Data Wrangling for EDSDers

Session 1

13-16 Nov, 2023

Tim Riffe
Universidad del País Vasco & Ikerbasque (Basque Foundation for Science)

13 Nov, 2023

About me

Hi, I'm Tim Riffe, a US-American, CED-educated (Spain) demographer living in Spain and working at the Universidad del País Vasco as an Ikerbasque Research Fellow. I've been hooked on `R` since 2009 when I did EDSD, and have since authored several packages, mostly that do demographic things, but sometimes that do plotting things. My own research data preparation, analyses, analytic plotting, and presentation-quality plotting are all done in `R`. I've led dozens of introductory workshops like this one, including the previous four editions of the present EDSD module here: 2019, 2020, and 2022), However, **your module will be unique**. My own focuses are in programming tools for demographers, formal demography, data visualization, and health demography. Although this module is not about any of those things I hope we get a chance to talk about your work and my work if you're interested.

About this module

The route to being an `R` data analyst could be via Base programming, as it was for me, but it is a longer road than simply jumping into the `tidyverse`. In this module we will use the tidyverse framework to work with data. However, we'll also make use of function programming, and we'll blend the two things. Such code tends to be nicely legible, and in some ways it is easier to fix and change code. For a few different reasons, we'll be mostly working with R markdown and not in R scripts (although we could). The way to follow along in this course is to get set up correctly and to type along.

All I know is that you've had an intro course, and I assume no more than that. Where necessary I will review concepts. I expect you'll be able to keep up if you type along and take notes as we go, and if you ask questions and tell me to stop, slow down, or repeat when needed. Real learning will happen when you try to apply these concepts on your own, and for this reason I'll give exercises, which we'll then solve together. Often I will set myself up for apparent failure by not having worked through a solution in advance, and you'll see me induce errors and use help resources in a natural way. It's important to see this messy side of programming so that you also learn to make natural use of such resources and to react intelligently to error messages.

This module consists in a bunch of worked examples, hopefully different enough from one another that by the end you will have mastered the basics, seen a good variety of functionality, and be able to extend your capabilities all by yourself. If you internalize this last bit then you can consider yourself empowered. The formal objective of this module is to get you able to construct *data processing pipelines* to succinctly process haphazard messy datasets into tidy analyzable datasets via a series of sequential steps, including but limited to reading, parsing, recoding, *pivoting*, harmonizing, joining, grouping, aggregating, and so forth. By the end of the week, you'll be on your way to being able to fluently mix a large palette of these tools in a modular way. We do this course near the beginning of EDSD because these are skills you will use throughout the coming year(s). Practically, we'll mostly be using `dplyr` (and friends) to data wrangle, however any tool that can do the job is in-universe, so we'll see what happens. My *personal* objective is to impart *attitude* and *confidence* when dealing with new and messy datasets. This class is a safe space, so please ask, complain, laugh, and join in my bewilderment at the contemporary datascape.

R markdown basics

Class will happen in R markdown. Or maybe quarto? We should vote. Indeed, this material was produced in markdown. I'll say why in class and then fill this in.

tidy data

Tidy data refers to tables of a particular layout, with one observation per row and one variable per column (Wickham et al. 2014). Tidy datasets processed using `tidyverse` tools allow for fast and understandable analyses that in many cases require no programming, whereas it often takes a certain amount of head-scratching (programming) to analyze not-tidy datasets.

Example of tidy data

For example, you're probably all familiar with HMD lifetables (Human Mortality Database 2019): This data is tidy. If you were to take all the HMD single age lifetables and stack them, adding new grouping columns for Sex and Country, then this would still be tidy. That is, a single HMD observation consists in a unique combination of Country, Sex, Year, and Age. Each lifetable column is a variable, so the tidy way to store them is the traditional way: in columns.

Examples of not tidy data.

1. The UN's DemoData database is not tidy: it's much longer than tidy data. All variables are stacked, and the variable type (qx, lx, counts, etc) is coded in an additional grouping column. This is efficient for storage, given that not every observation has every variable. It is not efficient for interactive programming because observations are potentially scattered over rows.

2. A matrix with ages in rows and years in columns can be very convenient for traditional programming, and it appeals to demographers' Lexis intuition, but it does not follow our definition of tidy. To make it tidy, we would need columns for age and year, and a new column containing the variable stored in the matrix.

3. Often panel data is delivered with each row representing an individual, and with many many variables. If there are 100 variables, and 10 waves, we end up with 1000 or so columns. For many analyses this is not considered tidy: an observation is a unique combination of individual and wave. To make it tidy, we'd need a column to store the wave variable, and each id would have a row for each wave where they were observed.

Different data structures exist for various reasons, but in the end tidy data facilitates the kind of thinking and operations needed for a particular system of flexible data analysis and visualization. The big question you need to answer is *what is an observation*, and this is determined by the target analysis.

the gapminder `data.frame`

Let's work with a tidy `data.frame`. These are like a rectangular spreadsheet, or like a dataset in `Stata`: `data.frame`s have rows and columns, and there can be different kinds of data between columns, but only one kind of data within a column. Load it along with `tidyverse` in case you didn't before:

```
library(gapminder)
library(tidyverse)
```

