How to do stuff in R and Stata

John Gardner

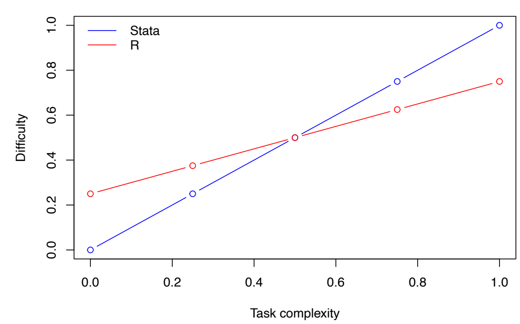
Introduction

This guide serves two purposes. First, it gives a quick introduction to common data management and econometric estimation tasks in both R and Stata. It's terse, and because those languages are complex, it isn't meant to be comprehensive (readers should consult the resources at the end for more detail). However, it does provide some examples of some intermediate-level programming techniques that are useful for applied projects. Second, it provides something of a two-way crosswalk between R and Stata, for users of one who need to work in, or are contemplating switching to, the other.

There are always multiple ways to do things, and I have tried to illustrate a variety of techniques. For R, I have tried to show how to use base R as well as the Tidyverse (a set of packages that provides alternative ways of working in R), but there are other tools, and other techniques within these tools.

My focus here is on data management and pre-programmed estimation routines, so there is no discussion of simulation techniques or programming custom estimators. Readers should be aware that both R and Stata can handle those things as well.

Should I use Stata or R? This is a subject of ongoing, and in my opinion mostly silly, debate. Stata and R are both really good, and almost anything can be done using either. In general, Stata is more user friendly and R is more flexible (see the graph on the right), but it's a close race.[1] I recommend learning a bit of both and using whatever works best for you. I tend to use Stata for more straightforward analyses and R for custom work, but this is mostly a matter of taste—I could also do everything in one or the other.



Some basic programming advice In Stata, 90% of programming comes down to knowing how to use for loops and local macros. Although loops are also useful in R, often the most effective way to do something is to supply a vector to a function.[2] If you haven't used R much, this might not make a lot of sense right now, but it is illustrated in some of the examples below.

Getting some practice This primer is accompanied by sample data, Stata and R files that implement all of the examples given below (as well as the R file used to create the sample datasets).

[1] According to its creator, Stata's syntax was designed (in pre-internet days) so that you could use it without carrying around the manual.

[2] I have seen several R experts note that it's perfectly fine to use loops in R, despite the general perception to the contrary in online help forums.

Preliminaries

R

Download R from https://www.r-project.org and R Studio from https://posit.co/download/rstudio-desktop/

You should primarily work with R by creating R script files (which R Studio is helpful for)

You can run an R script using the “run” command in R Studio

starts a line comment

You can change the working directory using

`setwd("path/to/file")`

Remove an object: `rm(object)`

Get help for a command: `help(commandname)`

Packages extend R’s functionality. Install them using

`install.packages("packagename")`

To use a package, type

`library(packagename)`

Note that you put quotes around `packagename` when installing but not when loading. Many of the examples in this guide will use the Tidyverse set of packages (which I assume you have loaded using `library(tidyverse)`

You can also use a command from a package without explicitly loading the package by typing `packagename::commandname`

You can save all of the output from your R script using `sink("filename.txt")`. This isn’t necessary if you save all of your results as objects, because then you can save your entire workspace and refer back to them (see the Data import/export section below)

Stata

Stata comes with a built-in “do file” editor, which you should use rather than entering commands interactively

You can run a do file by typing `do filename.do` in the command window, or by clicking on the “do” button the the do file editor

You can change the working directory using

`cd "path/to/file"`

By default, Stata output fills the screen, then stops until the user hits enter to advance through the results. You can turn this off using `set more off`

Stata allows you to keep a log of all of the inputs and results from running your do file:

```
capture log close
capture log using filename.log, text replace
<commands here>
capture log close
```

Here, we close the log if one is currently running, start a new log saved as `filename.log` in text format, replacing it if there is already one with that name, then close the log at the end of the file

