BRENTON KENKEL

PRACTICAL DATA ANALYSIS FOR POLITICAL SCIENTISTS

Contents

1 About This Book

This book contains the course notes for Brenton Kenkel's course Statistics for Political Research II (PSCI 8357 at Vanderbilt University). It covers the basics of statistical modeling and programming with linear models, along with applications in R.

This book is written in R Markdown and published via Bookdown on GitHub Pages. You can find the R Markdown source files at `https://github.com/brentonk/pdaps`.

2
Principles of Programming

It may seem strange to begin a statistics class with two weeks on programming. It is strange. Here is why I have made this strange choice.

First, as a working social scientist, most of the time you spend on data analysis won't be on the *analysis* part. It'll be on obtaining and cleaning the data, to get it in a form that makes sense to analyze. Good programming skills will let you spend less time cleaning data and more time publishing papers.

Second, even if you don't want to develop good programming habits, journals are going to force you to. Every reputable political science journal requires that you provide replication scripts, and some of the best (e.g., *American Journal of Political Science*) have begun auditing the replication materials as a condition of publication. Better to learn The Right Way now when you have lots of time than to be forced to when you're writing a dissertation or on the market or teaching your own courses.

Third, while I feel embarrassed to invoke the cliché that is Big Data, that doesn't mean it's not a real thing. Political scientists have access to more data and more computing power than ever before. You can't collect, manage, clean, and analyze large quantities of data without understanding the basic principles of programming.

As Bowers (2011) puts it, "Data analysis is computer programming." By getting a PhD in political science,[1] by necessity you're going to become a computer programmer. The choice before you is whether to be a good one or a bad one.

[1] Or whatever other social science field.

Wilson et al. (2014) list eight "best practices for scientific computing." The first two encapsulate most of what you need to know:

1. Write programs for people, not computers.
2. Let the computer do the work.

2.1 *Write Programs for People, Not Computers*

The first two words here—*write programs*—are crucial. When you are doing analysis for a research project, you should be writing and running scripts, not typing commands into the R (or Stata) console. The console is ephemeral, but scripts are forever, at least if you save them.

Like the manuscripts you will write to describe your findings, your analysis scripts are a form of scientific communication. You wouldn't write a paper that is disorganized, riddled with grammatical errors, or incomprehensible to anyone besides yourself. Don't write your analysis scripts that way either.

Each script should be self-contained, ideally accomplishing one major task. Using an omnibus script that runs every bit of analysis is like writing a paper without paragraph breaks. A typical breakdown of scripts for a project of mine looks like:

- `0-download.r`: downloads the data
- `1-clean.r`: cleans the data
- `2-run.r`: runs the main analysis
- `3-figs.r`: generates figures

The exact structure varies depending on the nature of the project. Notice that the scripts are numbered in the order they should be run.

Within each script, write the code to make it as easy as possible for your reader to follow what you're doing. You should indent your code according to style conventions such as `http://adv-r.had.co.nz/Style.html`. Even better, use the `Code -> Reindent Lines` menu option in R Studio to automatically indent according to a sane style.

```r
# Bad
my_results <- c(mean(variable),
quantile(variable,
probs = 0.25),
max(variable))

# Better
my_results <- c(mean(variable),
                quantile(variable,
                         probs = 0.25),
                max(variable))
```

Another way to make your code readable—one that, unfortunately, cannot be accomplished quite so algorithmically—is to add explanatory comments. The point of comments is not to document how the language works. The following comment is an extreme example of a useless comment.

```r
# Take the square root of the errors and assign them to
# the output variable
output <- sqrt(errors)
```

A better use for the comment would be to explain *why* you're taking the square root of the errors, at least if your purpose in doing so would be unclear to a hypothetical reader of the code.

My basic heuristic for code readability is *If I got hit by a bus tomorrow, could one of my coauthors figure out what the hell I was doing and finish the paper?*

2.2 Let the Computer Do the Work

Computers are really good at structured, repetitive tasks. If you ever find yourself entering the same thing into the computer over and over again, you are Doing It Wrong. Your job as the human directing the computer is to figure out the structure that underlies the repeated task and to program the computer to do the repetition.

For example, imagine you have just run a large experiment and you want to estimate effects by subgroups.[2] Your respondents differ across four variables—party ID (R or D), gender (male or female), race (white or nonwhite), and education (college degree or not)—giving you 16 subgroups. You *could* copy and paste your code to estimate the treatment effect 16 times. But this is a bad idea for a few reasons.

[2] There could be statistical problems with this kind of analysis, at least if the subgroups were specified *post hoc*. See `https://xkcd.com/882/` ("Significant"). We're going to leave this issue aside for now, but we'll return to it later when we discuss the statistical crisis in science.

- Copy-paste doesn't scale. Copy-paste is managable (albeit misguided) for 16 iterations, but probably not for 50 and definitely not for more than 100.