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

```
str(gapminder)
```

```
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
##  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 163
##  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

Basic `ggplot2` with the gapminder data

This course isn't about `ggplot2`, but we'll use it from time to time, mostly so that there's a little prize at the end of each data processing pipeline. So we now have an explicit section on this will make this less mysterious.

Overview

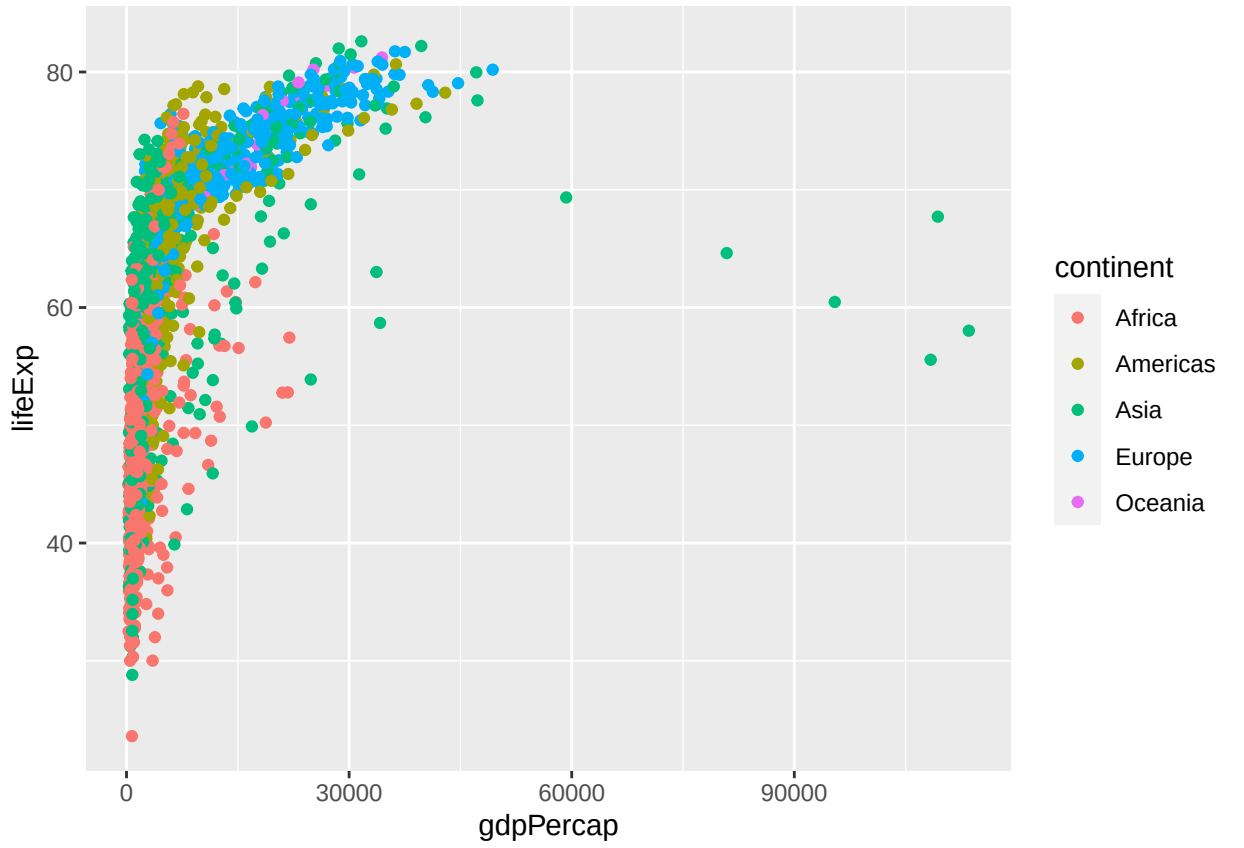
Tidy datasets such as this can also be visualized without further ado using a systematic grammar (Wilkinson 2012) implemented in the `ggplot2` package (Wickham (2016), this loads automatically with `tidyverse`). The `gapminder` examples I'll give today and tomorrow are either gratuitously lifted or modified from Healy (2019), which you can either purchase as the book, or refer to the free version online: www.socviz.co. It's a fabulous book.

NOTE: when following along in these examples, you should **type along**, and not copy-paste the code.

Let's take a first look:

Map to aesthetics

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



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

Notes: The first argument specifies the dataset. The `mapping` argument is specified always as `mapping = aes()`, where *aes* stands for aesthetics. At a minimum you'll want to map to `x`, but in this case also `y` and `color`. The things that you can map are variables in the tidy dataset. And where necessary `ggplot2` guesses whether they are quantitative or categorical in nature. You need to give a mapping to `ggplot()`, but you can also give different mappings for different geoms specified directly in the `geom` functions. That only makes sense if you'll have more than one `geom` going into the plot (we'll see this). Further modifications to the coordinate base created by `ggplot()` are *added* in to the expression with `+`.

Adjust axis scales

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

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

`geoms` and `facets`

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

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

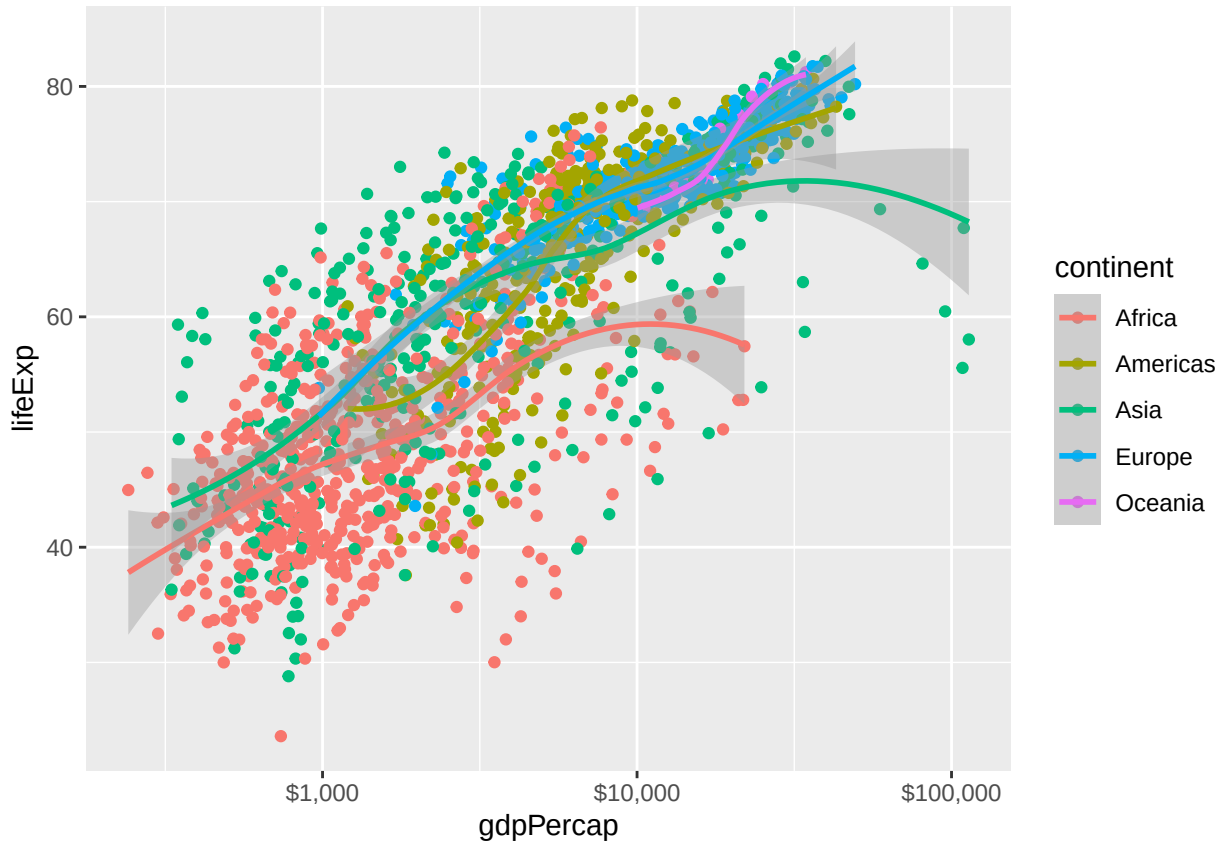
```
ggplot(gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
    geom_point(mapping = aes(color = continent)) +
    geom_smooth(method = "loess") +
    scale_x_log10(labels = dollar)
```

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

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

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

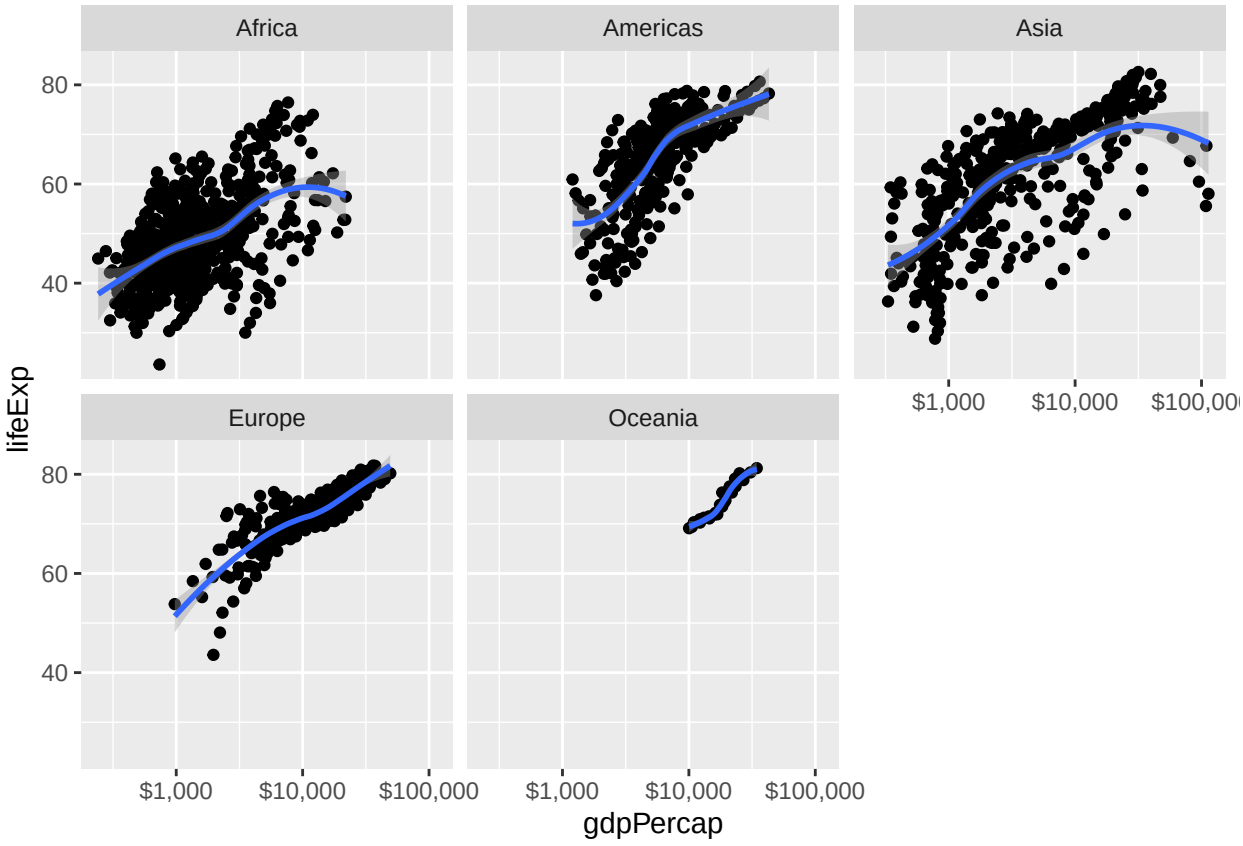
```
## `geom_smooth()` using formula = 'y ~ x'
```



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

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

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

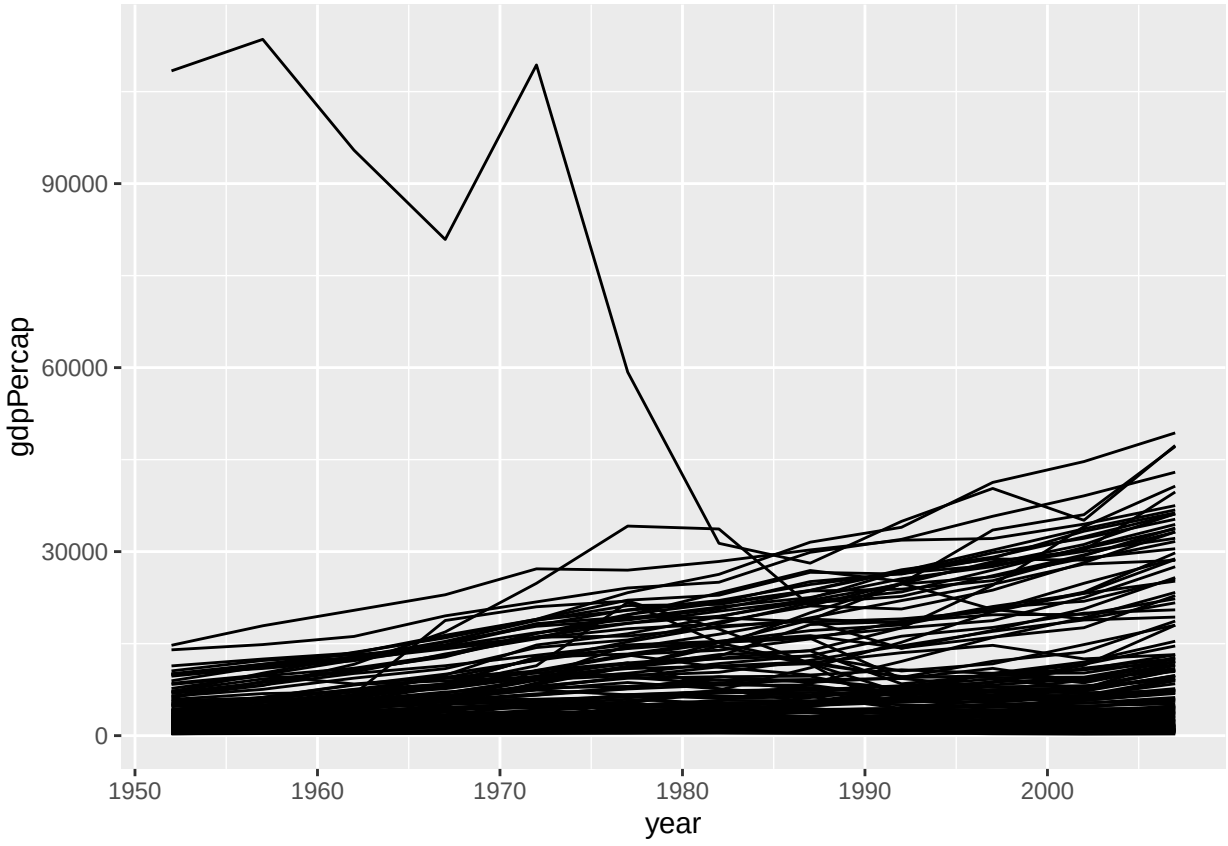


