Into the `tidyverse`

Shuxiao Chen

1/22/2021

Many thanks to Shuxiao Chen, who prepared this tutorial for STAT 471 in Spring 2021. You can view the corresponding video here.

Data science workflow

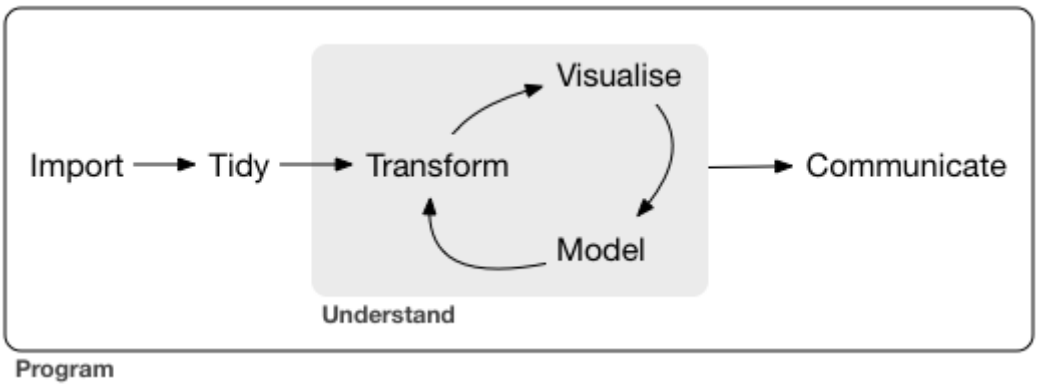


Figure 1: Data science workflow (R for Data Science).

We will cover Chapter 3 (visualization), 5 (transformation), 11 (import), and 12 (tidy data) of R for Data Science (R4DS).

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

Importing data

The materials in this section comes from Chapter 11 of R4DS.

The basics

The `readr` package can read all kinds of data into R. The functions related to data importing are named as `read_something()`. For example:

- `read_csv()` reads *comma separated values* files;
- `read_tsv()` reads *tab separated values* files;
- `read_delim()` reads files with an arbitrary (but user-defined) delimiter.

All such functions share a similar syntax and we will focus on `read_csv` here.

```r
# read .csv files from a path
heights <- read_csv("../data/heights.csv")
```

```
## Rows: 1192 Columns: 6
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): sex, race
## dbl (4): earn, height, ed, age
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
heights
```

```
## # A tibble: 1,192 x 6
##     earn height sex       ed   age race
##    <dbl>  <dbl> <chr>  <dbl> <dbl> <chr>
##  1 50000   74.4 male      16    45 white
##  2 60000   65.5 female    16    58 white
##  3 30000   63.6 female    16    29 white
##  4 50000   63.1 female    16    91 other
##  5 51000   63.4 female    17    39 white
##  6  9000   64.4 female    15    26 white
##  7 29000   61.7 female    12    49 white
##  8 32000   72.7 male      17    46 white
##  9  2000   72.0 male      15    21 hispanic
## 10 27000   72.2 male      12    26 white
## # ... with 1,182 more rows
```

```r
# read an inline csv file
manual_data <- read_csv(
  "a,b,c
  1,2,3
  4,5,6")
```

```
## Rows: 2 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): a, b, c
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
manual_data
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

```r
# skip a few lines
manual_data <- read_csv(
  "The first line of metadata
  The second line of metadata
```

```
  x,y,z
  1,2,3", skip = 2)
```

```
## Rows: 1 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): x, y, z
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
manual_data
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

```
# skip all lines that start with a "#" sign
manual_data <- read_csv(
  "# A comment I want to skip
  x,y,z
  1,2,3", comment = "#")
```

```
## Rows: 1 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): x, y, z
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
manual_data
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

```
# read files without headings
manual_data <- read_csv(
  "1,2,3
  4,5,6", col_names = FALSE)
```

```
## Rows: 2 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): X1, X2, X3
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
manual_data
```

```
## # A tibble: 2 x 3
```

```
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

```
# read files without headings + specify the headings
manual_data <- read_csv(
  "1,2,3
  4,5,6", col_names = c("x", "y", "z"))
```

```
## Rows: 2 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): x, y, z
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
manual_data
```

```
## # A tibble: 2 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

```
# dealing with NA (not available, e.g., missing) values
manual_data <- read_csv(
  "a,b,c
  1,2,.", na = ".")
```

```
## Rows: 1 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (2): a, b
## lgl (1): c
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
manual_data
```

```
## # A tibble: 1 x 3
##       a     b c
##   <dbl> <dbl> <lgl>
## 1     1     2 NA
```

Manually specifying column types

```
heights <- read_csv(
  "../data/heights.csv",
  col_types = cols(
      earn = col_double(),
      height = col_double(),
      sex = col_factor(),
```

```
      ed = col_integer(),
      age = col_integer(),
      race = col_factor()
  )
)

heights
```

```
## # A tibble: 1,192 x 6
##     earn height sex       ed   age race
##    <dbl>  <dbl> <fct>  <int> <int> <fct>
##  1 50000   74.4 male      16    45 white
##  2 60000   65.5 female    16    58 white
##  3 30000   63.6 female    16    29 white
##  4 50000   63.1 female    16    91 other
##  5 51000   63.4 female    17    39 white
##  6  9000   64.4 female    15    26 white
##  7 29000   61.7 female    12    49 white
##  8 32000   72.7 male      17    46 white
##  9  2000   72.0 male      15    21 hispanic
## 10 27000   72.2 male      12    26 white
## # ... with 1,182 more rows
```

Tidying Data

The materials in this section comes from Chapter 12 of R4DS.

After getting your data into R, you need to bring it into a form that will be easy to analyze. This form is called "tidy data" and the process of bringing it into that form is called "tidying." Most of the datasets you encounter in homework problems will already be tidy, but when you go out into the real world to find data (including for your final projects!), it will likely be much more messy.

What kinds of data are "tidy"?

A single dataset can be represented in multiple ways.

```
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
```

```
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```
# stores cases
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
# stores population
table4b
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

How to make a dataset tidy

pivot_longer()

Let us look at `table4a` more closely.

```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
```

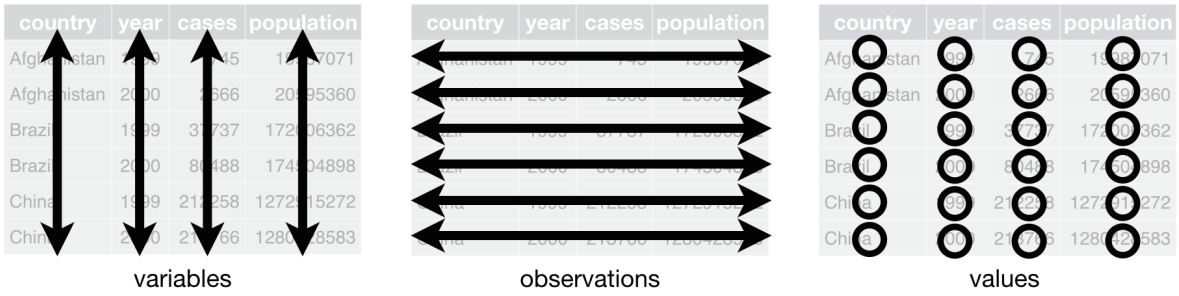


Figure 2: Rules of tidy data.

```
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

`table4a` itself is not tidy: 1999 and 2000 are values, not variables. Ideally we want to modify it as follows:

Figure 3: Pivot `table4a` to be longer.

We then pivot it to be longer.

```
table4a %>%
  pivot_longer(cols = c(`1999`, `2000`),
               names_to = "year",
               values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

pivot_wider()

Let us look at `table2` more closely.

