

# Date Science with R

Tidy data

Peng Zhang

School of Mathematical Sciences, Zhejiang University

2025/06/28

# Data wrangling: dplyr

Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in the mundane labor of collecting and preparing data, before it can be explored for useful information. - NYTimes (2014)

What are some common things you like to do with your data? Maybe remove rows or columns, do calculations and maybe add new columns? This is called **data wrangling**. It's not data management or data manipulation: you **keep the raw data raw** and do these things programatically in R with the tidyverse.

We are going to introduce you to data wrangling in R first with the tidyverse. The tidyverse is a suite of packages that match a philosophy of data science developed by Hadley Wickham and the RStudio team. I find it to be a more straight-forward way to learn R. We will also show you by comparison what code will look like in “Base R”.

# Objectives

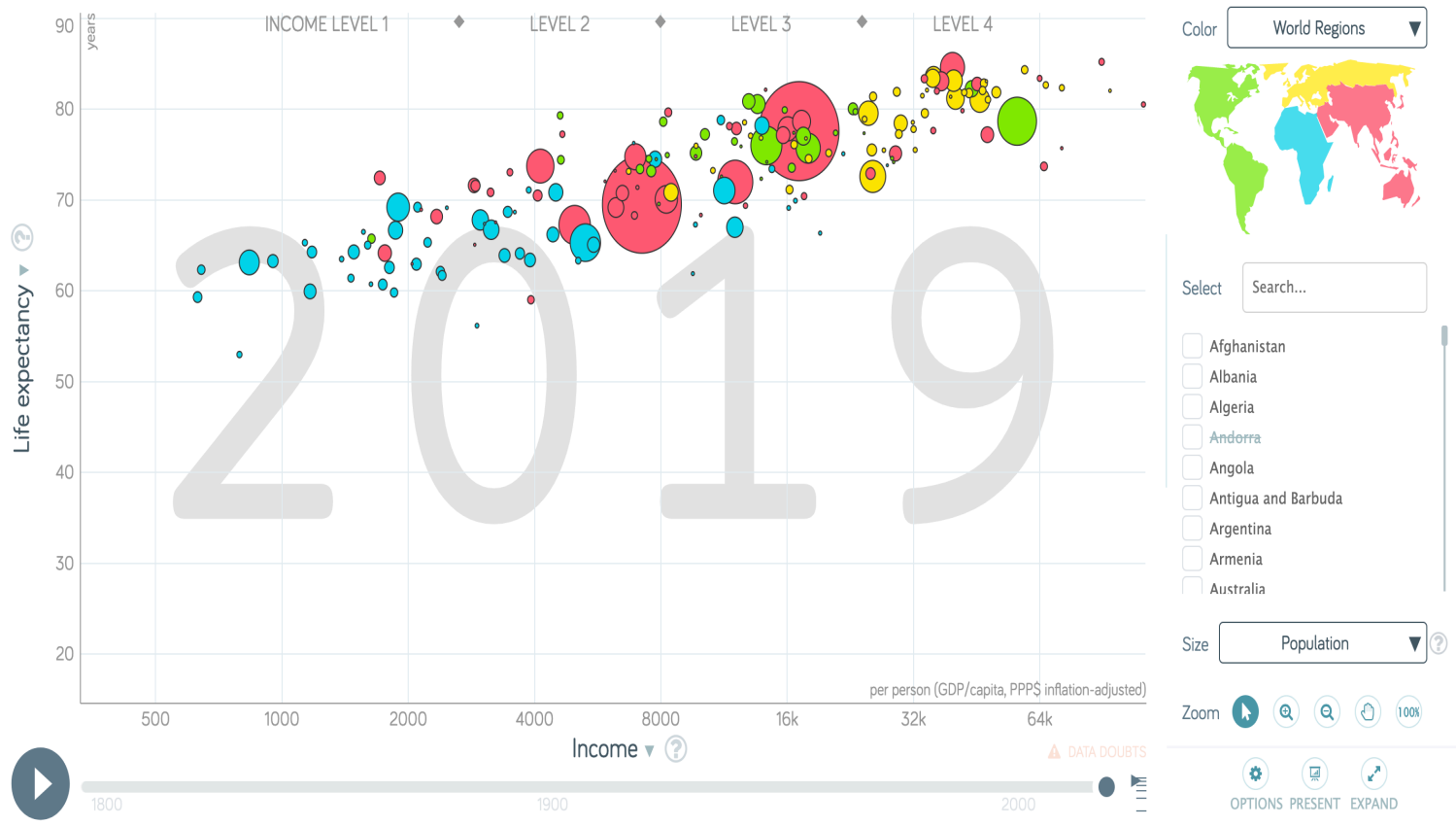
- discuss tidy data
- read data from online into R
- explore gapminder data with base-R functions
- wrangle gapminder data with dplyr tidyverse functions
- practice RStudio-GitHub workflow

## Data and packages

### Gapminder data

We'll be using Gapminder data, which represents the health and wealth of nations. It was pioneered by Hans Rosling, who is famous for describing the prosperity of nations over time through famines, wars and other historic events with this beautiful data visualization in his [2006 TED Talk: The best stats you've ever seen](#):

## Gapminder Motion Chart



We'll use the package `dplyr`, which is bundled within the `tidyverse` package. Please install the `tidyverse` ahead of time:

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

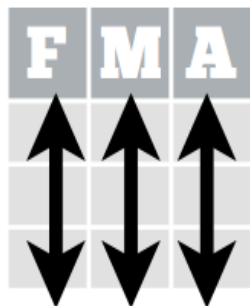
# Tidy Data

Let's start off discussing Tidy Data.

Hadley Wickham, RStudio's Chief Scientist, and his team have been building R packages for data wrangling and visualization based on the idea of tidy data.

Tidy data has a simple convention: put variables in the columns and observations in the rows.

In a tidy  
data set:



Each **variable** is saved  
in its own **column**

&

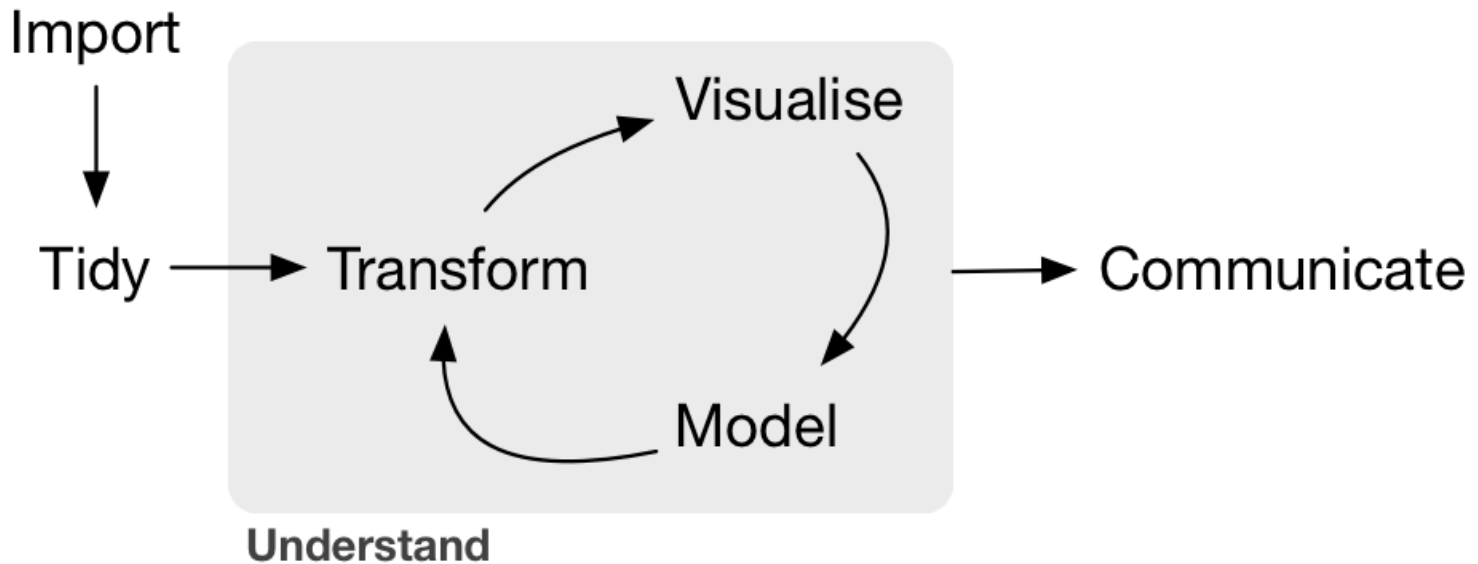


Each **observation** is  
saved in its own **row**

When data are tidy, you are set up to work with it for your analyses, plots, etc.

region	state	code	park_name	type	visitors	year
PW	CA	CHIS	Channel Islands National Park	National Park	1200	1963
PW	CA	CHIS	Channel Islands National Park	National Park	1500	1964
PW	CA	CHIS	Channel Islands National Park	National Park	1600	1965
PW	CA	CHIS	Channel Islands National Park	National Park	300	1966
PW	CA	CHIS	Channel Islands National Park	National Park	15700	1967
PW	CA	CHIS	Channel Islands National Park	National Park	31000	1968
PW	CA	CHIS	Channel Islands National Park	National Park	33100	1969
PW	CA	CHIS	Channel Islands National Park	National Park	32000	1970

Right now we are going to use `dplyr` to wrangle this tidy-ish data set (the transform part of the cycle), and then come back to tidying messy data using `tidyr` once we've had some fun wrangling. These are both part of the tidyverse package that we've already installed.



Conceptually, making data tidy first is really critical. Instead of building your analyses around whatever (likely weird) format your data are in, take deliberate steps to make your data tidy. When your data are tidy, you can use a growing assortment of powerful analytical and visualization tools instead of inventing home-grown ways to accommodate your data. This will save you time since you aren't reinventing the wheel, and will make your work more clear and understandable to your collaborators (most importantly, Future You).

## Setup

We'll do this in a new RMarkdown file.

Here's what to do:

Clear your workspace (Session > Restart R)

1. New File > R Markdown
2. Save as `gapminder-wrangle.Rmd`
3. Delete the irrelevant text and write a little note to yourself about how we'll be wrangling gapminder data using `dplyr`. You can edit the title too if you need to.