We get panel plots like this by specifying a layout formula in `facet_wrap()`, where `~` separates left and right sides of the formula, and where left means rows in a panel layout and right means columns. Since there's nothing on the left it just orders the continents. Color is no longer needed since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

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

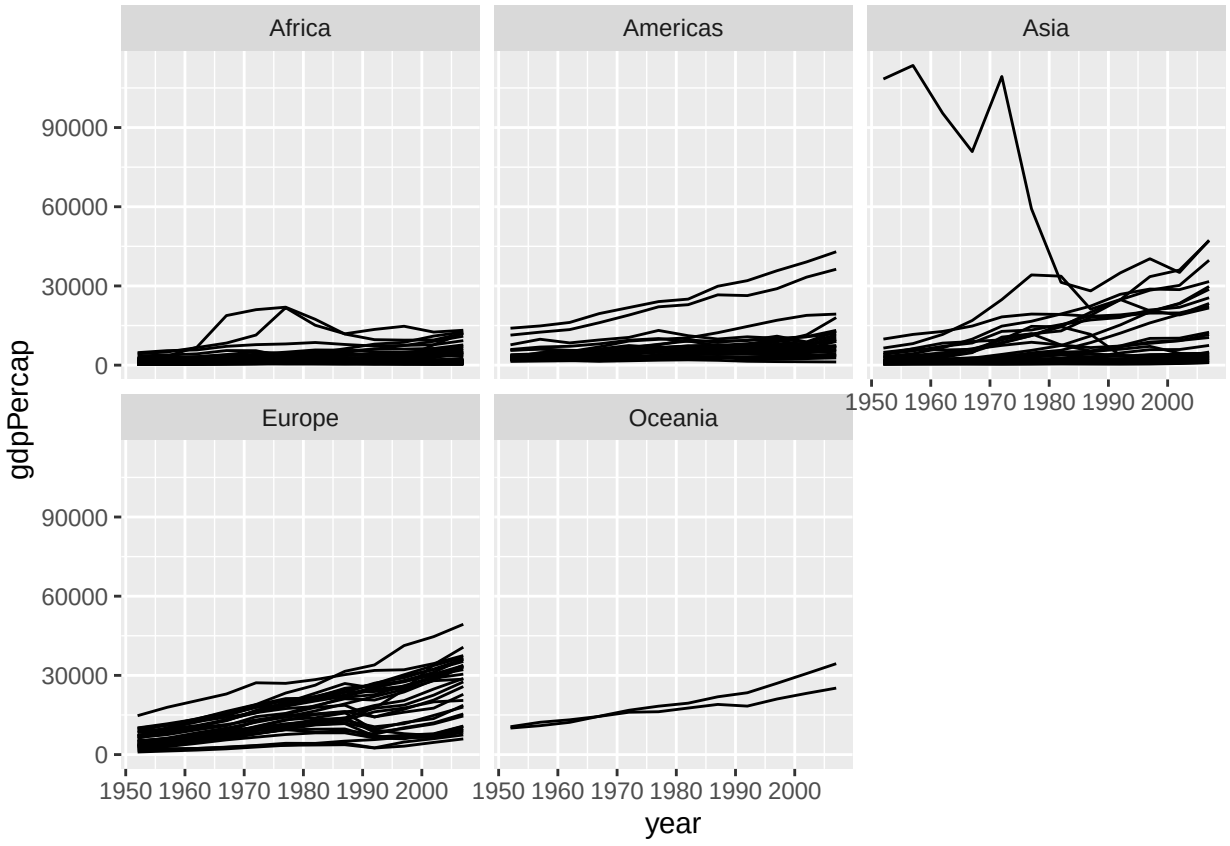
Too busy?

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



Use faceting to separate the continents

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



Summary lessons

More sophisticated usage of `ggplot2` builds on these basics in a rather consistent way. You can deepen your fluency by learning new `geoms`, learning what things can be *mapped* to aesthetics, and learning options to scale axes and mappings. To get an overview, refer to the `ggplot2` cheat sheet. For a very approachable introduction, try the Healy book. For the most thorough and advanced treatment, refer to the Wickham book. To learn usage, look at help examples, or R graph gallery examples (add links). In following lessons I will sometimes use `ggplot2`, and I will slip in further tips in an unstructured way.

Basic `dplyr`

`ggplot2` is a wonderful tool if your data is already both *tidy* and *mappable*. But source data rarely is. To feed a dataset to `ggplot2` often we'll want to pre-process data, or even model it. The tools for data management, shaping, matching, sorting, joining, filtering, and so forth (stuff to prep for `ggplot2`) blend seamlessly into the tools of data analysis and modeling when one uses the `dplyr` approach. Some basic concepts will help us get started:

1. `dplyr` functions are *verbal* because they *do* things to data.
2. A bunch of `dplyr` functions executed in sequence on a dataset for a *pipeline*.
3. Data processing pipelines read as sentences, because verbs are strung together using pipes. This is the pipe symbol: `%>%`
4. Data processing pipelines are quick to build, easy to read, and help keep your `R` workspace less cluttered than most alternatives. Clean coding and clean workspaces means code is easier to maintain, even when not written by yourself!

This probably sounds jargony, so let's hop to it:

Verbs

Overview

Here are a selection of `dplyr` verbs. Some things to note: the first argument of each is a data object, usually something like a `data.frame`. These do things to the dataset, and they return the modified dataset back.

1. `group_by()` allows operations on subgroups
2. `pivot_longer()` make a wide range of columns to long format
3. `pivot_wider()` does the inverse
4. `mutate()` make new columns, potentially using other columns, no loss of rows
5. `summarize()` aggregates over rows. Usually reduces nr of rows
6. `select()` selects columns
7. `filter()` selects rows (subsets)

We'll see other `dplyr` verbs here and there during this module

Flip through the tabs in order.

`group_by()` When you want to operate on groups in a dataset, then it is necessary to declare the grouping.