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases            37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases            80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases           212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases           213766
## 12 China        2000 population 1280428583
```

`table2` is not tidy: one observation is scattered across two rows. Ideally we want to modify it as follows:

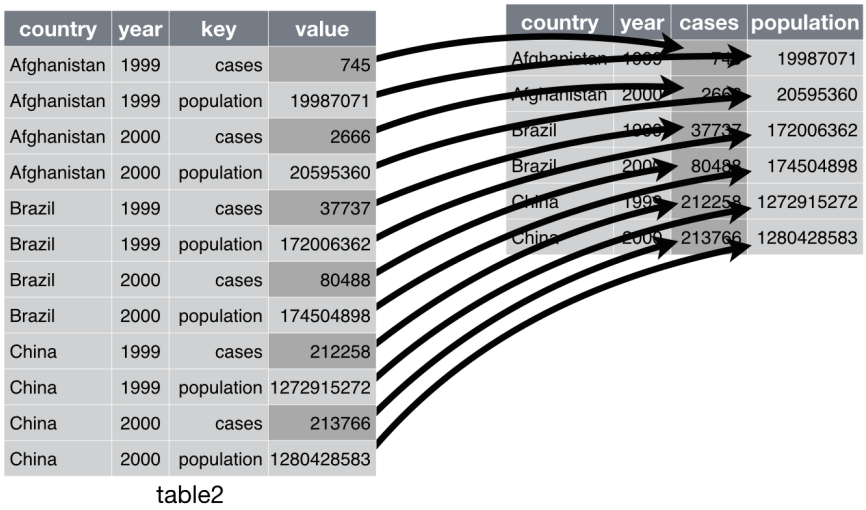


Figure 4: Pivot `table2` to be wider.

We then pivot it to be wider.

```
table2 %>%
    pivot_wider(names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

separate()

Let us look at `table3` more closely.

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

`table3` is not tidy: two values are squeezed into one cell.

```
table3 %>%
  separate(col = rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

unite()

Let us look at a new `table5`:

```
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
## * <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

```
table5 %>%
  unite(col = new, century, year)
```

```
## # A tibble: 6 x 3
##   country     new   rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 19_99 745/19987071
## 2 Afghanistan 20_00 2666/20595360
## 3 Brazil      19_99 37737/172006362
## 4 Brazil      20_00 80488/174504898
## 5 China       19_99 212258/1272915272
## 6 China       20_00 213766/1280428583
```

Dealing with missing values

Types of missing-ness

A value can be missing in two possible ways:

- *Explicitly*, i.e, flagged with `NA`.
- *Implicitly*, i.e. simply not present in the data.

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)

stocks
```

```
## # A tibble: 7 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

The return of the observation with `year=2015, qtr=4` is *explicitly* missing, whereas the observation with `year=2016, qtr=1` is *implicitly* missing.

complete()

We make the implicitly missing values explicit by `complete()`.

```
stocks %>%
  complete(year, qtr)
```

```
## # A tibble: 8 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     1  NA
## 6  2016     2   0.92
## 7  2016     3   0.17
## 8  2016     4   2.66
```

fill()

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
```

```
)
treatment
```

```
## # A tibble: 4 x 3
##   person           treatment response
##   <chr>                <dbl>    <dbl>
## 1 Derrick Whitmore         1        7
## 2 <NA>                     2       10
## 3 <NA>                     3        9
## 4 Katherine Burke          1        4
```

The `fill()` function takes a set of columns where you want missing values to be replaced by the most recent non-missing value (sometimes called last observation carried forward).

```
treatment %>%
  fill(person)
```

```
## # A tibble: 4 x 3
##   person           treatment response
##   <chr>                <dbl>    <dbl>
## 1 Derrick Whitmore         1        7
## 2 Derrick Whitmore         2       10
## 3 Derrick Whitmore         3        9
## 4 Katherine Burke          1        4
```

Transforming Data

Overview

Below is a list of popular `dplyr` commands. We will go through each one.

- `filter()`: filter out rows (i.e., observations) according to certain conditions;
- `select()`: select columns (i.e., variables) according to certain conditions;
- `distinct()`:
- `arrange()`: re-order rows;
- `rename()`: rename columns;
- `mutate()`: create new columns;
- `group_by()`: "split" dataset into groups;
- `summarise()`: creating summary statistics.

To illustrate these commands, we will use a pre-existing dataset the contains 336,776 flights that departed from New York City in 2013. `Dplyr` allows you to gather insight from a dataset without altering the original dataset. It is considered best practice not to alter the original dataset. For example in this case, we will never overwrite the existing dataset 'flights'. We will first take a look at the summary statistics.

```
# install.packages("nycflights13")
library(nycflights13)
summary(flights)
```

```
##       year          month             day          dep_time    sched_dep_time
##  Min.   :2013   Min.   : 1.000   Min.   : 1.00   Min.   :   1   Min.   : 106
##  1st Qu.:2013   1st Qu.: 4.000   1st Qu.: 8.00   1st Qu.: 907   1st Qu.: 906
##  Median :2013   Median : 7.000   Median :16.00   Median :1401   Median :1359
##  Mean   :2013   Mean   : 6.549   Mean   :15.71   Mean   :1349   Mean   :1344
##  3rd Qu.:2013   3rd Qu.:10.000   3rd Qu.:23.00   3rd Qu.:1744   3rd Qu.:1729
##  Max.   :2013   Max.   :12.000   Max.   :31.00   Max.   :2400   Max.   :2359
##                                                  NA's   :8255
```

```
##    dep_delay          arr_time    sched_arr_time   arr_delay
##  Min.   : -43.00   Min.   :   1   Min.   :   1   Min.   : -86.000
##  1st Qu.:  -5.00   1st Qu.:1104   1st Qu.:1124   1st Qu.: -17.000
##  Median :  -2.00   Median :1535   Median :1556   Median :  -5.000
##  Mean   :  12.64   Mean   :1502   Mean   :1536   Mean   :   6.895
##  3rd Qu.:  11.00   3rd Qu.:1940   3rd Qu.:1945   3rd Qu.:  14.000
##  Max.   :1301.00   Max.   :2400   Max.   :2359   Max.   :1272.000
##  NA's   :8255      NA's   :8713                  NA's   :9430
##    carrier             flight       tailnum             origin
##  Length:336776      Min.   :   1   Length:336776      Length:336776
##  Class :character   1st Qu.: 553   Class :character   Class :character
##  Mode  :character   Median :1496   Mode  :character   Mode  :character
##                     Mean   :1972
##                     3rd Qu.:3465
##                     Max.   :8500
##
##      dest              air_time        distance         hour
##  Length:336776      Min.   : 20.0   Min.   :  17   Min.   : 1.00
##  Class :character   1st Qu.: 82.0   1st Qu.: 502   1st Qu.: 9.00
##  Mode  :character   Median :129.0   Median : 872   Median :13.00
##                     Mean   :150.7   Mean   :1040   Mean   :13.18
##                     3rd Qu.:192.0   3rd Qu.:1389   3rd Qu.:17.00
##                     Max.   :695.0   Max.   :4983   Max.   :23.00
##                     NA's   :9430
##      minute        time_hour
##  Min.   : 0.00   Min.   :2013-01-01 05:00:00
##  1st Qu.: 8.00   1st Qu.:2013-04-04 13:00:00
##  Median :29.00   Median :2013-07-03 10:00:00
##  Mean   :26.23   Mean   :2013-07-03 05:22:54
##  3rd Qu.:44.00   3rd Qu.:2013-10-01 07:00:00
##  Max.   :59.00   Max.   :2013-12-31 23:00:00
##
```

Pipes

The `%>%` command is called a pipe. This means that the result of the code before `%>%` is sent, or "piped", to the one after after `%>%`. Piping is a powerful tool for clearly expressing a sequence of multiple operations, as we will see shortly.

filter()

The filter command will only display the subset of your dataset that match a certain condition. This command will only show flights on Jan 1st, 2013.