## load tidyverse (which has dplyr inside)

In your R Markdown file, let's make sure we've got our libraries loaded. Write the following:

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

This is becoming standard practice for how to load a library in a file, and if you get an error that the library doesn't exist, you can install the package easily by running the code within the comment (highlight `install.packages("tidyverse")` and run it).

# Explore the gapminder data.frame

## read data with readr::read\_csv()

In our R Markdown, let's read this csv file and name the variable "gapminder". We will use the read\_csv() function from the readr package (part of the tidyverse, so it's already installed!).

Let's inspect:

```
View(gapminder)
```

Let's use head and tail:

```
head(gapminder) # shows first 6
```

```
## # A tibble: 6 × 6
##   country      year      pop continent lifeExp gdpPercap
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>
## 1 Afghanistan 1952  8425333 Asia      28.8     779.
## 2 Afghanistan 1957  9240934 Asia      30.3     821.
## 3 Afghanistan 1962 10267083 Asia      32.0     853.
## 4 Afghanistan 1967 11537966 Asia      34.0     836.
## 5 Afghanistan 1972 13079460 Asia      36.1     740.
## 6 Afghanistan 1977 14880372 Asia      38.4     786.
```

```
tail(gapminder) # shows last 6
```

```
## # A tibble: 6 × 6
##   country    year      pop continent lifeExp gdpPercap
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>
## 1 Zimbabwe  1982  7636524 Africa     60.4     789.
## 2 Zimbabwe  1987  9216418 Africa     62.4     706.
## 3 Zimbabwe  1992 10704340 Africa     60.4     693.
## 4 Zimbabwe  1997 11404948 Africa     46.8     792.
## 5 Zimbabwe  2002 11926563 Africa     40.0     672.
## 6 Zimbabwe  2007 12311143 Africa     43.5     470.
```

```
head(gapminder, 10) # shows first X that you indicate
```

```
## # A tibble: 10 × 6
##   country    year      pop continent lifeExp gdpPercap
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>
## 1 Afghanistan 1952  8425333 Asia      28.8     779.
## 2 Afghanistan 1957  9240934 Asia      30.3     821.
## 3 Afghanistan 1962 10267083 Asia      32.0     853.
## 4 Afghanistan 1967 11537966 Asia      34.0     836.
## 5 Afghanistan 1972 13079460 Asia      36.1     740.
## 6 Afghanistan 1977 14880372 Asia      38.4     786.
## 7 Afghanistan 1982 12881816 Asia      39.9     978.
## 8 Afghanistan 1987 13867957 Asia      40.8     852.
## 9 Afghanistan 1992 16317921 Asia      41.7     649.
## 10 Afghanistan 1997 22227415 Asia      41.8     635.
```

`str()` will provide a sensible description of almost anything: when in doubt, inspect using `str()` on some of the recently created objects to get some ideas about what to do next.

```
str(gapminder) # ?str - displays the structure of an object
```

```
## spc_tbl_ [1,704 × 6] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ country   : chr [1:1704] "Afghanistan" "Afghanistan" "Afghanistan" "Afgh
## $ year      : num [1:1704] 1952 1957 1962 1967 1972 ...
## $ pop       : num [1:1704] 8425333 9240934 10267083 11537966 13079460 ...
## $ continent: chr [1:1704] "Asia" "Asia" "Asia" "Asia" ...
## $ lifeExp   : num [1:1704] 28.8 30.3 32 34 36.1 ...
## $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
## - attr(*, "spec")=
## .. cols(
## ..   country = col_character(),
## ..   year = col_double(),
## ..   pop = col_double(),
## ..   continent = col_character(),
## ..   lifeExp = col_double(),
## ..   gdpPercap = col_double()
## .. )
## - attr(*, "problems")=<externalptr>
```

`gapminder` is a `data.frame`. It is also a `tibble`, a modern extended structure based on `data.frame`.

- Tibbles are `data.frames` but modify some older behaviours to make life a little easier
- Preferred data format in the tidyverse
- No need to worry about this!

# dplyr basics

There are five dplyr functions that you will use to do the vast majority of data manipulations:

- `filter()`: pick observations by their values
- `select()`: pick variables by their names
- `mutate()`: create new variables with functions of existing variables
- `summarise()`: collapse many values down to a single summary
- `arrange()`: reorder the rows

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using `$`.
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

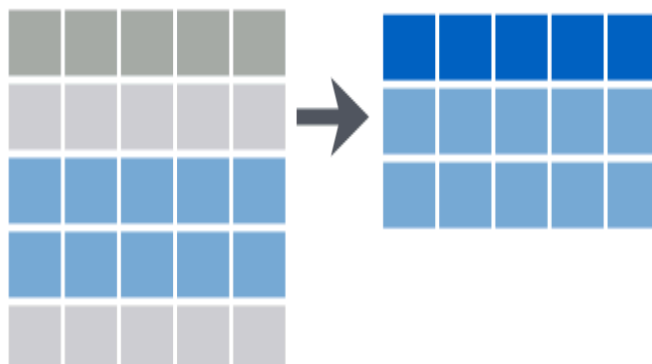
## **`filter()` subsets data row-wise (observations)**

You will want to isolate bits of your data; maybe you want to only look at a single country or a few years. R calls this subsetting.

`filter()` is a function in `dplyr` that takes logical expressions and returns the rows for which all are TRUE.

Visually, we are doing this (thanks RStudio for your cheatsheet):

# Subset Observations (Rows)



Remember your logical expressions? Conditional operators accepted in filter: ==, <, >, <=, >=, is.na(), !is.na(), %in%, !, |, &, xor()

```
filter(gapminder, lifeExp < 29)
```

```
## # A tibble: 2 × 6
##   country      year      pop continent lifeExp gdpPercap
##   <chr>      <dbl>   <dbl> <chr>      <dbl>     <dbl>
## 1 Afghanistan 1952 8425333 Asia       28.8       779.
## 2 Rwanda      1992 7290203 Africa     23.6       737.
```

You can say this out loud: “Filter the gapminder data for life expectancy less than 29”. Notice that when we do this, all the columns are returned, but only the rows that have the life expectancy less than 29. We’ve subsetting by row.



Let's try another: "Filter the gapminder data for the country Mexico".

```
filter(gapminder, country == "Mexico")
```

```
## # A tibble: 12 × 6
```

##	country	year	pop	continent	lifeExp	gdpPercap
##	<chr>	<dbl>	<dbl>	<chr>	<dbl>	<dbl>
##	1 Mexico	1952	30144317	Americas	50.8	3478.
##	2 Mexico	1957	35015548	Americas	55.2	4132.
##	3 Mexico	1962	41121485	Americas	58.3	4582.
##	4 Mexico	1967	47995559	Americas	60.1	5755.
##	5 Mexico	1972	55984294	Americas	62.4	6809.
##	6 Mexico	1977	63759976	Americas	65.0	7675.
##	7 Mexico	1982	71640904	Americas	67.4	9611.
##	8 Mexico	1987	80122492	Americas	69.5	8688.
##	9 Mexico	1992	88111030	Americas	71.5	9472.
##	10 Mexico	1997	95895146	Americas	73.7	9767.
##	11 Mexico	2002	102479927	Americas	74.9	10742.
##	12 Mexico	2007	108700891	Americas	76.2	11978.

How about if we want two country names? We can't use the == operator here, because it can only operate on one thing at a time. We will use the %in% operator:

```
filter(gapminder, country %in% c("Mexico", "Peru"))
```

```
## # A tibble: 24 × 6
##   country  year      pop continent lifeExp gdpPercap
##   <chr>    <dbl>    <dbl> <chr>         <dbl>    <dbl>
## 1 Mexico  1952 30144317 Americas     50.8    3478.
## 2 Mexico  1957 35015548 Americas     55.2    4132.
## 3 Mexico  1962 41121485 Americas     58.3    4582.
## 4 Mexico  1967 47995559 Americas     60.1    5755.
## 5 Mexico  1972 55984294 Americas     62.4    6809.
## 6 Mexico  1977 63759976 Americas     65.0    7675.
## 7 Mexico  1982 71640904 Americas     67.4    9611.
## 8 Mexico  1987 80122492 Americas     69.5    8688.
## 9 Mexico  1992 88111030 Americas     71.5    9472.
## 10 Mexico 1997 95895146 Americas     73.7    9767.
## # i 14 more rows
```

How about if we want Mexico in 2002? You can pass filter different criteria:

```
filter(gapminder, country == "Mexico", year == 2002)
```

```
## # A tibble: 1 × 6
##   country year      pop continent lifeExp gdpPercap
##   <chr>   <dbl>    <dbl> <chr>      <dbl>    <dbl>
## 1 Mexico  2002 102479927 Americas    74.9    10742.
```

## Exercise

What was the average life expectancy in Brazil between 1987 and 2007? Hint: do this in 2 steps by assigning a variable and then using the `mean()` function.

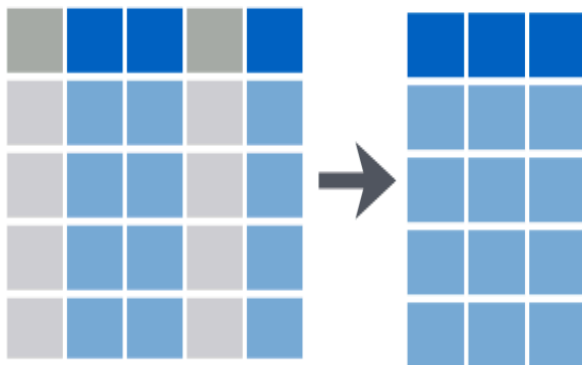
Then, sync to Github.com (pull, stage, commit, push).

`select()` subsets data column-wise (variables)

We use `select()` to subset the data on variables or columns.

