR294W         Uracil     0.05      -5.55 sulfate transport     sulf~
 7 SFC1   YJR095W         Phosphate  0.2       -5.52 fumarate transport*   succ~
 8 JEN1   YKL217W         Phosphate  0.3       -5.44 lactate transport     lact~
 9 MHT1   YLL062C         Phosphate  0.05      -5.36 sulfur amino acid me~ homo~
10 SFC1   YJR095W         Phosphate  0.25      -5.35 fumarate transport*   succ~
# i 198,420 more rows
```

```
# arrange by decreasing expression
arrange(ydat, desc(expression))
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient   rate expression bp                    mf
   <chr>  <chr>           <chr>     <dbl>      <dbl> <chr>                 <chr>
 1 GAP1   YKR039W         Ammonia    0.05       6.64 amino acid transport* L-pr~
 2 DAL5   YJR152W         Ammonia    0.05       6.64 allantoate transport  alla~
 3 GAP1   YKR039W         Ammonia    0.1        6.64 amino acid transport* L-pr~
 4 DAL5   YJR152W         Ammonia    0.1        6.64 allantoate transport  alla~
 5 DAL5   YJR152W         Ammonia    0.15       6.64 allantoate transport  alla~
 6 DAL5   YJR152W         Ammonia    0.2        6.64 allantoate transport  alla~
 7 DAL5   YJR152W         Ammonia    0.25       6.64 allantoate transport  alla~
 8 DAL5   YJR152W         Ammonia    0.3        6.64 allantoate transport  alla~
 9 GIT1   YCR098C         Phosphate  0.05       6.64 glycerophosphodieste~ glyc~
10 PHM6   YDR281C         Phosphate  0.05       6.64 biological process u~ mole~
# i 198,420 more rows
```

Exercise 2

1. First, re-run the command you used above to filter the data for genes involved in the "leucine biosynthesis" biological process *and* where the limiting nutrient is Leucine.
2. Wrap this entire filtered result with a call to `arrange()` where you'll arrange the result of #1 by the gene symbol.
3. Wrap this entire result in a `View()` statement so you can see the entire result.

3.3.5 summarize()

The `summarize()` function summarizes multiple values to a single value. On its own the `summarize()` function doesn't seem to be all that useful. The dplyr package provides a few convenience functions called `n()` and `n_distinct()` that tell you the number of observations or the number of distinct values of a particular variable.

Notice that summarize takes a data frame and returns a data frame. In this case it's a 1x1 data frame with a single row and a single column. The name of the column, by default is whatever the expression was used to summarize the data. This usually isn't pretty, and if we wanted to work with this resulting data frame later on, we'd want to name that returned value something easier to deal with.

```
# Get the mean expression for all genes
summarize(ydat, mean(expression))
```

```
# A tibble: 1 x 1
  `mean(expression)`
              <dbl>
1            0.00337
```

```
# Use a more friendly name, e.g., meanexp, or whatever you want to call it.
summarize(ydat, meanexp=mean(expression))
```

```
# A tibble: 1 x 1
  meanexp
    <dbl>
1 0.00337
```

```
# Measure the correlation between rate and expression
summarize(ydat, r=cor(rate, expression))
```

```
# A tibble: 1 x 1
        r
    <dbl>
1 -0.0220
```

```
# Get the number of observations
summarize(ydat, n())
```

```
# A tibble: 1 x 1
   `n()`
   <int>
1 198430
```

```
# The number of distinct gene symbols in the data
summarize(ydat, n_distinct(symbol))
```

```
# A tibble: 1 x 1
  `n_distinct(symbol)`
                 <int>
1                 4211
```

3.3.6 group_by()

We saw that `summarize()` isn't that useful on its own. Neither is `group_by()` All this does is takes an existing data frame and coverts it into a grouped data frame where operations are performed by group.

```
ydat
```

```
# A tibble: 198,430 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
```

```
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

```
  group_by(ydat, nutrient)
```

```
# A tibble: 198,430 x 7
# Groups:   nutrient [6]
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

```
  group_by(ydat, nutrient, rate)
```

```
# A tibble: 198,430 x 7
# Groups:   nutrient, rate [36]
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Glucose   0.05      -0.24 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Glucose   0.05       0.28 biological process un~ mole~
 3 QRI7   YDL104C         Glucose   0.05      -0.02 proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Glucose   0.05      -0.33 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Glucose   0.05       0.05 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Glucose   0.05      -0.69 biological process un~ mole~
```

```
 7 RIB2   YOL066C         Glucose   0.05      -0.55 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Glucose   0.05      -0.75 vacuolar acidification hydr~
 9 EDC3   YEL015W         Glucose   0.05      -0.24 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Glucose   0.05      -0.16 protein retention in ~ prot~
# i 198,420 more rows
```

The real power comes in where `group_by()` and `summarize()` are used together. First, write the `group_by()` statement. Then wrap the result of that with a call to `summarize()`.

```
# Get the mean expression for each gene
# group_by(ydat, symbol)
summarize(group_by(ydat, symbol), meanexp=mean(expression))
```

```
# A tibble: 4,211 x 2
   symbol  meanexp
   <chr>     <dbl>
 1 AAC1     0.529
 2 AAC3    -0.216
 3 AAD10    0.438
 4 AAD14   -0.0717
 5 AAD16    0.242
 6 AAD4    -0.792
 7 AAD6     0.290
 8 AAH1     0.0461
 9 AAP1    -0.00361
10 AAP1'   -0.421
# i 4,201 more rows
```

```
# Get the correlation between rate and expression for each nutrient
# group_by(ydat, nutrient)
summarize(group_by(ydat, nutrient), r=cor(rate, expression))
```

```
# A tibble: 6 x 2
  nutrient        r
  <chr>       <dbl>
1 Ammonia   -0.0175
2 Glucose   -0.0112
3 Leucine   -0.0384
4 Phosphate -0.0194
5 Sulfate   -0.0166
6 Uracil    -0.0353
```

3.4 The pipe: |>

3.4.1 How |> works

This is where things get awesome. The dplyr package imports functionality from the magrittr package that lets you *pipe* the output of one function to the input of another, so you can avoid nesting functions. It looks like this: `|>`. You don't have to load the magrittr package to use it since dplyr imports its functionality when you load the dplyr package.

Here's the simplest way to use it. Remember the `tail()` function. It expects a data frame as input, and the next argument is the number of lines to print. These two commands are identical:

```
tail(ydat, 5)
```

```
# A tibble: 5 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 KRE1   YNL322C         Uracil     0.3       0.28 cell wall organization~ stru~
2 MTL1   YGR023W         Uracil     0.3       0.27 cell wall organization~ mole~
3 KRE9   YJL174W         Uracil     0.3       0.43 cell wall organization~ mole~
4 UTH1   YKR042W         Uracil     0.3       0.19 mitochondrion organiza~ mole~
5 <NA>   YOL111C         Uracil     0.3       0.04 biological process unk~ mole~
```

```
ydat |> tail(5)
```

```
# A tibble: 5 x 7
  symbol systematic_name nutrient  rate expression bp                      mf
  <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                   <chr>
1 KRE1   YNL322C         Uracil     0.3       0.28 cell wall organization~ stru~
2 MTL1   YGR023W         Uracil     0.3       0.27 cell wall organization~ mole~
3 KRE9   YJL174W         Uracil     0.3       0.43 cell wall organization~ mole~
4 UTH1   YKR042W         Uracil     0.3       0.19 mitochondrion organiza~ mole~
5 <NA>   YOL111C         Uracil     0.3       0.04 biological process unk~ mole~
```

Let's use one of the dplyr verbs.

```
filter(ydat, nutrient=="Leucine")
```

```
# A tibble: 33,178 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Leucine   0.05       0.18 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Leucine   0.05       0.16 biological process un~ mole~
 3 QRI7   YDL104C         Leucine   0.05      -0.3  proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Leucine   0.05      -0.27 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Leucine   0.05      -0.59 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Leucine   0.05      -0.17 biological process un~ mole~
 7 RIB2   YOL066C         Leucine   0.05      -0.02 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Leucine   0.05      -0.11 vacuolar acidification hydr~
 9 EDC3   YEL015W         Leucine   0.05       0.12 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Leucine   0.05      -0.2  protein retention in ~ prot~
# i 33,168 more rows
```

```
ydat |> filter(nutrient=="Leucine")
```

```
# A tibble: 33,178 x 7
   symbol systematic_name nutrient  rate expression bp                     mf
   <chr>  <chr>           <chr>    <dbl>      <dbl> <chr>                  <chr>
 1 SFB2   YNL049C         Leucine   0.05       0.18 ER to Golgi transport  mole~
 2 <NA>   YNL095C         Leucine   0.05       0.16 biological process un~ mole~
 3 QRI7   YDL104C         Leucine   0.05      -0.3  proteolysis and pepti~ meta~
 4 CFT2   YLR115W         Leucine   0.05      -0.27 mRNA polyadenylylatio~ RNA ~
 5 SSO2   YMR183C         Leucine   0.05      -0.59 vesicle fusion*        t-SN~
 6 PSP2   YML017W         Leucine   0.05      -0.17 biological process un~ mole~
 7 RIB2   YOL066C         Leucine   0.05      -0.02 riboflavin biosynthes~ pseu~
 8 VMA13  YPR036W         Leucine   0.05      -0.11 vacuolar acidification hydr~
 9 EDC3   YEL015W         Leucine   0.05       0.12 deadenylylation-indep~ mole~
10 VPS5   YOR069W         Leucine   0.05      -0.2  protein retention in ~ prot~
# i 33,168 more rows
```

3.4.2 Nesting versus |>

So what?

Now, think about this for a minute. What if we wanted to get the correlation between the growth rate and expression separately for each limiting nutrient only for genes in the leucine biosynthesis pathway, and return a sorted list of those correlation coeffients rounded to two digits? Mentally we would do something like this:

0. Take the `ydat` dataset
1. *then* `filter()` it for genes in the leucine biosynthesis pathway
2. *then* `group_by()` the limiting nutrient
3. *then* `summarize()` to get the correlation (`cor()`) between rate and expression
4. *then* `mutate()` to round the result of the above calculation to two significant digits
5. *then* `arrange()` by the rounded correlation coefficient above

But in code, it gets ugly. First, take the `ydat` dataset

```
ydat
```

then `filter()` it for genes in the leucine biosynthesis pathway

```
filter(ydat, bp=="leucine biosynthesis")
```

then `group_by()` the limiting nutrient

```
group_by(filter(ydat, bp=="leucine biosynthesis"), nutrient)
```

then `summarize()` to get the correlation (`cor()`) between rate and expression

```
summarize(group_by(filter(ydat, bp == "leucine biosynthesis"), nutrient), r = cor(rate,
    expression))
```

then `mutate()` to round the result of the above calculation to two significant digits

```
mutate(summarize(group_by(filter(ydat, bp == "leucine biosynthesis"), nutrient),
    r = cor(rate, expression)), r = round(r, 2))
```

then `arrange()` by the rounded correlation coefficient above

```
arrange(
  mutate(
    summarize(
      group_by(
        filter(ydat, bp=="leucine biosynthesis"),
      nutrient),
    r=cor(rate, expression)),
  r=round(r, 2)),
r)
```

```
# A tibble: 6 x 2
```

```
  nutrient      r
  <chr>     <dbl>
1 Leucine   -0.58
2 Glucose   -0.04
3 Ammonia    0.16
4 Sulfate    0.33
5 Phosphate  0.44
6 Uracil     0.58
```

Now compare that with the mental process of what you're actually trying to accomplish. The way you would do this without pipes is completely inside-out and backwards from the way you express in words and in thought what you want to do. The pipe operator |> allows you to pass the output data frame from one function to the input data frame to another function.

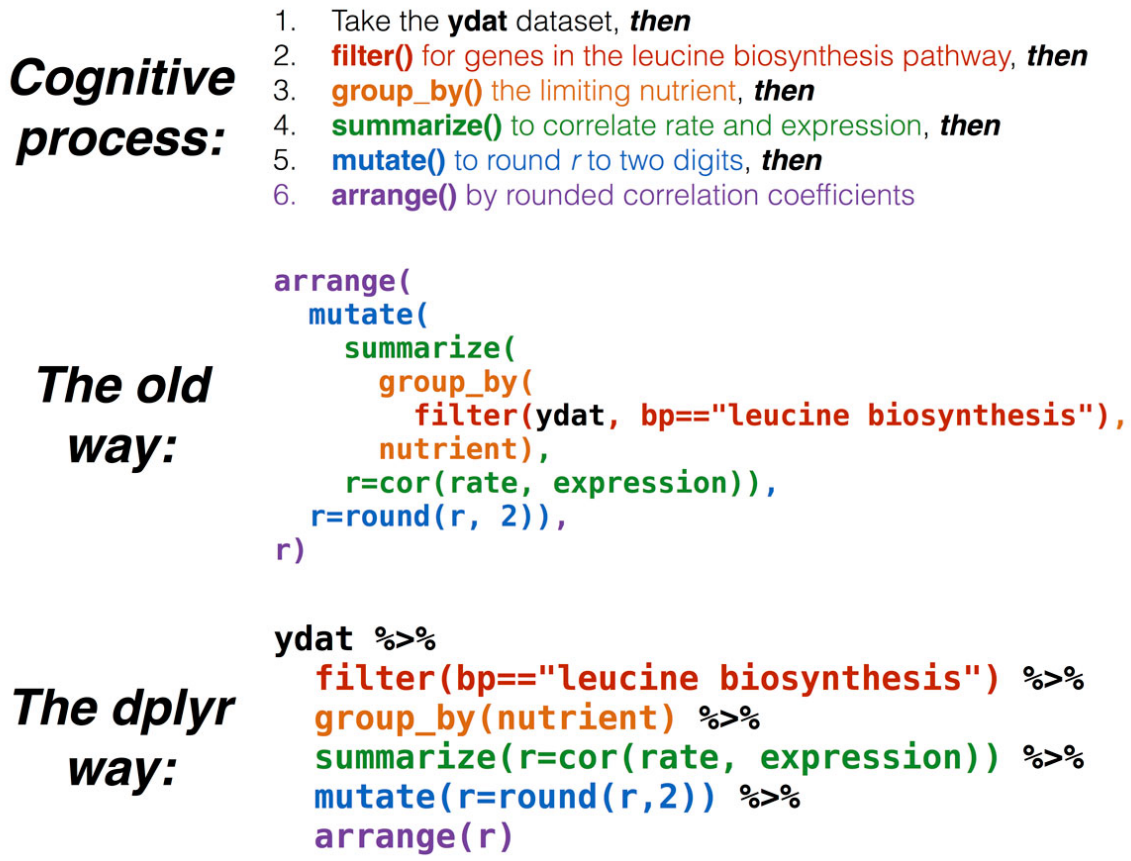


Figure 3.1: Nesting functions versus piping

This is how we would do that in code. It's as simple as replacing the word "then" in words to the symbol |> in code. (There's a keyboard shortcut that I'll use frequently to insert the

|> sequence – you can see what it is by clicking the *Tools* menu in RStudio, then selecting *Keyboard Shortcut Help*. On Mac, it's CMD-SHIFT-M.)

```
ydat |>
  filter(bp=="leucine biosynthesis") |>
  group_by(nutrient) |>
  summarize(r=cor(rate, expression)) |>
  mutate(r=round(r,2)) |>
  arrange(r)
```

```
# A tibble: 6 x 2
  nutrient      r
  <chr>     <dbl>
1 Leucine   -0.58
2 Glucose   -0.04
3 Ammonia    0.16
4 Sulfate    0.33
5 Phosphate  0.44
6 Uracil     0.58
```

3.5 Exercises

Here's a warm-up round. Try the following.

> Exercise 3
>
> Show the limiting nutrient and expression values for the gene ADH2 when the growth rate is restricted to 0.05. *Hint:* 2 pipes: `filter` and `select`.
>
> ```
> # A tibble: 6 x 2
>   nutrient  expression
>   <chr>          <dbl>
> 1 Glucose         6.28
> 2 Ammonia         0.55
> 3 Phosphate      -4.6
> 4 Sulfate        -1.18
> 5 Leucine         4.15
> 6 Uracil          0.63
> ```

Exercise 4

What are the four most highly expressed genes when the growth rate is restricted to 0.05 by restricting glucose? Show only the symbol, expression value, and GO terms. *Hint:* 4 pipes: `filter`, `arrange`, `head`, and `select`.

```
# A tibble: 4 x 4
  symbol expression bp                  mf
  <chr>       <dbl> <chr>               <chr>
1 ADH2         6.28 fermentation*       alcohol dehydrogenase activity
2 HSP26        5.86 response to stress* unfolded protein binding
3 MLS1         5.64 glyoxylate cycle    malate synthase activity
4 HXT5         5.56 hexose transport    glucose transporter activity*
```

Exercise 5

When the growth rate is restricted to 0.05, what is the average expression level across all genes in the "response to stress" biological process, separately for each limiting nutrient? What about genes in the "protein biosynthesis" biological process? *Hint:* 3 pipes: `filter`, `group_by`, `summarize`.

```
# A tibble: 6 x 2
  nutrient  meanexp
  <chr>       <dbl>
1 Ammonia     0.943
2 Glucose     0.743
3 Leucine     0.811
4 Phosphate   0.981
5 Sulfate     0.743
6 Uracil      0.731

# A tibble: 6 x 2
  nutrient  meanexp
  <chr>       <dbl>
1 Ammonia    -1.61
2 Glucose    -0.691
3 Leucine    -0.574
4 Phosphate  -0.750
5 Sulfate    -0.913
6 Uracil     -0.880
```

That was easy, right? How about some tougher ones.

Exercise 6

First, some review. How do we see the number of distinct values of a variable? Use `n_distinct()` within a `summarize()` call.

```
ydat |> summarize(n_distinct(mf))
```

```
# A tibble: 1 x 1
  `n_distinct(mf)`
             <int>
1             1086
```

Exercise 7

Which 10 biological process annotations have the most genes associated with them? What about molecular functions? *Hint:* 4 pipes: `group_by`, `summarize` with `n_distinct`, `arrange`, `head`.