```
flights %>%
  filter(month == 1 & day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
```

```
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

This code is the same as doing `filter(flights, month == 1 & day == 1)` since the `%>%` command passes the `flights` dataframe to the filter command.

It is important to remember that this command does not alter the orignal `flight` dataset. If we want to save this subset as its own object, we run the following. Remember the `<-` is the assignment operator in `R`.

```
filteredFlight <- flights %>%
                  filter(month == 1 & day == 1)

filteredFlight
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Multiple conditions can be included in a filter command. The command below shows any flights from Jan through June to PHL or SLC airports.

```
flights %>%
  filter(dest %in% c("PHL","SLC") & month <= 6)
```

```
## # A tibble: 2,116 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      655            655         0     1021           1030
##  2  2013     1     1      908            915        -7     1004           1033
##  3  2013     1     1     1047           1050        -3     1401           1410
##  4  2013     1     1     1245           1245         0     1616           1615
##  5  2013     1     1     1323           1300        23     1651           1608
##  6  2013     1     1     1543           1550        -7     1933           1925
##  7  2013     1     1     1600           1610       -10     1712           1729
##  8  2013     1     1     1909           1912        -3     2239           2237
##  9  2013     1     1     1915           1920        -5     2238           2257
## 10  2013     1     1     2000           2000         0     2054           2110
## # ... with 2,106 more rows, and 11 more variables: arr_delay <dbl>,
```

```
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

select()

Select will only return columns that are listed. In this case, the resulting dataset will consist of the Origin, Destination, and Carrier of flights that were destined for PHL or SLC in the first 6 months of the year. Remember, the pipe command sends the result of the current line to the next line. In this case, the filtered dataset is then piped into the select command.

```
flights %>%
  filter(dest %in% c("PHL","SLC") & month <= 6) %>%
  select(origin, dest, carrier)
```

```
## # A tibble: 2,116 x 3
##    origin dest  carrier
##    <chr>  <chr> <chr>
##  1 JFK    SLC   DL
##  2 LGA    PHL   US
##  3 JFK    SLC   DL
##  4 JFK    SLC   DL
##  5 EWR    SLC   DL
##  6 JFK    SLC   DL
##  7 JFK    PHL   9E
##  8 JFK    SLC   B6
##  9 JFK    SLC   DL
## 10 JFK    PHL   9E
## # ... with 2,106 more rows
```

On the contrary, we can use `-` to deselect columns. If we want to drop `year`, `month` and `day`, we just need to prefix `-` to each column name.

```
flights %>%
  filter(dest %in% c("PHL","SLC") & month <= 6) %>%
  select(-year, -month, -day)
```

```
## # A tibble: 2,116 x 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##       <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
##  1      655            655         0     1021           1030        -9 DL
##  2      908            915        -7     1004           1033       -29 US
##  3     1047           1050        -3     1401           1410        -9 DL
##  4     1245           1245         0     1616           1615         1 DL
##  5     1323           1300        23     1651           1608        43 DL
##  6     1543           1550        -7     1933           1925         8 DL
##  7     1600           1610       -10     1712           1729       -17 9E
##  8     1909           1912        -3     2239           2237         2 B6
##  9     1915           1920        -5     2238           2257       -19 DL
## 10     2000           2000         0     2054           2110       -16 9E
## # ... with 2,106 more rows, and 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

distinct()

Distinct will remove any duplicate rows from the given dataset. Notice in the previous command, it returned a subset with 2116 rows, but with distinct, we can see that only 8 carriers flew to PHL or SLC in the first half of the year.

```
flights %>%
  filter(dest %in% c("PHL","SLC") & month <= 6) %>%
  select(origin, dest, carrier) %>%
  distinct()
```

```
## # A tibble: 8 x 3
##   origin dest  carrier
##   <chr>  <chr> <chr>
## 1 JFK    SLC   DL
## 2 LGA    PHL   US
## 3 EWR    SLC   DL
## 4 JFK    PHL   9E
## 5 JFK    SLC   B6
## 6 EWR    PHL   EV
## 7 JFK    PHL   US
## 8 JFK    PHL   DL
```

arrange()

Arrange puts your data into alphabetical order. In this case the order is first by origin, then descending alphabetical order of the destination, then alphabetical order of carrier.

```
flights %>%
  filter(dest %in% c("PHL","SLC") & month <= 6) %>%
  select(origin, dest, carrier) %>%
  distinct() %>%
  arrange(origin, desc(dest), carrier)
```

```
## # A tibble: 8 x 3
##   origin dest  carrier
##   <chr>  <chr> <chr>
## 1 EWR    SLC   DL
## 2 EWR    PHL   EV
## 3 JFK    SLC   B6
## 4 JFK    SLC   DL
## 5 JFK    PHL   9E
## 6 JFK    PHL   DL
## 7 JFK    PHL   US
## 8 LGA    PHL   US
```

rename()

The Rename function can be used to easily rename a column Header. Here, we rename carrier to airline.

```
flights %>%
  filter(dest %in% c("PHL","SLC") & month <= 6) %>%
  select(origin, dest, carrier) %>%
  distinct() %>%
  arrange(origin, desc(dest), carrier) %>%
  rename(airline = carrier)
```

```
## # A tibble: 8 x 3
##   origin dest  airline
##   <chr>  <chr> <chr>
## 1 EWR    SLC   DL
## 2 EWR    PHL   EV
## 3 JFK    SLC   B6
## 4 JFK    SLC   DL
## 5 JFK    PHL   9E
## 6 JFK    PHL   DL
## 7 JFK    PHL   US
## 8 LGA    PHL   US
```

mutate()

Mutate is used to create new columns based on current ones. This feature is very useful. Here, we create three new variables "gain", "speed", and "gain_per_hour". Notice how "gain_per_hour" uses the column "gain", which was created in the same mutate statement.

```
flights %>%
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    gain_per_hour = gain / (air_time / 60)) %>%
  select(dep_delay, arr_delay, gain,
         distance, distance, air_time,
         speed, gain_per_hour)
```

```
## # A tibble: 336,776 x 7
##    dep_delay arr_delay  gain distance air_time speed gain_per_hour
##        <dbl>     <dbl> <dbl>    <dbl>    <dbl> <dbl>         <dbl>
##  1         2        11    -9     1400      227  370.         -2.38
##  2         4        20   -16     1416      227  374.         -4.23
##  3         2        33   -31     1089      160  408.        -11.6
##  4        -1       -18    17     1576      183  517.          5.57
##  5        -6       -25    19      762      116  394.          9.83
##  6        -4        12   -16      719      150  288.         -6.4
##  7        -5        19   -24     1065      158  404.         -9.11
##  8        -3       -14    11      229       53  259.         12.5
##  9        -3        -8     5      944      140  405.          2.14
## 10        -2         8   -10      733      138  319.         -4.35
## # ... with 336,766 more rows
```

group_by()

This function groups the rows of a dataset according to a column. It is usually used in conjunction with `summerise()` to produce summary statistics of each group.

Here, the origin column had three categories, `EWR`, `JFK`, & `LGA`. The `group_by(origin)` command organizes the data by the three origins. Then `summarise()` is used to get metrics related to each origin.

From this table, we can see that EWR had the most flights with 120835, and LGA had the lowest avg delay at 10.34

```
flights %>%
  group_by(origin) %>%
  summarise(
```

```
    num_of_flights = n(),
    avg_delay = mean(dep_delay, na.rm = TRUE)
  ) # na.rm removes any NA data
```

```
## # A tibble: 3 x 3
##   origin num_of_flights avg_delay
##   <chr>           <int>     <dbl>
## 1 EWR            120835      15.1
## 2 JFK            111279      12.1
## 3 LGA            104662      10.3
```

group_by can also take expressions. The following returns the number of flights that started late but arrived early (or on time), started and arrived late etc.