Visually, we are doing this (thanks RStudio for your cheatsheet):

## Subset Variables (Columns)



- We can select multiple columns with a comma, after we specify the data frame (gapminder).

```
gap1 <- dplyr::select(gapminder, year, country, lifeExp)
head(gap1, 3)
```

```
## # A tibble: 3 × 3
##   year country    lifeExp
##   <dbl> <chr>      <dbl>
## 1  1952 Afghanistan  28.8
## 2  1957 Afghanistan  30.3
## 3  1962 Afghanistan  32.0
```

- We can select a range of variables with a semicolon.

```
gap2 <- dplyr::select(gapminder, year:lifeExp)
head(gap2, 3)
```

```
## # A tibble: 3 × 4
##   year      pop continent lifeExp
##   <dbl>   <dbl> <chr>      <dbl>
## 1  1952  8425333 Asia      28.8
## 2  1957  9240934 Asia      30.3
## 3  1962 10267083 Asia      32.0
```

- We can select columns with indices.

```
gap3 <- dplyr::select(gapminder, 1, 2, 4)
head(gap3, 3)
```

```
## # A tibble: 3 × 3
##   country      year continent
##   <chr>      <dbl> <chr>
## 1 Afghanistan 1952 Asia
## 2 Afghanistan 1957 Asia
## 3 Afghanistan 1962 Asia
```

- We can also use `-` to deselect columns

```
gap4 <- dplyr::select(gapminder, -continent, -lifeExp) # you can use
head(gap4, 3)
```

```
## # A tibble: 3 × 4
##   country      year      pop gdpPercap
##   <chr>      <dbl>   <dbl>   <dbl>
## 1 Afghanistan 1952 8425333    779.
## 2 Afghanistan 1957 9240934    821.
## 3 Afghanistan 1962 10267083    853.
```

## Use `select()` and `filter()` together

Let's filter for Cambodia and remove the continent and lifeExp columns. We'll save this as a variable. Actually, as two temporary variables, which means that for the second one we need to operate on `gap_cambodia`, not `gapminder`.

```
gap_cambodia <- filter(gapminder, country == "Cambodia")  
gap_cambodia2 <- dplyr::select(gap_cambodia, -continent, -lifeExp)
```

We also could have called them both `gap_cambodia` and overwritten the first assignment. Either way, naming them and keeping track of them gets super cumbersome, which means more time to understand what's going on and opportunities for confusion or error.

# Meet the new pipe %>% ( | > ) operator

Before we go any further, we should explore the new pipe operator that `dplyr` imports from the `magrittr` package by Stefan Bache. This is going to **change your life**. You no longer need to enact multi-operation commands by nesting them inside each other. And we won't need to make temporary variables like we did in the Cambodia example above. This new syntax leads to code that is much easier to write and to read: it actually tells the story of your analysis.

Here's what it looks like: %>%. The RStudio keyboard shortcut: Ctrl + Shift + M (Windows), Cmd + Shift + M (Mac). Or you can use the base R pipe |> built into R 4.1 and later version. Go to the Global Options window, select "Code" and you'll see an option for "use native pipe operator, |>".

Let's demo then I'll explain:

```
gapminder |> head(3)
```

```
## # A tibble: 3 × 6
##   country      year      pop continent lifeExp gdpPercap
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>
## 1 Afghanistan 1952  8425333 Asia      28.8      779.
## 2 Afghanistan 1957  9240934 Asia      30.3      821.
## 3 Afghanistan 1962 10267083 Asia      32.0      853.
```



This is equivalent to `head(gapminder, 3)`. This pipe operator takes the thing on the left-hand-side and **pipes** it into the function call on the right-hand-side. It literally drops it in as the first argument.

**You should think “and then” whenever you see the pipe operator, `|>`.**

One of the most awesome things about this is that you START with the data before you say what you’re doing to DO to it. So above: “take the gapminder data, and then give me the first three entries”.

This means that instead of this:

```
## instead of this...
gap_cambodia <- filter(gapminder, country == "Cambodia")
gap_cambodia2 <- dplyr::select(gap_cambodia, -continent, -lifeExp)

## ...we can do this
gap_cambodia <- gapminder |> filter(country == "Cambodia")
gap_cambodia2 <- gap_cambodia |> dplyr::select(-continent, -lifeExp)
```

So you can see that we’ll start with `gapminder` in the first example line, and then `gap_cambodia` in the second. This makes it a bit easier to see what data we are starting with and what we are doing to it.

But, we still have those temporary variables so we’re not truly that better off. But get ready to be majorly impressed:

## Revel in the convenience

We can use the pipe to chain those two operations together:

```
gap_cambodia <- gapminder |>  
  filter(country == "Cambodia") |>  
  dplyr::select(-continent, -lifeExp)
```

What's happening here? In the second line, we were able to delete `gap_cambodia2 <- gap_cambodia`, and put the pipe operator above. This is possible since we wanted to operate on the `gap_cambodia` data anyways. And we weren't truly excited about having a second variable named `gap_cambodia2` anyways, so we can get rid of it. This is huge, because most of your data wrangling will have many more than 2 steps, and we don't want a `gap_cambodia14`!

By using multiple lines I can actually read this like a story and there aren't temporary variables that get super confusing. In my head:

“start with the gapminder data, and then filter for Cambodia, and then drop the variables continent and lifeExp.”

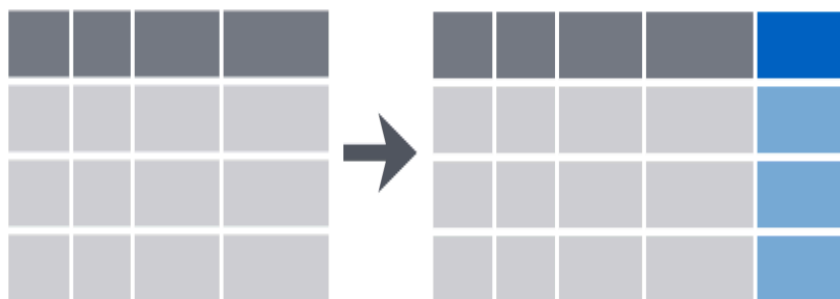
Being able to read a story out of code like this is really game-changing. We'll continue using this syntax as we learn the other dplyr verbs.

## `mutate()` adds new variables

Let's say we needed to add an index column so we know which order these data came in. Let's not make a new variable, let's add a column to our `gapminder` data frame. How do we do that? With the `mutate()` function.

Visually, we are doing this (thanks RStudio for your cheatsheet):

# Make New Variables



Imagine we want to know each country's annual GDP. We can multiply pop by gdpPercap to create a new column named gdp.

```
gapminder |>
  mutate(gdp = pop * gdpPercap)
```

```
## # A tibble: 1,704 × 7
##   country      year      pop continent lifeExp gdpPercap      gdp
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>    <dbl>
## 1 Afghanistan 1952  8425333 Asia      28.8      779.  6567086330.
## 2 Afghanistan 1957  9240934 Asia      30.3      821.  7585448670.
## 3 Afghanistan 1962 10267083 Asia      32.0      853.  8758855797.
## 4 Afghanistan 1967 11537966 Asia      34.0      836.  9648014150.
## 5 Afghanistan 1972 13079460 Asia      36.1      740.  9678553274.
## 6 Afghanistan 1977 14880372 Asia      38.4      786. 11697659231.
## 7 Afghanistan 1982 12881816 Asia      39.9      978. 12598563401.
## 8 Afghanistan 1987 13867957 Asia      40.8      852. 11820990309.
## 9 Afghanistan 1992 16317921 Asia      41.7      649. 10595901589.
## 10 Afghanistan 1997 22227415 Asia      41.8      635. 14121995875.
## # i 1,694 more rows
```

## Your turn

Calculate the population in thousands for all Asian countries in the year 2007 and add it as a new column.

Then, sync to Github.com (pull, stage, commit, push).

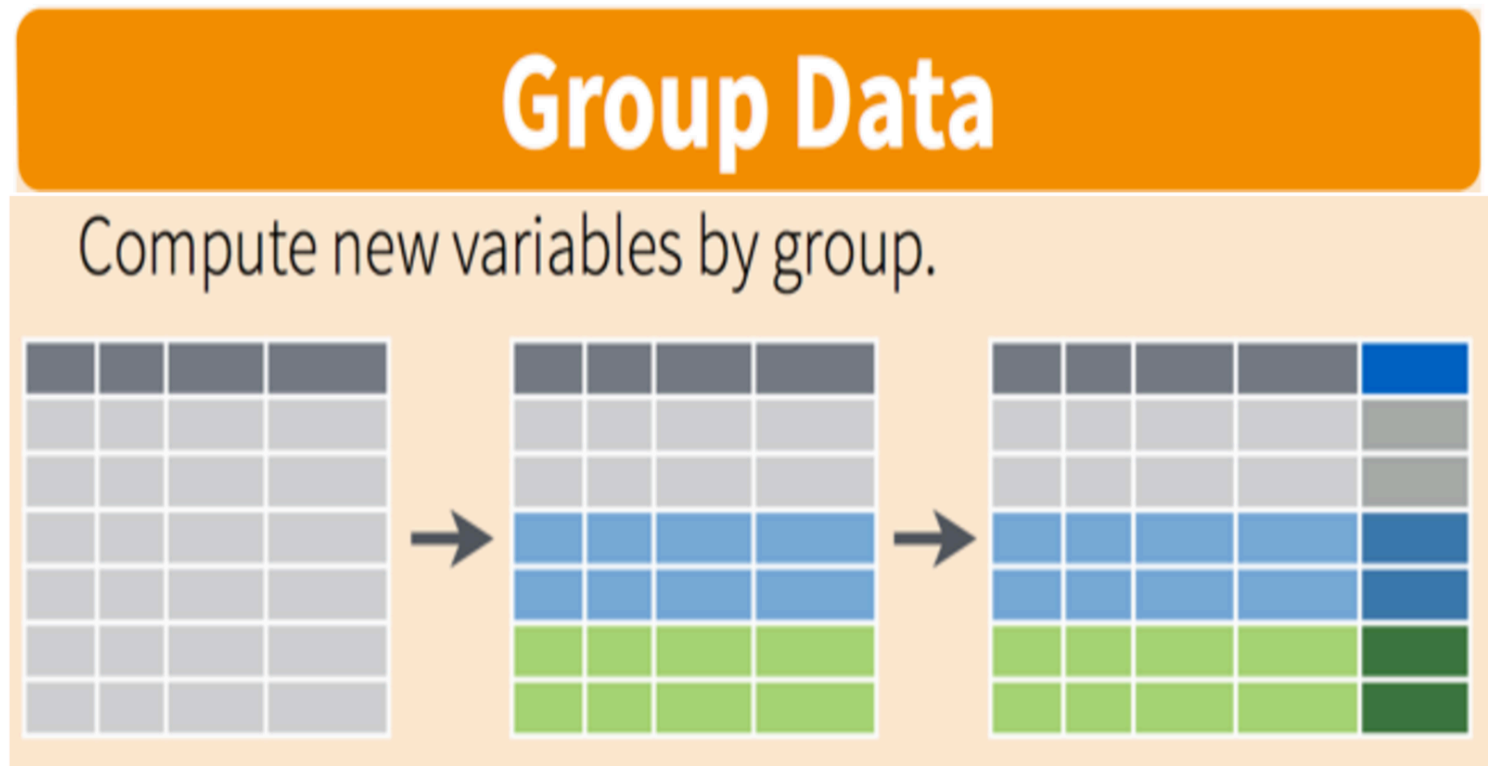
# Functions in `mutate()`

- arithmetic operators: `+`, `-`, `*`, `/`, `^`
- modular arithmetic: `%/%`, `%%`
- logs: `log()`, `log2()`, `log10()`
- offsets: `lead()`, `lag()`
- cumulative and rolling aggregates: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`, `cummean()`
- logical comparisons: `<`, `<=`, `>`, `>=`, `!=`
- ranking: `min_rank()`, `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`, `ntile()`
- Any R or custom function that returns a **vector** with the same length as the number of rows

## group\_by ( ) operates on groups

What if we wanted to know the total population on each continent in 2002?  
Answering this question requires a **grouping variable**.