```
gap_grouped    <- group_by(gapminder, country, year, continent)
head(gap_grouped)
```

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

Grouping does not do anything to the data directly. All values will be identical, and the dimensions are the same. It just adds a piece of metadata to the object that declares the grouping.

NB: the object created here is used in the following code chunk.

`pivot_longer()` Here we use `pivot_longer()` to pull all the gapminder value columns into a single key-value pair, where the key is the variable column name, and value is the contents. The resulting `data.frame` is three times longer because there are three variable columns (and three key columns), and this mimics the setup of your own `DemoData` SQL database.

```
gapminder_long <- pivot_longer(gap_grouped,
                          lifeExp:gdpPercap,
                          names_to = "varname",
                          values_to = "value")
```

Note: `group_by()` is a necessary pre-step in order to maintain the unique key combinations. We could consider `continent` either a key or a variable. It's not strictly needed as a key because country-year uniquely identifies observations, but I left it there to not over-lengthen the dataset.

You can get back to the original format with the `pivot_wider()`, which is the opposite of `pivot_longer()`.

```
pivot_wider(gapminder_long, names_from = "varname", values_from = "value")
```

`mutate()` `mutate()` adds columns to, or modifies columns in a `data.frame`/`tibble`. Think of the example of lifetable construction: You might start with deaths and exposures, and all other columns are created based on these. This is a case for `mutate()`. The main thing to remember is that `mutate()` does not change the number of rows. Keeping with our gapminder example, say we want to add a column for the mean life expectancy, but repeated for each row within each group. To get these means for each country, we'll want to group by country first (our previous grouping also grouped by year).