```
flights %>%
  filter(!is.na(dep_delay) & !is.na(arr_delay)) %>%
  group_by(dep_delay > 0, arr_delay > 0) %>%
  summarise(num_of_flights = n())
```

```
## `summarise()` has grouped output by 'dep_delay > 0'. You can override using the `.groups` argument.
```

```
## # A tibble: 4 x 3
## # Groups:   dep_delay > 0 [2]
##   `dep_delay > 0` `arr_delay > 0` num_of_flights
##   <lgl>           <lgl>                    <int>
## 1 FALSE           FALSE                   158900
## 2 FALSE           TRUE                     40701
## 3 TRUE            FALSE                    35442
## 4 TRUE            TRUE                     92303
```

summarise()

Summarise has a number of other functions that can be used within it. `n_distinct(dest)` returns the number of distinct destinations. From this table we can see that EWR has flights to the largest number of destinations (56). We can also see LGA flights has a lower average distance than those of EWR & JFK.

```
flights %>%
  group_by(origin) %>%
  summarise(destinations = n_distinct(dest),
            avg_distance = mean(distance, na.rm = TRUE))
```

```
## # A tibble: 3 x 3
##   origin destinations avg_distance
##   <chr>         <int>        <dbl>
## 1 EWR              86        1057.
## 2 JFK              70        1266.
## 3 LGA              68         780.
```

Here we summarise the whole dataset. We can see we have 337,776 observations, 105 distinct destinations and a 12.6 min avg delay.

```
flights %>%
  summarise(num_of_flights = n(),
            destinations = n_distinct(dest),
            avg_delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 3
##   num_of_flights destinations avg_delay
```

```
##            <int>        <int>     <dbl>
## 1         336776          105      12.6
```

`dpylr` is a great way to answer initial questions about a dataset. For example, say we want to know what the farthest flight to leave NYC is.

To answer this, we can group by origin and destination, summarise the max distance for each pair, and then order by the maximum distance value we created. It is now easy to see that the max distance was from EWR or JFK to HNL.

```
flights %>%
  group_by(origin, dest) %>%
  summarise(max_distance = max(distance)) %>%
  arrange(desc(max_distance))
```

```
## `summarise()` has grouped output by 'origin'. You can override using the `.groups` argument.
```

```
## # A tibble: 224 x 3
## # Groups:   origin [3]
##    origin dest  max_distance
##    <chr>  <chr>        <dbl>
##  1 JFK    HNL           4983
##  2 EWR    HNL           4963
##  3 EWR    ANC           3370
##  4 JFK    SFO           2586
##  5 JFK    OAK           2576
##  6 JFK    SJC           2569
##  7 EWR    SFO           2565
##  8 JFK    SMF           2521
##  9 JFK    LAX           2475
## 10 JFK    BUR           2465
## # ... with 214 more rows
```

More details can be found in Chapter 5 of R4DS.

Visualizing Data

Building blocks of `ggplot`

We now move on to `ggplot`. The basic idea of `ggplot` is to independently specify building blocks and combine them to create just about any kind of graphical display you want. Building blocks of a graph include:

- data
- aesthetic mapping
- geometric object
- faceting

Aesthetic Mappings

In `ggplot` land aesthetic means "something you can see". Examples include:

- position (i.e., on the x and y axes)
- color ("outside" color)
- fill ("inside" color)
- shape (of points)
- size

We now use a different dataset, gapminder. Let's do a quick summary.