- Making changes becomes painful. Suppose you decide to change how you calculate the estimate. Now you have to go back and individually edit 16 chunks of code.

- Copy-paste is error-prone, and insidiously so. If you do the calculation wrong all 16 times, you'll probably notice. But what if you screwed up for just one or two cases? Are you *really* going to go through and check that you did everything right in each individual case?

We're going to look at the most basic ways to get the computer to repeat structured tasks—functions and control flow statements. To illustrate these, we will use a result you discussed in Stat I: the central limit theorem.

The central limit theorem concerns the *sampling distribution* of the sample mean,

$$\bar{X} = \frac{1}{N} \sum_{n=1}^{N} X_n,$$

where each X_n is independent and identically distributed with mean μ and variance σ^2. Loosely speaking, the CLT says that as N grows large, the sampling distribution of $\bar{X}$ becomes approximately normal with mean μ and variance σ^2/N.

Here's what we would need to do to see the CLT in practice. We'd want to take a bunch of samples, each of size N, and calculate the sample mean of each. Then we'd have a sample of sample means, and we could check to verify that they are approximately normally distributed with mean μ and variance σ^2/N. This is a structured, repetitive task—exactly the kind of thing that should be programmed. We'll try it out with a random variable from a Poisson distribution with $\lambda = 3$, which has mean $\mu = 3$ and variance $\sigma^2 = 3$.

First things first. We can use the `rpois` function to draw a random sample of N numbers from the Poisson distribution.

```
samp <- rpois(10, lambda = 3)
samp
```

```
##  [1] 2 3 8 3 5 4 3 4 2 2
```

To calculate the sample mean, we simply use the `mean` function.

```
mean(samp)
```

```
## [1] 3.6
```

We are interested in the distribution of the sample mean across many samples like this one. To begin, we will write a **function** that automates our core task—drawing a sample of N observations from Poisson(3) and calculating the sample mean. A function consists of a set of *arguments* (the inputs) and a *body* of code specifying which calculations to perform on the inputs to produce the output.

```
pois_mean <- function(n_obs) {
    samp <- rpois(n_obs, lambda = 3)
    ans <- mean(samp)
    return(ans)
}
```

This code creates a function called `pois_mean`. It has a single argument, called `n_obs`. It generates a random sample of `n_obs` draws from Poisson(3) and calculates its sample mean. It then `return`s the sample mean as the output.

Let's try calling this function a few times, each with a sample size of $N = 30$. Your output will differ slightly from what's printed here, since the function is generating random numbers.

```
pois_mean(n_obs = 30)
```

```
## [1] 3.0333
```

```
pois_mean(n_obs = 30)
```

```
## [1] 2.4667
```

```
pois_mean(n_obs = 30)
```

```
## [1] 2.9667
```

Remember that what we're interested in is the *sampling distribution* of the sample mean—the distribution of the sample mean across every possible sample of N observations. We can approximate this distribution by running `pois_mean` many times (e.g., 1000 or more). This would be infeasible via copy-paste. Instead, we will use a **for loop**.

```
# Set up a vector to store the output
n_replicates <- 1000
sampling_dist <- rep(NA, n_replicates)

for (i in 1:n_replicates) {
    sampling_dist[i] <- pois_mean(n_obs = 30)
}
```

Here's how the for loop works. We specified `i` as the name of the index variable, with values `1:n_replicates`. The for loop takes each value in the sequence, assigns it to the variable `i`, runs the given expression (in this case, assigning the output of `pois_mean` to the `i`'th element of `sampling_dist`), and then moves on to the next value in sequence, until it reaches the end.

Let's take a look at the results and compare them to our expectations.

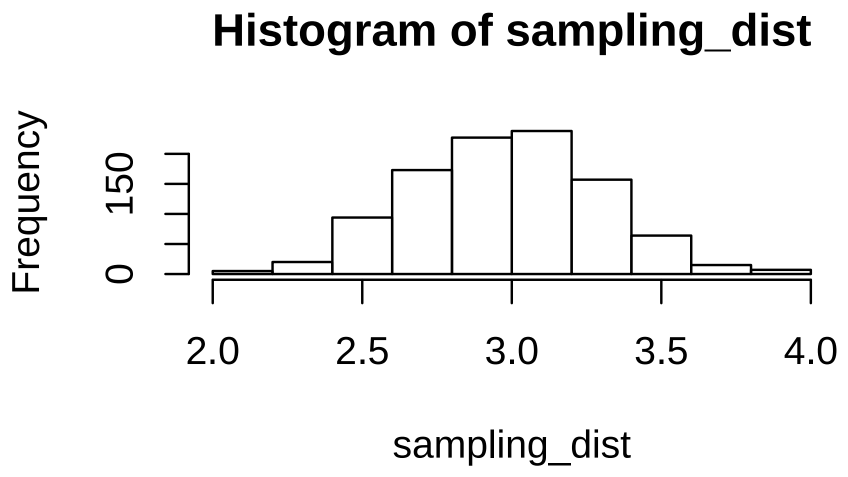
```
mean(sampling_dist)  # Expect 3
```

```
## [1] 2.9952
```

```
var(sampling_dist)  # Expect 1/10
```

```
## [1] 0.096944
```

```r
hist(sampling_dist)  # Expect roughly normal
```