```
# A tibble: 10 x 2
   bp                                                          n
   <chr>                                                   <int>
 1 biological process unknown                                269
 2 protein biosynthesis                                      182
 3 protein amino acid phosphorylation*                        78
 4 protein biosynthesis*                                      73
 5 cell wall organization and biogenesis*                     64
 6 regulation of transcription from RNA polymerase II promoter*    49
 7 nuclear mRNA splicing, via spliceosome                     47
 8 DNA repair*                                                44
 9 ER to Golgi transport*                                     42
10 aerobic respiration*                                       42
```

```
# A tibble: 10 x 2
   mf                                      n
   <chr>                               <int>
 1 molecular function unknown            886
 2 structural constituent of ribosome    185
 3 protein binding                       107
 4 RNA binding                            63
 5 protein binding*                       53
 6 DNA binding*                           44
 7 structural molecule activity           43
```

```
 8 GTPase activity                            40
 9 structural constituent of cytoskeleton     39
10 transcription factor activity              38
```

Exercise 8

How many distinct genes are there where we know what process the gene is involved in but we don't know what it does? *Hint:* 3 pipes; `filter` where `bp!="biological process unknown" & mf=="molecular function unknown"`, and after `select`ing columns of interest, pipe the output to `distinct()`. The answer should be **737**, and here are a few:

```
# A tibble: 737 x 3
   symbol bp                                                                mf
   <chr>  <chr>                                                             <chr>
 1 SFB2   ER to Golgi transport                                             molec~
 2 EDC3   deadenylylation-independent decapping                             molec~
 3 PER1   response to unfolded protein*                                     molec~
 4 PEX25  peroxisome organization and biogenesis*                           molec~
 5 BNI5   cytokinesis*                                                      molec~
 6 CSN12  adaptation to pheromone during conjugation with cellular fusion   molec~
 7 SEC39  secretory pathway                                                 molec~
 8 ABC1   ubiquinone biosynthesis                                           molec~
 9 PRP46  nuclear mRNA splicing, via spliceosome                            molec~
10 MAM3   mitochondrion organization and biogenesis*                       molec~
# i 727 more rows
```

Exercise 9

When the growth rate is restricted to 0.05 by limiting Glucose, which biological processes are the most upregulated? Show a sorted list with the most upregulated BPs on top, displaying the biological process and the average expression of all genes in that process rounded to two digits. *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `mutate`, `arrange`.

```
# A tibble: 881 x 2
   bp                                          meanexp
   <chr>                                         <dbl>
 1 fermentation*                                  6.28
 2 glyoxylate cycle                               5.28
 3 oxygen and reactive oxygen species metabolism  5.04
 4 fumarate transport*                            5.03
 5 acetyl-CoA biosynthesis*                       4.32
 6 gluconeogenesis                                3.64
```

```
 7 fatty acid beta-oxidation                    3.57
 8 lactate transport                            3.48
 9 carnitine metabolism                         3.3
10 alcohol metabolism*                          3.25
# i 871 more rows
```

Exercise 10

Group the data by limiting nutrient (primarily) then by biological process. Get the average expression for all genes annotated with each process, separately for each limiting nutrient, where the growth rate is restricted to 0.05. Arrange the result to show the most upregulated processes on top. The initial result will look like the result below. Pipe this output to a `View()` statement. What's going on? Why didn't the `arrange()` work? *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `arrange`, `View`.

```
# A tibble: 5,257 x 3
# Groups:   nutrient [6]
   nutrient  bp                                                meanexp
   <chr>     <chr>                                               <dbl>
 1 Ammonia   allantoate transport                                6.64
 2 Ammonia   amino acid transport*                               6.64
 3 Phosphate glycerophosphodiester transport                     6.64
 4 Glucose   fermentation*                                       6.28
 5 Ammonia   allantoin transport                                 5.56
 6 Glucose   glyoxylate cycle                                    5.28
 7 Ammonia   proline catabolism*                                 5.14
 8 Ammonia   urea transport                                      5.14
 9 Glucose   oxygen and reactive oxygen species metabolism       5.04
10 Glucose   fumarate transport*                                 5.03
# i 5,247 more rows
```

Exercise 11

Let's try to further process that result to get only the top three most upregulated biolgocal processes for each limiting nutrient. Google search "dplyr first result within group." You'll need a `filter(row_number()......)` in there somewhere. *Hint:* 5 pipes: `filter`, `group_by`, `summarize`, `arrange`, `filter(row_number()....`. *Note:* dplyr's pipe syntax used to be `%.%` before it changed to `|>`. So when looking around, you might still see some people use the old syntax. Now if you try to use the old syntax, you'll get a deprecation warning.

```
# A tibble: 18 x 3
```

```
# Groups:   nutrient [6]
   nutrient  bp                                                meanexp
   <chr>     <chr>                                             <dbl>
 1 Ammonia   allantoate transport                               6.64
 2 Ammonia   amino acid transport*                             6.64
 3 Phosphate glycerophosphodiester transport                   6.64
 4 Glucose   fermentation*                                     6.28
 5 Ammonia   allantoin transport                               5.56
 6 Glucose   glyoxylate cycle                                  5.28
 7 Glucose   oxygen and reactive oxygen species metabolism     5.04
 8 Uracil    fumarate transport*                               4.32
 9 Phosphate vacuole fusion, non-autophagic                    4.20
10 Leucine   fermentation*                                     4.15
11 Phosphate regulation of cell redox homeostasis*             4.03
12 Leucine   fumarate transport*                               3.72
13 Leucine   glyoxylate cycle                                  3.65
14 Sulfate   protein ubiquitination                            3.4
15 Sulfate   fumarate transport*                               3.27
16 Uracil    pyridoxine metabolism                             3.11
17 Uracil    asparagine catabolism*                            3.06
18 Sulfate   sulfur amino acid metabolism*                     2.69
```

Exercise 12

There's a slight problem with the examples above. We're getting the average expression of all the biological processes separately by each nutrient. But some of these biological processes only have a single gene in them! If we tried to do the same thing to get the correlation between rate and expression, the calculation would work, but we'd get a warning about a standard deviation being zero. The correlation coefficient value that results is `NA`, i.e., missing. While we're summarizing the correlation between rate and expression, let's also show the number of distinct genes within each grouping.

```
ydat |>
  group_by(nutrient, bp) |>
  summarize(r=cor(rate, expression), ngenes=n_distinct(symbol))
```

```
Warning: There was 1 warning in `summarize()`.
i In argument: `r = cor(rate, expression)`.
i In group 110: `nutrient = "Ammonia"` and `bp = "allantoate transport"`.
Caused by warning in `cor()`:
! the standard deviation is zero
```

```
# A tibble: 5,286 x 4
# Groups:   nutrient [6]
   nutrient bp                                              r ngenes
   <chr>    <chr>                                       <dbl>  <int>
 1 Ammonia  'de novo' IMP biosynthesis*                 0.312      8
 2 Ammonia  'de novo' pyrimidine base biosynthesis     -0.0482     3
 3 Ammonia  'de novo' pyrimidine base biosynthesis*     0.167      4
 4 Ammonia  35S primary transcript processing           0.508     13
 5 Ammonia  35S primary transcript processing*          0.424     30
 6 Ammonia  AMP biosynthesis*                           0.464      1
 7 Ammonia  ATP synthesis coupled proton transport      0.112     15
 8 Ammonia  ATP synthesis coupled proton transport*    -0.541      2
 9 Ammonia  C-terminal protein amino acid methylation   0.813      1
10 Ammonia  D-ribose metabolism                        -0.837      1
# i 5,276 more rows
```

Take the above code and continue to process the result to show only results where the process has at least 5 genes. Add a column corresponding to the absolute value of the correlation coefficient, and show for each nutrient the singular process with the highest correlation between rate and expression, regardless of direction. *Hint:* 4 more pipes: `filter`, `mutate`, `arrange`, and `filter` again with `row_number()==1`. Ignore the warning.

```
# A tibble: 6 x 5
# Groups:   nutrient [6]
  nutrient  bp                                              r ngenes  absr
  <chr>     <chr>                                       <dbl>  <int> <dbl>
1 Glucose   telomerase-independent telomere maintenance -0.95      7  0.95
2 Ammonia   telomerase-independent telomere maintenance -0.91      7  0.91
3 Leucine   telomerase-independent telomere maintenance -0.9       7  0.9
4 Phosphate telomerase-independent telomere maintenance -0.9       7  0.9
5 Uracil    telomerase-independent telomere maintenance -0.81      7  0.81
6 Sulfate   translational elongation*                     0.79      5  0.79
```

4 Data Visualization with ggplot2

This section will cover fundamental concepts for creating effective data visualization and will introduce tools and techniques for visualizing large, high-dimensional data using R. We will review fundamental concepts for visually displaying quantitative information, such as using series of small multiples, avoiding "chart-junk," and maximizing the data-ink ratio. We will cover the grammar of graphics (geoms, aesthetics, stats, and faceting), and using the ggplot2 package to create plots layer-by-layer.

This lesson assumes a basic familiarity with R (Chapter 1), data frames (Chapter 2), and manipulating data with dplyr and `|>` (Chapter 3).

4.1 Review

4.1.1 Gapminder data

We're going to work with a different dataset for this section. It's a cleaned-up excerpt from the Gapminder data. Download the **gapminder.csv** data by clicking here or using the link above.

Let's read in the data to an object called `gm` and take a look with `View`. Remember, we need to load both the dplyr and readr packages for efficiently reading in and displaying this data.

```
# Load packages
library(readr)
library(dplyr)

# Download the data locally and read the file
gm <- read_csv(file="data/gapminder.csv")

# Show the first few lines of the data
gm
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
```

```
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# i 1,694 more rows
```

```
# Optionally bring up data in a viewer window.
# View(gm)
```

This particular excerpt has 1704 observations on six variables:

- `country` a categorical variable 142 levels
- `continent`, a categorical variable with 5 levels
- `year`: going from 1952 to 2007 in increments of 5 years
- `pop`: population
- `gdpPercap`: GDP per capita
- `lifeExp`: life expectancy

4.1.2 dplyr review

The dplyr package gives you a handful of useful **verbs** for managing data. On their own they don't do anything that base R can't do. Here are some of the *single-table* verbs we'll be working with in this lesson (single-table meaning that they only work on a single table – contrast that to *two-table* verbs used for joining data together). They all take a `data.frame` or `tbl` as their input for the first argument, and they all return a `data.frame` or `tbl` as output.

1. `filter()`: filters *rows* of the data where some condition is true
2. `select()`: selects out particular *columns* of interest
3. `mutate()`: adds new columns or changes values of existing columns
4. `arrange()`: arranges a data frame by the value of a column
5. `summarize()`: summarizes multiple values to a single value, most useful when combined with…
6. `group_by()`: groups a data frame by one or more variable. Most data operations are useful done on groups defined by variables in the the dataset. The `group_by` function takes an existing data frame and converts it into a grouped data frame where `summarize()` operations are performed *by group*.

Additionally, the |> operator allows you to "chain" operations together. Rather than nesting functions inside out, the |> operator allows you to write operations left-to-right, top-to-bottom. Let's say we wanted to get the average life expectancy and GDP (not GDP per capita) for Asian countries for each year.

Cognitive process:

1. Take the **gm** data, **then**
2. **Mutate** it to add "gdp" variable, **then**
3. **Filter** where continent=="Asia", **then**
4. **Group by** year, **then**
5. **Summarize** to get mean life exp & GDP

The old way:

```
summarize(
  group_by(
    filter(
      mutate(gm, gdp=gdpPercap*pop),
    continent=="Asia"),
  year),
mean(lifeExp), mean(gdp))
```

The dplyr way:

```
gm %>%
  mutate(gdp=gdpPercap*pop) %>%
  filter(continent=="Asia") %>%
  group_by(year) %>%
  summarize(mean(lifeExp), mean(gdp))
```

The |> would allow us to do this:

```
gm |>
  mutate(gdp=gdpPercap*pop) |>
  filter(continent=="Asia") |>
  group_by(year) |>
  summarize(mean(lifeExp), mean(gdp))
```

```
# A tibble: 12 x 3
    year `mean(lifeExp)`   `mean(gdp)`
   <dbl>           <dbl>         <dbl>
 1  1952            46.3  34095762661.
```

```
 2  1957            49.3  47267432088.
 3  1962            51.6  60136869012.
 4  1967            54.7  84648519224.
 5  1972            57.3 124385747313.
 6  1977            59.6 159802590186.
 7  1982            62.6 194429049919.
 8  1987            64.9 241784763369.
 9  1992            66.5 307100497486.
10  1997            68.0 387597655323.
11  2002            69.2 458042336179.
12  2007            70.7 627513635079.
```

Instead of this:

```
summarize(
  group_by(
    filter(
      mutate(gm, gdp=gdpPercap*pop),
    continent=="Asia"),
  year),
mean(lifeExp), mean(gdp))
```

4.2 About ggplot2

ggplot2 is a widely used R package that extends R's visualization capabilities. It takes the hassle out of things like creating legends, mapping other variables to scales like color, or faceting plots into small multiples. We'll learn about what all these things mean shortly.

Where does the "gg" in ggplot2 come from? The **ggplot2** package provides an R implementation of Leland Wilkinson's *Grammar of Graphics* (1999). The *Grammar of Graphics* allows you to think beyond the garden variety plot types (e.g. scatterplot, barplot) and the consider the components that make up a plot or graphic, such as how data are represented on the plot (as lines, points, etc.), how variables are mapped to coordinates or plotting shape or color, what transformation or statistical summary is required, and so on.

Specifically, **ggplot2** allows you to build a plot layer-by-layer by specifying:

- a **geom**, which specifies how the data are represented on the plot (points, lines, bars, etc.),
- **aesthetics** that map variables in the data to axes on the plot or to plotting size, shape, color, etc.,
- a **stat**, a statistical transformation or summary of the data applied prior to plotting,

- **facets**, which we've already seen above, that allow the data to be divided into chunks on the basis of other categorical or continuous variables and the same plot drawn for each chunk.

First, a note about `qplot()`. The `qplot()` function is a quick and dirty way of making ggplot2 plots. You might see it if you look for help with ggplot2, and it's even covered extensively in the ggplot2 book. And if you're used to making plots with built-in base graphics, the `qplot()` function will probably feel more familiar. But the sooner you abandon the `qplot()` syntax the sooner you'll start to really understand ggplot2's approach to building up plots layer by layer. So we're not going to use it at all in this class.

4.3 Plotting bivariate data: continuous Y by continuous X

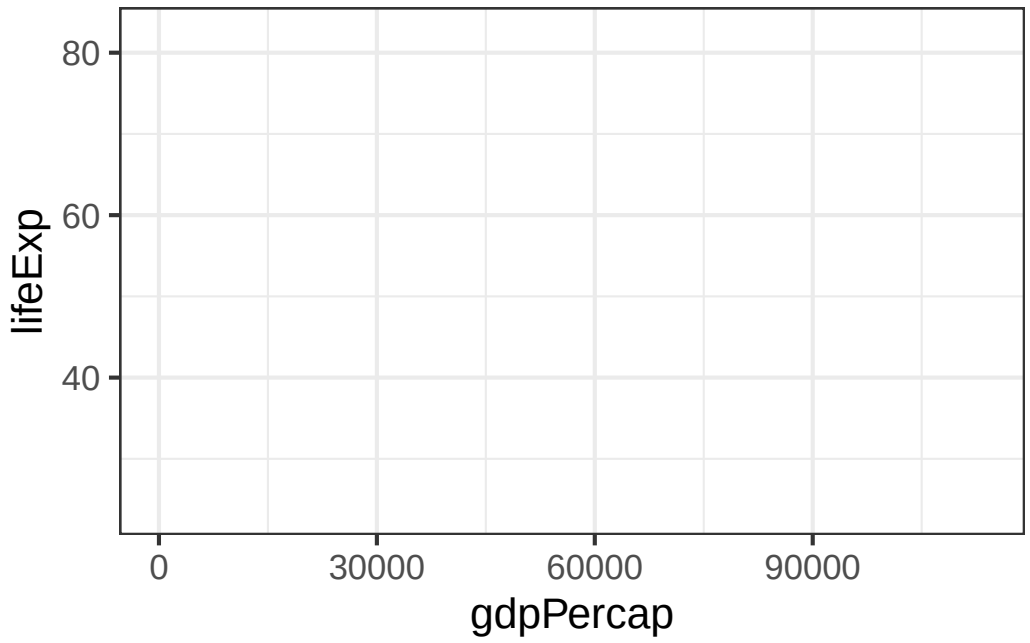
The `ggplot` function has two required arguments: the *data* used for creating the plot, and an *aesthetic* mapping to describe how variables in said data are mapped to things we can see on the plot.

First let's load the package:

```
library(ggplot2)
```

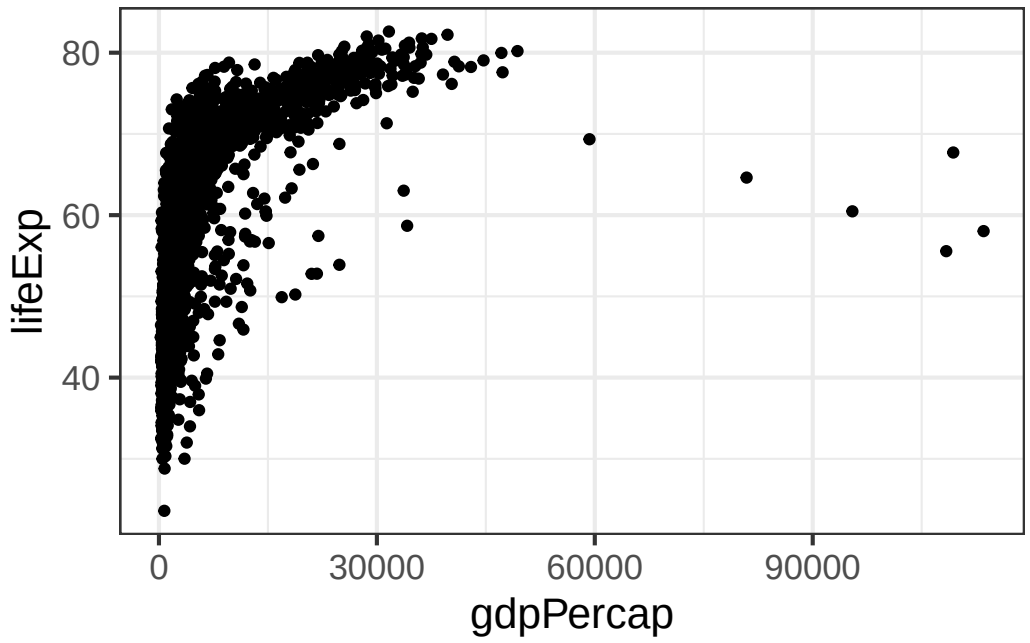
Now, let's lay out the plot. If we want to plot a continuous Y variable by a continuous X variable we're probably most interested in a scatter plot. Here, we're telling ggplot that we want to use the `gm` dataset, and the aesthetic mapping will map `gdpPercap` onto the x-axis and `lifeExp` onto the y-axis. Remember that the variable names are case sensitive!