```
library(gapminder)
summary(gapminder)
```

```
##         country        continent        year         lifeExp
##  Afghanistan:  12   Africa  :624   Min.   :1952   Min.   :23.60
##  Albania    :  12   Americas:300   1st Qu.:1966   1st Qu.:48.20
##  Algeria    :  12   Asia    :396   Median :1980   Median :60.71
##  Angola     :  12   Europe  :360   Mean   :1980   Mean   :59.47
##  Argentina  :  12   Oceania : 24   3rd Qu.:1993   3rd Qu.:70.85
##  Australia  :  12                  Max.   :2007   Max.   :82.60
##  (Other)    :1632
##       pop              gdpPercap
##  Min.   :6.001e+04   Min.   :   241.2
##  1st Qu.:2.794e+06   1st Qu.:  1202.1
##  Median :7.024e+06   Median :  3531.8
##  Mean   :2.960e+07   Mean   :  7215.3
##  3rd Qu.:1.959e+07   3rd Qu.:  9325.5
##  Max.   :1.319e+09   Max.   :113523.1
##
```

Plots by Data Types

Data	Plots	Geom (ggplot command)
One Continuous	Histogram	geom_histogram
One Continuous + One Categorical	Boxplot	geom_boxplot
Two Continuous	Scatter Plot	geom_point
Three Continuous	Scatter Plot + Size	geom_point w/ size aesthetic
Two Continuous + One Categorical	Scatter Plot + Color	geom_point w/ color aesthetic
Categorical with reasonable number of levels	Faceting!!	facet_wrap()

Note: Time is always the x-axis.

There are many more geom types, but we will focus on the ones listed in the table above.