```
gap_grouped <- group_by(gapminder, country)
gap_mutated <- mutate(gap_grouped,
       mean_lifeExp = mean(lifeExp))
head(gap_mutated)
```

```
## # A tibble: 6 x 7
## # Groups:   country [1]
##   country     continent  year lifeExp      pop gdpPercap mean_lifeExp
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>        <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.         37.5
## 2 Afghanistan Asia       1957    30.3  9240934      821.         37.5
## 3 Afghanistan Asia       1962    32.0 10267083      853.         37.5
## 4 Afghanistan Asia       1967    34.0 11537966      836.         37.5
## 5 Afghanistan Asia       1972    36.1 13079460      740.         37.5
## 6 Afghanistan Asia       1977    38.4 14880372      786.         37.5
```

You can also just keep comma separating newly made columns, like so:

```
mutate(gap_grouped,
       mean_lifeExp = mean(lifeExp),
       sd_lifeExp = sd(lifeExp),
       cov_lifeExp = sd_lifeExp / mean_lifeExp,
       cov_pop = sd(population) / mean(population))
```

You see how `cov_lifeExp` is created using columns being created within the same call? Pretty neat! Expressions can also be more complicated, see where `cov_pop` is created using two function calls.

NB: The object `gap_grouped` created here is also used in the next code chunk.

`summarize()` If instead we wished to collapse the dataset to have one observation per country, then we use `summarize()` (this can also by lossy with respect to columns). Here is that first `mutate` expression when executed with `summarize()`:

```
gap_summarize <- summarize(gap_grouped,
       mean_lifeExp = mean(lifeExp))
head(gap_summarize)
```

```
## # A tibble: 6 x 2
##   country     mean_lifeExp
##   <fct>              <dbl>
## 1 Afghanistan         37.5
## 2 Albania             68.4
## 3 Algeria             59.0
## 4 Angola              37.9
## 5 Argentina           69.1
## 6 Australia           74.7
```

```
dim(gap_summarize) # collapsed
```

```
## [1] 142   2
```

If you want to summarize more than one column, then this works, also with sequential dependency over columns created on the fly:

```
gap_summarize <- summarize(gap_grouped,
       mean_lifeExp = mean(lifeExp),
       mean_gdpPercap = mean(gdpPercap),
       sd_gdpPercap = sd(gdpPercap),
       cov_gdpPercap = sd_gdpPercap / mean_gdpPercap)
head(gap_summarize)
```

```
## # A tibble: 6 x 5
##   country     mean_lifeExp mean_gdpPercap sd_gdpPercap cov_gdpPercap
##   <fct>              <dbl>          <dbl>        <dbl>         <dbl>
## 1 Afghanistan         37.5           803.         108.         0.135
## 2 Albania             68.4          3255.        1192.         0.366
## 3 Algeria             59.0          4426.        1310.         0.296
## 4 Angola              37.9          3607.        1166.         0.323
## 5 Argentina           69.1          8956.        1863.         0.208
## 6 Australia           74.7         19981.        7815.         0.391
```

You can get a lot of mileage out of `group_by()`, `mutate()` and `summarize()`.

`select()` `select()` picks out columns, and potentially renames them:

```
head(select(gapminder, country, GDP = gdpPercap))
```

```
## # A tibble: 6 x 2
##   country       GDP
##   <fct>       <dbl>
## 1 Afghanistan  779.
## 2 Afghanistan  821.
## 3 Afghanistan  853.
## 4 Afghanistan  836.
## 5 Afghanistan  740.
## 6 Afghanistan  786.
```

If data are grouped, the key columns are retained (and it tells you so in the console):

```
gap_grouped <- group_by(gapminder, country, year)
head(select(gap_grouped, GDP = gdpPercap))
```

```
## Adding missing grouping variables: `country`, `year`
```

```
## # A tibble: 6 x 3
## # Groups:   country, year [6]
##   country      year   GDP
##   <fct>       <int> <dbl>
## 1 Afghanistan  1952  779.
## 2 Afghanistan  1957  821.
## 3 Afghanistan  1962  853.
## 4 Afghanistan  1967  836.
## 5 Afghanistan  1972  740.
## 6 Afghanistan  1977  786.
```

`filter()` More often we want to subset rows to those that satisfy certain selection criteria. Use `filter()` for this:

```
gap_filtered <- filter(gapminder,
                       year >= 2000,
                       year <= 2010,
                       continent == "Africa")
str(gap_filtered)
```

```
## tibble [104 x 6] (S3: tbl_df/tbl/data.frame)
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 3 3 4 4 11 11 14 14 17 17 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int [1:104] 2002 2007 2002 2007 2002 2007 2002 2007 2002 2007 ...
##  $ lifeExp  : num [1:104] 71 72.3 41 42.7 54.4 ...
##  $ pop      : int [1:104] 31287142 33333216 10866106 12420476 7026113 8078314 1630347 1639131 122512
##  $ gdpPercap: num [1:104] 5288 6223 2773 4797 1373 ...
```

There are three logical expressions here, each producing a value of TRUE or FALSE for each row. Comma separation between these is interpreted as `&`, so the full logical expression evaluated is interpreted as: `year >= 2000 & year <= 2010 & continent == "Africa"` Note that double equals `==` is for the logical evaluation of equality! The other logical operators are pretty straightforward. A vertical bar `|` is used for *or*.

- There are other packages in `R` that have functions called `filter()`, uh oh! If this ever comes up and causes problems for you, just write it as `dplyr::filter()` instead and it won't get mixed up.

`ungroup()` It is often a good idea to remove a grouping attribute from a data object when you're done using it. Otherwise you might operate within groups when you don't mean to. Then execute `ungroup()` on the object. This is also a clean way to remove a grouping before declaring a new and different one. For example, say we want to make a column for the mean `lifeExp` within the series for each country, and also the mean over countries within each year. Then we'll need to switch groupings:

```
gap_gr1 <- group_by(gapminder, country)
gap_gr1 <- mutate(gap_gr1,
                  mean_lifeExp_country = mean(lifeExp))
```

```
# now undo to operate on a different group
gap_gr2 <- ungroup(gap_gr1)
gap_gr2 <- group_by(gap_gr2, year)
gap_gr2 <- mutate(gap_gr2, mean_lifeExp_year = mean(lifeExp))
head(gap_gr2)
```

```
## # A tibble: 6 x 8
## # Groups:   year [6]
##   country     continent  year lifeExp      pop gdpPercap mean_lifeExp_country mean_lifeExp_year
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>                <dbl>              <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.                 37.5               49.1
## 2 Afghanistan Asia       1957    30.3  9240934      821.                 37.5               51.5
## 3 Afghanistan Asia       1962    32.0 10267083      853.                 37.5               53.6
## 4 Afghanistan Asia       1967    34.0 11537966      836.                 37.5               55.7
## 5 Afghanistan Asia       1972    36.1 13079460      740.                 37.5               57.6
## 6 Afghanistan Asia       1977    38.4 14880372      786.                 37.5               59.6
```

Pipes

I had to force myself not to use pipes in the previous section! Really `dplyr` functions can be executed in sequence without the need to create intermediate objects. If you executed the previous examples, look at how cluttered RStudio's environment tab has become. We can avoid all that by stringing things together using the pipe operator from the `magrittr` package that loads automatically with `tidyverse`. here are some of those example reworked. See how they read as sentences?

Before:

```
gap_grouped <- group_by(gapminder, country)
gap_mutated <- mutate(gap_grouped,
       mean_lifeExp = mean(lifeExp))
head(gap_mutated)
```

After:

```
gapminder %>%
    group_by(country) %>%
    mutate(mean_lifeExp = mean(lifeExp)) %>%
    head()
```

Note: the first argument (`data`) is automatically filled with the object running through the pipeline. You should imagine `gapminder` running through a staged pipeline, getting modified at each step. When it gets to the end we just show the first several rows in the console.

Before:

```
gap_gr1 <- group_by(gapminder, country)
gap_gr1 <- mutate(gap_gr1,
                  mean_lifeExp_country = mean(lifeExp))
# now undo to operate on a different group
gap_gr2 <- ungroup(gap_gr1)
gap_gr2 <- group_by(gap_gr2, year)
gap_gr2 <- mutate(gap_gr2, mean_lifeExp_year = mean(lifeExp))
head(gap_gr2)
```

After:

```
gapminder %>%
    group_by(country) %>%
    mutate(mean_lifeExp_country = mean(lifeExp)) %>%
    ungroup() %>%
    group_by(year) %>%
    mutate(mean_lifeExp_year = mean(lifeExp)) %>%
    head()
```

Instead of ending in `head()`, we could assign the result to a new object. This is usually done at the start, like so:

```
gap_mutated <-
    gapminder %>%
    group_by(country) %>%
    mutate(mean_lifeExp_country = mean(lifeExp)) %>%
    ungroup() %>%
    group_by(year) %>%
    mutate(mean_lifeExp_year = mean(lifeExp))
```

The object `gap_mutated` consists in the fully-executed pipeline.

Note: We usually construct pipelines like these incrementally, often checking as we to make sure things work as expected. Often that checking is done like we saw above, with `%>% head()` or similar as the last step. As we determine next steps, this checker line gets pushed further down in the chain of execution. Just remember to remove it when you're done building the pipeline! Another great tool is the new `ViewPipeSteps` addin, which opens up a tab for each step in a pipeline! You need to install it from github like so:

```
remotes::install_github("daranzolin/ViewPipeSteps")
```

Select this text, then from the addins menu in `RStudio`, select `View Pipe Chain Steps`

```
library(ViewPipeSteps)
gapminder %>%
    group_by(country) %>%
  mutate(mean_lifeExp_country = mean(lifeExp)) %>%
    ungroup() %>%
  group_by(year) %>%
  mutate(mean_lifeExp_year = mean(lifeExp))
```

```
## # A tibble: 1,704 x 8
## # Groups:   year [12]
##    country     continent  year lifeExp      pop gdpPercap mean_lifeExp_country mean_lifeExp_year
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>                <dbl>              <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.                 37.5               49.1
##  2 Afghanistan Asia       1957    30.3  9240934      821.                 37.5               51.5
##  3 Afghanistan Asia       1962    32.0 10267083      853.                 37.5               53.6
##  4 Afghanistan Asia       1967    34.0 11537966      836.                 37.5               55.7
##  5 Afghanistan Asia       1972    36.1 13079460      740.                 37.5               57.6
##  6 Afghanistan Asia       1977    38.4 14880372      786.                 37.5               59.6
##  7 Afghanistan Asia       1982    39.9 12881816      978.                 37.5               61.5
##  8 Afghanistan Asia       1987    40.8 13867957      852.                 37.5               63.2
```

```
##  9 Afghanistan Asia       1992    41.7 16317921      649.                37.5              64.2
## 10 Afghanistan Asia       1997    41.8 22227415      635.                37.5              65.0
## # i 1,694 more rows
```

Note also: We use indentation and alignment to make the code more legible.

As mentioned, `dplyr` pipelines also read as sentences. This last one can be verbalized as "first take the gapminder dataset, then group it by country, then make a new column for the average life expectancy over time within the country, then regroup by year and make a new column for the average life expectancy over countries within the year". Such sentences don't make for good lit but they do make your code accessible to readers, which is a major advantage. You can (should) also comment as you go.

Legibility of a script is achieved by reducing clutter and by using succinct pipelines. This is important because your processing/analysis code becomes all the more shareable, both with others and with future you. Who hasn't looked back at your own code written years ago and wonders what the heck it does? It feels awful to have to reverse engineer your own logic and coding. Yes, it can be overcome with good annotations, but let's obviate the need altogether if we can.

As mentioned, I'll slip in pipelines and `ggplot2` here and there in the coming days, so there will be several opportunities for practice and repetition.

Exercises

A `dplyr` worked example

Let's download this spreadsheet and figure out how to work with it! The exercise is to produce a ridgeplot where levels are TFR intervals, and two single age asfr distributions are shown, picked out as those that have the highest and lowest mean age at childbearing (MAB) respectively. Each code chunk is described right below it.

http://data.un.org/DocumentData.aspx?id=319

Download the spreadsheet and stick it in a new folder called `Data`. We're going to figure this thing out. For the intrepid, you can create the folder and automatically download the data like so:

```
if (!dir.exists("Data")){
  dir.create("Data")
}
data_download_trigger_url <-
  "http://data.un.org/Handlers/DocumentDownloadHandler.ashx?id=319&t=bin"
# give R some extra time in case we have choppy connection
options(timeout = 200)
# download triggers properly with above url
download.file(url = data_download_trigger_url,
              destfile = "Data/un_fertility.xls")
```

```
library(readxl)
ASFR <- read_excel(
  "Data/un_fertility.xls",
    na = "..",
    skip = 4) %>% # Ctrl + Shift + m
  select(1:13) %>%
  dplyr::rename("Country" = "...1", "LocID" = "...2",
```

```
                      "PeriodVerbose" = "...3","TFR" = "...6",
                      "15" = "15-19","20" = "20-24",
                      "25" = "25-29","30" = "30-34",
                      "35" = "35-39","40" = "40-44",
                      "45" = "45-49")
```

```
## New names:
## * `` -> `...1`
## * `` -> `...2`
## * `` -> `...3`
## * `` -> `...6`
## * `` -> `...14`
## * `` -> `...15`
## * `` -> `...16`
## * `` -> `...17`
## * `` -> `...18`
## * `` -> `...19`
## * `` -> `...20`
## * `` -> `...21`
## * `` -> `...22`
```

Here we have a 3-step pipeline! Yay. The first step is syntax to read in from a spreadsheet, generated interactively using the import data feature of `RStudio`. Copy. Paste. Bam. Step 2 `select()`s just the columns we want for now. Step 3 `rename()`s the columns manually to something useful.

Pipelines read like sentences. Because they are made of verbs and the pipe operator reads as *and then do this*.