For loops are fun, but don't overuse them. Many simple operations are **vectorized** and don't require a loop. For example, suppose you want to take the square of a sequence of numbers. You could use a for loop …

```r
input <- c(1, 3, 7, 29)
output <- rep(NA, length(input))

for (i in 1:length(input)) {
    output[i] <- input[i]^2
}

output
```

```
## [1]   1   9  49 841
```

… but it's faster (in terms of computational speed) and easier to just take advantage of vectorization:

```r
input^2
```

```
## [1]   1   9  49 841
```

Another useful piece of control flow is **if/else statements**. These check a logical condition—an expression whose value is `TRUE` or `FALSE`—and run different code depending on the value of the expression. (You may want to catch up on the comparison operators: `==`, `>`, `>=`, `<`, `<=`, etc.)

Let's edit the `pois_mean` function to allow us to calculate the median instead of the mean. We'll add a second argument to the function, and implement the option using an if/else statement.

```r
pois_mean <- function(n_obs, use_median = FALSE) {
    samp <- rpois(n_obs, lambda = 3)
    if (use_median) {
        ans <- median(samp)
    } else {
        ans <- mean(samp)
    }
    return(ans)
}
```

A couple of things to notice about the structure of the function. We use a comma to separate multiple function arguments. Also, we've specified `FALSE` as the *default* for the `use_median` argument. If we call the function without explicitly specifying a value for `use_median`, the function sets it to `FALSE`.

```r
pois_mean(n_obs = 9)
```

```
## [1] 3.7778
```

```r
pois_mean(n_obs = 9, use_median = TRUE)
```

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

```r
pois_mean(n_obs = 9, use_median = FALSE)
```

```
## [1] 2.6667
```

There is a vectorized version of if/else statements called, naturally, the `ifelse` function. This function takes three arguments, each a vector of the same length: (1) a logical condition, (2) an output value if the condition is `TRUE`, (3) an output value if the condition is `FALSE`.

```r
x <- 1:10
big_x <- x * 100
small_x <- x * -100

ifelse(x > 5, big_x, small_x)
```

```
##  [1] -100 -200 -300 -400 -500  600  700  800
##  [9]  900 1000
```

Functions, for loops, and if/else statements are just a few of the useful tools for programming in R.[3] But even these simple tools are enough to allow you to do much more at scale than you could with a copy-paste philosophy.

[3] Others include the `replicate` function, the `apply` family of functions (`sapply`, `lapply`, `tapply`, `mapply`, ...), the **foreach** package, the **purrr** package, just to name a few of the most useful off the top of my head.

3 Working with Data

Some material in this chapter is adapted from notes Matt DiLorenzo wrote for the Spring 2016 session of PSCI 8357.

Let me repeat something I said last week. In your careers as social scientists, starting with your dissertation research—if not earlier—you will probably spend more time collecting, merging, and cleaning data than you will on statistical analysis. So it's worth taking some time to learn how to do this well.

Best practices for data management can be summarized in a single sentence: *Record and document everything you do to the data.*

The first corollary of this principle is that raw data is sacrosanct. You should never edit raw data "in place". Once you download the raw data file, that file should never change.[1]

[1] Even if it's data you collected yourself, that data should still have a "canonical" representation that never gets overwritten. See Leek (2015) for more on distributing your own data.

In almost any non-trivial analysis, the "final" data—the format you plug into your analysis—will differ significantly from the raw data. It may consist of information merged from multiple sources. The variables may have been transformed, aggregated, or otherwise altered. The unit of observation may even differ from the original source. You must document every one of these changes, so that another researcher working from the exact same raw data will end up with the exact same final data.

The most sensible way to achieve this level of reproducibility is to do all of your data merging and cleaning in a script. In other words, no going into Excel and mucking around manually. Like any other piece of your analysis, your pipeline from raw data to final data should follow the principles of programming that we discussed last week.

Luckily for you,[2] the **tidyverse** suite of R packages (including **dplyr**, **tidyr**, and others) makes it easy to script your "data pipeline". We'll begin by loading the package.

[2] But not for me, because these tools didn't exist when I was a PhD student. Also, get off my lawn!

```r
library("tidyverse")
```

3.1 Loading

The first step in working with data is to acquire some data. Depending on the nature of your research, you will be getting some or all of your data from sources available online. When you download data from online repositories, you should keep track of where you got it from. The best way to do so is—you guessed it—to script your data acquisition.