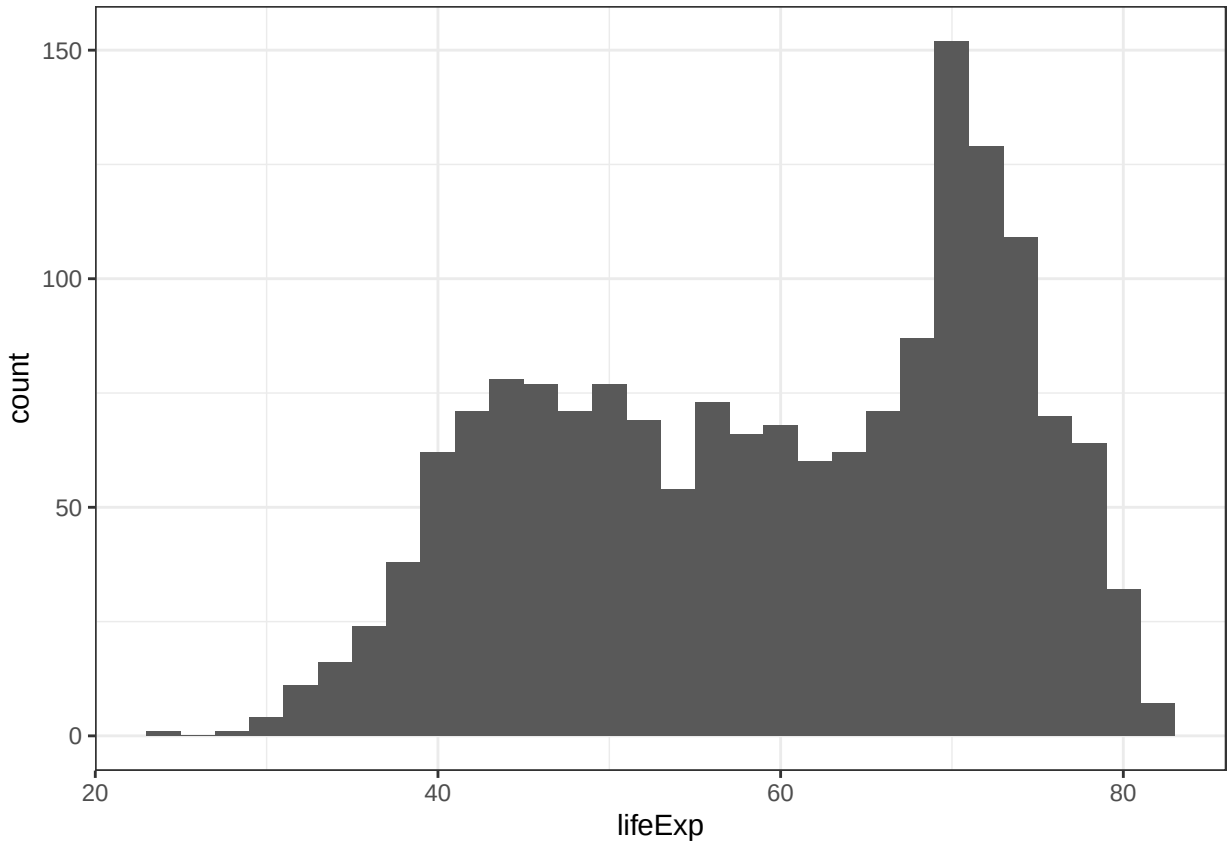
Here is an extremely useful cheatsheet that shows all of ggplots functions and how to use them.

One Continous / Geom_Histogram

The following shows the histogram of life Expectancy in 2007. Life expectancy is a continuous variable, so we use `geom_histogram()`.

Note how the `%>%` or "piping" also works with ggplot. If you are not piping in a dataframe, the first input to ggplot should be your dataframe. For example, the command would become `ggplot(gapminder, aes(x = lifeExP)) + geom_histogram(binwidth = 2)`

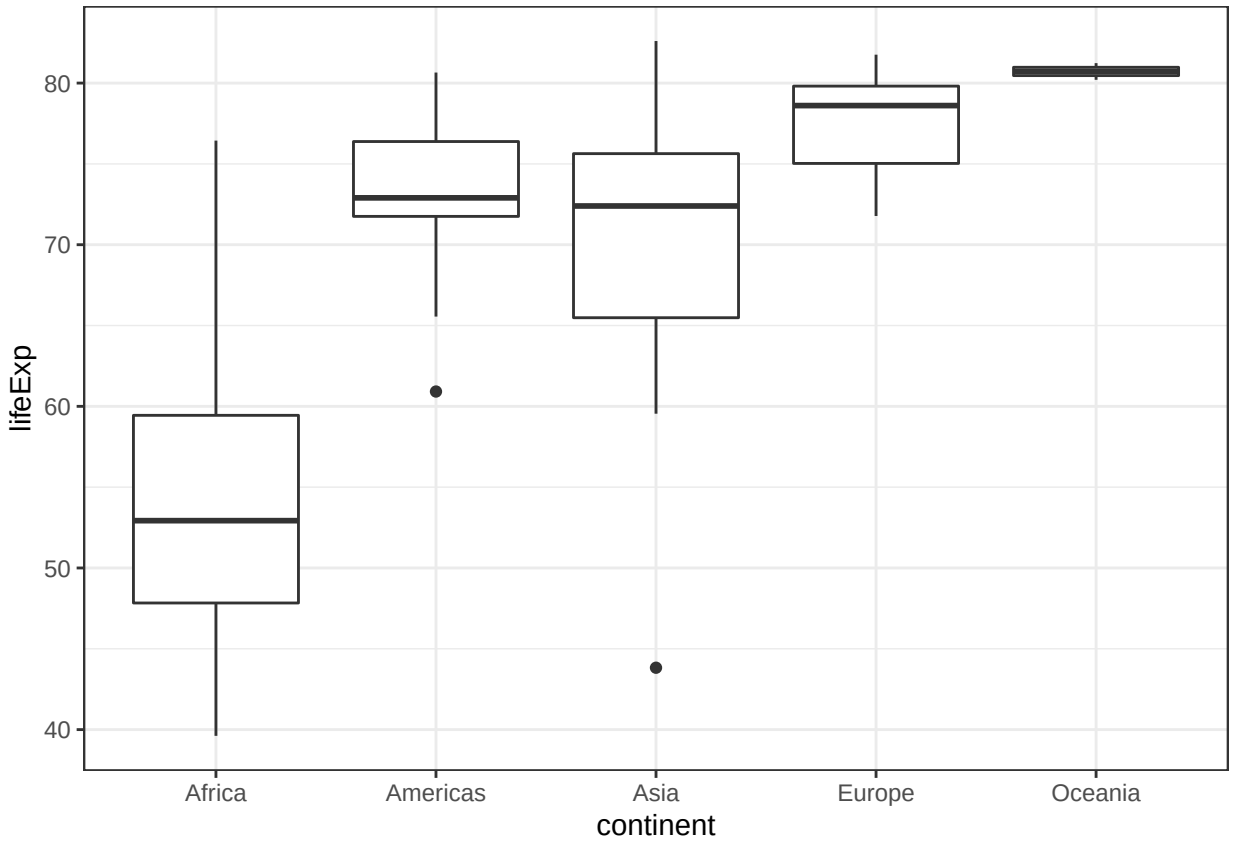
```
#hist(gapminder$lifeExp)
gapminder %>%
  ggplot(aes(x = lifeExp)) + geom_histogram(binwidth = 2) +
  theme_bw()
```



One Continuous + One Categorical / Geom_boxplot

Now, we want to show `lifeExp` broken down by continent. `Continent` is a categorical variable, also called factors in R. For this, we use the geom_boxplot() command.

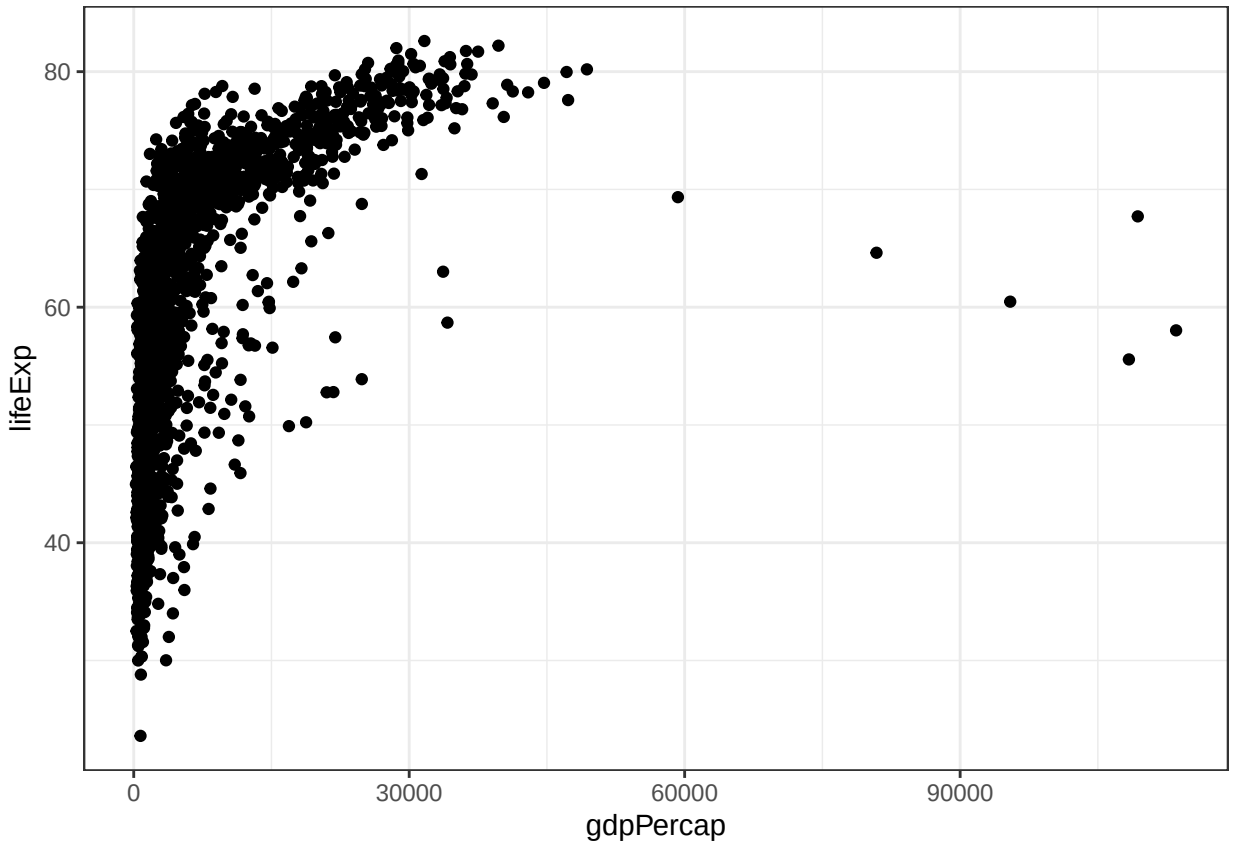
```
gapminder %>%
  filter(year == 2007) %>%
  ggplot(aes(x = continent, y = lifeExp)) + geom_boxplot() +
  theme_bw()
```



Two Continous / Geom_Point

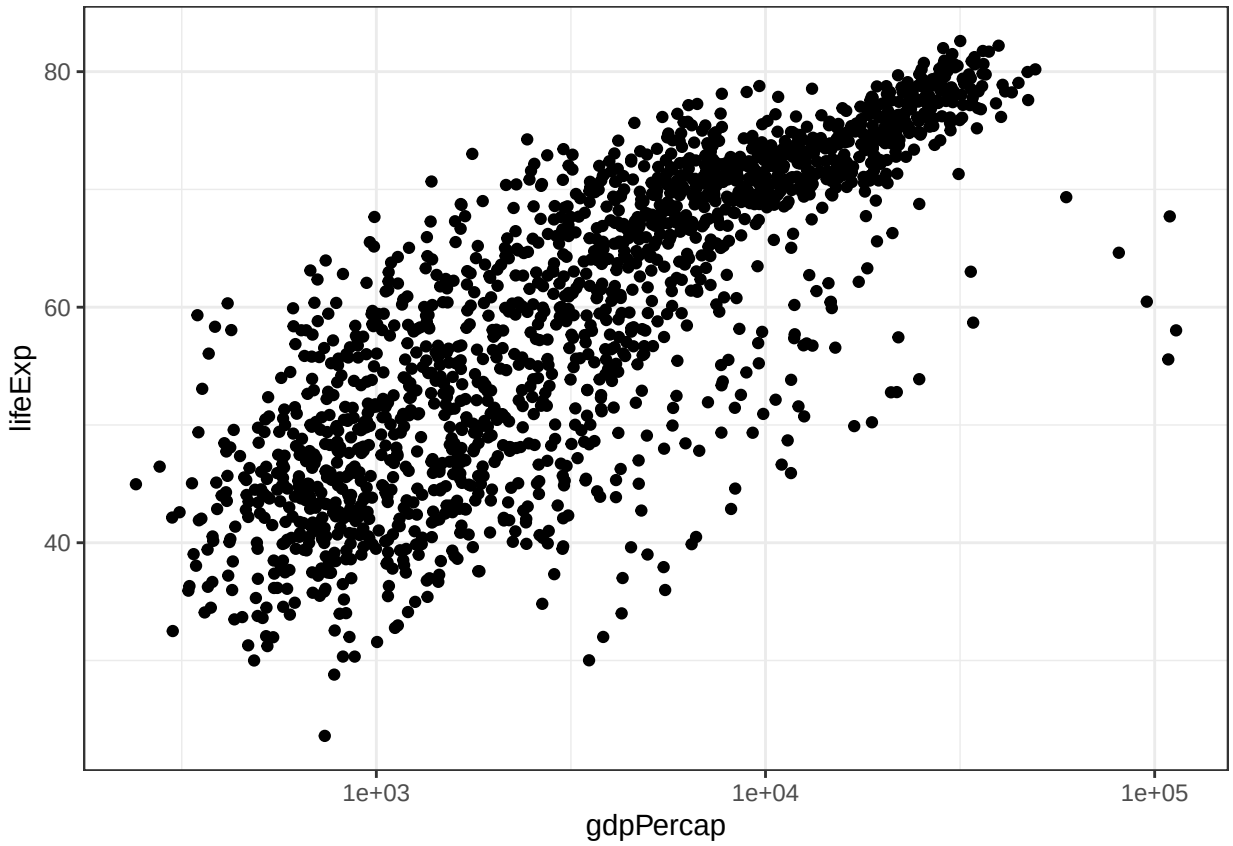
Using `geom_point()` we create a scatter plot of our two continuous variables, `gdpPercap` and `LifeExp`.