```
# here we're assigning at the top, but it's getting
# what comes out the bottom
ASFR <- ASFR %>%
  select(-c(PeriodVerbose, TFR, Period)) %>%
  pivot_longer(cols = `15`:`45`,
               names_to = "Age",
               values_to = "asfr") %>%
  mutate(Age = as.integer(Age),
         asfr = asfr / 1000) %>%
  filter(!is.na(asfr)) %>%
  group_by(Country, Year) %>%
  mutate(TFR = sum(asfr, na.rm = TRUE) * 5,
         MAB = sum((5 * asfr) * (Age + 2.5), na.rm = TRUE) / TFR,
         # how many rows in this subset?
         n = n()) %>%
  ungroup() %>%
  # just keep usable subsets
  filter(TFR > 0,
         n > 4)  # ASFR up top gets what comes out here
```

What method do we use to split these data to single ages?

Suggestions from the field:

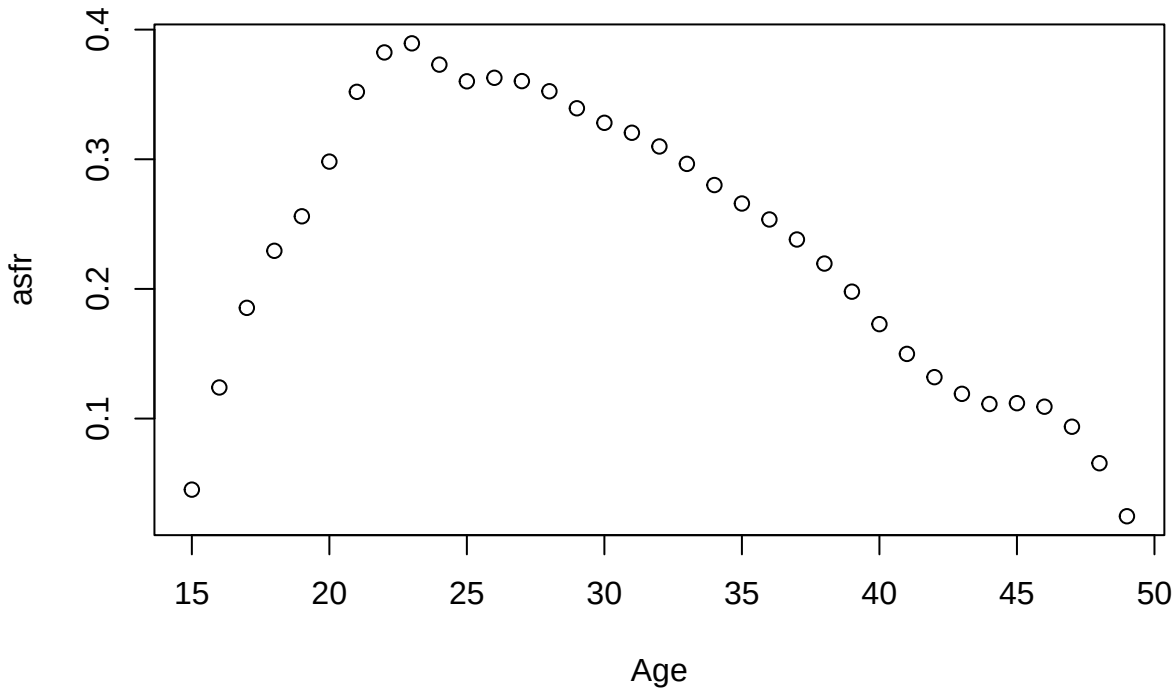
1. PCLM

2. Sprague (or Beers, or Grabill, or . . .)
3. smooth spline through uniform repeated values in age groups (not constrained, but could work OK)
4. monotonic spline through cumulative distribution
5. use a fertility model, e.g. Coale-Trussel and pals
6. TOPALS http://schmert.net/calibrated-spline/ Super awesome love it

We will try the monotonic spline method because it seems easier to write fresh, even though you probably wouldn't use this as your first choice in real applications.

```
mono_graduate <- function(Age5, asfr5){

  # assume NAs are 0s, sometimes these are found
  # in the first or last age groups.
  # Rather than setting to 0, we could also throw these out,
  # and adapt Age5 variable made below...
  asfr5[is.na(asfr5)] <- 0
  # anchor the endpoints (added double anchor)
  Age5                <- c(13,14, Age5+4, 50,51)
  asfr5               <- c(0,0, asfr5, 0,0)
  # scale up asfr
  asfr5               <- asfr5 * 5
  # accumualte it
  casfr5              <- cumsum(asfr5)
  # start at 14 because we need firs differences
  # to accumulate...
  predict_ages        <- 14:49
  # fit the spline
  xy <- splinefun(x = Age5,
            y = casfr5,
            method = "monoH.FC",
            ties = "min")(predict_ages)
  # return tibble
  tibble(Age = 15:49,
         asfr = diff(xy))
}
chunk <- filter(ASFR, LocID == 4, Year == 1973)
plot(mono_graduate(Age5 = chunk$Age,
              asfr5 = chunk$asfr))
```



Great now we have a spline function, let's see if we can figure out how to use it in a pipeline.

```
ASFR1 <-
  ASFR %>%
  group_by(Country, LocID, Year) %>%
  group_modify(~mono_graduate(.x$Age, .x$asfr)) %>%
  ungroup()

nrow(ASFR)
```

```
## [1] 7268
```

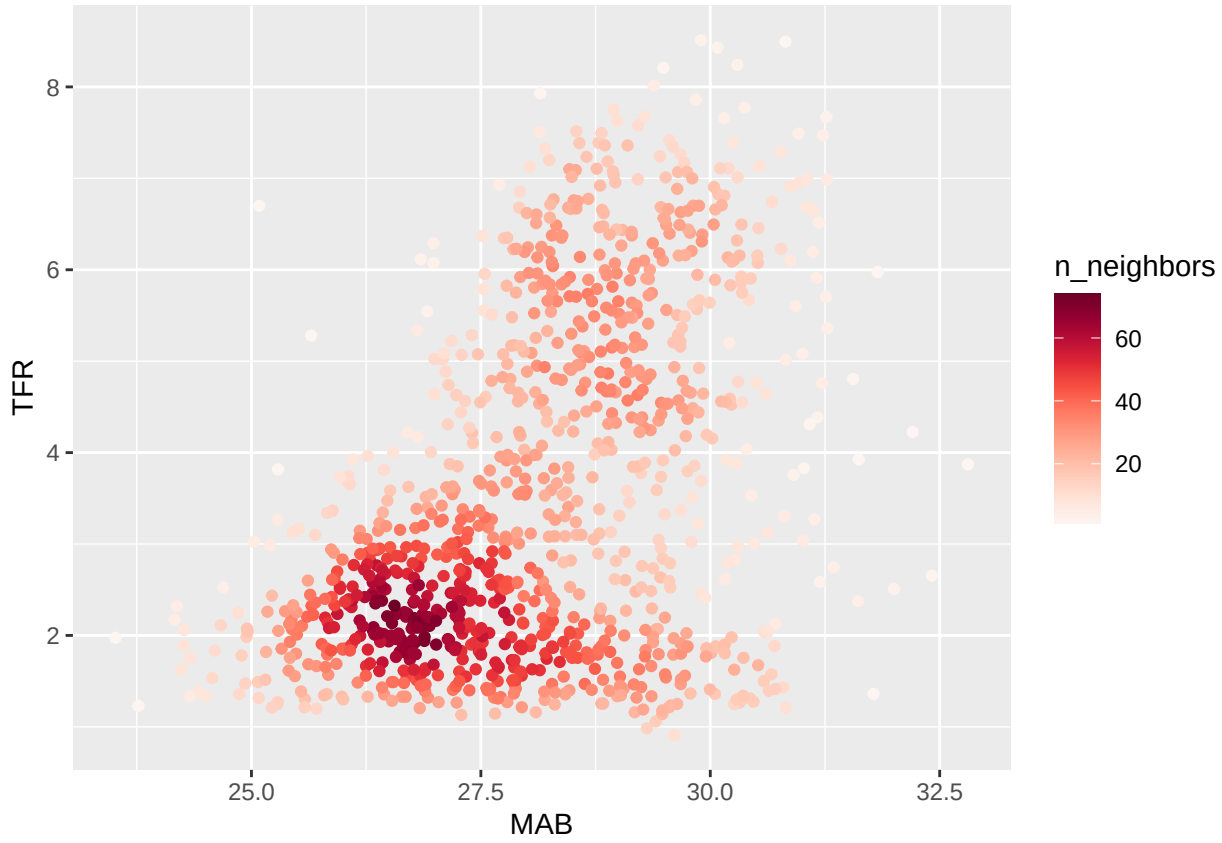
```
nrow(ASFR1) # yup it's bigger now
```

```
## [1] 36400
```

Each subset now has single ages 15-49. We don't know yet whether there were any pathological cases. Let's make a scatterplot of MAB by TFR to find out.