The R function `download.file()` is the easiest way to download files from URLs from within R. Just specify where you're getting the file from and where you want it to go. For the examples today, we'll use an "untidied" version of the World Development Indicators data from the World Bank that I've posted to my website.

```
download.file(url = "http://bkenkel.com/data/untidy-data.csv",
    destfile = "my-untidy-data.csv")
```

Once you've got the file stored locally, use the utilities from the **readr** package (part of **tidyverse**) to read it into R as a data frame.[3] We have a CSV file, so we will use `read_csv`. See `help(package = "readr")` for other possibilities.

[3] More precisely, the **readr** functions produce output of class `"tbl_df"` (pronounced "tibble diff," I'm told), which are like data frames but better. See `help(package = "tibble")` for what can be done with `tbl_df`s.

```
untidy_data <- read_csv(file = "my-untidy-data.csv")
```

```
## Parsed with column specification:
## cols(
##   country = col_character(),
##   gdp.2005 = col_double(),
##   gdp.2006 = col_double(),
##   gdp.2007 = col_double(),
##   gdp.2008 = col_double(),
##   pop.2005 = col_double(),
##   pop.2006 = col_double(),
##   pop.2007 = col_double(),
##   pop.2008 = col_double(),
##   unemp.2005 = col_double(),
##   unemp.2006 = col_double(),
##   unemp.2007 = col_double(),
##   unemp.2008 = col_double()
## )
```

Remember that each column of a data frame might be a different type, or more formally *class*, of object. `read_csv` and its ilk try to guess the type of data each column contains: character, integer, decimal number ("double" in programming-speak), or something else.

The readout above tells you what guesses it made. If it gets something wrong—say, reading a column as numbers that ought to be characters—you can use the `col_types` argument to set it straight.

FYI, you could also run `read_csv()` directly on a URL, as in:

```
read_csv("http://bkenkel.com/data/untidy-data.csv")
```

However, in analyses intended for publication, it's usually preferable to download and save the raw data. What's stored at a URL might change or disappear, and you'll need to have a hard copy of the raw data for replication purposes.

Now let's take a look at the data we've just loaded in.

```
head(untidy_data)
```

```
## # A tibble: 6 × 13
##   country gdp.2005 gdp.2006 gdp.2007
##     <chr>    <dbl>    <dbl>    <dbl>
## 1      AD   3.8423   4.0184   4.0216
## 2      AE 253.9655 278.9489 287.8318
## 3      AF   9.7630  10.3052  11.7212
## 4      AG   1.1190   1.2687   1.3892
## 5      AL   9.2684   9.7718  10.3483
## 6      AM   7.6678   8.6797   9.8731
## # ... with 9 more variables: gdp.2008 <dbl>,
## #   pop.2005 <dbl>, pop.2006 <dbl>,
## #   pop.2007 <dbl>, pop.2008 <dbl>,
## #   unemp.2005 <dbl>, unemp.2006 <dbl>,
## #   unemp.2007 <dbl>, unemp.2008 <dbl>
```

We have a `country` variable giving country abbreviations. The other variables are numerical values: the country's GDP in 2005, 2006, 2007, and 2008; then the same for population and unemployment. Let's get this into a format we could use for analysis.

3.2 Tidying

Wickham (2014) outlines three qualities that make data "tidy":

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

For one thing, this means that whether a dataset is tidy or not depends—at least in part (some data collections are messy from any angle)—on the purpose it's being analyzed for.

Each row of `untidy_data` is a country. In observational studies in comparative politics and international relations, more commonly the unit of observation is the country-year.[4] How can we take `untidy_data` and easily make it into country-year data?

[4] Insert lame joke about how Americanists haven't heard of other countries. But, seriously, if you're confused because you haven't heard of other countries, just think of "state-years".

We'll use the **tidyr** package (again, part of **tidyverse**) to clean up this data. The biggest problem right now is that each column, besides the country identifier, really encodes two pieces of information: the year of observation and the variable being observed. To deal with this, we'll have to first transform the data from one untidy format to another. We're going to use the `gather()` function to make each row a country-year-variable.

What `gather()` does is make a row for each entry from a set of columns. It's probably easiest to understand it by seeing it in practice:

```
long_data <- gather(untidy_data, key = variable,
    value = number, gdp.2005:unemp.2008)
head(long_data)
```

```
## # A tibble: 6 × 3
##   country variable   number
##     <chr>    <chr>    <dbl>
## 1      AD gdp.2005   3.8423
## 2      AE gdp.2005 253.9655
## 3      AF gdp.2005   9.7630
## 4      AG gdp.2005   1.1190
## 5      AL gdp.2005   9.2684
## 6      AM gdp.2005   7.6678
```

With the first argument, we told `gather()` to use the `untidy_data` data frame. With the last argument, we told it the set of columns to "gather" together into a single column. The `key` column specifies the name of the variable to store the "key" (original column name) in, and the `value` column specifies the name of the variable to store the associated value. For example, the second row of `long_data` encodes what we previously saw as the `gdp.2005` column of `untidy_data`.