Visually, we are doing this (thanks RStudio for your cheatsheet):



By using `group_by()` we can set our grouping variable to `continent` and create a new column called `cont_pop` that will add up all country populations by their associated continents.

```
gapminder |>
  filter(year == 2002) |>
  group_by(continent) |>
  mutate(cont_pop = sum(pop))
```

```
## # A tibble: 142 × 7
## # Groups:   continent [5]
##   country      year      pop continent lifeExp gdpPercap  cont_pop
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>    <dbl>
## 1 Afghanistan 2002  25268405 Asia       42.1     727.  3601802203
## 2 Albania     2002   3508512 Europe     75.7    4604.   578223869
## 3 Algeria     2002  31287142 Africa     71.0    5288.   833723916
## 4 Angola      2002  10866106 Africa     41.0    2773.   833723916
## 5 Argentina   2002  38331121 Americas  74.3    8798.   849772762
## 6 Australia   2002  19546792 Oceania    80.4   30688.   23454829
## 7 Austria     2002   8148312 Europe     79.0   32418.   578223869
## 8 Bahrain     2002    656397 Asia      74.8   23404.  3601802203
## 9 Bangladesh  2002 135656790 Asia      62.0    1136.  3601802203
## 10 Belgium    2002  10311970 Europe     78.3   30486.   578223869
## # i 132 more rows
```



What if we don't care about the other columns and we only want each continent and their population in 2002? Here's the next function:

## `summarize()` with `group_by()`

We want to operate on a group, but actually collapse or distill the output from that group. The `summarize()` function will do that for us.

Visually, we are doing this (thanks RStudio for your cheatsheet):

# Summarise Data



Here we go:

```
gapminder |>
  group_by(continent) |>
  summarize(cont_pop = sum(pop)) |>
  ungroup()
```

```
## # A tibble: 5 × 2
##   continent    cont_pop
##   <chr>         <dbl>
## 1 Africa      6187585961
## 2 Americas    7351438499
## 3 Asia        30507333902
## 4 Europe      6181115304
## 5 Oceania     212992136
```

`summarize()` will actually only keep the columns that are `grouped_by` or summarized. So if we wanted to keep other columns, we'd have to do have a few more steps. `ungroup()` removes the grouping and it's good to get in the habit of using it after a `group_by()`.

We can use more than one grouping variable. Let's get total populations by continent and year.

```
gapminder |>
  group_by(continent, year) |>
  summarize(cont_pop = sum(pop))
```

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

```
## # A tibble: 60 × 3
## # Groups:   continent [5]
##   continent year cont_pop
##   <chr>      <dbl>    <dbl>
## 1 Africa    1952  237640501
## 2 Africa    1957  264837738
## 3 Africa    1962  296516865
## 4 Africa    1967  335289489
## 5 Africa    1972  379879541
## 6 Africa    1977  433061021
## 7 Africa    1982  499348587
## 8 Africa    1987  574834110
## 9 Africa    1992  659081517
## 10 Africa   1997  743832984
## # i 50 more rows
```

# Functions in summarise()

- location: `mean(x)`, `median(x)`
- spread: `sd(x)`, `IQR(x)`, `mad(x)`
- rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`
- position: `first(x)`, `nth(x, 2)`, `last(x)`
- count: `n(x)`, `sum(!is.na(x))`, `n_distinct(x)`
- any base R or custom function that returns **one summary value**

## arrange() orders columns

This is ordered alphabetically, which is cool. But let's say we wanted to order it in ascending order for year. The dplyr function is `arrange()`.

```
gapminder |>
  group_by(continent, year) |>
  summarize(cont_pop = sum(pop)) |>
  arrange(year)
```

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

```
## # A tibble: 60 × 3
## # Groups:   continent [5]
##   continent year   cont_pop
##   <chr>      <dbl>     <dbl>
## 1 Africa    1952  237640501
## 2 Americas 1952  345152446
## 3 Asia     1952 1395357352.
## 4 Europe   1952  418120846
## 5 Oceania  1952   10686006
## 6 Africa   1957  264837738
## 7 Americas 1957  386953916
## 8 Asia     1957 1562780599
## 9 Europe   1957  437890351
```

## Your turn

What is the maximum GDP per continent across all years?

## Your turn

1. arrange your data frame in descending order (opposite of what we've done). Expect that this is possible: `?arrange`
2. save your data frame as a variable
3. find the maximum life expectancy for countries in Asia. What is the earliest year you encounter? The latest? Hint: you can use `base::max` and `dplyr::arrange()`
4. Knit your RMarkdown file, and sync it to GitHub (pull, stage, commit, push)

# All together now

We have done a pretty incredible amount of work in a few lines. Our whole analysis is this. Imagine the possibilities from here. It's very readable: you see the data as the first thing, it's not nested. Then, you can read the verbs. This is the whole thing, with explicit package calls from `readr::` and `dplyr::`:

```
## load libraries
library(tidyverse) ## install.packages('tidyverse')

## read in data
gapminder <- readr::read_csv('data/gapminder.csv')

## summarize
gap_max_life_exp <- gapminder |>
  dplyr::select(-continent, -lifeExp) |> # or select(country, year, pop)
  dplyr::group_by(country) |>
  dplyr::mutate(gdp = pop * gdpPercap) |>
  dplyr::summarize(max_gdp = max(gdp)) |>
  dplyr::ungroup()
```

## Compare to base R

Instead of calculating the max for each country like we did with dplyr above, here we will calculate the max for one country, Mexico.

```
gapminder <- read.csv('data/gapminder.csv', stringsAsFactors = FALSE)
x1 <- gapminder[ , c('country', 'year', 'pop', 'gdpPercap')] # subset
mex <- x1[x1$country == "Mexico", ] # subset rows
mex$gdp <- mex$pop * mex$gdpPercap # add new columns
mex$max_gdp <- max(mex$gdp)
```

Note too that the chain operator `|>` that we used with the tidyverse lets us get away from the temporary variable `x1`.



# Joining datasets

Most of the time you will have data coming from different places or in different files, and you want to put them together so you can analyze them. Datasets you'll be joining can be called relational data, because it has some kind of relationship between them that you'll be acting upon. In the tidyverse, combining data that has a relationship is called “joining”.

From the RStudio cheatsheet (note: this is an earlier version of the cheatsheet but I like the graphics):

## Combine Data Sets

a		b			
x1	x2			x1	x3
A	1			A	T
B	2			B	F
C	3			D	T

+

=

Let's have a look at this and pretend that the x1 column is a study site and x2 is the variables we've recorded (like species count) and x3 is data from an instrument (like temperature data). Notice how you may not have exactly the same observations in the two datasets: in the x1 column, observations A and B appear in both datasets, but notice how the table on the left has observation C, and the table on the right has observation D.

If you wanted to combine these two tables, how would you do it? There are some decisions you'd have to make about what was important to you. The cheatsheet visualizes it for us:

## Combine Data Sets

a		b	
x1	x2	x1	x3
A	1	A	T
B	2	B	F
C	3	D	T

+

=

### Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

**dplyr::left\_join(a, b, by = "x1")**  
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

**dplyr::right\_join(a, b, by = "x1")**  
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

**dplyr::inner\_join(a, b, by = "x1")**  
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T


**dplyr::full\_join(a, b, by = "x1")**  
Join data. Retain all values, all rows.

We will only talk about this briefly here, but you can refer to this more as you have your own datasets that you want to join. This describes the figure above:

- `left_join` keeps everything from the left table and matches as much as it can from the right table. In R, the first thing that you type will be the left table (because it's on the left)
- `right_join` keeps everything from the right table and matches as much as it can from the left table
- `inner_join` only keeps the observations that are similar between the two tables
- `full_join` keeps all observations from both tables.

Let's play with these CO2 emissions data to illustrate:

```
## read in the data. (same URL as yesterday, with co2.csv instead of  
co2 <- read_csv("data/co2.csv")  
  
## explore  
co2 |> head()
```



```
## # A tibble: 6 × 2  
##   country      co2_2007  
##   <chr>         <dbl>  
## 1 Afghanistan    2938.  
## 2 Albania         4218.  
## 3 Algeria       105838.  
## 4 American Samoa    18.4  
## 5 Angola        17405.  
## 6 Anguilla        12.4
```

```
co2 |> dim() # 12
```

```
## [1] 12  2
```

```
## create new variable that is only 2007 data
gap_2007 <- gapminder |>
  filter(year == 2007)
gap_2007 |> dim() # 142
```

```
## [1] 142    6
```

```
## left_join gap_2007 to co2
lj <- left_join(gap_2007, co2, by = "country")

## explore
lj |> dim() #142
```

```
## [1] 142    7
```

```
lj |> head(3) # lots of NAs in the co2_2017 column
```

```
## # A tibble: 3 × 7
##   country      year      pop continent lifeExp gdpPercap co2_2007
##   <chr>      <dbl>    <dbl> <chr>      <dbl>    <dbl>    <dbl>
## 1 Afghanistan 2007 31889923 Asia      43.8      975.    2938.
## 2 Albania      2007  3600523 Europe    76.4     5937.   4218.
## 3 Algeria      2007 33333216 Africa    72.3     6223.  105838.
```

```
## right_join gap_2007 and co2
rj <- right_join(gap_2007, co2, by = "country")
```

```
## explore
rj |> dim() # 12
```

```
## [1] 12  7
```

```
rj |> head(3)
```

```
## # A tibble: 3 × 7
```

	country	year	pop	continent	lifeExp	gdpPercap	co2_2007
	<chr>	<dbl>	<dbl>	<chr>	<dbl>	<dbl>	<dbl>
## 1	Afghanistan	2007	31889923	Asia	43.8	975.	2938.
## 2	Albania	2007	3600523	Europe	76.4	5937.	4218.
## 3	Algeria	2007	33333216	Africa	72.3	6223.	105838.