```
ggplot(gm, aes(x = gdpPercap, y = lifeExp))
```



When we do that we get a blank canvas with no data showing (you might get an error if you're using an old version of ggplot2). That's because all we've done is laid out a two-dimensional plot specifying what goes on the x and y axes, but we haven't told it what kind of geometric object to plot. The obvious choice here is a point. Check out docs.ggplot2.org to see what kind of geoms are available.

```
ggplot(gm, aes(x = gdpPercap, y = lifeExp)) + geom_point()
```



Here, we've built our plot in layers. First, we create a canvas for plotting layers to come using the `ggplot` function, specifying which **data** to use (here, the **gm** data frame), and an **aesthetic mapping** of `gdpPercap` to the x-axis and `lifeExp` to the y-axis. We next add a layer to the plot, specifying a **geom**, or a way of visually representing the aesthetic mapping.

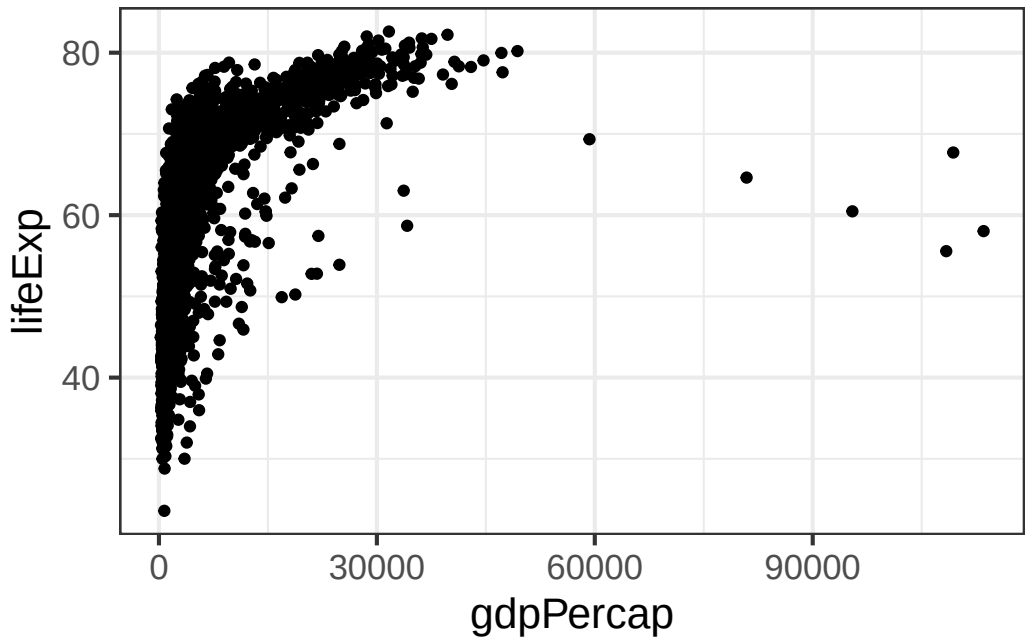
Now, the typical workflow for building up a ggplot2 plot is to first construct the figure and save that to a variable (for example, `p`), and as you're experimenting, you can continue to re-define the `p` object as you develop "keeper commands".

First, let's construct the graphic. Notice that we don't have to specify `x=` and `y=` if we specify the arguments in the correct order (x is first, y is second).

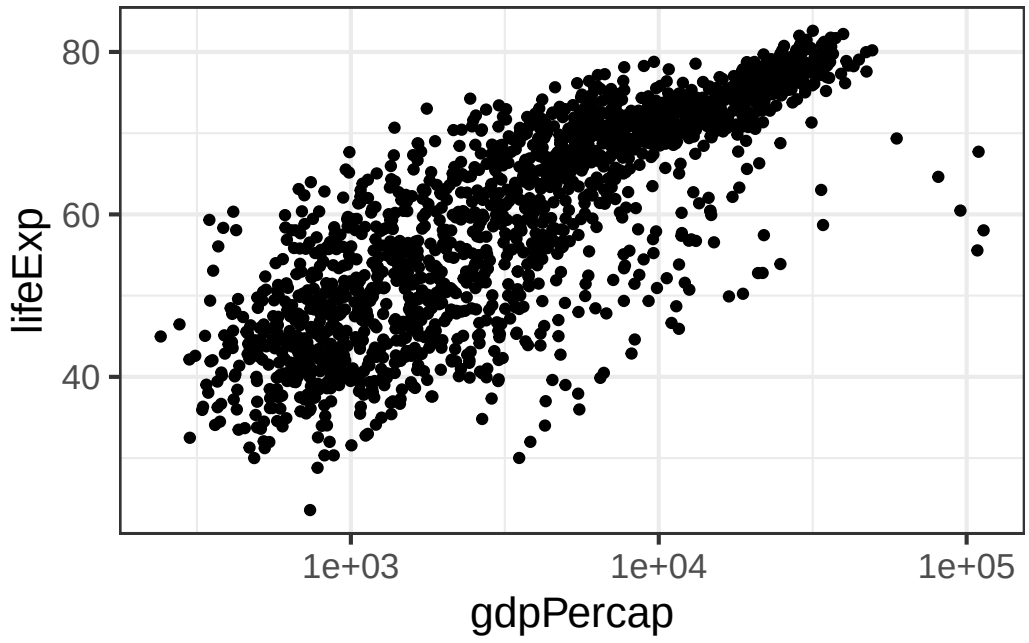
```
p <- ggplot(gm, aes(gdpPercap, lifeExp))
```

The `p` object now contains the canvas, but nothing else. Try displaying it by just running `p`. Let's experiment with adding points and a different scale to the x-axis.

```
# Experiment with adding poings
p + geom_point()
```




```
# Experiment with a different scale
p + geom_point() + scale_x_log10()
```

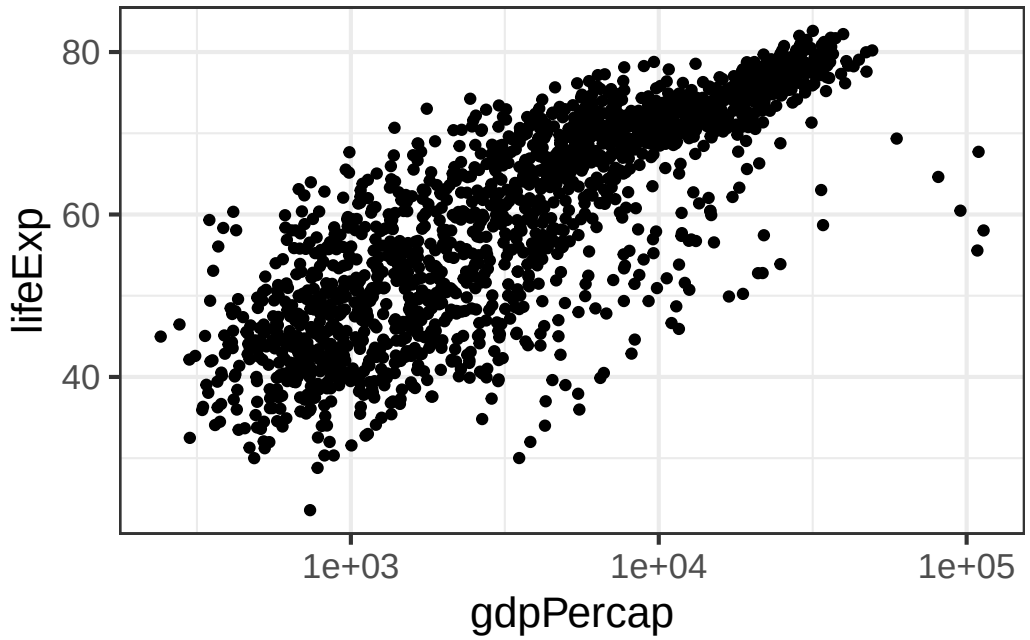


I like the look of using a log scale for the x-axis. Let's make that stick.

```
p <- p + scale_x_log10()
```

Now, if we re-ran `p` still nothing would show up because the `p` object just contains a blank canvas. Now, re-plot again with a layer of points:

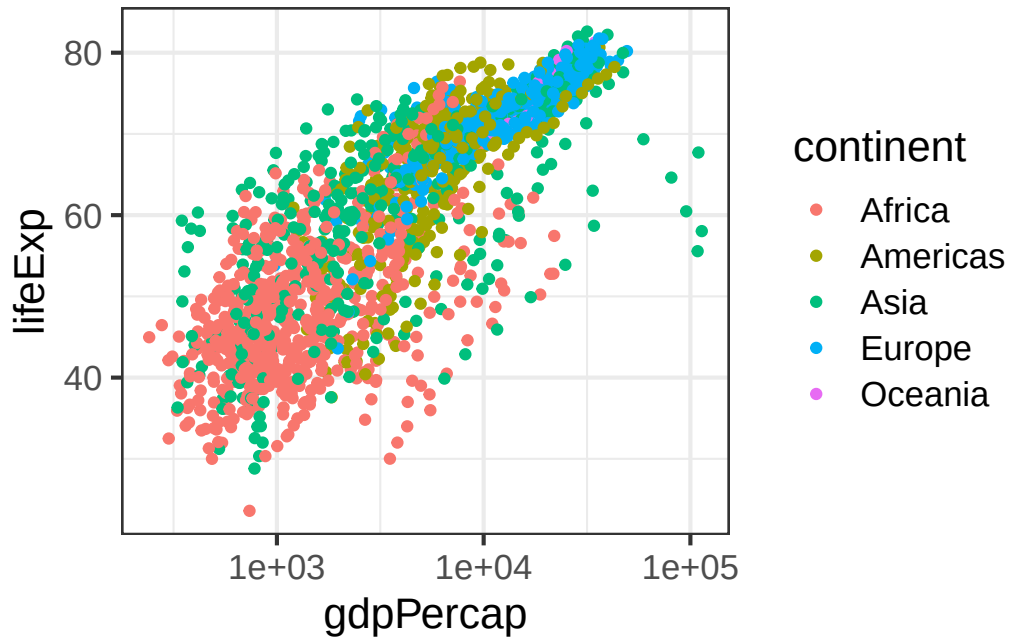
```
p + geom_point()
```



Now notice what I've saved to `p` at this point: only the basic plot layout and the log10 mapping on the x-axis. I didn't save any layers yet because I want to fiddle around with the points for a bit first.

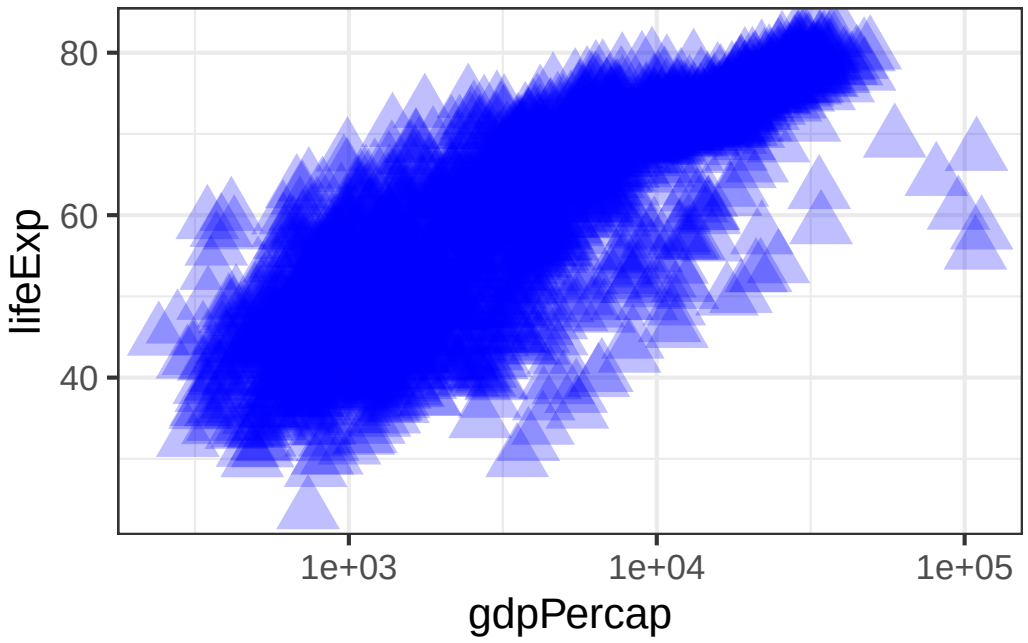
Above we implied the aesthetic mappings for the x- and y- axis should be `gdpPercap` and `lifeExp`, but we can also add aesthetic mappings to the geoms themselves. For instance, what if we wanted to color the points by the value of another variable in the dataset, say, continent?

```
p + geom_point(aes(color=continent))
```



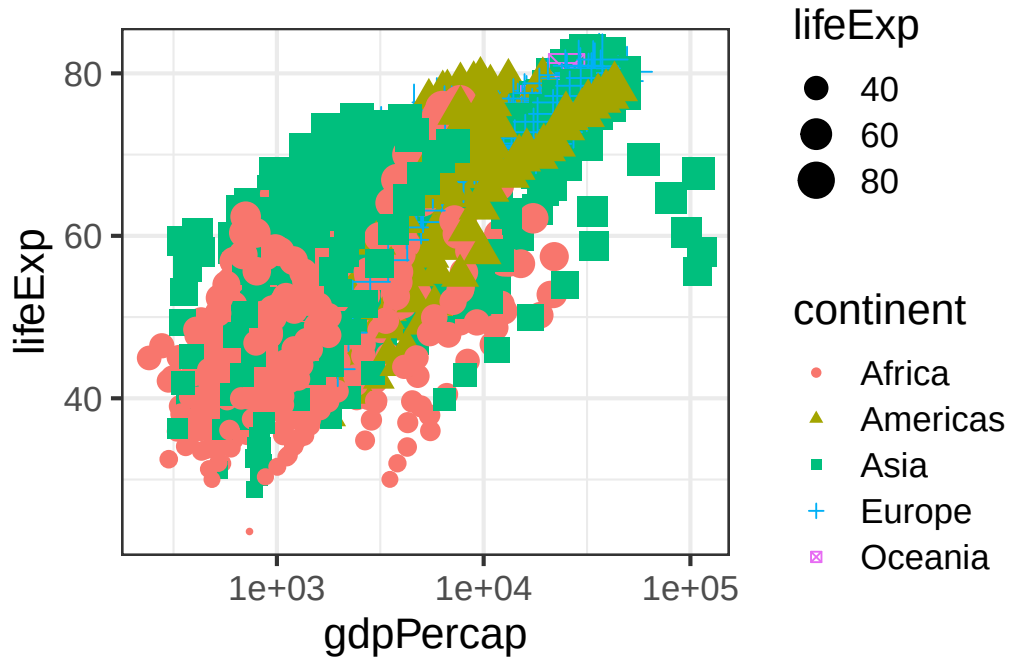
Notice the difference here. If I wanted the colors to be some static value, I wouldn't wrap that in a call to `aes()`. I would just specify it outright. Same thing with other features of the points. For example, lets make all the points huge (`size=8`) blue (`color="blue"`) semitransparent (`alpha=(1/4)`) triangles (`pch=17`):

```
p + geom_point(color="blue", pch=17, size=8, alpha=1/4)
```



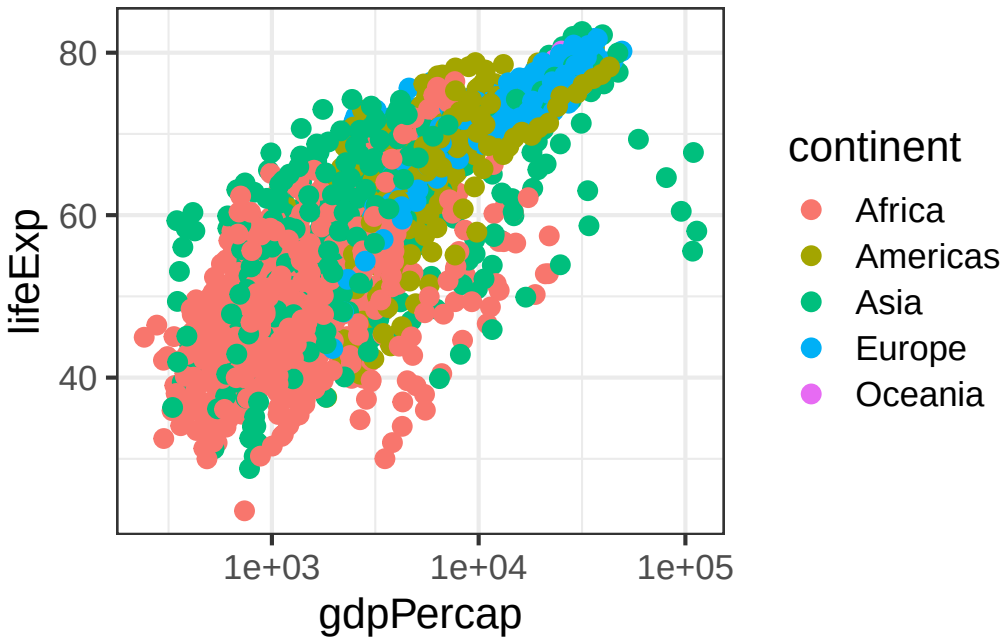
Now, this time, let's map the aesthetics of the point character to certain features of the data. For instance, let's give the points different colors and character shapes according to the continent, and map the size of the point onto the life Expectancy:

```
p + geom_point(aes(col=continent, shape=continent, size=lifeExp))
```



Now, this isn't a great plot because there are several aesthetic mappings that are redundant. Life expectancy is mapped to both the y-axis and the size of the points – the size mapping is superfluous. Similarly, continent is mapped to both the color and the point character (the shape is superfluous). Let's get rid of that, but let's make the points a little bigger outsize of an aesthetic mapping.

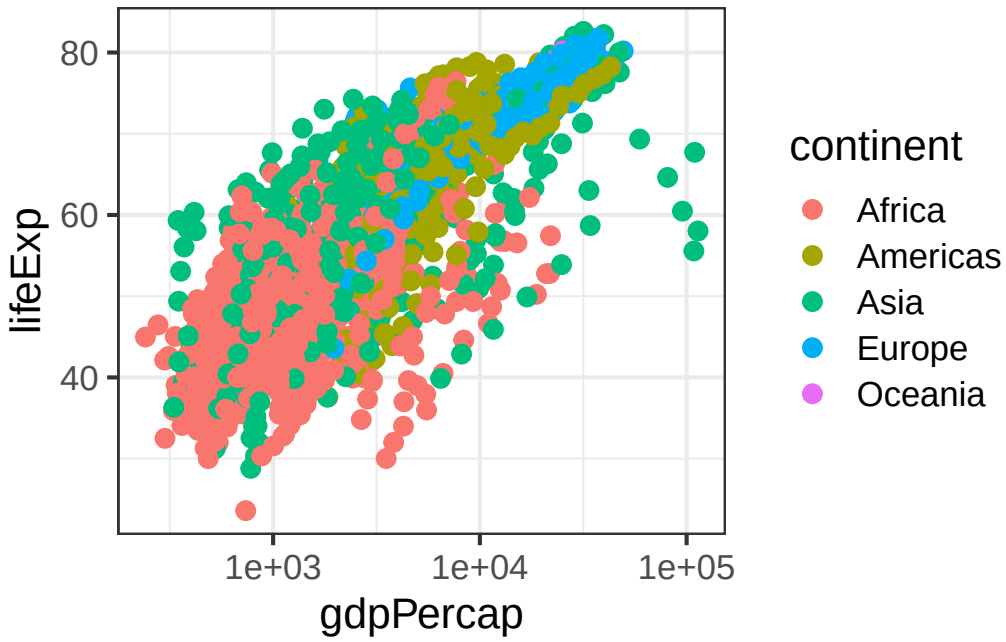
```
p + geom_point(aes(col=continent), size=3)
```



Exercise 1

Re-create this same plot from scratch without saving anything to a variable. That is, start from the `ggplot` call.

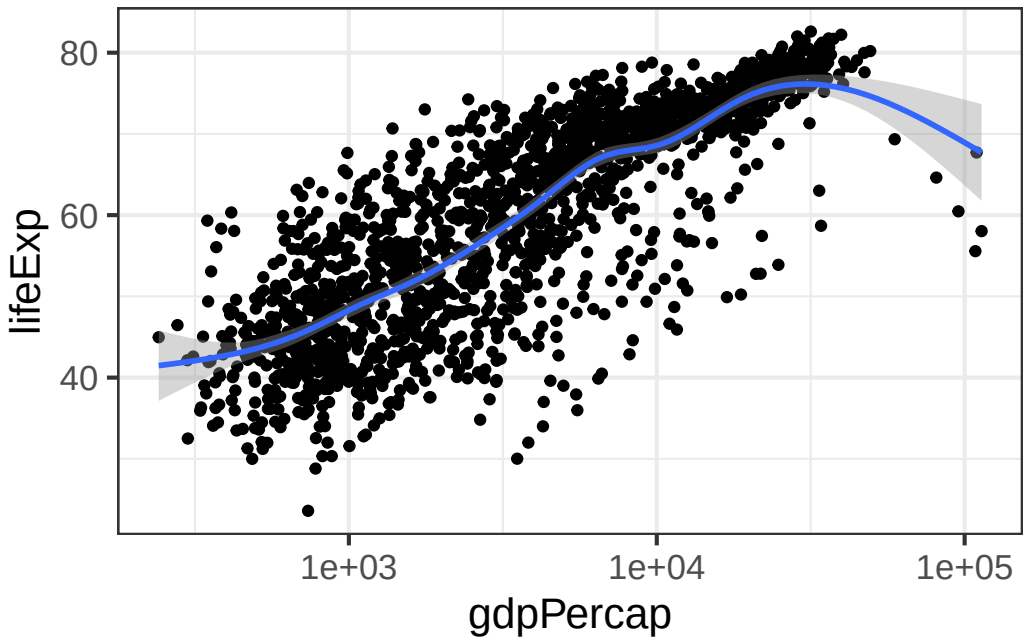
- Start with the `ggplot()` function.
- Use the gm data.
- Map `gdpPercap` to the x-axis and `lifeExp` to the y-axis.
- Add points to the plot
  - Make the points size 3
  - Map continent onto the aesthetics of the point
- Use a log10 scale for the x-axis.



4.3.1 Adding layers

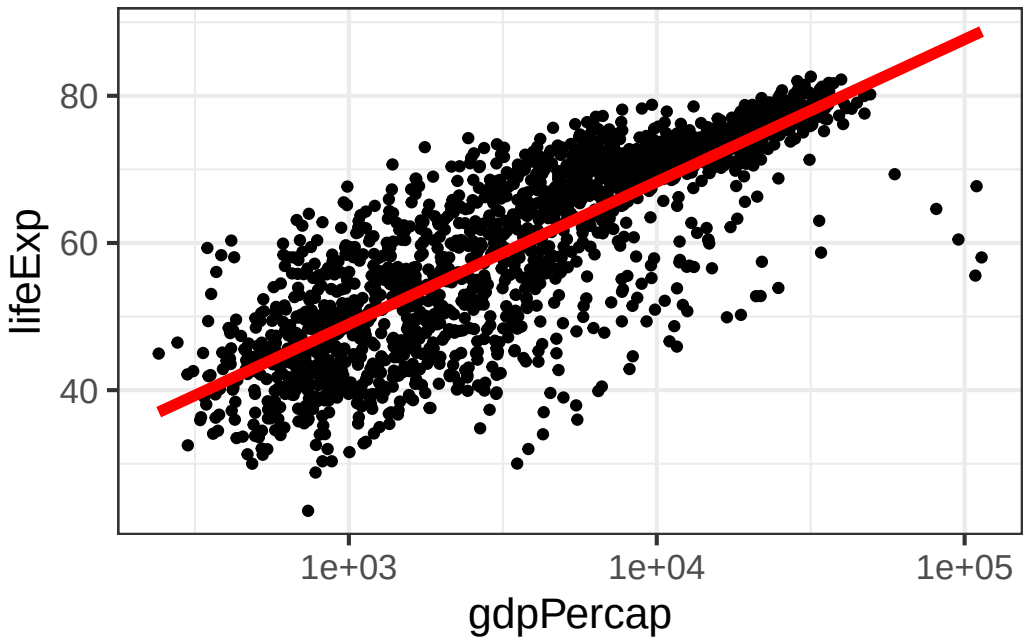
Let's add a fitted curve to the points. Recreate the plot in the `p` object if you need to.

```
p <- ggplot(gm, aes(gdpPercap, lifeExp)) + scale_x_log10()
p + geom_point() + geom_smooth()
```



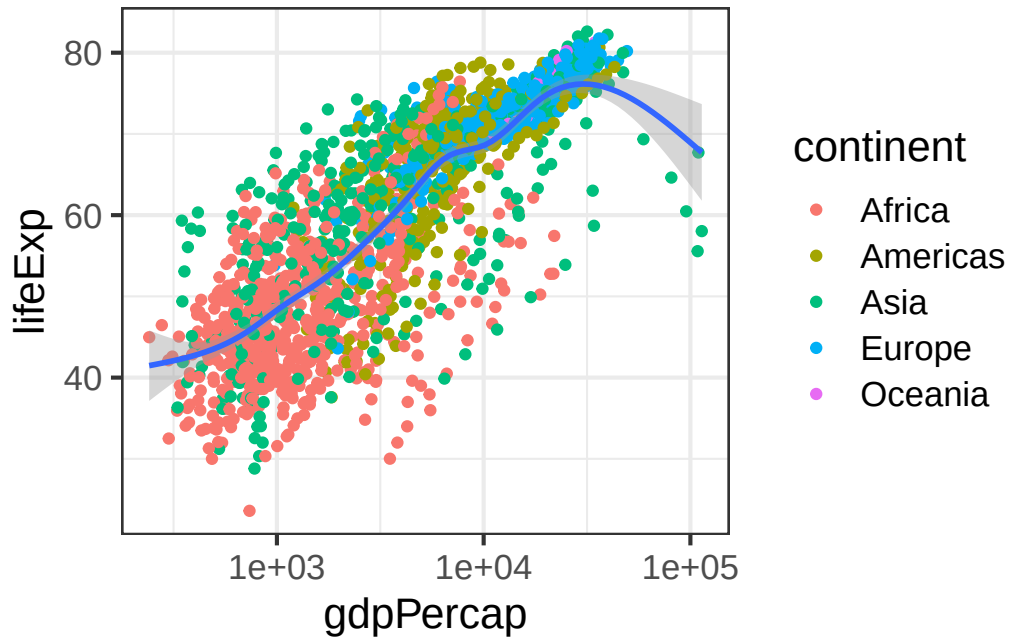
By default `geom_smooth()` will try to lowess for data with n<1000 or generalized additive models for data with n>1000. We can change that behavior by tweaking the parameters to use a thick red line, use a linear model instead of a GAM, and to turn off the standard error stripes.

```
p + geom_point() + geom_smooth(lwd=2, se=FALSE, method="lm", col="red")
```



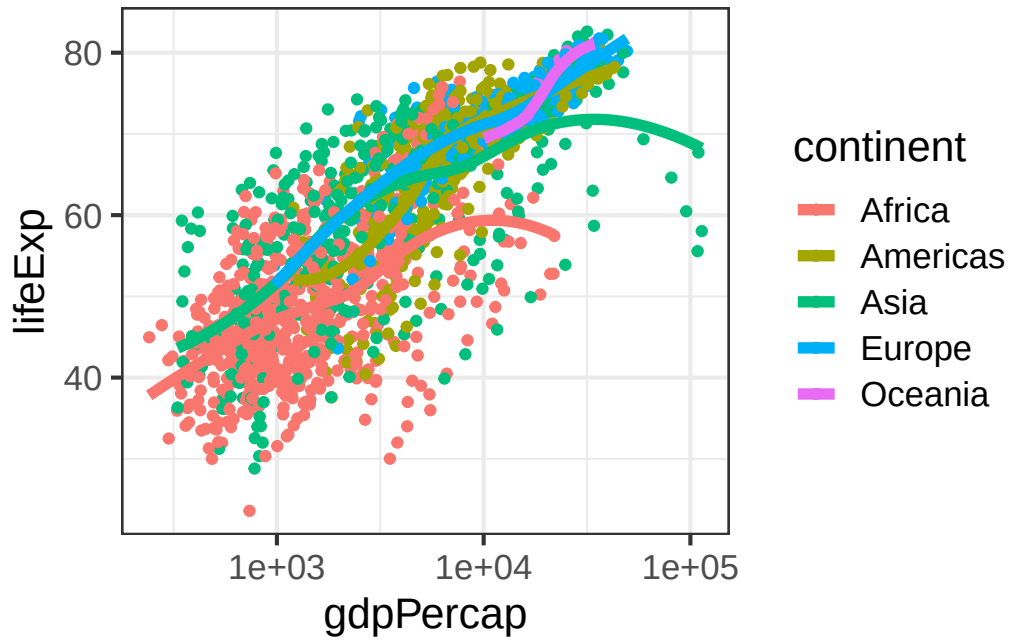
But let's add back in our aesthetic mapping to the continents. Notice what happens here. We're mapping continent as an aesthetic mapping *to the color of the points only* – so `geom_smooth()` still works only on the entire data.

```
p + geom_point(aes(color = continent)) + geom_smooth()
```



But notice what happens here: we make the call to `aes()` outside of the `geom_point()` call, and the continent variable gets mapped as an aesthetic to any further geoms. So here, we get separate smoothing lines for each continent. Let's do it again but remove the standard error stripes and make the lines a bit thicker.

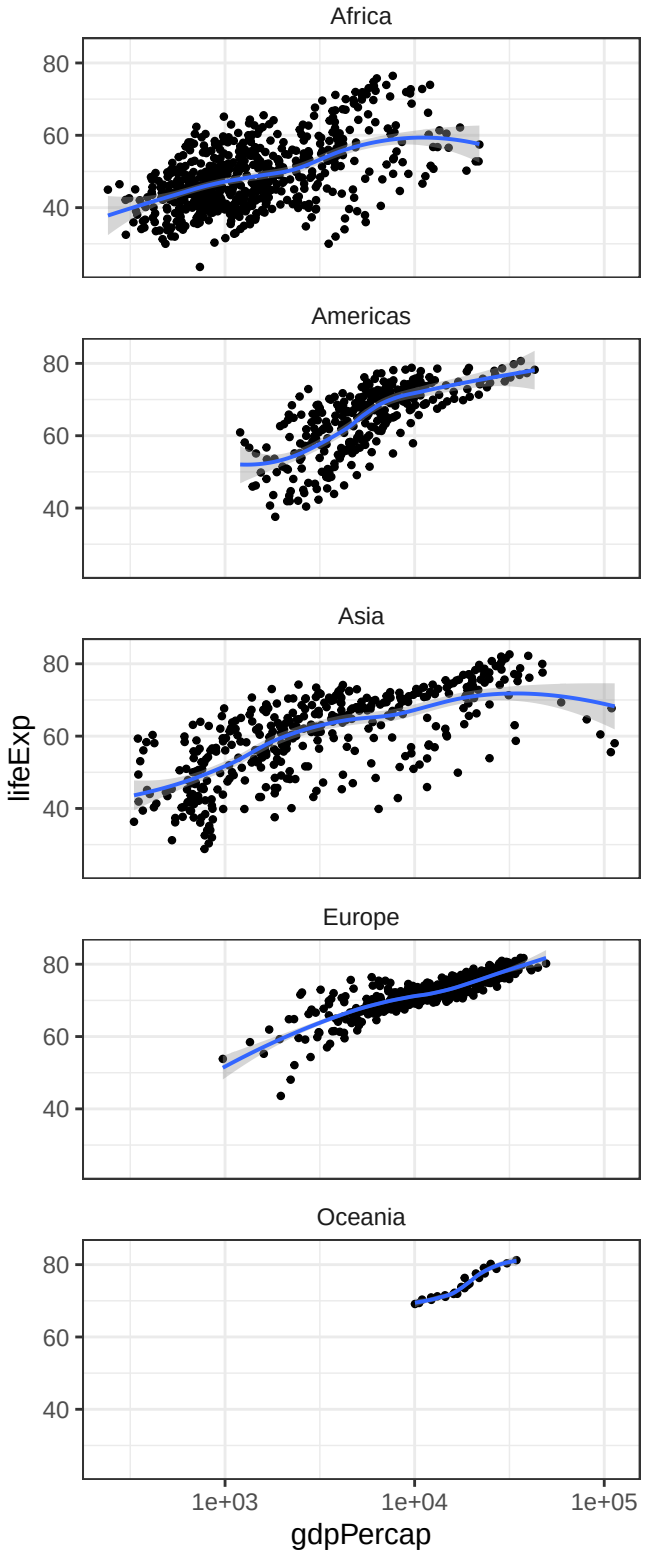
```
p + aes(color = continent) + geom_point() + geom_smooth()
p + aes(color = continent) + geom_point() + geom_smooth(se=F, lwd=2)
```



4.3.2 Faceting

Facets display subsets of the data in different panels. There are a couple ways to do this, but `facet_wrap()` tries to sensibly wrap a series of facets into a 2-dimensional grid of small multiples. Just give it a formula specifying which variables to facet by. We can continue adding more layers, such as smoothing. If you have a look at the help for `?facet_wrap()` you'll see that we can control how the wrapping is laid out.

```
p + geom_point() + facet_wrap(~continent)
p + geom_point() + geom_smooth() + facet_wrap(~continent, ncol=1)
```


Africa
Americas
Asia
Europe
Oceania
lifeExp
80
60
40
1e+03
1e+04
1e+05
gdpPercap

4.3.3 Saving plots

There are a few ways to save ggplots. The quickest way, that works in an interactive session, is to use the `ggsave()` function. You give it a file name and by default it saves the last plot that was printed to the screen.

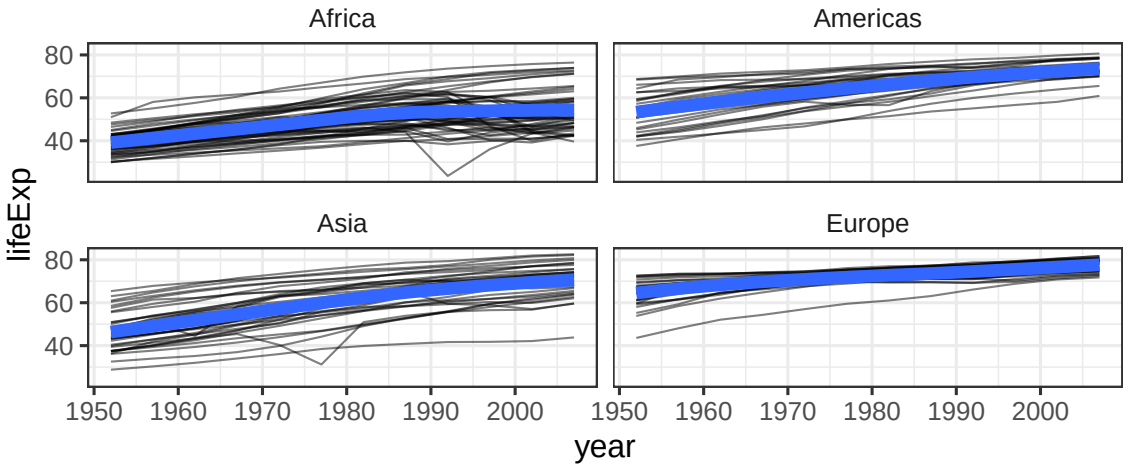
```
p + geom_point()
ggsave(file="myplot.png")
```

But if you're running this through a script, the best way to do it is to pass `ggsave()` the object containing the plot that is meant to be saved. We can also adjust things like the width, height, and resolution. `ggsave()` also recognizes the name of the file extension and saves the appropriate kind of file. Let's save a PDF.