Now we have a new problem, which is that `variable` encodes two pieces of information: the variable and the year of its observation. **tidyr** provides the `separate()` function to solve that, splitting a single variable into two.

```
long_data <- separate(long_data, col = variable,
    into = c("var", "year"))
head(long_data)
```

```
## # A tibble: 6 × 4
##   country   var  year   number
##     <chr> <chr> <chr>    <dbl>
## 1      AD   gdp  2005   3.8423
## 2      AE   gdp  2005 253.9655
## 3      AF   gdp  2005   9.7630
## 4      AG   gdp  2005   1.1190
## 5      AL   gdp  2005   9.2684
## 6      AM   gdp  2005   7.6678
```

So now we have country-year-variable data, with the year and variable conveniently stored in different columns. To turn this into country-year data, we can use the `spread()` function, which is like the inverse of `gather()`. `spread()` takes a key column and a value column, and turns each different key into a column of its own.

```
clean_data <- spread(long_data, key = var, value = number)
head(clean_data)
```

```
## # A tibble: 6 × 5
##   country  year      gdp      pop unemp
##     <chr> <chr>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## 6      AE  2006 278.9489 5.171255   3.3
```

When using `spread()` on data that you didn't previously `gather()`, be sure to set the `fill` argument to tell it how to fill in empty cells. A simple example:

```
test_data
```

```
## # A tibble: 3 × 3
##        id     k     v
##     <chr> <chr> <dbl>
## 1 brenton     a    10
## 2 brenton     b    20
## 3 patrick     b     5
```

```
spread(test_data, key = k, value = v)
```

```
## # A tibble: 2 × 3
##        id     a     b
## *   <chr> <dbl> <dbl>
## 1 brenton    10    20
## 2 patrick    NA     5
```

```
spread(test_data, key = k, value = v, fill = 100)
```

```
## # A tibble: 2 × 3
##         id     a     b
## *    <chr> <dbl> <dbl>
## 1 brenton    10    20
## 2 patrick   100     5
```

One more important note on **tidyverse** semantics. It includes a fabulous feature called the *pipe*, `%>%`, which makes it easy to string together a truly mind-boggling number of commands.

In pipe syntax, `x %>% f()` is equivalent to `f(x)`. That seems like a wasteful and confusing way to write `f(x)`, and it is. But if you want to string together a bunch of commands, it's much easier to comprehend

```
x %>% f() %>% g() %>% h() %>% i()
```

than `i(h(g(f(x))))`.

You can pass function arguments using the pipe too. For example, `f(x, bear = "moose")` is equivalent to `x %>% f(bear = "moose")`.

The key thing about the **tidyverse** functions is that each of them takes a data frame as its first argument, and returns a data frame as its output. This makes them highly amenable to piping. For example, we can combine all three steps of our tidying above with a single command, thanks to the pipe:[5]

[5] If you are reading the PDF copy of these notes (i.e., the ones I hand out in class), the line breaks are eliminated, making the piped commands rather hard to read. I am working on fixing this. For now, you may find the online notes at `http://bkenkel.com/pdaps` easier to follow.

```
untidy_data %>% gather(key = variable, value = number,
    gdp.2005:unemp.2008) %>% separate(col = variable,
    into = c("var", "year")) %>% spread(key = var,
    value = number)
```

```
## # A tibble: 860 × 5
##   country  year      gdp      pop unemp
## *   <chr> <chr>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## # ... with 855 more rows
```

Without the pipe, if we wanted to run all those commands together, we would have to write:

```
spread(separate(gather(untidy_data, key = variable,
    value = number, gdp.2005:unemp.2008), col = variable,
    into = c("var", "year")), key = var, value = number)
```

Sad!

3.3 Transforming and Aggregating

Tidying the data usually isn't the end of the process. If you want to perform further calculations on the raw, that's where the tools in **dplyr** (part of, you guessed it, the **tidyverse**) come in.

Perhaps the simplest **dplyr** function (or "verb", as the R hipsters would say) is `rename()`, which lets you rename columns.

```
clean_data %>% rename(gross_domestic_product = gdp)
```

```
## # A tibble: 860 × 5
##   country  year gross_domestic_product
## *   <chr> <chr>                  <dbl>
## 1      AD  2005                 3.8423
## 2      AD  2006                 4.0184
## 3      AD  2007                 4.0216
## 4      AD  2008                 3.6759
## 5      AE  2005               253.9655
## # ... with 855 more rows, and 2 more
## #   variables: pop <dbl>, unemp <dbl>
```

The **dplyr** functions, like the vast majority of R functions, do not modify their inputs. In other words, running `rename()` on `clean_data` will return a renamed copy of `clean_data`, but won't overwrite the original.

```
clean_data
```

```
## # A tibble: 860 × 5
##   country  year      gdp      pop unemp
## *   <chr> <chr>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## # ... with 855 more rows
```