# Key Points

Data manipulation functions in `dplyr` allow you to `filter()` by rows and `select()` by columns, create new columns with `mutate()`, and `group_by()` unique column values to apply `summarize()` for new columns that define aggregate values across groupings. The “then” operator `|>` allows you to chain successive operations without needing to define intermediary variables for creating the most parsimonious, easily read analysis.

## Error: unexpected SPECIAL in "|>"

If you get this error, it is probably because you have a line that starts with a pipe. The pipe should be at the end of the previous line, not the start of the current line.

### Yes:

```
gap_cambodia <- gapminder |> filter(country == "Cambodia") |>
  select(-continent, -lifeExp)
```

### No:

```
gap_cambodia <- gapminder |> filter(country == "Cambodia")
|> select(-continent, -lifeExp)
# Error: unexpected SPECIAL in "|>"
```



# Data Wrangling: `tidyr`

Now you have some experience working with tidy data and seeing the logic of wrangling when data are structured in a tidy way. But ‘real’ data often don’t start off in a tidy way, and require some reshaping to become tidy. The `tidyr` package is for reshaping data. You won’t use `tidyr` functions as much as you use `dplyr` functions, but it is incredibly powerful when you need it.

Why is this important? Well, if your data are formatted in a standard way, you will be able to use analysis tools that operate on that standard way. Your analyses will be streamlined and you won’t have to reinvent the wheel every time you see data in a different.

Data are often entered in a *wide* format where each row is often a site/subject/patient and you have multiple observation variables containing the same type of data.

An example of data in a *wide* format is the AirPassengers dataset which provides information on monthly airline passenger numbers from 1949-1960. You'll notice that each row is a single year and the columns are each month Jan - Dec.

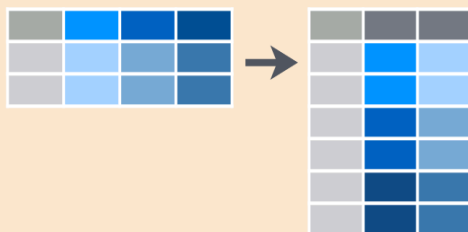
#### AirPassengers

##		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
##	1949	112	118	132	129	121	135	148	148	136	119	104	118
##	1950	115	126	141	135	125	149	170	170	158	133	114	140
##	1951	145	150	178	163	172	178	199	199	184	162	146	166
##	1952	171	180	193	181	183	218	230	242	209	191	172	194
##	1953	196	196	236	235	229	243	264	272	237	211	180	201
##	1954	204	188	235	227	234	264	302	293	259	229	203	229
##	1955	242	233	267	269	270	315	364	347	312	274	237	278
##	1956	284	277	317	313	318	374	413	405	355	306	271	306
##	1957	315	301	356	348	355	422	465	467	404	347	305	336
##	1958	340	318	362	348	363	435	491	505	404	359	310	337
##	1959	360	342	406	396	420	472	548	559	463	407	362	405
##	1960	417	391	419	461	472	535	622	606	508	461	390	432

This format is intuitive for data entry, but less so for data analysis. If you wanted to calculate the monthly mean, where would you put it? As another row?

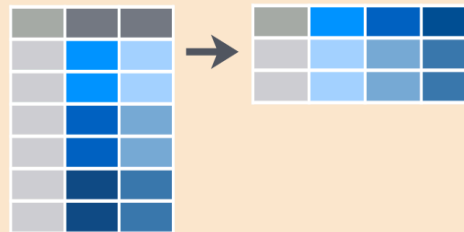
Often, data must be reshaped for it to become tidy data. What does that mean? There are four main verbs we'll use, which are essentially pairs of opposites:

- turn columns into rows (`gather()`),
- turn rows into columns (`spread()`),
- turn a character column into multiple columns (`separate()`),
- turn multiple character columns into a single column (`unite()`)



**tidyr::gather(cases, "year", "n", 2:4)**

Gather columns into rows.



**tidyr::spread(pollution, size, amount)**

Spread rows into columns.



**tidyr::separate(storms, date, c("y", "m", "d"))**

Separate one column into several.



**tidyr::unite(data, col, ..., sep)**

Unite several columns into one.

# Explore gapminder dataset

First have a look at the *wide* format data.

You can see there are a lot more columns than the version we looked at before. This format is pretty common, because it can be a lot more intuitive to enter data in this way.

continent	country	gdpPercap_1952	gdpPercap_1957	gdpPercap_1962	gdpPercap_1967	gdpPercap_1972	gdpPercap_1977
Africa	Algeria	2449.008185	3013.976023	2550.81688	3246.991771	4182.663766	4910.416756
Africa	Angola	3520.610273	3827.940465	4269.276742	5522.776375	5473.288005	3008.647355
Africa	Benin	1062.7522	959.6010805	949.4990641	1035.831411	1085.796879	1029.161251
Africa	Botswana	851.2411407	918.2325349	983.6539764	1214.709294	2263.611114	3214.857818
Africa	Burkina Faso	543.2552413	617.1834648	722.5120206	794.8265597	854.7359763	743.3870368
Africa	Burundi	339.2964587	379.5646281	355.2032273	412.9775136	464.0995039	556.1032651
Africa	Cameroon	1172.667655	1313.048099	1399.607441	1508.453148	1684.146528	1783.432873
Africa	Central African Republic	1071.310713	1190.844328	1193.068753	1136.056615	1070.013275	1109.374338
Africa	Chad	1178.665927	1308.495577	1389.817618	1196.810565	1104.103987	1133.98495
Africa	Comoros	1102.990936	1211.148548	1406.648278	1876.029643	1937.577675	1172.603047
Africa	Congo Dem. Rep.	780.5423257	905.8602303	896.3146335	861.5932424	904.8960685	795.757282
Africa	Congo Rep.	2125.621418	2315.056572	2464.783157	2677.939642	3213.152683	3259.178978
Africa	Cote d'Ivoire	1388.594732	1500.895925	1728.869428	2052.050473	2378.201111	2517.736547

# Setup

We'll learn `tidyr` in an RMarkdown file within a GitHub repository so we can practice what we've learned so far. You can either continue from the same RMarkdown as yesterday, or begin a new one.

## Here's what to do:

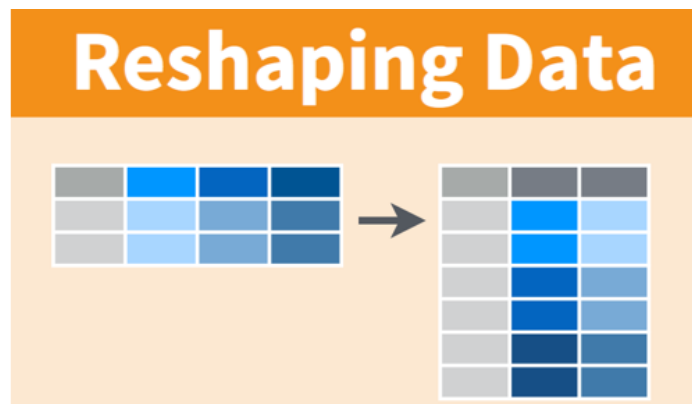
1. Clear your workspace (Session > Restart R)
2. New File > R Markdown..., save as something other than `gapminder-wrangle.Rmd` and delete irrelevant info, or just continue using `gapminder-wrangle.Rmd`

## load tidyverse (which has tidyr inside)

First load `tidyr` in an R chunk. You already have installed the tidyverse, so you should be able to just load it like this (using the comment so you can run `install.packages("tidyverse")` easily if need be):

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

## gather ( ) data from wide to long format



Read in the data. Let's also read in the gapminder data from yesterday so that we can use it to compare later on.

```
## wide format
gap_wide <- readr::read_csv('data/gapminder_wide.csv')
gapminder <- readr::read_csv('data/gapminder.csv')
```

Let's have a look:

```
#head(gap_wide)
str(gap_wide)
```

```
## spc_tbl_ [142 × 38] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ continent      : chr [1:142] "Africa" "Africa" "Africa" "Africa" ...
## $ country        : chr [1:142] "Algeria" "Angola" "Benin" "Botswana" ...
## $ gdpPercap_1952: num [1:142] 2449 3521 1063 851 543 ...
## $ gdpPercap_1957: num [1:142] 3014 3828 960 918 617 ...
## $ gdpPercap_1962: num [1:142] 2551 4269 949 984 723 ...
## $ gdpPercap_1967: num [1:142] 3247 5523 1036 1215 795 ...
## $ gdpPercap_1972: num [1:142] 4183 5473 1086 2264 855 ...
## $ gdpPercap_1977: num [1:142] 4910 3009 1029 3215 743 ...
## $ gdpPercap_1982: num [1:142] 5745 2757 1278 4551 807 ...
## $ gdpPercap_1987: num [1:142] 5681 2430 1226 6206 912 ...
## $ gdpPercap_1992: num [1:142] 5023 2628 1191 7954 932 ...
## $ gdpPercap_1997: num [1:142] 4797 2277 1233 8647 946 ...
## $ gdpPercap_2002: num [1:142] 5288 2773 1373 11004 1038 ...
## $ gdpPercap_2007: num [1:142] 6223 4797 1441 12570 1217 ...
```

While wide format is nice for data entry, it's not nice for calculations. Some of the columns are a mix of variable (e.g. "gdpPercap") and data ("1952"). What if you were asked for the mean population after 1990 in Algeria? Possible, but ugly. But we know it doesn't need to be so ugly. Let's tidy it back to the format we've been using.

Question: let's talk this through together. If we're trying to turn the `gap_wide` format into `gapminder` format, what structure does it have that we like? And what do we want to change?

- We like the continent and country columns. We won't want to change those.
- We want 1 column identifying the variable name (`tidyr` calls this a '**key**'), and 1 column for the data (`tidyr` calls this the '**value**').
- We actually want 3 different columns for variable: `gdpPercap`, `lifeExp`, and `pop`.
- We would like year as a separate column.



Let's get it to long format. We'll have to do this in 2 steps. The first step is to take all of those column names (e.g. `lifeExp_1970`) and make them a variable in a new column, and transfer the values into another column.

■ Question: What is our **key-value pair**?