```
#plot(gapminder$gdpPercap, gapminder$lifeExp, pch=16)
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  theme_bw()
```



Some relationships will look better on different scales, and ggplot allows you to change scales very quickly. Here we log the x-axis, with `scale_x_log10()`, which makes the relationship between these two varibles much clearer.

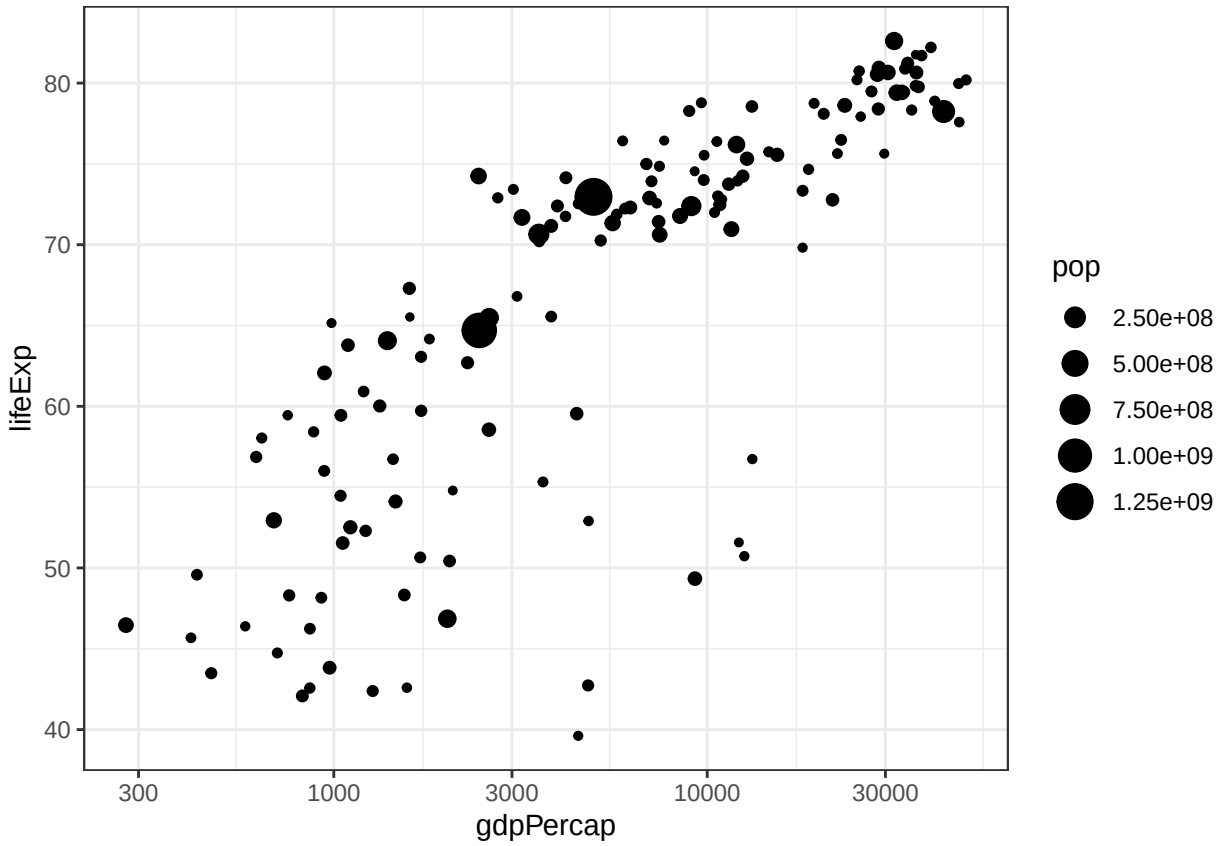
```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_log10() +
  theme_bw()
```



Three Continuous / Geom_point With Size Aesthetic

If we want to show three continuous variables at the same time, we can use the size aesthetic in ggplot. This will alter the size of the point by the value in the `pop` column of the gapminder data frame.

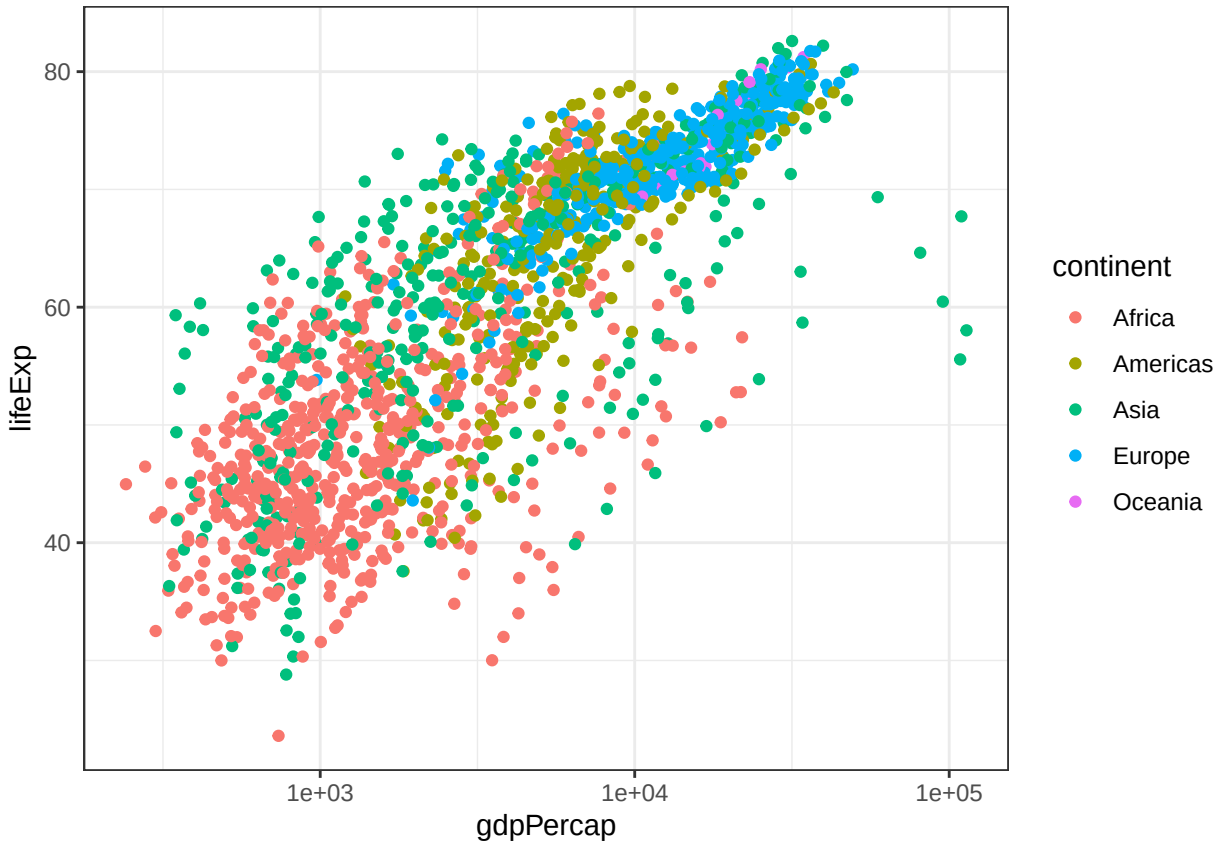
```
gapminder %>%
  filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop)) +
  geom_point() +
  scale_x_log10() +
  theme_bw()
```



Two Continuous + One Categorical / Geom_point With Color Aesthetic

To show more insight into this graph, we can show each point by which continent it is from. Adding the color Aesthetic allows us to show a categorical variable, `continent`, as each point is colored by what continent it is from.

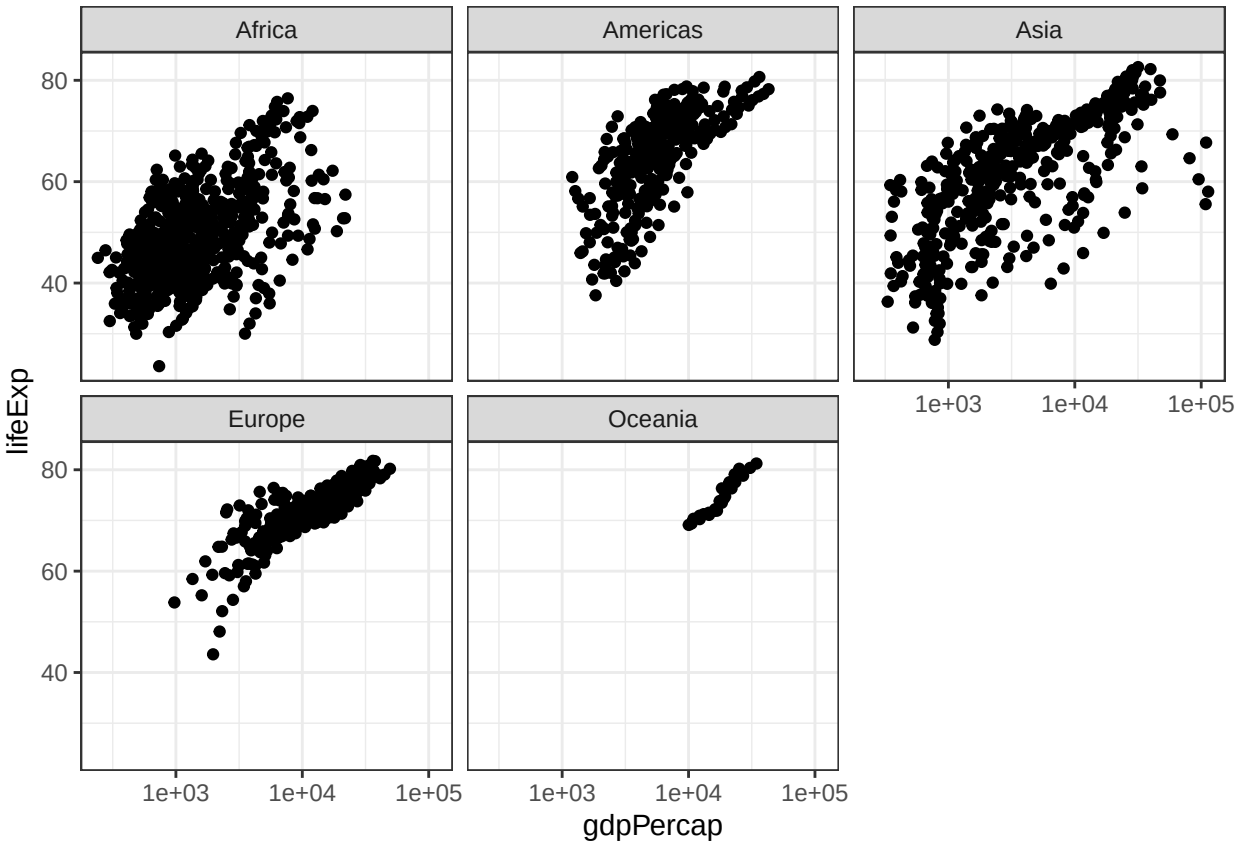
```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_bw()
```



Faceting

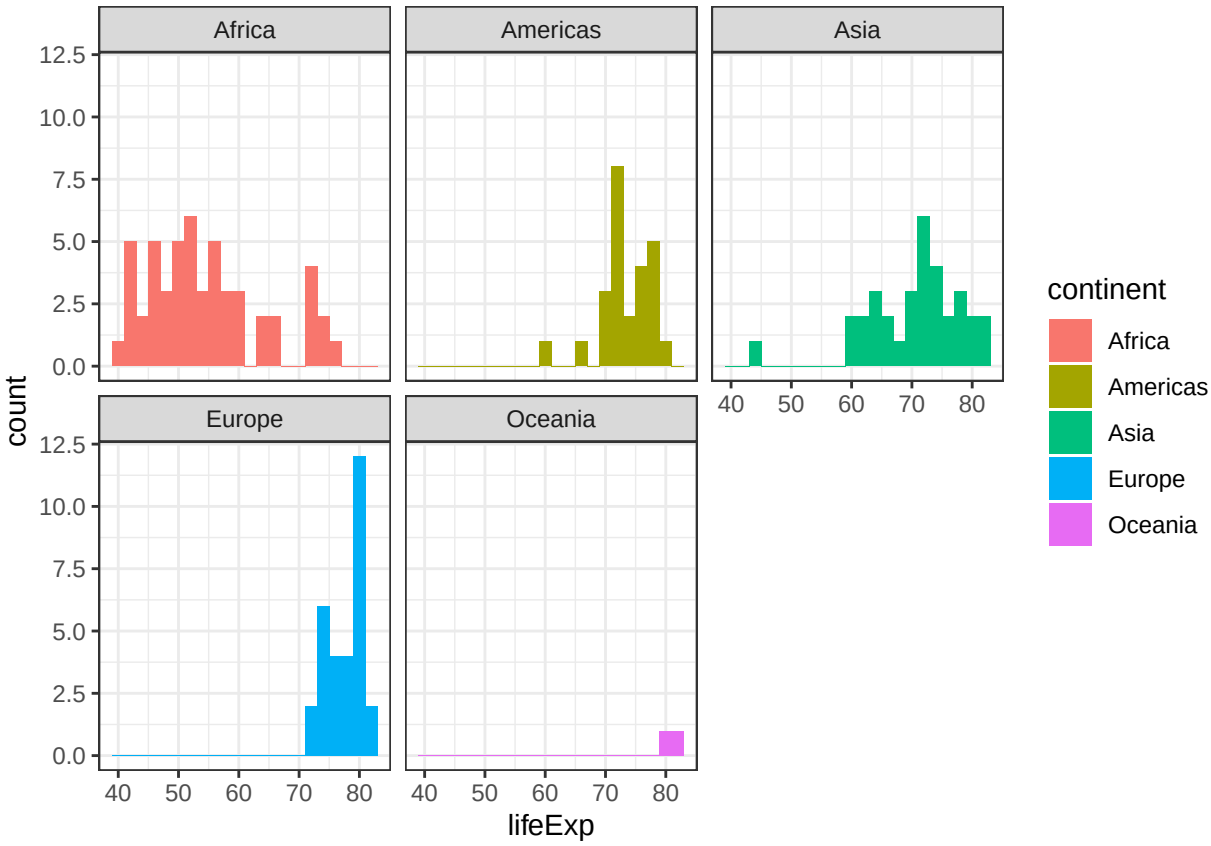
Instead of changing the color of points on the graph by continent, you can also create a different graph for each continent by 'faceting'. Depending on the number of factors and your dataset, faceting may look better than just changing colors. To do this we add the `facet_wrap(~ continent)` command.