```
pfinal <- p + geom_point() + geom_smooth() + facet_wrap(~continent, ncol=1)
ggsave(pfinal, file="myplot.pdf", width=5, height=15)
```

Exercise 2

1. Make a scatter plot of `lifeExp` on the y-axis against `year` on the x.
2. Make a series of small multiples faceting on continent.
3. Add a fitted curve, smooth or lm, with and without facets.
4. **Bonus**: using `geom_line()` and and aesthetic mapping `country` to `group=`, make a "spaghetti plot", showing *semitransparent* lines connected for each country, faceted by continent. Add a smoothed loess curve with a thick (`lwd=3`) line with no standard error stripe. Reduce the opacity (`alpha=`) of the individual black lines. *Don't* show Oceania countries (that is, `filter()` the data where `continent!="Oceania"` before you plot it).



4.4 Plotting bivariate data: continuous Y by categorical X

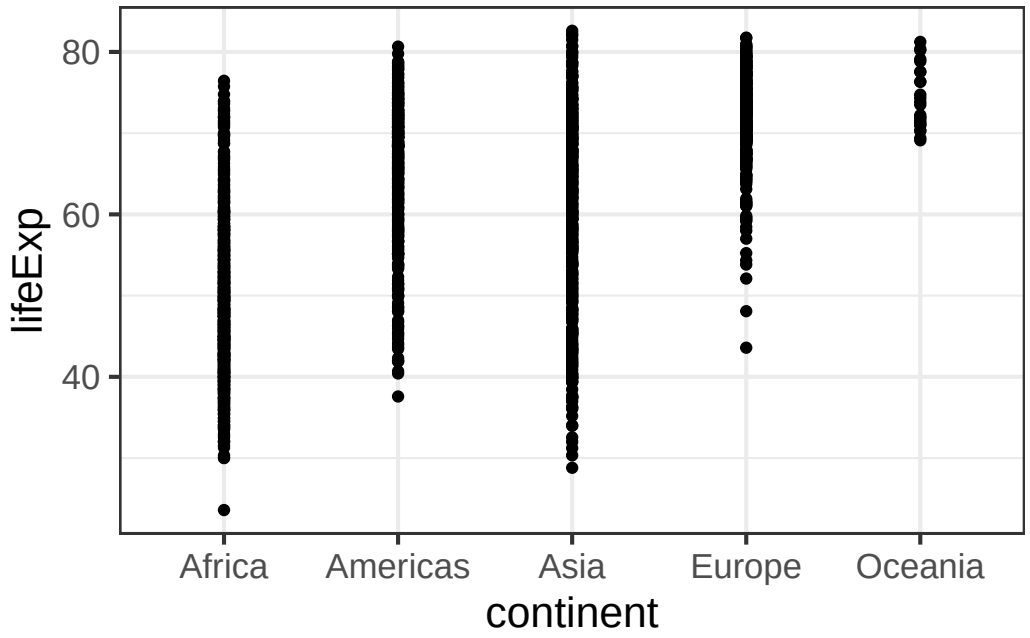
With the last example we examined the relationship between a continuous Y variable against a continuous X variable. A scatter plot was the obvious kind of data visualization. But what if we wanted to visualize a continuous Y variable against a categorical X variable? We sort of saw what that looked like in the last exercise. `year` is a continuous variable, but in this dataset, it's broken up into 5-year segments, so you could almost think of each year as a categorical variable. But a better example would be life expectancy against continent or country.

First, let's set up the basic plot:

```
p <- ggplot(gm, aes(continent, lifeExp))
```

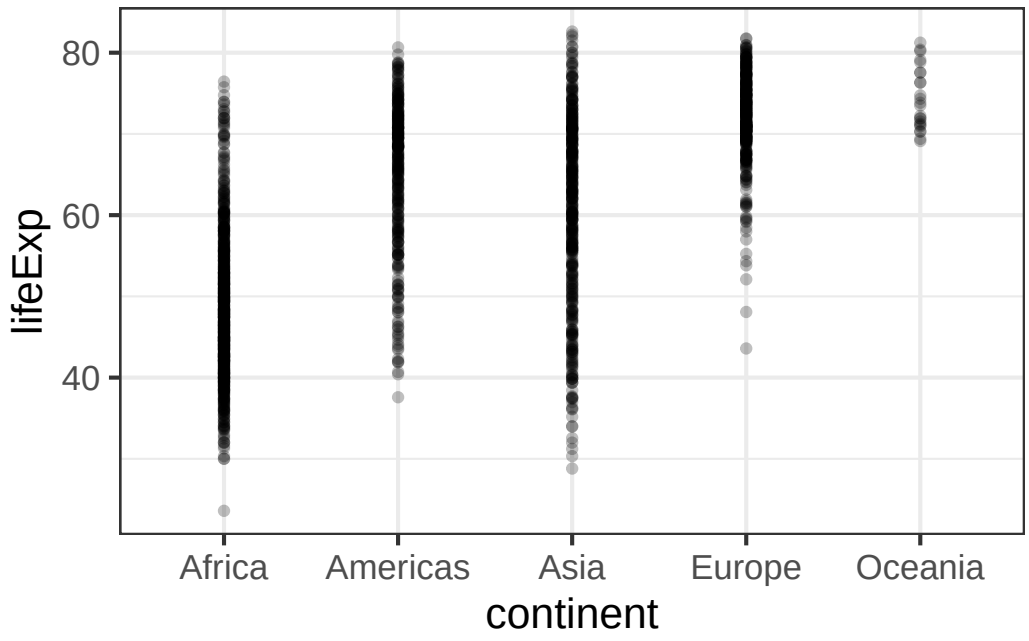
Then add points:

```
p + geom_point()
```



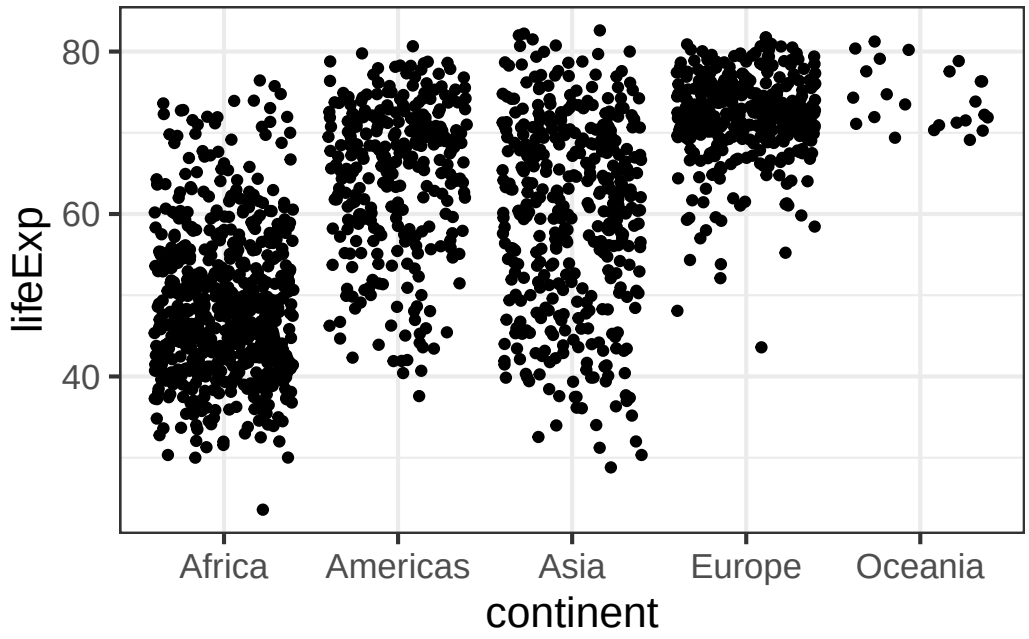
That's not terribly useful. There's a big overplotting problem. We can try to solve with transparency:

```
p + geom_point(alpha=1/4)
```



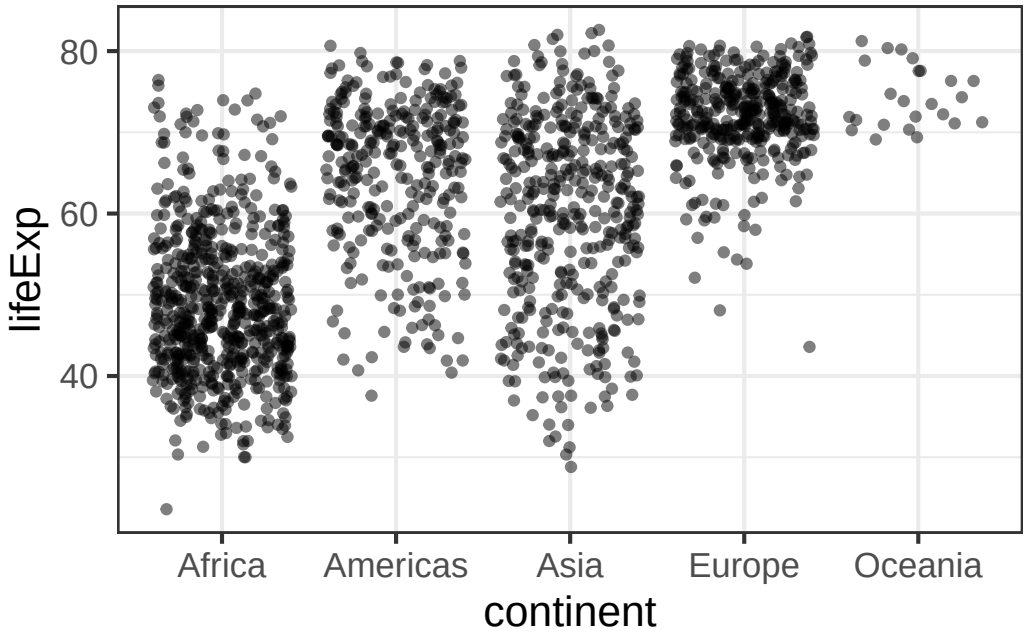
But that really only gets us so far. What if we spread things out by adding a little bit of horizontal noise (aka "jitter") to the data.

```
p + geom_jitter()
```



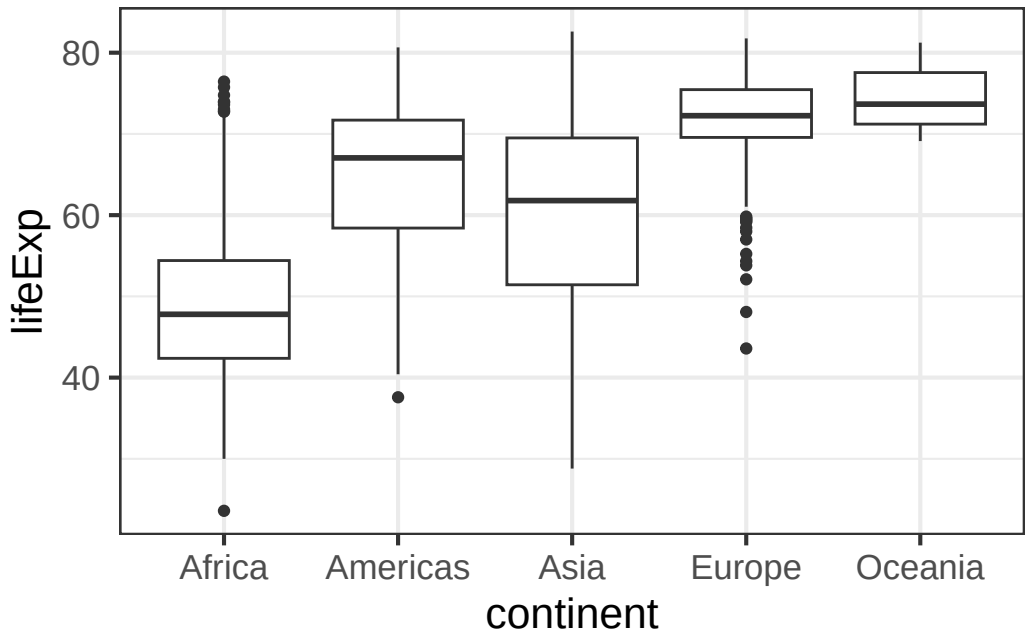
Note that the little bit of horizontal noise that's added to the jitter is random. If you run that command over and over again, each time it will look slightly different. The idea is to visualize the density at each vertical position, and spreading out the points horizontally allows you to do that. If there were still lots of over-plotting you might think about adding some transparency by setting the `alpha=` value for the jitter.

```
p + geom_jitter(alpha=1/2)
```



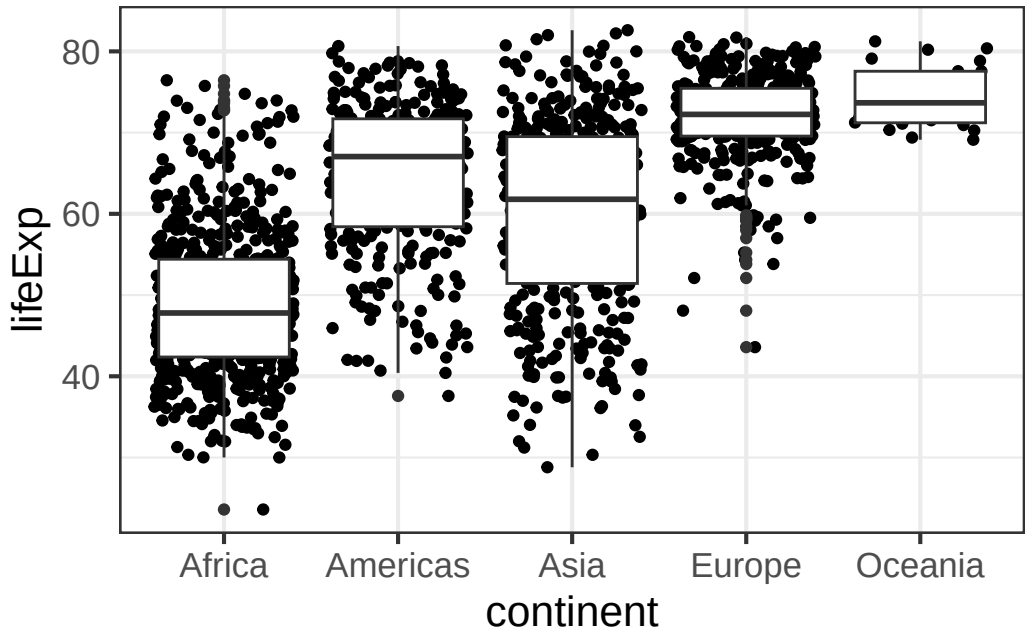
Probably a more common visualization is to show a box plot:

```
p + geom_boxplot()
```



But why not show the summary and the raw data?

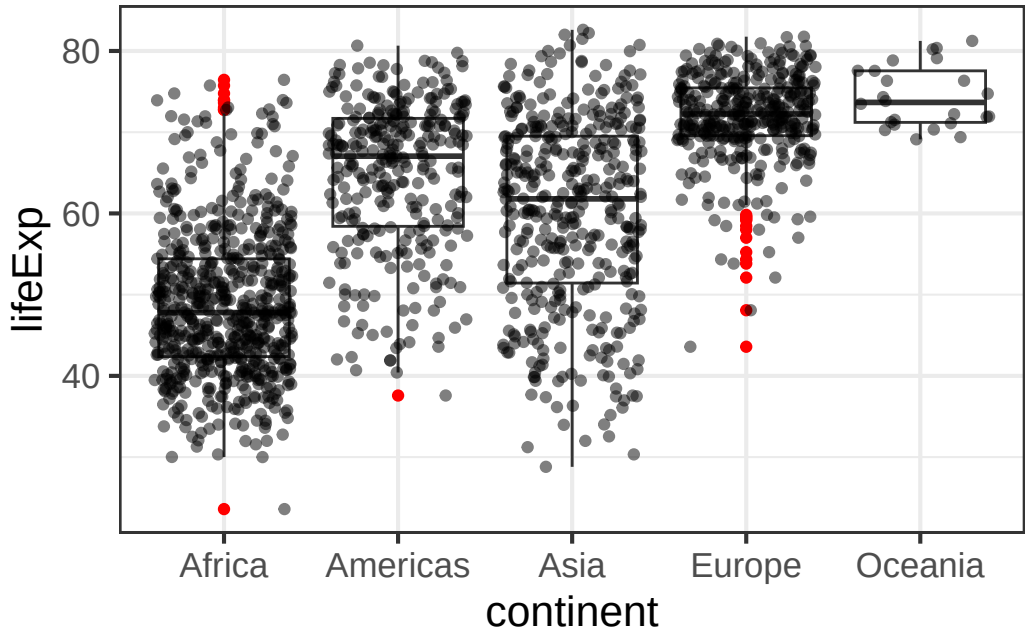
```
p + geom_jitter() + geom_boxplot()
```



Notice how in that example we first added the jitter layer then added the boxplot layer. But

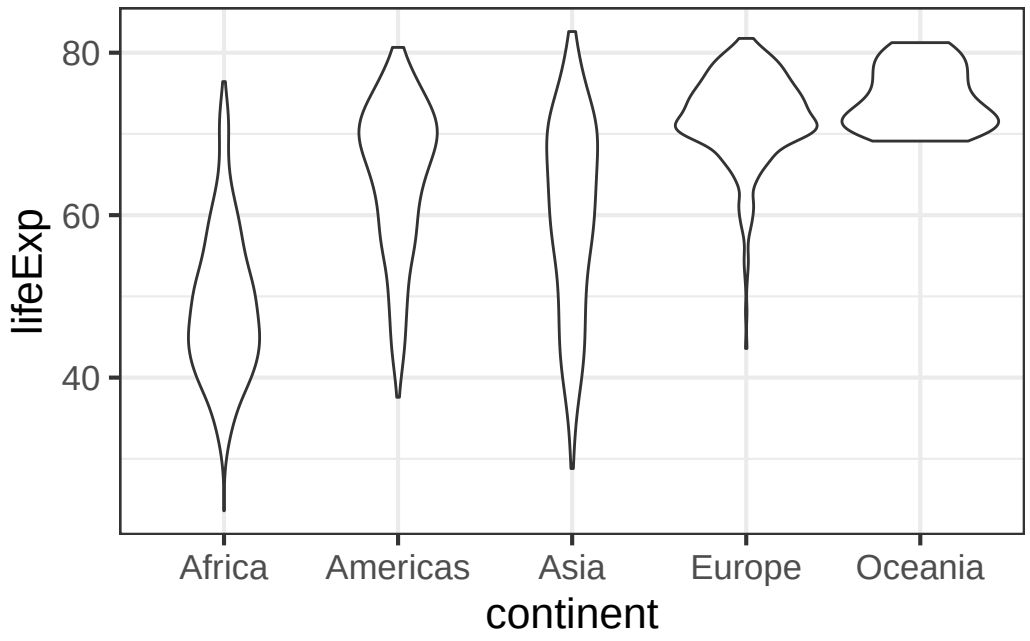
the boxplot is now superimposed over the jitter layer. Let's make the jitter layer go on top. Also, go back to just the boxplots. Notice that the outliers are represented as points. But there's no distinction between the outlier point from the boxplot geom and all the other points from the jitter geom. Let's change that. Notice the British spelling.