If you wanted to make the change stick, you would have to run:

```
clean_data <- clean_data %>% rename(gross_domestic_product = gdp)
```

`select()` lets you keep a couple of columns and drop all the others. Or vice versa if you use minus signs.

```
clean_data %>% select(country, gdp)
```

```
## # A tibble: 860 × 2
##   country      gdp
## *   <chr>    <dbl>
## 1      AD   3.8423
## 2      AD   4.0184
## 3      AD   4.0216
## 4      AD   3.6759
## 5      AE 253.9655
## # ... with 855 more rows
```

```
clean_data %>% select(-pop)
```

```
## # A tibble: 860 × 4
##   country  year      gdp unemp
## *   <chr> <chr>    <dbl> <dbl>
## 1      AD  2005   3.8423    NA
## 2      AD  2006   4.0184    NA
## 3      AD  2007   4.0216    NA
## 4      AD  2008   3.6759    NA
## 5      AE  2005 253.9655   3.1
## # ... with 855 more rows
```

`mutate()` lets you create new variables that are transformations of old ones.

```
clean_data %>% mutate(gdppc = gdp/pop, log_gdppc = log(gdppc))
```

```
## # A tibble: 860 × 7
##   country  year      gdp      pop unemp
##     <chr> <chr>    <dbl>    <dbl> <dbl>
## 1      AD  2005   3.8423 0.081223    NA
## 2      AD  2006   4.0184 0.083373    NA
## 3      AD  2007   4.0216 0.084878    NA
## 4      AD  2008   3.6759 0.085616    NA
## 5      AE  2005 253.9655 4.481976   3.1
## # ... with 855 more rows, and 2 more
## #   variables: gdppc <dbl>, log_gdppc <dbl>
```

`filter()` cuts down the data according to the logical condition(s) you specify.

```
clean_data %>% filter(year == 2006)
```

```
## # A tibble: 215 × 5
##   country  year      gdp       pop unemp
##     <chr> <chr>    <dbl>     <dbl> <dbl>
## 1      AD  2006   4.0184  0.083373    NA
## 2      AE  2006 278.9489  5.171255   3.3
## 3      AF  2006  10.3052 25.183615   8.8
## 4      AG  2006   1.2687  0.083467    NA
## 5      AL  2006   9.7718  2.992547  12.4
## # ... with 210 more rows
```

`summarise()` calculates summaries of the data. For example, let's find the maximum unemployment rate.

```
clean_data %>% summarise(max_unemp = max(unemp,
    na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##   max_unemp
##       <dbl>
## 1      37.6
```

This seems sort of useless, until you combine it with the `group_by()` function. If you group the data before `summarise`-ing it, you'll calculate a separate summary for each group. For example, let's calculate the maximum unemployment rate for each year in the data.

```
clean_data %>% group_by(year) %>% summarise(max_unemp = max(unemp,
    na.rm = TRUE))
```

```
## # A tibble: 4 × 2
##    year max_unemp
##   <chr>     <dbl>
## 1  2005      37.3
## 2  2006      36.0
## 3  2007      34.9
## 4  2008      37.6
```

`summarise()` produces a "smaller" data frame than the input—one row per group. If you want to do something similar, but preserving the structure of the original data, use `mutate` in combination with `group_by`.

```
clean_data %>% group_by(year) %>% mutate(max_unemp = max(unemp,
    na.rm = TRUE), unemp_over_max = unemp/max_unemp) %>%
    select(country, year, contains("unemp"))
```

```
## Source: local data frame [860 x 5]
## Groups: year [4]
##
##   country  year unemp max_unemp
##     <chr> <chr> <dbl>     <dbl>
## 1      AD  2005    NA      37.3
## 2      AD  2006    NA      36.0
## 3      AD  2007    NA      34.9
## 4      AD  2008    NA      37.6
## 5      AE  2005   3.1      37.3
## # ... with 855 more rows, and 1 more
## #   variables: unemp_over_max <dbl>
```

This gives us back the original data, but with a `max_unemp` variable recording the highest unemployment level that year. We can then calculate each individual country's unemployment as a percentage of the maximum. Whether grouped `mutate` or `summarise` is better depends, of course, on the purpose and structure of your analysis.

Notice how I selected all of the unemployment-related columns with `contains("unemp")`. See `?select_helpers` for a full list of helpful functions like this for `select`-ing variables.

3.4 Appendix: Creating the Example Data

I used the same tools this chapter introduces to create the untidy data. I may as well include the code to do it, in case it helps further illustrate how to use the **tidyverse** tools (and, as a bonus, the **WDI** package for downloading World Development Indicators data).