The `capture` prefix tells Stata to keep running even if it encounters an error (e.g., if you try to close a log but there isn’t one open)

* and // are single-line comments. ‘`/* comment here */` is a multiple-line comment. /// continues a command on the next line

You can get help for a command using `help commandname`

User-written modules extend Stata’s capabilities. Install them using

`ssc install packagename`

If that doesn’t work, try

`findit packagename`

Data import/export

R

Open an RData file:

```
load("filename.RData")
```

Read a CSV file (can also use `read_csv` from the Haven package, but some packages work better with the built-in format)

```
dat <- read.csv("filename.csv")
```

Read an Excel file:

```
library(readxl)
dat <- read_excel("filename.xls",
  sheet="sheetname")
```

Read a Stata file (also see the foreign package):

```
library(haven)
dat <- read_dta("filename.dta")
```

Save all objects in workspace to an R file:

```
save.image("filename.RData")
```

Save a CSV file:

```
write.csv(dat, "filename.csv")
```

Drop/keep variables (several ways)

```
dat <- dat[c("y1", "x1", "z")]
dat[, c("y1", "x1", "z")]
subset(dat, select=c(y1, x1, z))
dat[!(names(dat) %in% c("x1", "x2"))]
dat[-c(5,7)]
subset(dat, select=-c(x1, x2))
dat[!startsWith(names(dat),"x")]
```

I only used `dat <-` to actually reassign the data in the first example. `c()` is the concatenate function, which makes a vector of its arguments

Tidyverse:

```
dat |> select(!c("x1", "x2"))
dat |> select(c("y1", "x1"))
dat |> select(!starts_with("x"))
```

"|>" *pipes* data into commands without requiring you to specify it within the command (also works in base R)

Subset data (two ways)

```
dat[dat$y1>=0 & dat$z!=2,]
subset(dat, y1 >= 0 & z!=2)
```

Tidyverse:

```
dat |> filter(y1 >= 0 & z!=2)
```

Rename variables:

```
names(dat)[names(dat)=="y1"] <- "why1"
names(dat)[6] <- "why1"
names(dat)[names(dat) %in% c("y1", "x1")] <-
  c("why1", "ecks1")
```

Tidyverse:

```
dat |> rename(why1 = y1, ecks1 = x1)
```

Stata

Clear existing data and open a Stata file:

```
clear
use "filename.dta"
```

Can use `use filename.dta, clear`, which will only clear existing data (using `clear all` will clear existing data and user-written programs)

Read a Stata file:

```
use "filename.dta"
```

Read a CSV file:

```
insheet using "filename.csv", comma
```

Read an Excel file:

```
import excel "filename.xlsx", ///
  sheet("sheetname")
```

Save a Stata file:

```
save "filename.dta", replace
```

Export a CSV file:

```
export delimited using "filename.csv", replace
```

Drop/keep variables:

```
drop x1 x2
keep y1 x1 z
drop x*
```

Subset data:

```
keep if y1 >= 0 & z!=2
drop if y1 < 0 | z==2
```

Rename:

```
rename y1 why1
rename (y1 x1) (why1 ecks1)
```

Recode variables

R

Recode variable according to condition:

```
dat$x1[dat$x1>0] <- 1
```

Tidyverse:

```
dat |> mutate(x1 = case_when(x1>0 ~ 1)
```

Do this across multiple variables (using a loop):

```
vars <- c("x1", "y1")
for (i in vars) {
  dat[dat[i]>0, i] <- 1
}
```

A non-looped version is:

```
dat[vars][dat[vars]>0] <- 1
```

Here, `dat[vars]` selects a subset of columns from `dat`, and `[dat[vars]>0]` selects the nonnegative rows of that subset...