```
#install.packages("ggpointdensity")
library(ggpointdensity)
library(colorspace)
ASFR1 <-
  ASFR1 %>%
  group_by(Country, LocID, Year) %>%
```

```
  mutate(MAB = sum(Age * asfr) / sum(asfr),
            TFR = sum(asfr),
            .groups = "drop")
ASFR1 %>%
  filter(Age == 30) %>%
  ggplot(mapping = aes(x = MAB, y = TFR)) +
  geom_pointdensity() +
  scale_color_continuous_sequential(palette = "Reds")
```



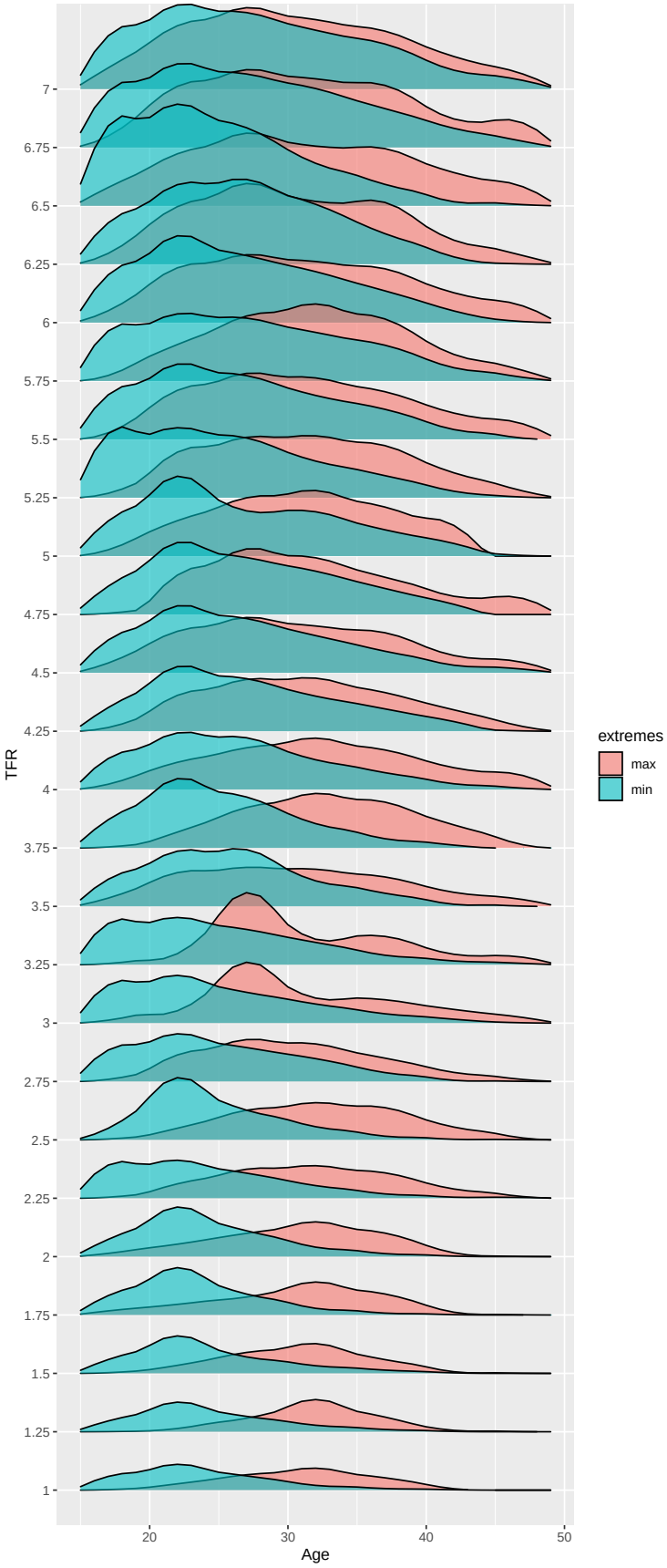
Now we start with the exercise. We bin TFRs into some intervals, let's just say .25 or so. This can happen in a `mutate()` call, and we do it by subtracting the modulo using `%%` (super awesome operator). Next step is to group by `TFRint` and pick out the minumum and maximum MAB in each interval band.

```
ASFRex <- ASFR1 %>%
 # filter(MAB > 20) %>%
  mutate(TFRint = TFR - TFR %% .25) %>%
  group_by(TFRint) %>%
  mutate(extremes = case_when(
                      MAB == max(MAB) ~ "max",
                      MAB == min(MAB) ~ "min",
                      TRUE ~ NA_character_
  )) %>%
 filter(!is.na(extremes))
```

Now this is something we can plot! Let's use the `ggridges` package to bring in the ridgeplot geom.

```r
#install.packages("ggridges")
library(ggridges)

ASFRex %>%
  filter(TFRint >= 1,
         TFRint < 7.1) %>%
  ggplot(mapping = aes(x = Age,
                y = factor(TFRint,
                                          levels = sort(unique(TFRint)),
                                          ordered = TRUE))) +
    geom_ridgeline(mapping = aes(x = Age,
                                              height = asfr,
                                              fill = extremes),
                   scale = 4,
                         alpha = .6) +
  labs(y = "TFR")
```


TFR
7
6.75
6.5
6.25
6
5.75
5.5
5.25
5
4.75
4.5
4.25
4
3.75
3.5
3.25
3
2.75
2.5
2.25
2
1.75
1.5
1.25
1
20
30
40
50
Age
extremes
max
min

We start by filtering out some of the extreme TFRs, which have few observations. Then we pass the main data object into `ggplot()`, mapping `Age` to `x` and `y` to `TFRint`. In this case, `y` means the ridge level, not the shifted height of each curve. Note since we want TFR to increase on the `y` axis, we need to set it as an ordered factor. This is a common ggplot2 trick, so not bad to see it. To draw the curves, we use the `geom_ridgeline()` geom, giving a new mapping there. Within the geom, `x` is `Age`, and `y` is `asfr`, and we finally map fill color to the extremes variable.

Looking at the results, we note a few odd cases, which could be problematic spline fits, or poor data quality. If we wanted to pursue this any further then these would require closer examination. I'm afraid some of these series might have poor quality, or else data gaps that cause the splines to behave bad, not sure. We could also try the Schmertmann or some other method. Let's chalk this up as a win for now!

References

Healy, Kieran. 2019. *Data Visualization: A Practical Introduction*. Princeton University Press.

Human Mortality Database. 2019. "University of California, Berkeley (USA) and Max Planck Institute for Demographic Research (Germany)."

Wickham, Hadley et al. 2014. "Tidy Data." *Journal of Statistical Software* 59 (10): 1–23.

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

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