First I load the necessary packages.

```r
library("tidyverse")
library("WDI")
library("countrycode")
library("stringr")
```

Next, I download the relevant WDI data. I used the `WDIsearch()` function to locate the appropriate indicator names.

```r
dat_raw <- WDI(country = "all",
               indicator = c("NY.GDP.MKTP.KD",  # GDP in 2000 USD
                             "SP.POP.TOTL",     # Total population
                             "SL.UEM.TOTL.ZS"), # Unemployment rate
               start = 2005,
               end = 2008)

head(dat_raw)
```

```
##   iso2c    country year NY.GDP.MKTP.KD
## 1    1A Arab World 2005     1.6428e+12
## 2    1A Arab World 2006     1.7629e+12
## 3    1A Arab World 2007     1.8625e+12
## 4    1A Arab World 2008     1.9799e+12
## 5    1W      World 2005     5.7703e+13
## 6    1W      World 2006     6.0229e+13
##   SP.POP.TOTL SL.UEM.TOTL.ZS
## 1   313430911        12.1402
## 2   320906736        11.3296
## 3   328766559        10.8961
## 4   336886468        10.5060
## 5  6513959904         6.1593
## 6  6594722462         5.9000
```

I want to get rid of the aggregates, like the "Arab World" and "World" we see here. As a rough tack at that, I'm going to exclude those so-called countries whose ISO codes don't appear in the **countrycode** package data.[6]

[6] **countrycode** is a very useful, albeit imperfect, package for converting between different country naming/coding schemes.

```
dat_countries <- dat_raw %>% filter(iso2c %in%
    countrycode_data$iso2c)
```

Let's check on which countries are left. (I cut it down to max six characters per country name for printing purposes.)

```
dat_countries$country %>% unique() %>% str_sub(start = 1,
    end = 6)
```

```
##   [1] "Andorr" "United" "Afghan" "Antigu"
##   [5] "Albani" "Armeni" "Angola" "Argent"
##   [9] "Americ" "Austri" "Austra" "Aruba"
##  [13] "Azerba" "Bosnia" "Barbad" "Bangla"
##  [17] "Belgiu" "Burkin" "Bulgar" "Bahrai"
##  [21] "Burund" "Benin"  "Bermud" "Brunei"
##  [25] "Bolivi" "Brazil" "Bahama" "Bhutan"
##  [29] "Botswa" "Belaru" "Belize" "Canada"
##  [33] "Congo," "Centra" "Congo," "Switze"
##  [37] "Cote d" "Chile"  "Camero" "China"
##  [41] "Colomb" "Costa " "Cuba"   "Cabo V"
##  [45] "Curaca" "Cyprus" "Czech " "German"
##  [49] "Djibou" "Denmar" "Domini" "Domini"
##  [53] "Algeri" "Ecuado" "Estoni" "Egypt,"
##  [57] "Eritre" "Spain"  "Ethiop" "Finlan"
##  [61] "Fiji"   "Micron" "Faroe " "France"
##  [65] "Gabon"  "United" "Grenad" "Georgi"
##  [69] "Ghana"  "Gibral" "Greenl" "Gambia"
##  [73] "Guinea" "Equato" "Greece" "Guatem"
##  [77] "Guam"   "Guinea" "Guyana" "Hong K"
##  [81] "Hondur" "Croati" "Haiti"  "Hungar"
##  [85] "Indone" "Irelan" "Israel" "Isle o"
##  [89] "India"  "Iraq"   "Iran, " "Icelan"
##  [93] "Italy"  "Jamaic" "Jordan" "Japan"
##  [97] "Kenya"  "Kyrgyz" "Cambod" "Kiriba"
## [101] "Comoro" "St. Ki" "Korea," "Korea,"
## [105] "Kuwait" "Cayman" "Kazakh" "Lao PD"
## [109] "Lebano" "St. Lu" "Liecht" "Sri La"
## [113] "Liberi" "Lesoth" "Lithua" "Luxemb"
## [117] "Latvia" "Libya"  "Morocc" "Monaco"
## [121] "Moldov" "Monten" "St. Ma" "Madaga"
## [125] "Marsha" "Macedo" "Mali"   "Myanma"
## [129] "Mongol" "Macao " "Northe" "Maurit"
## [133] "Malta"  "Maurit" "Maldiv" "Malawi"
## [137] "Mexico" "Malays" "Mozamb" "Namibi"
## [141] "New Ca" "Niger"  "Nigeri" "Nicara"
## [145] "Nether" "Norway" "Nepal"  "Nauru"
## [149] "New Ze" "Oman"   "Panama" "Peru"
## [153] "French" "Papua " "Philip" "Pakist"
## [157] "Poland" "Puerto" "West B" "Portug"
## [161] "Palau"  "Paragu" "Qatar"  "Romani"
## [165] "Serbia" "Russia" "Rwanda" "Saudi "
## [169] "Solomo" "Seyche" "Sudan"  "Sweden"
## [173] "Singap" "Sloven" "Slovak" "Sierra"
## [177] "San Ma" "Senega" "Somali" "Surina"
## [181] "South " "Sao To" "El Sal" "Sint M"
## [185] "Syrian" "Swazil" "Turks " "Chad"
## [189] "Togo"   "Thaila" "Tajiki" "Timor-"
## [193] "Turkme" "Tunisi" "Tonga"  "Turkey"
## [197] "Trinid" "Tuvalu" "Tanzan" "Ukrain"
## [201] "Uganda" "United" "Urugua" "Uzbeki"
## [205] "St. Vi" "Venezu" "Britis" "Virgin"
## [209] "Vietna" "Vanuat" "Samoa"  "Yemen,"
## [213] "South " "Zambia" "Zimbab"
```