```
p + geom_boxplot(outlier.colour = "red") + geom_jitter(alpha=1/2)
```

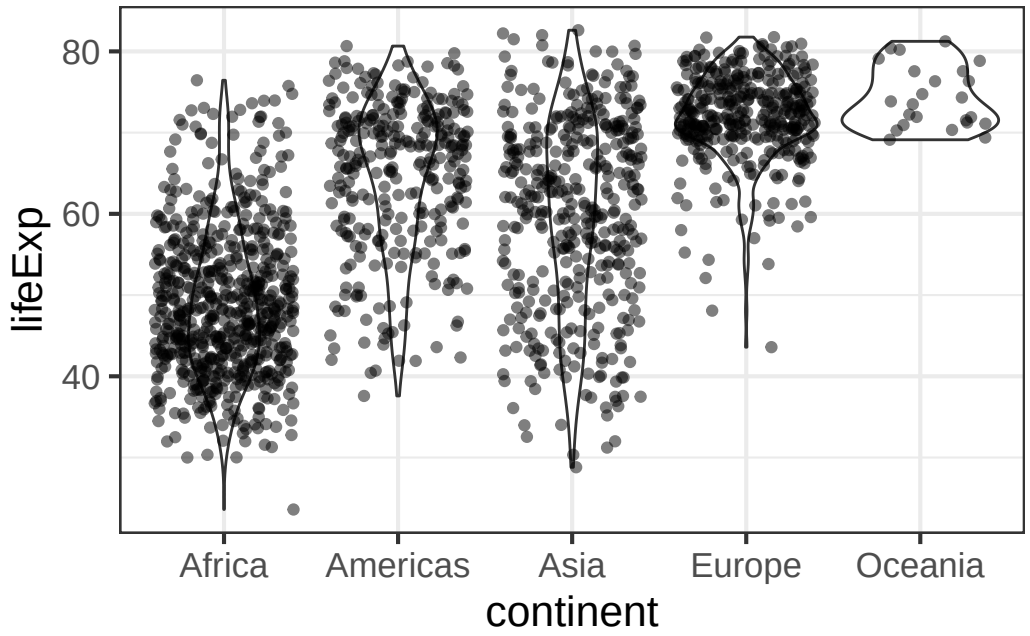


There's another geom that's useful here, called a voilin plot.

```
p + geom_violin()
```

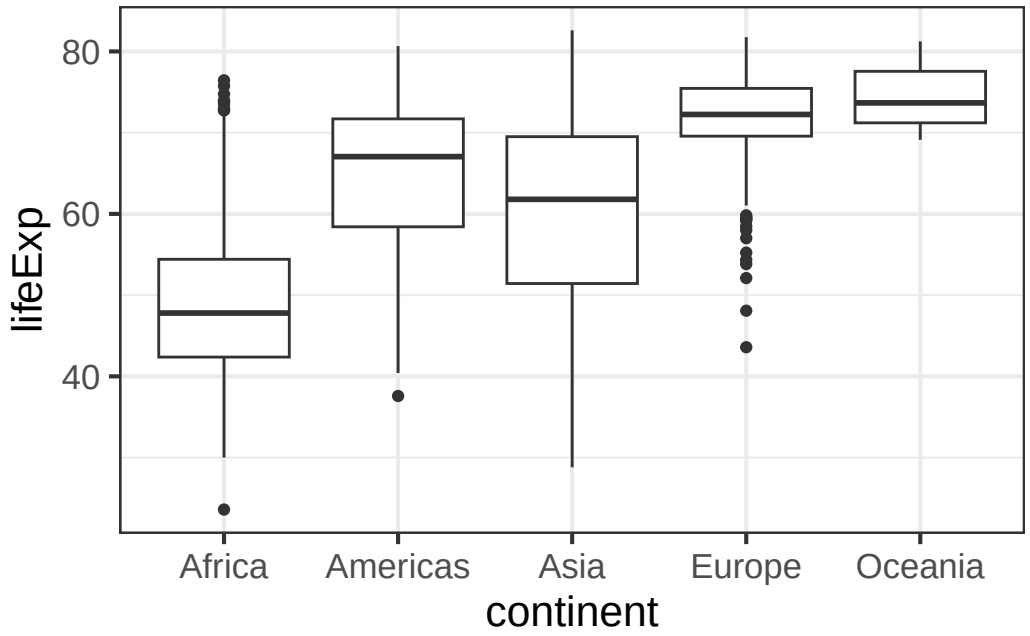



```
p + geom_violin() + geom_jitter(alpha=1/2)
```



Let's go back to our boxplot for a moment.

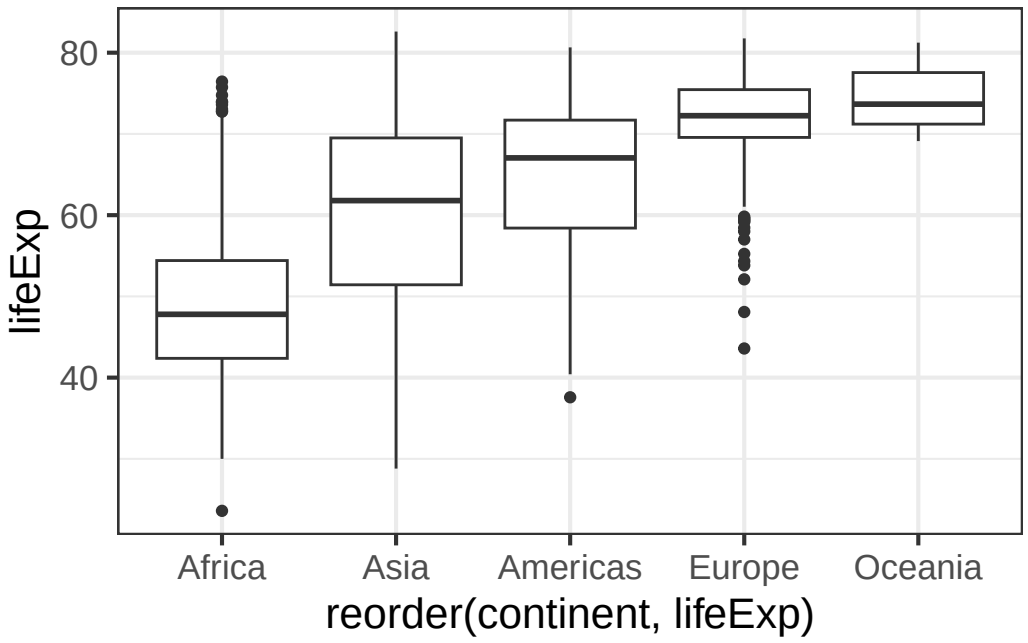
```
p + geom_boxplot()
```



This plot would be a lot more effective if the continents were shown in some sort of order other than alphabetical. To do that, we'll have to go back to our basic build of the plot again and use the `reorder` function in our original aesthetic mapping. Here, reorder is taking the first variable, which is some categorical variable, and ordering it by the level of the mean of the second variable, which is a continuous variable. It looks like this

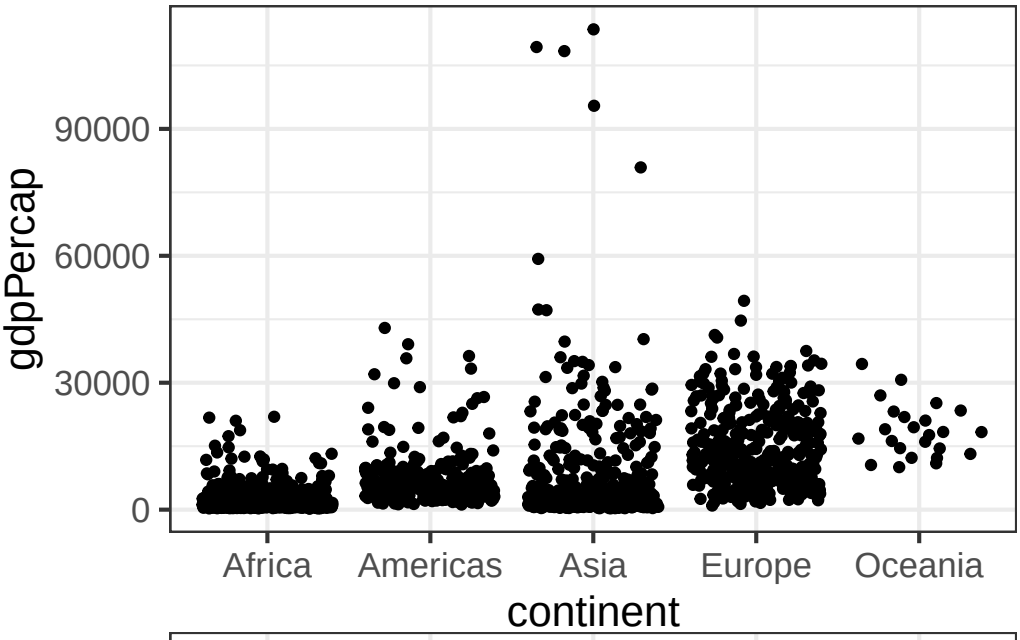
```
p <- ggplot(gm, aes(x=reorder(continent, lifeExp), y=lifeExp))
```

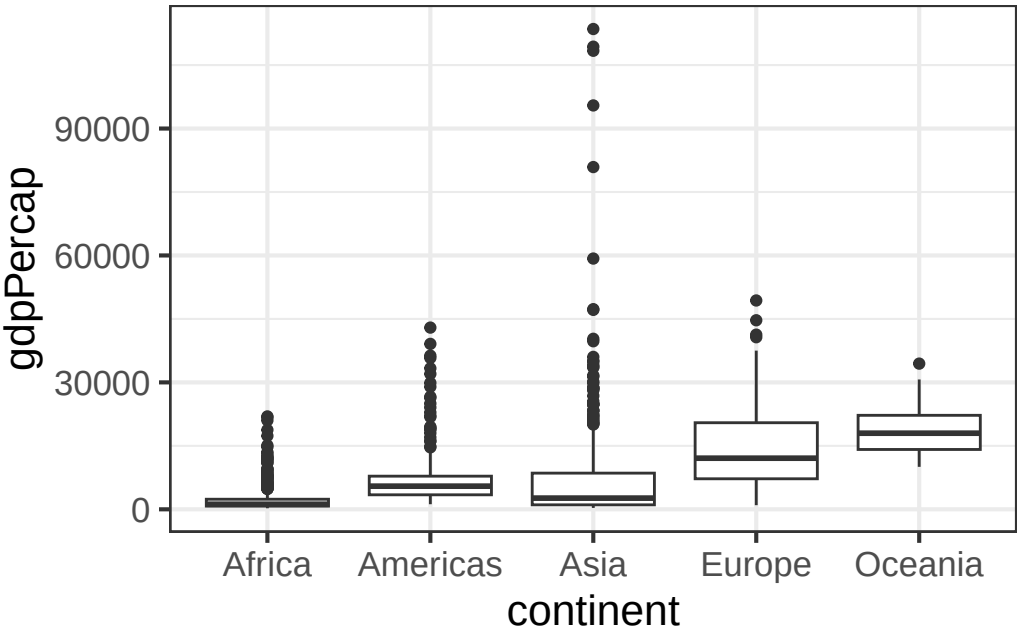
```
p + geom_boxplot()
```

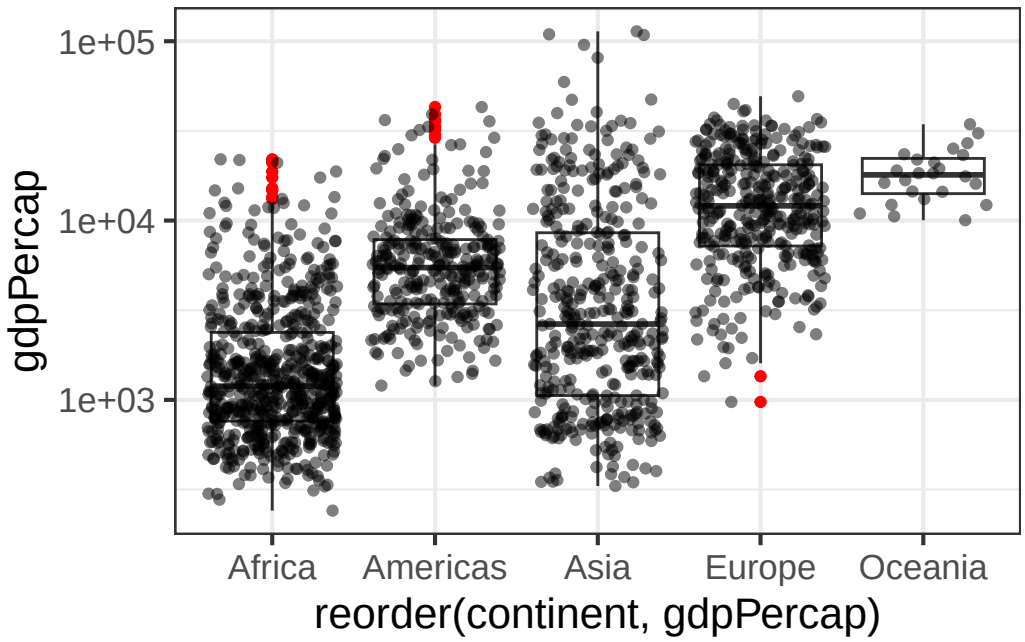


Exercise 3

1. Make a jittered strip plot of GDP per capita against continent.
2. Make a box plot of GDP per capita against continent.
3. Using a log10 y-axis scale, overlay semitransparent jittered points on top of box plots, where outlying points are colored.
4. **BONUS**: Try to reorder the continents on the x-axis by GDP per capita. Why isn't this working as expected? See `?reorder` for clues.

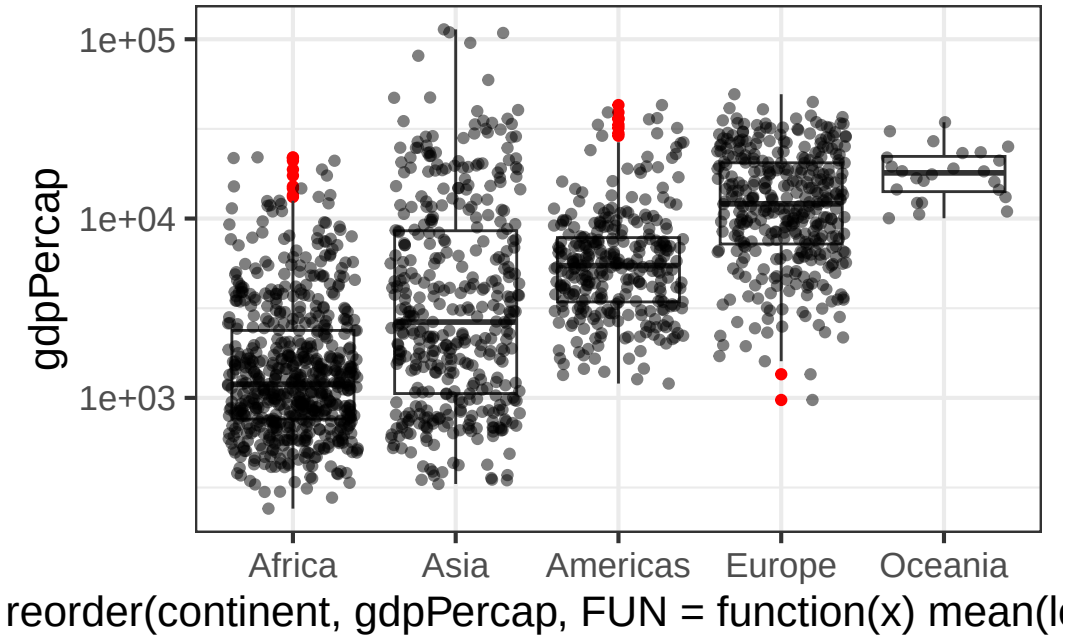

gdpPercap
90000
60000
30000
0
Africa
Americas
Asia
Europe
Oceania
continent


gdpPercap
90000
60000
30000
0
Africa
Americas
Asia
Europe
Oceania
continent




```
# A tibble: 5 x 2
  continent `mean(gdpPercap)`
  <chr>                   <dbl>
1 Africa                  2194.
2 Americas                7136.
3 Asia                    7902.
4 Europe                 14469.
5 Oceania                18622.