We need to name two new variables in the key-value pair, one for the key, one for the value. It can be hard to wrap your mind around this, so let's give it a try. Let's name them `obstype_year` and `obs_values`.

Here's the start of what we'll do:

```
gap_long <- gap_wide |>
  gather(key = obstype_year,
         value = obs_values)
```

Let's inspect our work.

```
str(gap_long)
```

```
## tibble [5,396 × 2] (S3: tbl_df/tbl/data.frame)
## $ obstype_year: chr [1:5396] "continent" "continent" "continent" "continent" ...
## $ obs_values  : chr [1:5396] "Africa" "Africa" "Africa" "Africa" ...
```

```
head(gap_long)
```

```
## # A tibble: 6 × 2
##   obstype_year obs_values
##   <chr>        <chr>
## 1 continent    Africa
## 2 continent    Africa
## 3 continent    Africa
## 4 continent    Africa
## 5 continent    Africa
## 6 continent    Africa
```

```
tail(gap_long)
```

```
## # A tibble: 6 × 2
##   obstype_year obs_values
##   <chr>        <chr>
## 1 pop_2007     9031088
## 2 pop_2007     7554661
## 3 pop_2007     71158647
## 4 pop_2007     60776238
## 5 pop_2007     20434176
## 6 pop_2007     4115771
```

We have reshaped our dataframe but this new format isn't really what we wanted.

What went wrong? Notice that it didn't know that we wanted to keep continent and country untouched; we need to give it more information about which columns we want reshaped. We can do this in several ways.

One way is to identify the columns by name. Listing them explicitly can be a good approach if there are just a few. But in our case we have 30 columns. I'm not going to list them out here since there is way too much potential for error if I tried to list gdpPercap\_1952, gdpPercap\_1957, gdpPercap\_1962 and so on. But we could use some of dplyr's awesome helper functions — because we expect that there is a better way to do this!

```
gap_long <- gap_wide |>
  gather(key = obstype_year,
         value = obs_values,
         dplyr::starts_with('pop'),
         dplyr::starts_with('lifeExp'),
         dplyr::starts_with('gdpPercap')) #here i'm listing all the
str(gap_long)
```

```
## tibble [5,112 × 4] (S3: tbl_df/tbl/data.frame)
## $ continent : chr [1:5112] "Africa" "Africa" "Africa" "Africa" ...
## $ country   : chr [1:5112] "Algeria" "Angola" "Benin" "Botswana" ...
## $ obstype_year: chr [1:5112] "pop 1952" "pop 1952" "pop 1952" "pop 1952" ...
```

```
head(gap_long)
```

```
## # A tibble: 6 × 4
##   continent country      obstype_year obs_values
##   <chr>      <chr>      <chr>          <dbl>
## 1 Africa    Algeria    pop_1952        9279525
## 2 Africa    Angola     pop_1952        4232095
## 3 Africa    Benin      pop_1952        1738315
## 4 Africa    Botswana   pop_1952         442308
## 5 Africa    Burkina Faso pop_1952        4469979
## 6 Africa    Burundi    pop_1952        2445618
```

```
tail(gap_long)
```

```
## # A tibble: 6 × 4
##   continent country      obstype_year  obs_values
##   <chr>      <chr>      <chr>          <dbl>
## 1 Europe    Sweden     gdpPercap_2007  33860.
## 2 Europe    Switzerland gdpPercap_2007  37506.
## 3 Europe    Turkey     gdpPercap_2007   8458.
## 4 Europe    United Kingdom gdpPercap_2007  33203.
## 5 Oceania   Australia   gdpPercap_2007  34435.
## 6 Oceania   New Zealand gdpPercap_2007  25185.
```

Success! And there is another way that is nice to use if your columns don't follow such a structured pattern: you can exclude the columns you don't want.

```
gap_long <- gap_wide |>
  gather(key = obstype_year,
         value = obs_values,
         -continent, -country)
```

```
str(gap_long)
```

```
## tibble [5,112 × 4] (S3: tbl_df/tbl/data.frame)
## $ continent : chr [1:5112] "Africa" "Africa" "Africa" "Africa" ...
## $ country   : chr [1:5112] "Algeria" "Angola" "Benin" "Botswana" ...
## $ obstype_year: chr [1:5112] "gdpPercap_1952" "gdpPercap_1952" "gdpPercap_1952" ...
## $ obs_values : num [1:5112] 2449 3521 1063 851 543 ...
```

```
head(gap_long, 3)
```

```
## # A tibble: 3 × 4
##   continent country obstype_year obs_values
##   <chr>      <chr>    <chr>          <dbl>
## 1 Africa    Algeria gdpPercap_1952 2449.
## 2 Africa    Angola  gdpPercap_1952 3521.
## 3 Africa    Benin   gdpPercap_1952 1063.
```

To recap:

Inside `gather()` we first name the new column for the new ID variable (`obstype_year`), the name for the new amalgamated observation variable (`obs_value`), then the names of the old observation variable. We could have typed out all the observation variables, but as in the `select()` function (see `dplyr` lesson), we can use the `starts_with()` argument to select all variables that starts with the desired character string. `Gather` also allows the alternative syntax of using the `-` symbol to identify which variables are not to be gathered (i.e. ID variables).

OK, but we're not done yet. `obstype_year` actually contains two pieces of information, the observation type (`pop`, `lifeExp`, or `gdpPercap`) and the year. We can use the `separate()` function to split the character strings into multiple variables.

?separate → the main arguments are separate(data, col, into, sep ...). So we need to specify which column we want separated, name the new columns that we want to create, and specify what we want it to separate by. Since the obstype\_year variable has observation types and years separated by a \_, we'll use that.

```
gap_long <- gap_wide |>
  gather(key = obstype_year,
        value = obs_values,
        -continent, -country) |>
  separate(obstype_year,
    into = c('obs_type', 'year'),
    sep = "_",
    convert = TRUE) #this ensures that the year column is an
str(gap_long)
```

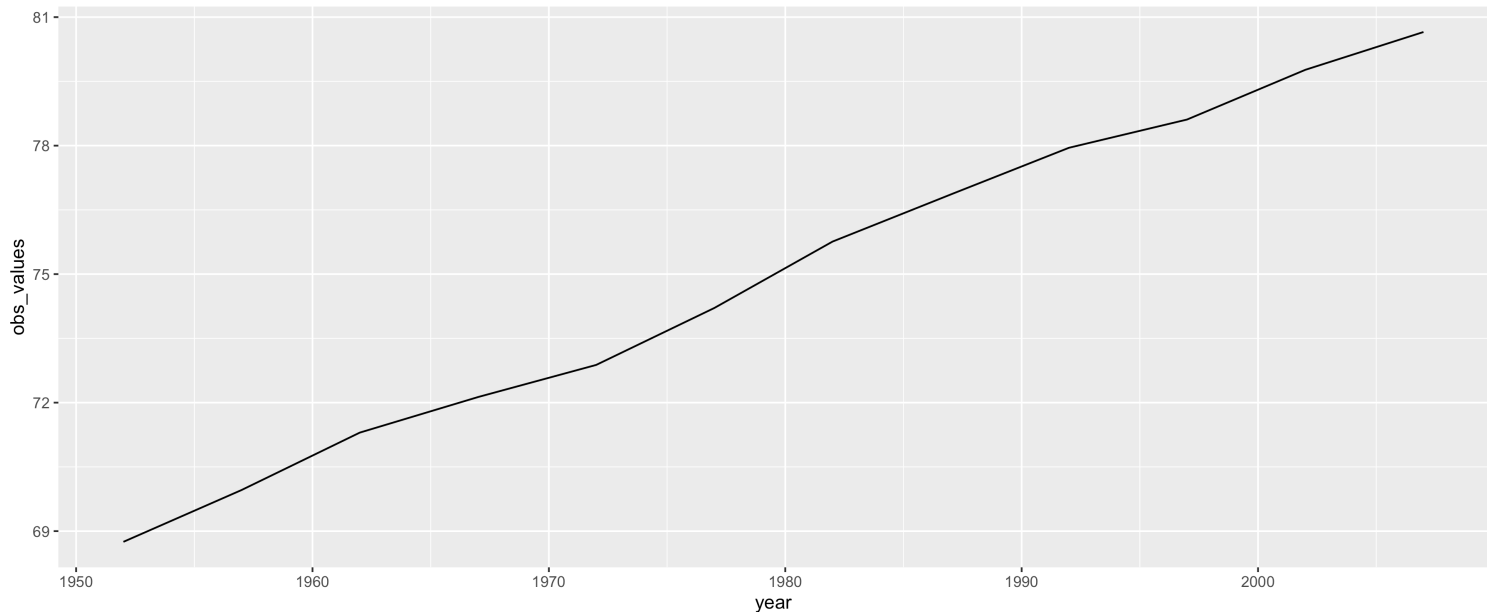
```
## tibble [5,112 × 5] (S3: tbl_df/tbl/data.frame)
## $ continent : chr [1:5112] "Africa" "Africa" "Africa" "Africa" ...
## $ country   : chr [1:5112] "Algeria" "Angola" "Benin" "Botswana" ...
## $ obs_type  : chr [1:5112] "gdpPercap" "gdpPercap" "gdpPercap" "gdpPercap" ...
## $ year      : int [1:5112] 1952 1952 1952 1952 1952 1952 1952 1952 1952 1952 ...
## $ obs_values: num [1:5112] 2449 3521 1063 851 543 ...
```

Excellent. This is long format: every row is a unique observation.

# Plot long format data

The long format is the preferred format for plotting with `ggplot2`. Let's look at an example by plotting just Canada's life expectancy.