With that out of the way, there's still some cleaning up to do. The magnitudes of GDP and population are too large, and the variable names are impenetrable. Also, the `country` variable, while helpful, is redundant now that we're satisfied with the list of countries remaining.

```r
dat_countries <- dat_countries %>% select(-country) %>%
    rename(gdp = NY.GDP.MKTP.KD, pop = SP.POP.TOTL,
        unemp = SL.UEM.TOTL.ZS, country = iso2c) %>%
    mutate(gdp = gdp/1e+09, pop = pop/1e+06)

head(dat_countries)
```

```
##   country year      gdp      pop unemp
## 1      AD 2005   3.8423 0.081223    NA
## 2      AD 2006   4.0184 0.083373    NA
## 3      AD 2007   4.0216 0.084878    NA
## 4      AD 2008   3.6759 0.085616    NA
## 5      AE 2005 253.9655 4.481976   3.1
## 6      AE 2006 278.9489 5.171255   3.3
```

Now I convert the data to “long” format.

```
dat_countries_long <- dat_countries %>% gather(key = variable,
    value = value, gdp:unemp)

head(dat_countries_long)
```

```
##   country year variable    value
## 1      AD 2005      gdp   3.8423
## 2      AD 2006      gdp   4.0184
## 3      AD 2007      gdp   4.0216
## 4      AD 2008      gdp   3.6759
## 5      AE 2005      gdp 253.9655
## 6      AE 2006      gdp 278.9489
```

I then smush `variable` and `year` into a single column, and drop the individual components.

```
dat_countries_long <- dat_countries_long %>% mutate(var_year = paste(variable,
    year, sep = ".")) %>% select(-variable, -year)

head(dat_countries_long)
```

```
##   country    value var_year
## 1      AD   3.8423 gdp.2005
## 2      AD   4.0184 gdp.2006
## 3      AD   4.0216 gdp.2007
## 4      AD   3.6759 gdp.2008
## 5      AE 253.9655 gdp.2005
## 6      AE 278.9489 gdp.2006
```

Finally, I “widen” the data, so that each `var_year` is a column of its own.

```
dat_countries_wide <- dat_countries_long %>% spread(key = var_year,
    value = value)

head(dat_countries_wide)
```

```
##   country gdp.2005 gdp.2006 gdp.2007
## 1      AD   3.8423   4.0184   4.0216
## 2      AE 253.9655 278.9489 287.8318
## 3      AF   9.7630  10.3052  11.7212
## 4      AG   1.1190   1.2687   1.3892
## 5      AL   9.2684   9.7718  10.3483
## 6      AM   7.6678   8.6797   9.8731
##   gdp.2008  pop.2005  pop.2006  pop.2007
## 1   3.6759  0.081223  0.083373  0.084878
## 2 297.0189  4.481976  5.171255  6.010100
## 3  12.1445 24.399948 25.183615 25.877544
## 4   1.3902  0.082565  0.083467  0.084397
## 5  11.1275  3.011487  2.992547  2.970017
## 6  10.5544  3.014917  3.002161  2.988117
##    pop.2008 unemp.2005 unemp.2006 unemp.2007
## 1  0.085616         NA         NA         NA
## 2  6.900142        3.1        3.3        3.4
## 3 26.528741        8.5        8.8        8.4
## 4  0.085350         NA         NA         NA
## 5  2.947314       12.5       12.4       13.5
## 6  2.975029       27.8       28.6       28.4
##   unemp.2008
## 1         NA
## 2        4.0
## 3        8.9
## 4         NA
## 5       13.0
## 6       16.4
```

Now we have some ugly data. I save the output to upload to my website.

```
write_csv(dat_countries_wide, path = "untidy-data.csv")
```

4
Bibliography

Bowers, J. (2011). Six Steps to a Better Relationship with Your Future Self. *The Political Methodologist*, 18(2):2–8.

Leek, J. (2015). *The Elements of Data Analytic Style*. Leanpub.

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, 59(10).

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., and Wilson, P. (2014). Best Practices for Scientific Computing. *PLOS Biology*, 12(1):e1001745.