```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_log10() +
  facet_wrap(~continent) +
  theme_bw()
```



You can facet with any geom type. Here is an example with `geom_histogram()`. It is also possible to color and facet on the same variable, as shown below.

```
gapminder %>%
  filter(year == 2007) %>%
  ggplot(aes(x = lifeExp, fill = continent)) +
  geom_histogram(binwidth = 2) +
  facet_wrap(~ continent) +
  theme_bw()
```

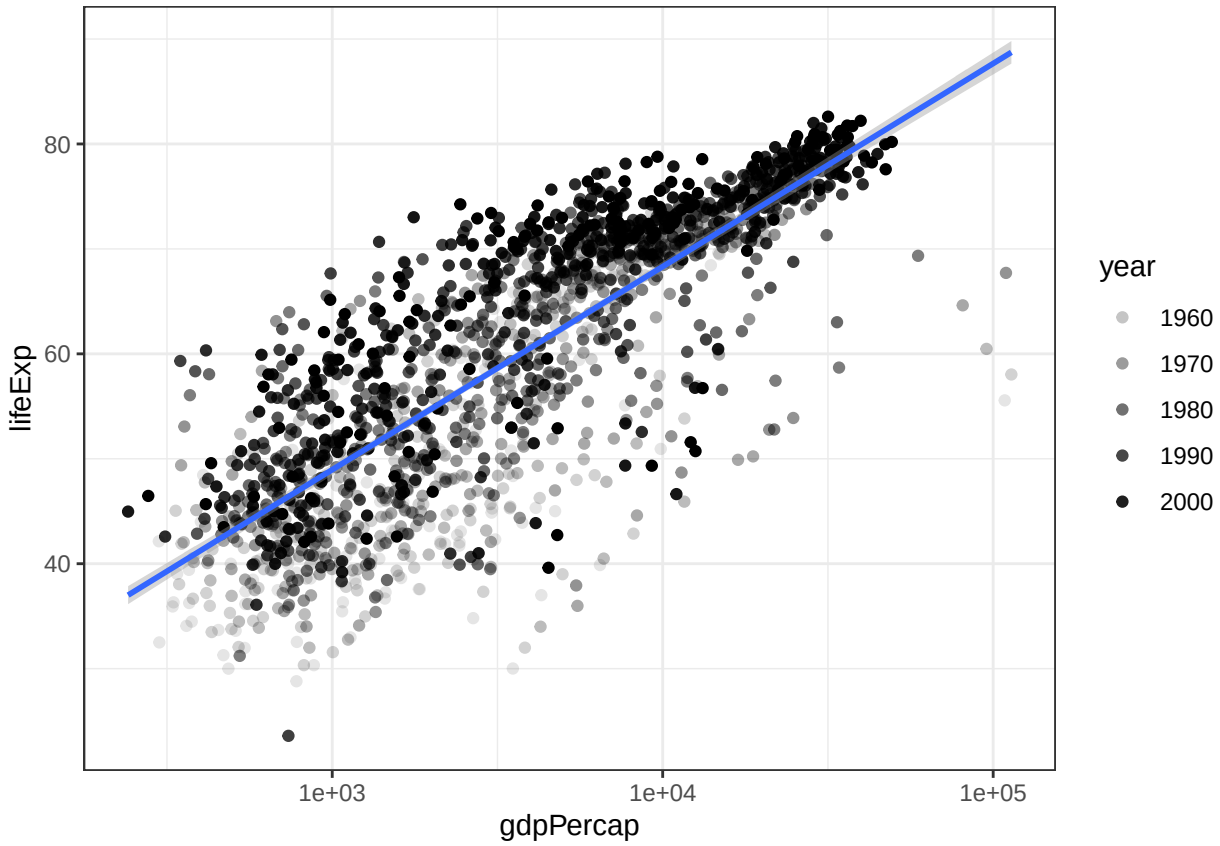


Adding a linear model line quickly / Geom_smooth

`ggplot` can also quickly add a linear model to a graph. There are also other models geom_smooth can do ("lm", "glm", "gam", "loess", "rlm"). If you leaving it blank it will automatically choose one for you, but that is not recommended.

To add the linear model line, we add `geom_smooth(method = 'lm', se = TRUE)` to the command. se = TRUE tells it to plot the standard error ranges on the graph.

```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(alpha = year)) +
  geom_smooth(method = 'lm', se = TRUE) +
  scale_x_log10() +
  theme_bw()
```


lifeExp
80
60
40
1e+03
1e+04
1e+05
gdpPercap
year
1960
1970
1980
1990
2000