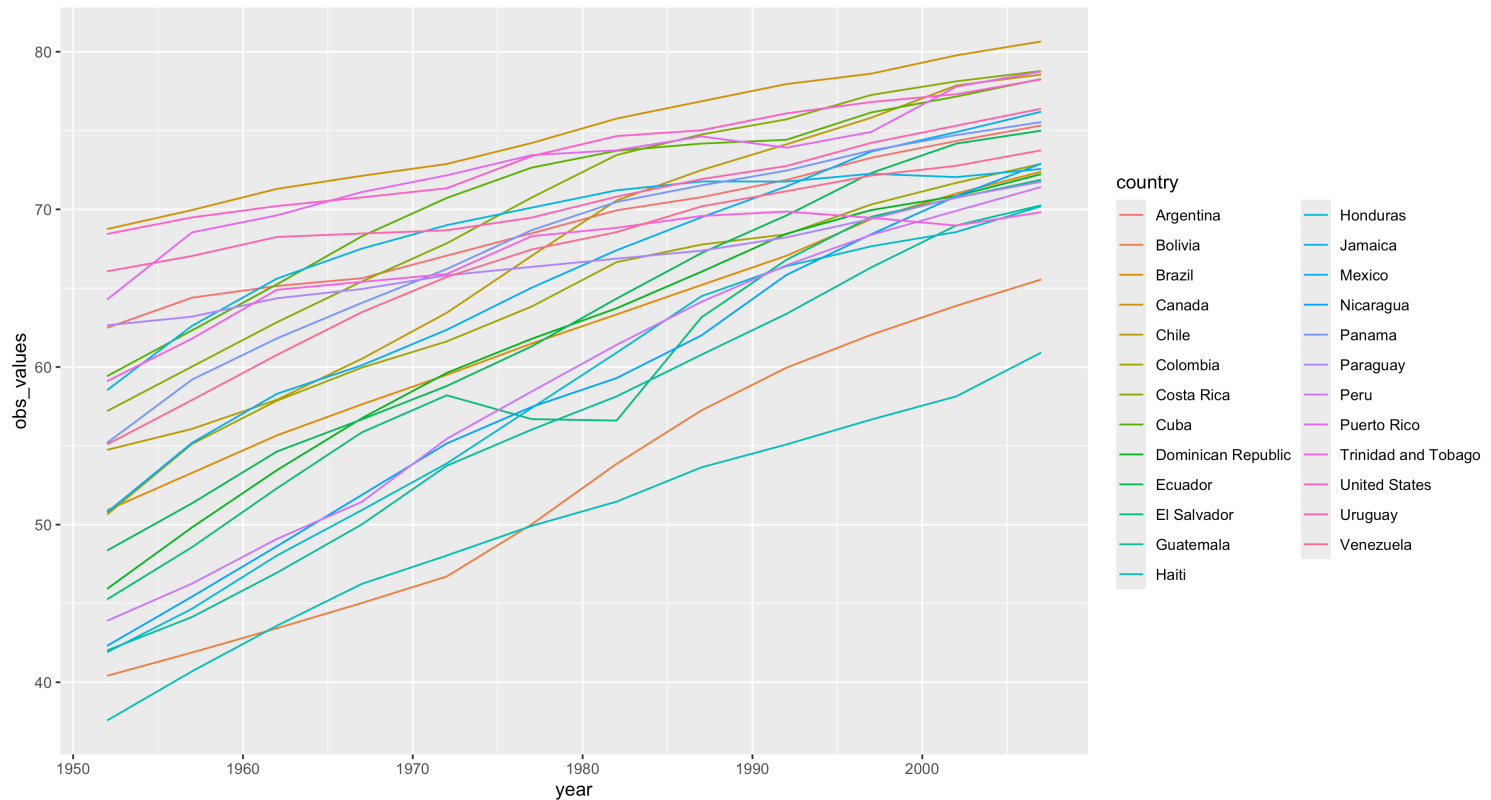
```
canada_df <- gap_long |>
  filter(obs_type == "lifeExp",
         country == "Canada")
ggplot(canada_df, aes(x = year, y = obs_values)) +
  geom_line()
```





We can also look at all countries in the Americas:

```
life_df <- gap_long |>
  filter(obs_type == "lifeExp",
         continent == "Americas")
ggplot(life_df, aes(x = year, y = obs_values, color = country)) +
  geom_line()
```



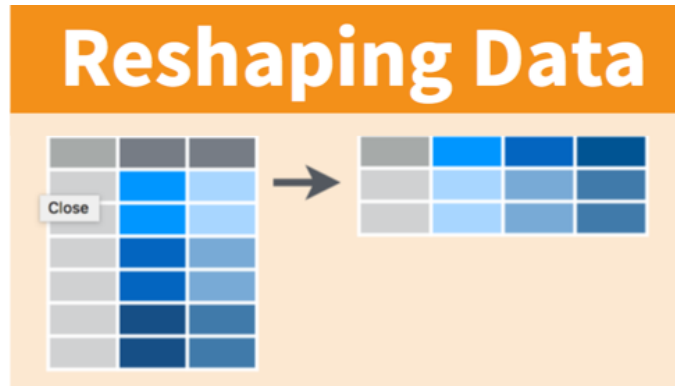
## Exercise

Using `gap_long`, calculate and plot the the mean life expectancy for each continent over time from 1982 to 2007. Give your plot a title and assign x and y labels. Hint: do this in two steps. First, do the logic and calculations using `dplyr::group_by()` and `dplyr::summarize()`. Second, plot using `ggplot()`.

# spread()

The function `spread()` is used to transform data from long to wide format

Alright! Now just to double-check our work, let's use the opposite of `gather()` to spread our observation variables back to the original format with the aptly named `spread()`. You pass `spread()` the key and value pair, which is now `obs_type` and `obs_values`.



```
gap_normal <- gap_long |>  
  spread(obs_type, obs_values)
```

```
dim(gap_normal)
```

```
## [1] 1704    6
```

```
dim(gapminder)
```

```
## [1] 1704    6
```

```
names(gap_normal)
```

```
## [1] "continent" "country"    "year"        "gdpPercap" "lifeExp"    "pop"
```

```
names(gapminder)
```

```
## [1] "country"    "year"        "pop"          "continent" "lifeExp"    "gdpPercap"
```

Now we've got a dataframe `gap_normal` with the same dimensions as the original `gapminder`.

## Exercise

1. Convert `gap_long` all the way back to `gap_wide`. Hint: Do this in 2 steps. First, create appropriate labels for all our new variables (variable\_year combinations) with the opposite of `separate`: `tidyr::unite()`. Second, `spread()` that variable\_year column into wider format.
2. Knit the R Markdown file and sync to Github (pull, stage, commit, push)

## clean up and save your .Rmd

Spend some time cleaning up and saving `gapminder-wrangle.Rmd` Restart R. In RStudio, use *Session > Restart R*. Otherwise, quit R with `q()` and re-launch it.

# Is there a relationship between life expectancy and GDP per capita?

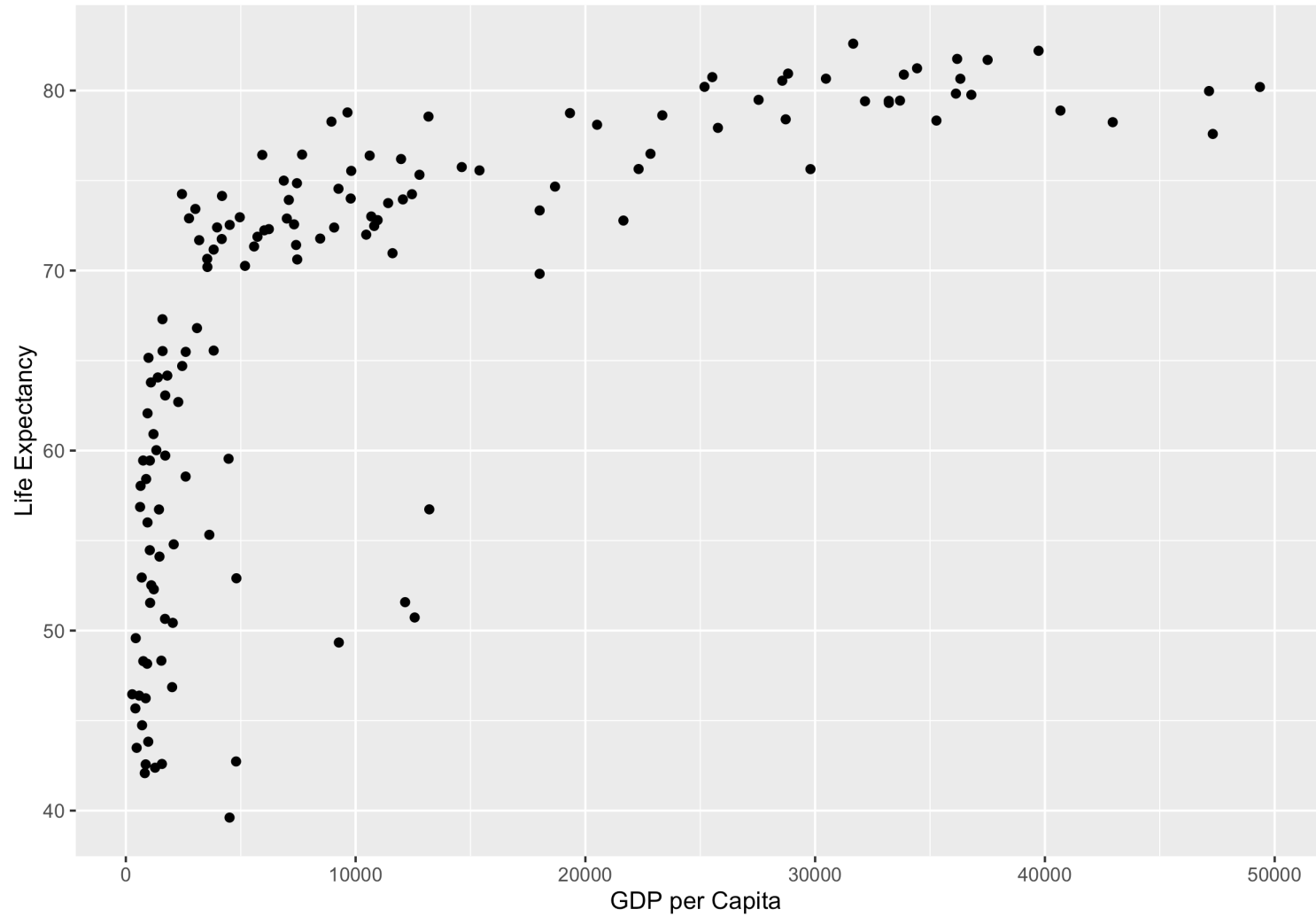
Let's use the gapminder data to answer this question. To try and answer it we will make a scatterplot. We will do this for the latest entry in the dataset which is:

```
max( gapminder$year )
```

```
## [1] 2007
```

```
gapminder |> filter(year==2007) |>  
  ggplot(aes(x = gdpPercap, y = lifeExp))+  
  geom_point()+  
  labs(x = "GDP per Capita",  
        y = "Life Expectancy",  
        title = "Data from Gapminder")
```