# A tibble: 5 x 2
  continent `mean(log10(gdpPercap))`
  <chr>                          <dbl>
1 Africa                          3.15
2 Americas                        3.74
3 Asia                            3.51
4 Europe                          4.06
5 Oceania                         4.25
```

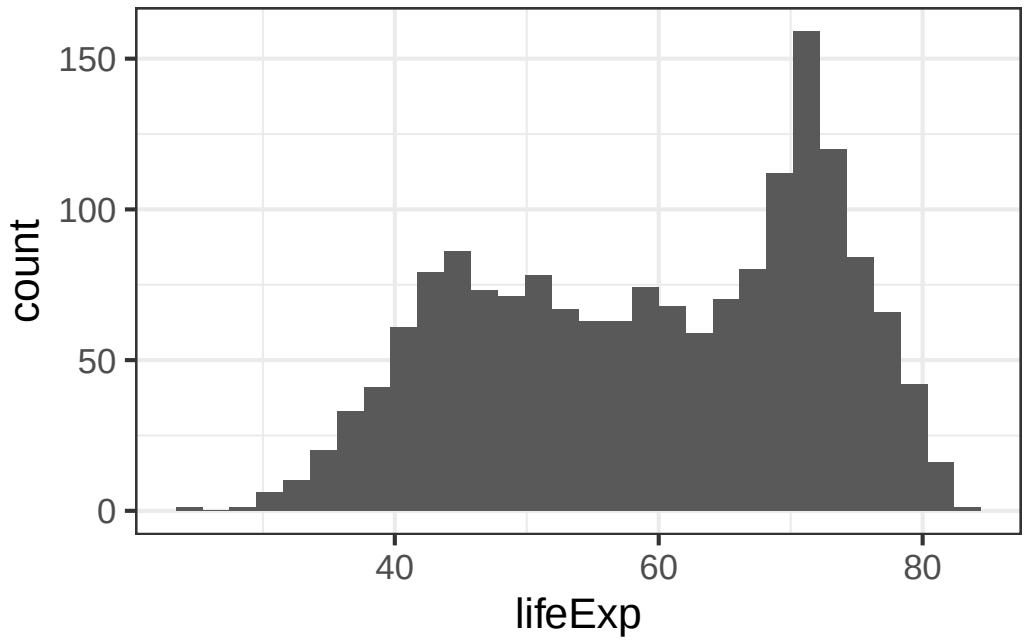


4.5 Plotting univariate continuous data

What if we just wanted to visualize distribution of a single continuous variable? A histogram is the usual go-to visualization. Here we only have one aesthetic mapping instead of two.

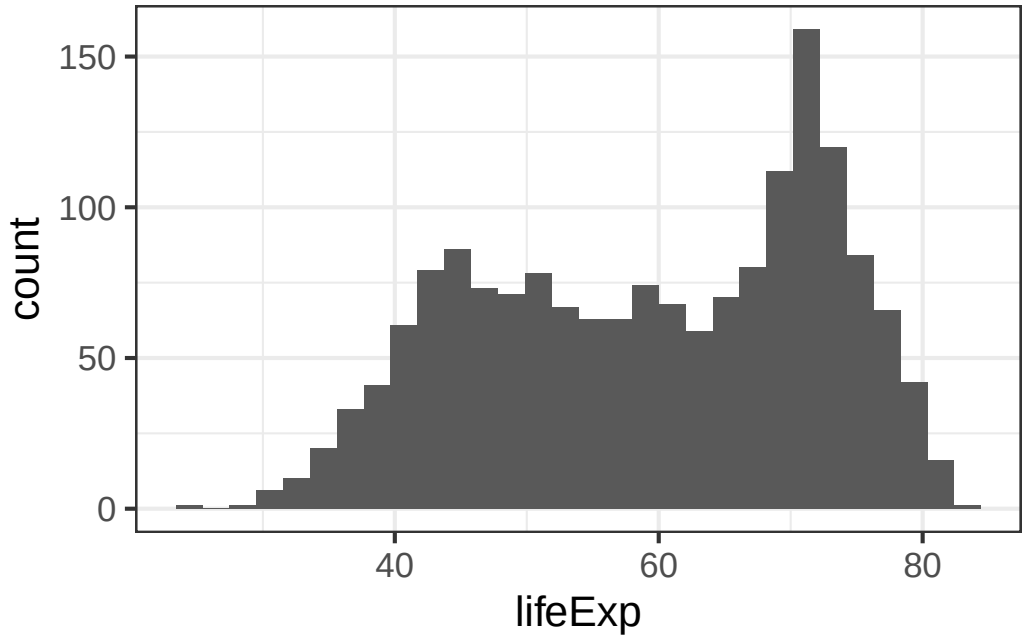
```
p <- ggplot(gm, aes(lifeExp))
```

```
p + geom_histogram()
```

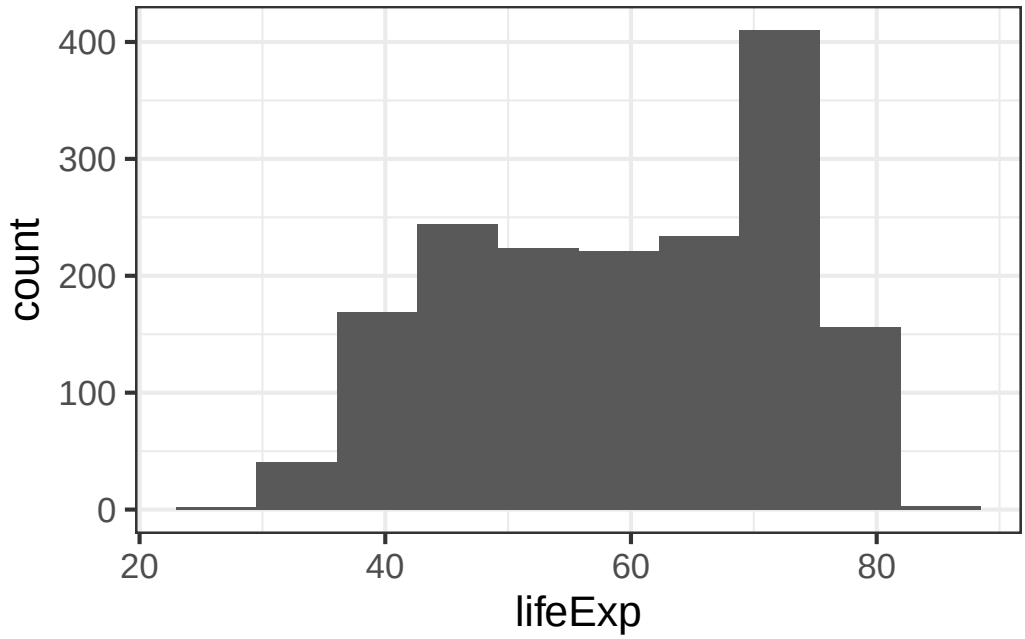


When we do this ggplot lets us know that we're automatically selecting the width of the bins, and we might want to think about this a little further.

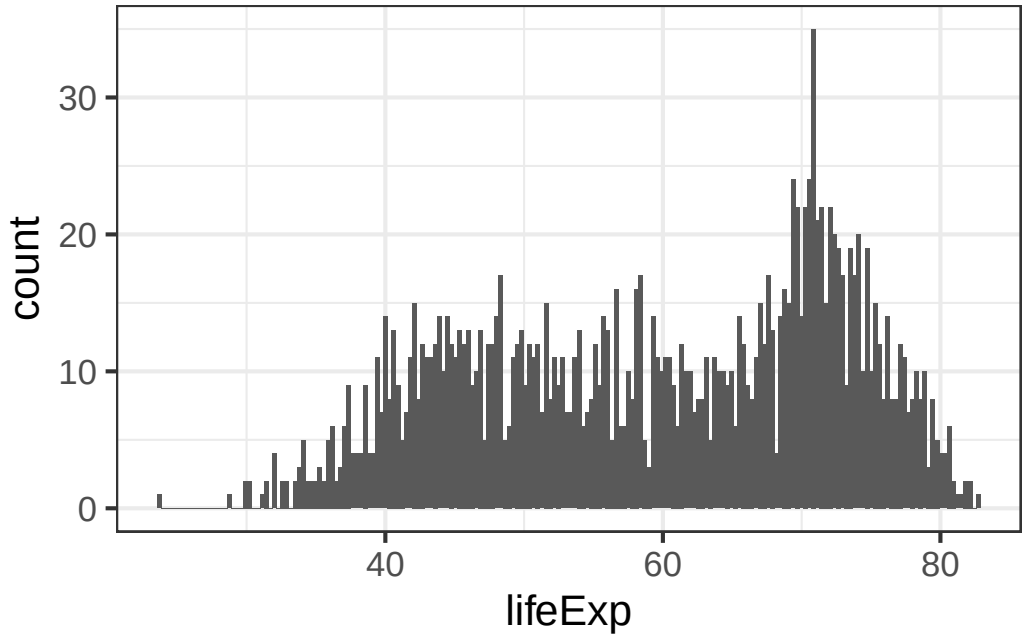
```
p + geom_histogram(bins=30)
```



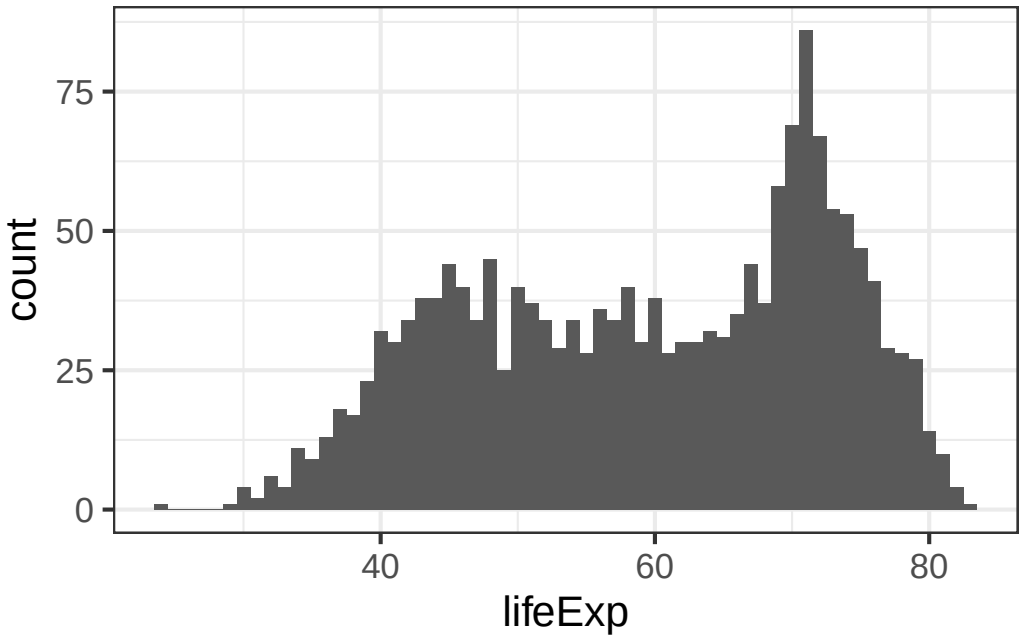
```
p + geom_histogram(bins=10)
```




```
p + geom_histogram(bins=200)
```

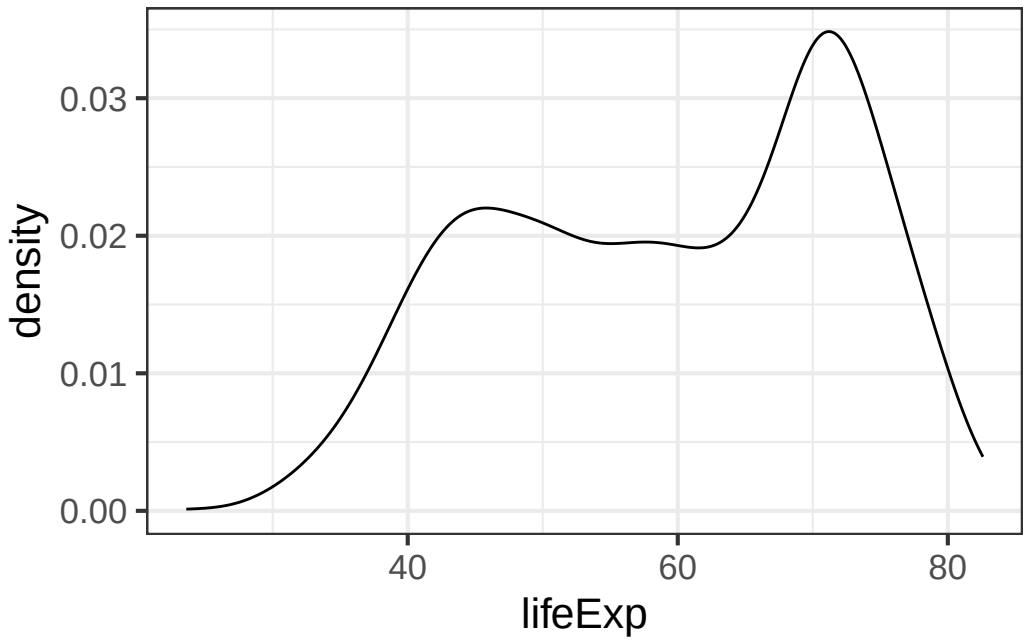


```
p + geom_histogram(bins=60)
```



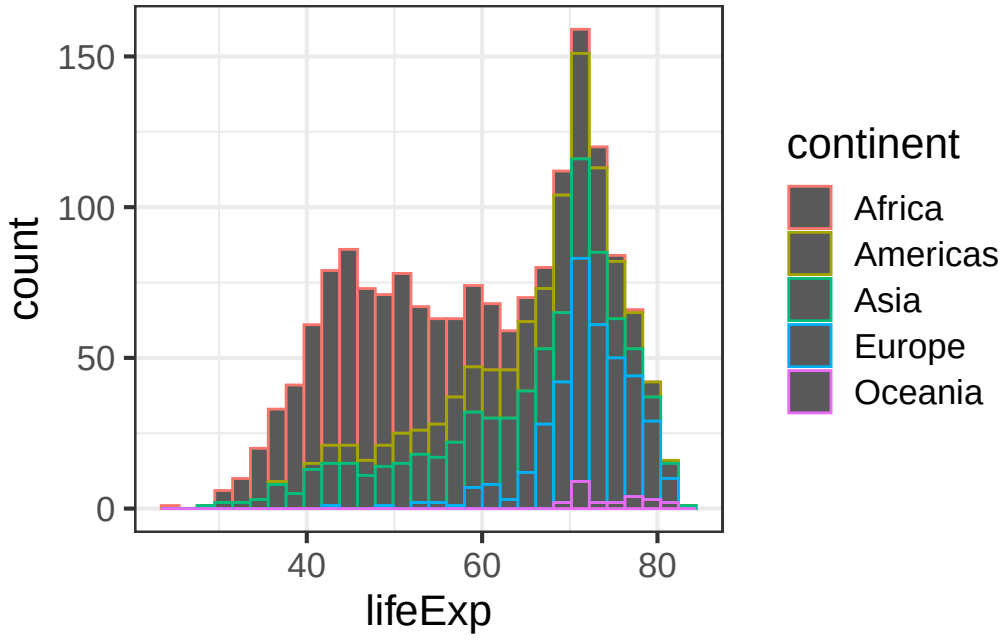
Alternative we could plot a smoothed density curve instead of a histogram:

```
p + geom_density()
```



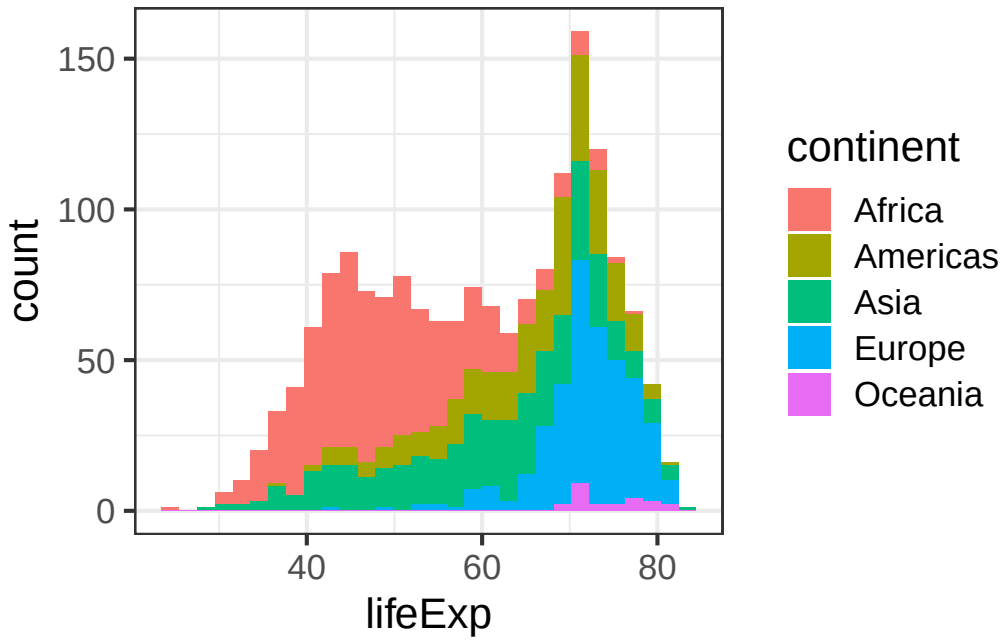
Back to histograms. What if we wanted to color this by continent?

```
p + geom_histogram(aes(color=continent))
```



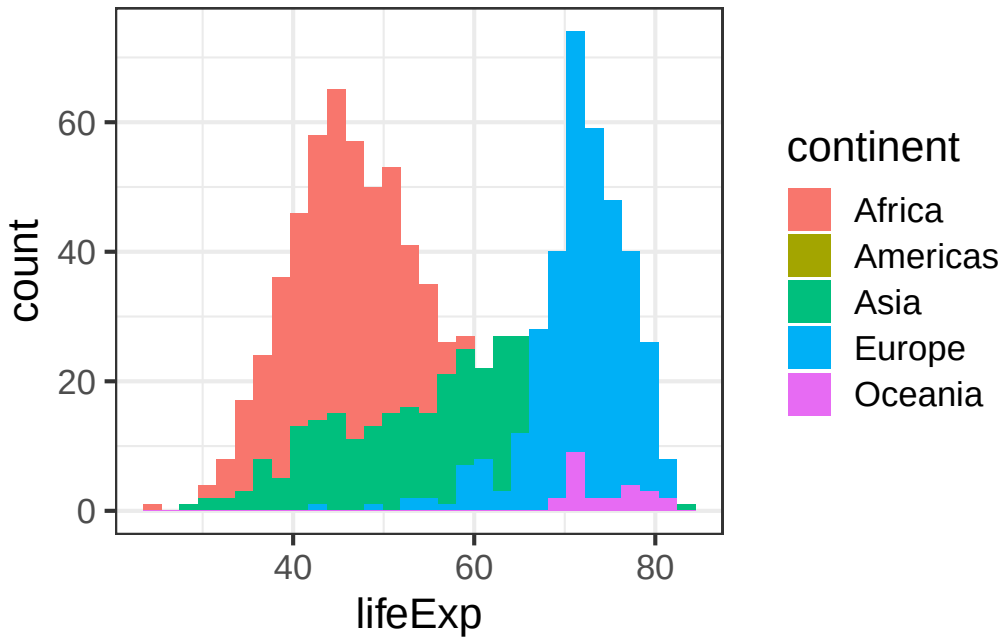
That's not what we had in mind. That's just the outline of the bars. We want to change the *fill* color of the bars.