Data from Gapminder

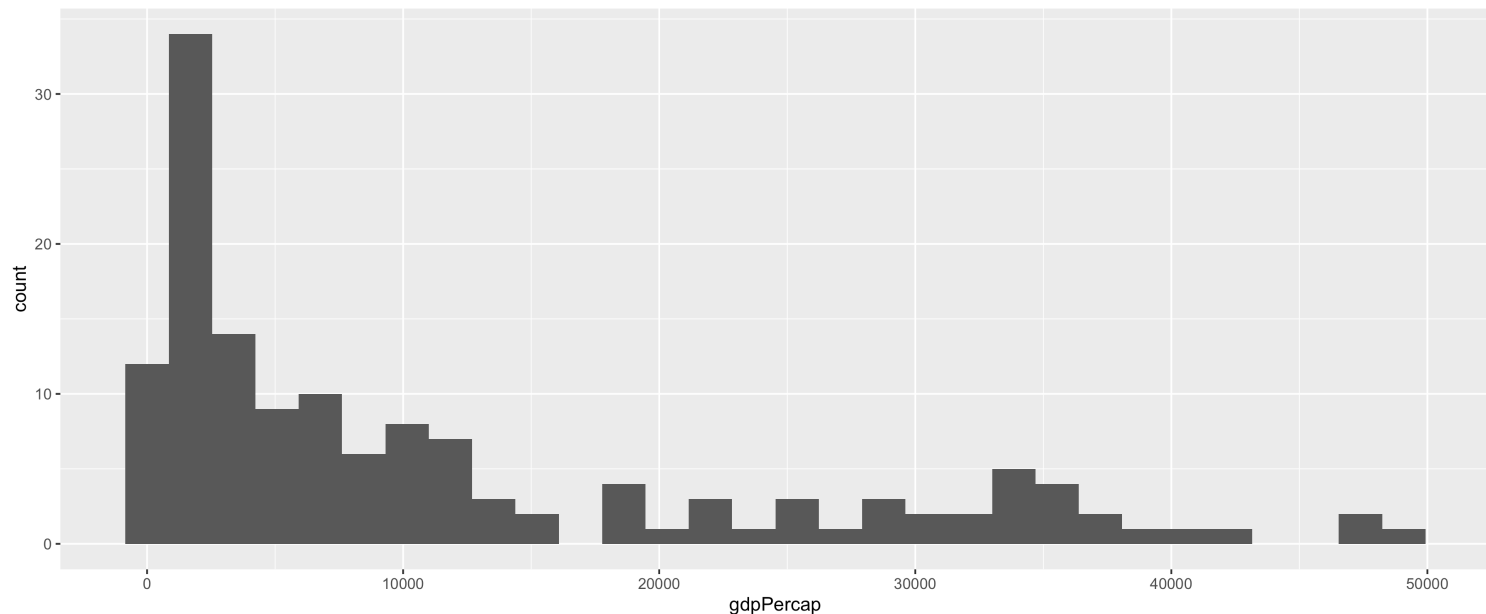


From this plot, we see that there is a wide variability in life expectancy for the lower income countries and then somewhat of a positive trend. However, there are many countries with incomes below 5,000 dollars per person and it is hard to see differences between these.

We can examine just this variable with a histogram.

```
gapminder |>  
  filter(year==2007) |>  
  ggplot(aes(x=gdpPercap)) + geom_histogram()
```

## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.

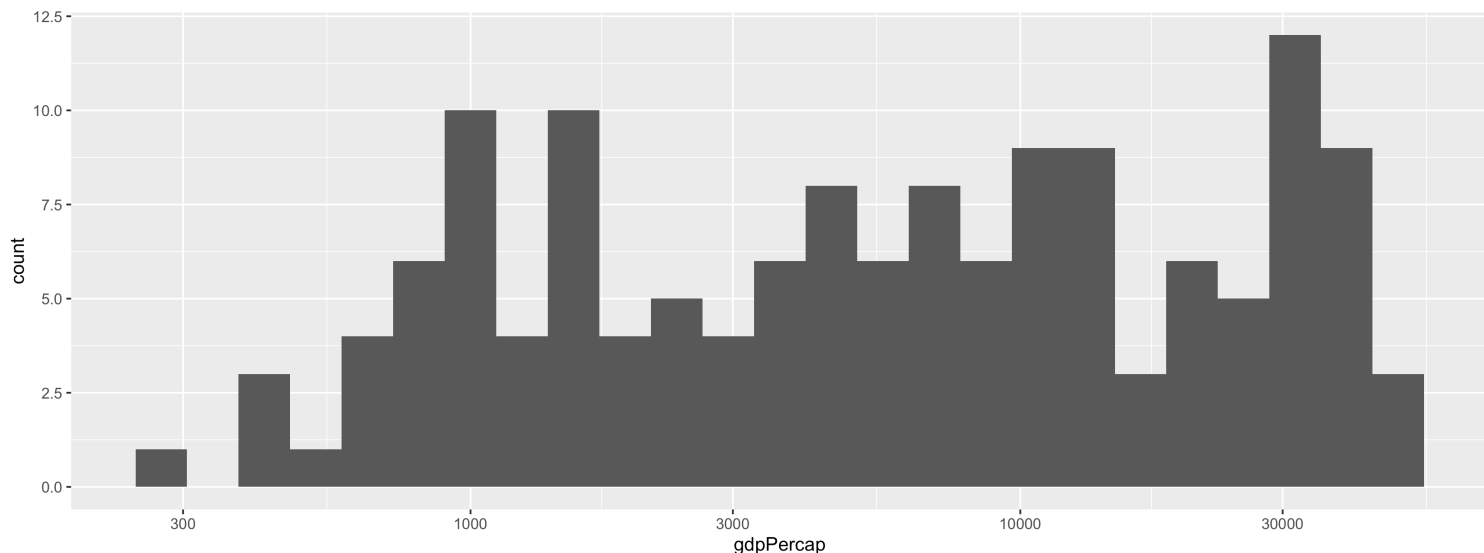




The histogram shows very large tails. We may do better by transforming the data. For data like this, the log transformation seems to work well. It also has a nice economic interpretation related to percent growth: in  $\log_{10}$  a change of 1 means the country is 10 times richer.

So how do we make the x-axis in the log scale? It is convenient to have this cheat sheet around when using `ggplot2`. From there we see that `scale_x_log10` does what we want.

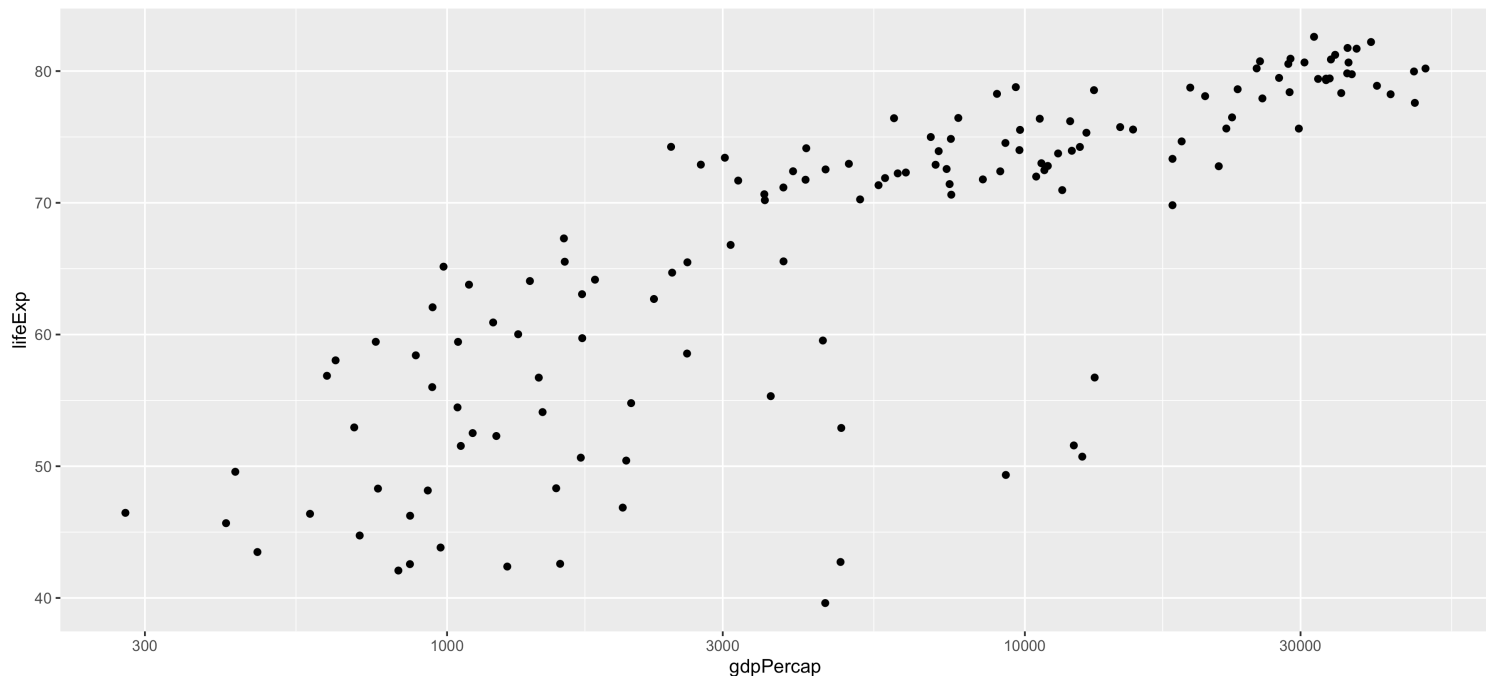
```
gapminder |>  
  filter(year==2007) |>  
  ggplot(aes(x=gdpPercap)) +  
  geom_histogram(bins=25) + scale_x_log10()
```



We no longer see extreme tails. The scatter plot now looks much more informative:

Now we can remake the scatter plot but now make sure the x-axis is in a log-scale.

```
gapminder |>  
  filter(year==2007) |>  
  ggplot(aes(x=gdpPercap, y = lifeExp)) +  
  geom_point() + scale_x_log10()
```



We can also use other really great packages, such as `ggrepel`:

```
library(ggrepel)
gapminder |>
  filter(year==2007) |>
  ggplot(aes(x=gdpPercap, y = lifeExp)) +
  geom_point(color = 'red',) + scale_x_log10() +
  geom_text_repel(aes(label = country), size = 2) +
  theme_classic()
```