```
p + geom_histogram(aes(fill=continent))
```



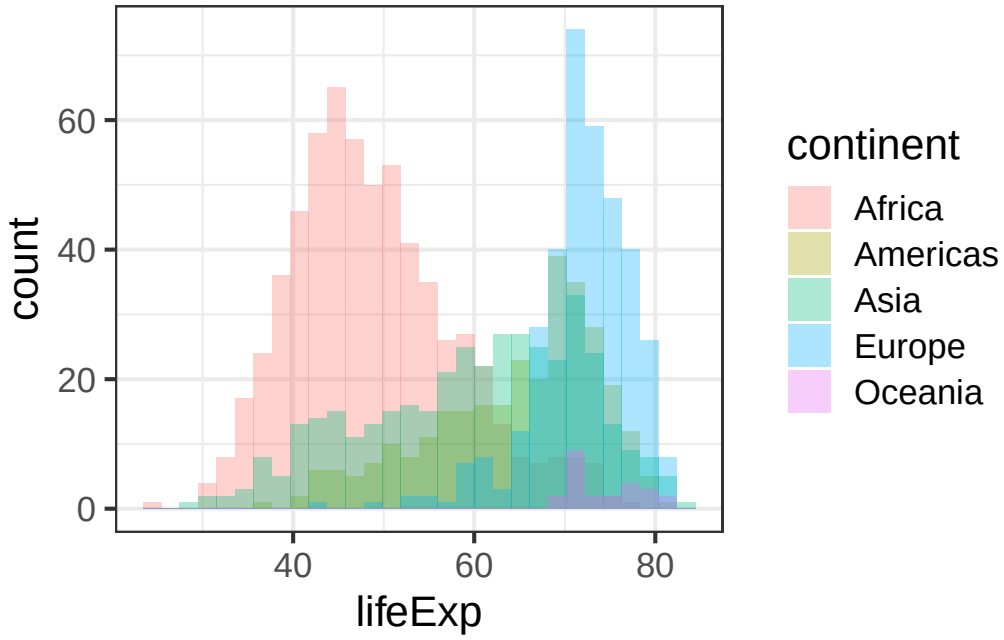
Well, that's not exactly what we want either. If you look at the help for `?geom_histogram` you'll see that by default it stacks overlapping points. This isn't really an effective visualization. Let's change the position argument.

```
p + geom_histogram(aes(fill=continent), position="identity")
```



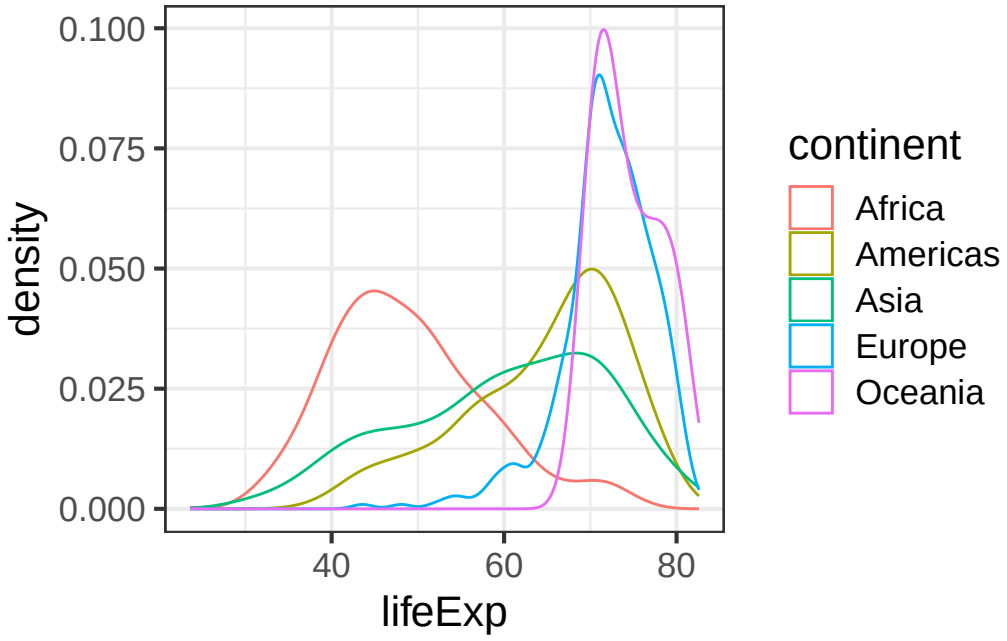
But the problem there is that the histograms are blocking each other. What if we tried transparency?

```
p + geom_histogram(aes(fill=continent), position="identity", alpha=1/3)
```



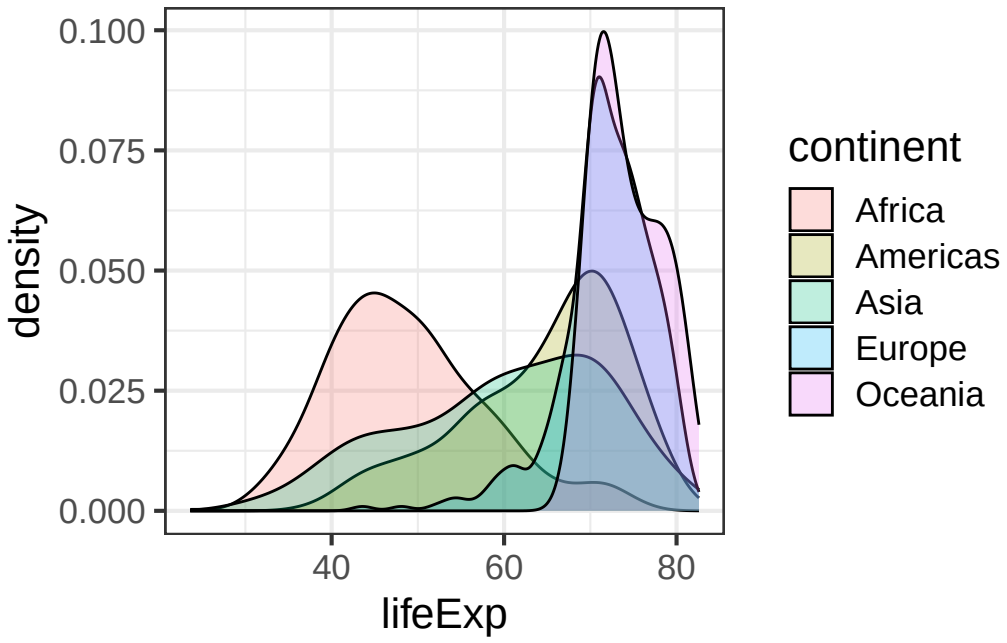
That's somewhat helpful, and might work for two distributions, but it gets cumbersome with 5. Let's go back and try this with density plots, first changing the color of the line:

```
p + geom_density(aes(color=continent))
```



Then by changing the color of the fill and setting the transparency to 25%:

```
p + geom_density(aes(fill=continent), alpha=1/4)
```



Exercise 4

1. Plot a histogram of GDP Per Capita.
2. Do the same but use a log10 x-axis.
3. Still on the log10 x-axis scale, try a density plot mapping continent to the fill of each density distribution, and reduce the opacity.
4. Still on the log10 x-axis scale, make a histogram faceted by continent *and* filled by continent. Facet with a single column (see `?facet_wrap` for help).
5. Save this figure to a 6x10 PDF file.

4.6 Publication-ready plots & themes

Let's make a plot we made earlier (life expectancy versus the log of GDP per capita with points colored by continent with lowess smooth curves overlaid without the standard error ribbon):

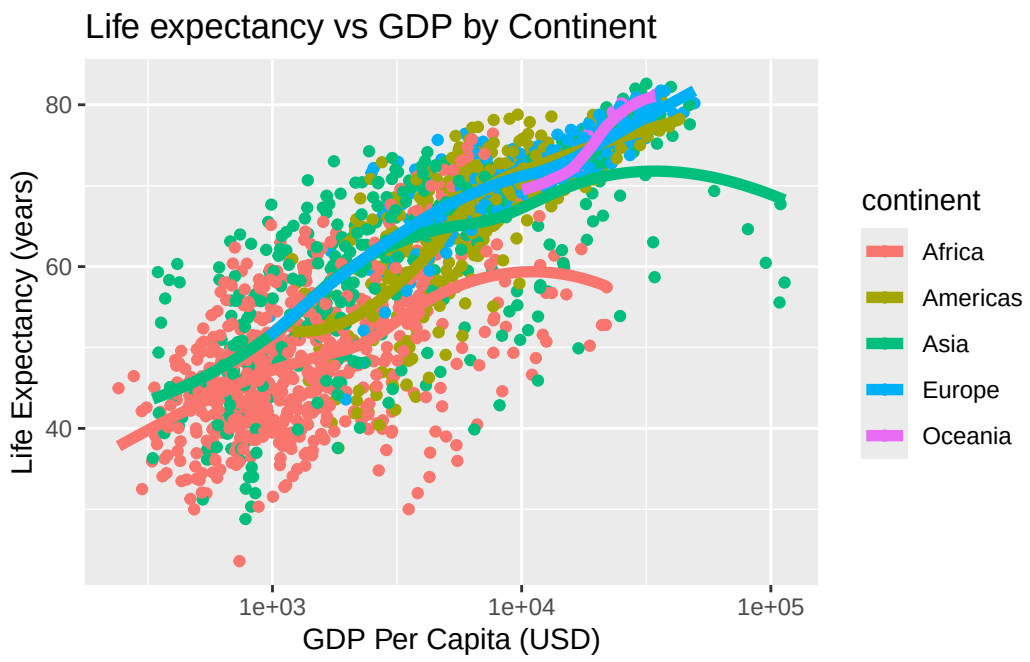
```
p <- ggplot(gm, aes(gdpPercap, lifeExp))
p <- p + scale_x_log10()
p <- p + aes(col=continent) + geom_point() + geom_smooth(lwd=2, se=FALSE)
```

Give the plot a title and axis labels:

```
p <- p + ggtitle("Life expectancy vs GDP by Continent")
p <- p + xlab("GDP Per Capita (USD)") + ylab("Life Expectancy (years)")
```

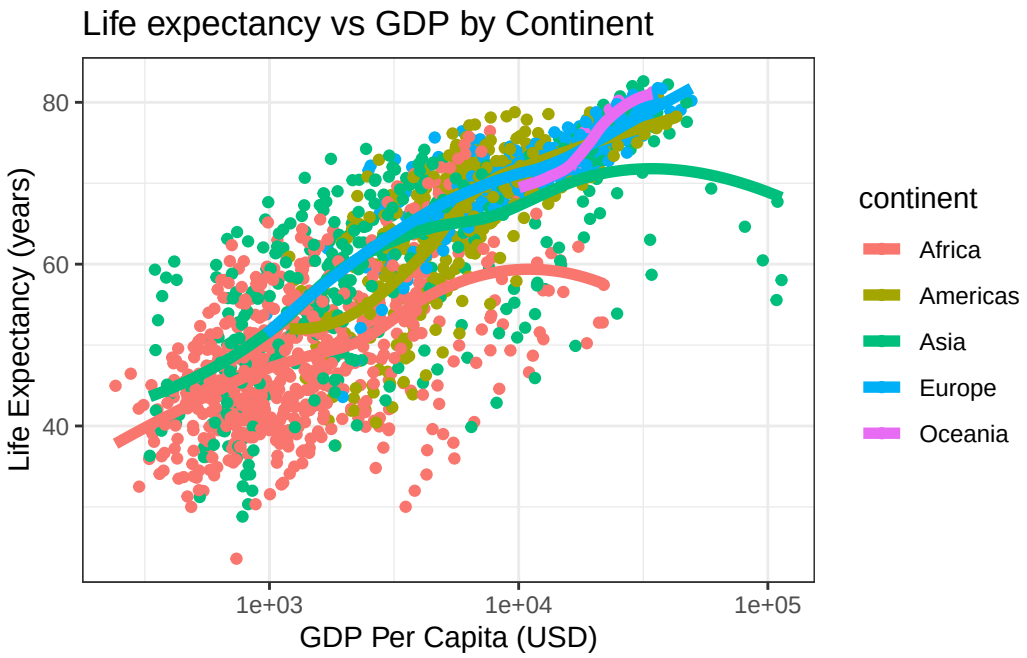
By default, the "gray" theme is the usual background (I've changed this course website to use the black and white background for all images).

```
p + theme_gray()
```



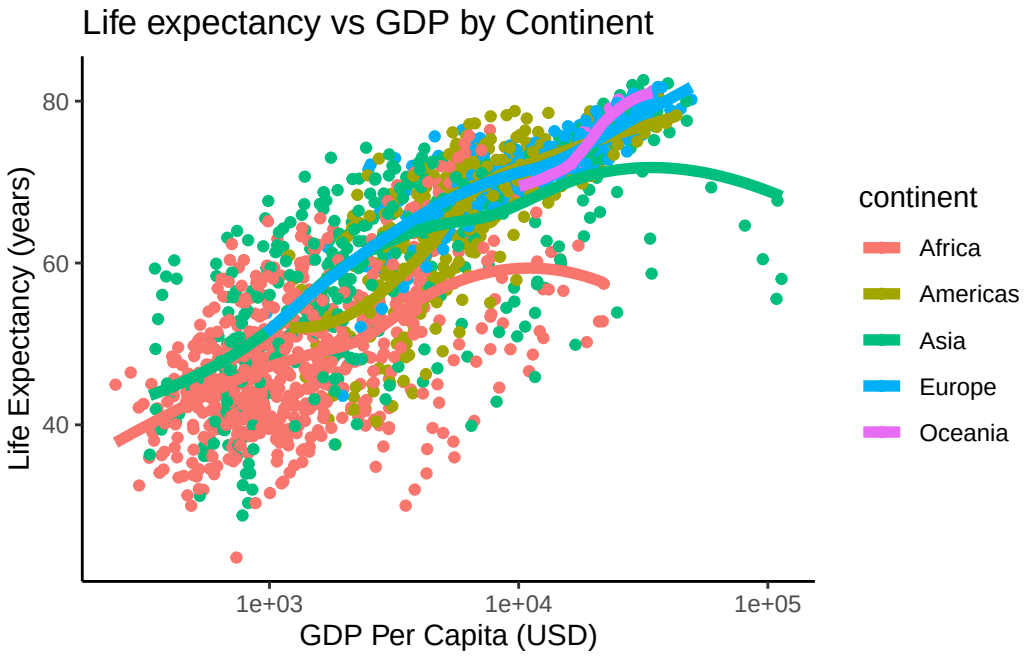
We could also get a black and white background:

```
p + theme_bw()
```



Or go a step further and remove the gridlines:

```
p + theme_classic()
```



Finally, there's another package that gives us lots of different themes. Install it if you don't have it already. Install all its dependencies along with it.

```
install.packages("ggthemes", dependencies = TRUE)
```

```
library(ggthemes)
p <- ggplot(gm, aes(gdpPercap, lifeExp))
p <- p + scale_x_log10()
p <- p + aes(col=continent) + geom_point() + geom_smooth(lwd=2, se=FALSE)
p + theme_excel()
p + theme_excel() + scale_colour_excel()
p + theme_gdocs() + scale_colour_gdocs()
p + theme_stata() + scale_colour_stata()
p + theme_wsj() + scale_colour_wsj()
p + theme_economist()
p + theme_fivethirtyeight()
p + theme_tufte()
```

5 Tidy EDA

Not much to see here...

6 R Markdown

Not much to see here...

Part II

Electives

7 Essential Statistics

Not much to see here...

8 Survival Analysis

Not much to see here...

9 Predictive Modeling

Not much to see here...

10 Probabilistic Forecasting

Not much to see here...

11 Text Mining

Not much to see here...

12 Phylogenetic Trees

Not much to see here...

13 RNA-seq

Not much to see here...

Summary

In summary, this book has no content whatsoever.

References

Bryan, Jennifer. 2019. "STAT 545: Data Wrangling, Exploration, and Analysis with r." https://stat545.com/.

Love, Michael I., Wolfgang Huber, and Simon Anders. 2014. "Moderated Estimation of Fold Change and Dispersion for RNA-seq Data with DESeq2." *Genome Biology* 15 (12): 1–21.

Robinson, David. 2015. "Variance Explained." http://varianceexplained.org/.

Silge, Julia, and David Robinson. 2017. *Text Mining with R: A Tidy Approach*. 1st edition. Beijing ; Boston: O'Reilly Media.

Teal, Tracy K., Karen A. Cranston, Hilmar Lapp, Ethan White, Greg Wilson, Karthik Ram, and Aleksandra Pawlik. 2015. "Data Carpentry: Workshops to Increase Data Literacy for Researchers."

Wilson, Greg. 2014. "Software Carpentry: Lessons Learned." *F1000Research* 3.

Yu, Guangchuang. 2022. "Ggtree: An r Package for Visualization of Tree and Annotation Data." http://bioconductor.org/packages/ggtree/.

Yu, Guangchuang, David K. Smith, Huachen Zhu, Yi Guan, and Tommy Tsan-Yuk Lam. 2017. "Ggtree: An R Package for Visualization and Annotation of Phylogenetic Trees with Their Covariates and Other Associated Data." *Methods in Ecology and Evolution* 8 (1): 28–36.

A Setup

A.1 Software

A.2 Data

1. **Option 1: Download all the data**. Download and extract **this zip file** (11.36 Mb) with all the data for the entire workshop. This may include additional datasets that we won't use here.
2. **Option 2: Download individual datasets as needed.**
    - Create a new folder somewhere on your computer that's easy to get to (e.g., your Desktop). Name it `bds`. Inside that folder, make a folder called `data`, all lowercase.
    - Download individual data files as needed, saving them to the new `bdsr/data` folder you just made. Click to download. If data displays in your browser, right-click and select *Save link as…* (or similar) to save to the desired location.

- data/airway_metadata.csv
- data/airway_scaledcounts.csv
- data/annotables_grch38.csv
- data/austen.csv
- data/brauer2007_messy.csv
- data/brauer2007_sysname2go.csv
- data/brauer2007_tidy.csv
- data/dmd.csv
- data/flu_genotype.csv
- data/gapminder.csv
- data/grads_dd.csv
- data/grads.csv
- data/h7n9_analysisready.csv
- data/h7n9.csv
- data/heartrate2dose.csv
- data/ilinet.csv
- data/movies_dd.csv
- data/movies_imdb.csv
- data/movies.csv

- data/nhanes_dd.csv
- data/nhanes.csv
- data/SRP026387_metadata.csv
- data/SRP026387_scaledcounts.csv
- data/stressEcho.csv

B Additional Resources

Not much to see here...