Tidyverse:

```
dat |> mutate(across(c(x1, y1),
  ~ case_when(.>0 ~ 1)))
```

Transform a variable:

```
dat$x1sq <- dat$x1^2
```

Tidyverse:

```
dat |> mutate(x1sq = x1^2)
```

Do this across multiple variables (loop version):

```
for (i in seq_along(vars)) {
  dat[, paste0(vars[i], "sq")] <-
    dat[vars[i]]^2
}
```

`seq_along` goes through all values of a vector (we could just loop through `vars` directly, as in our recoding example)

Vector version:

```
dat[paste0(c("x1", "y1"),"sq")] <-
  dat[c("x1", "y1")]^2
```

Tidyverse:

```
dat |> mutate(across(c(x1,y1),
  .fns=list(sq=~.x^2),
  .names="{col}_{fn}"))
```

Can replace `c(x,y)` with `num_range("x", 1:2)` to select variables `x1` and `x2`, or with `starts_with("x")` to select all variables that start with "x"

Stata

Recode variable according to condition:

```
replace x1 = 1 if x1>0 & !mi(x1)
```

Since Stata stores missing values as large numbers, we use `!mi(x1)` to avoid coding missing values as 1

Do this across multiple variables:

```
foreach x of varlist x1 y1 {
  replace `x' = 1 if `x'>0 & !mi(`x')
}
```

Note that when we *dereference* the macro, we use different tick marks on the left and right of `` `x' ``

An alternative syntax saves the varlist in a *local macro*:

```
local vars x1 y1
foreach x of varlist `vars' {
  replace `x'=1 if `x'>0 & !mi(`x')
}
```

You could use `` foreach x in `vars' `` instead

Transform a variable:

```
gen x1sq = x1^2
```

Do this across multiple variables:

```
foreach x of varlist x1 y1 {
  gen `x'sq = `x'^2
}
```

Can use `foreach x of varlist x*` to select all variables that start with 'x'

Recode numerically named variables:

```
forvalues i=1/2 {
  gen x`i'_sq = x`i'^2
}
```

Merge and reshape data

R

Merge variables from `new.dat` to `dat` according to their value of `varname`:

```
merge(dat, new.dat, by="id")
```

Tidyverse:

```
dat |> left_join(new.dat, join_by(id))
```

You can merge on multiple variables using `by=c("v1", "v2")` in the base R case or `join_by(v1, v2)` in the Tidyverse case

Reshape from wide to long:

```
long1 <- reshape(dat, idvar="id",
  varying=c("x1", "x2", "y1", "y2"),
  sep="", direction="long")
```

Tidyverse:

```
long2 <- dat |> pivot_longer(
  cols=c("x1", "x2", "y1", "y2"),
  names_to=c(".value", "year"),
  names_pattern = "([A-Za-z]+)(\\d+)")
```

Reshape from long to wide:

```
reshape(long1, idvar="id", timevar="time",
  direction="wide", v.names=c("x", "y"))
```

Tidyverse:

```
pivot_wider(long2, names_from = "year",
  values_from=c("x", "y"),
  names_glue ="{.value}_{year}")
```

Stata

Merge variables from `new_dat.dta` according to their value of `varname`:

```
merge m:1 id using new_dat
```

Use `merge m:1 v1 v2 using new_dat` to merge on multiple variables

Reshape from wide to long:

```
reshape long x y, i(id) j(year)
```

Reshape from long to wide:

```
reshape wide x y, i(id) j(year)
```

Long and wide-form datasets

A long-form dataset looks like:

id	year	x
1	1	2
1	2	4

A wide-form dataset looks like:

id	year	x1	x2
1	1	2	4

Aggregate data

R

Get average of y1 by groups defined by the variables v1 and v2:

```
aggregate(y1 ~ v1 + v2, dat, mean)
```

Tidyverse:

```
dat |>
  group_by(v1, v2) |>
  summarize(mean(y1))
```

Aggregate over multiple variables:

```
aggregate(cbind(y1, y2) ~ v1 + v2, dat, mean)
```

cbind ("column bind") allows you use multiple variables on the left side of formula objects

Tidyverse:

```
dat |> group_by(v1, v2) |>
  summarize(across(c(y1, y2), mean))
```

Instead of means, you can also produce standard deviations, observation counts, and other statistics

To get the mean and median (base/Tidyverse):

```
aggregate(cbind(y1, y2) ~ v1 + v2, dat,
  \(x) c(mean = mean(x, na.rm=TRUE),
  median = median(x, na.rm=TRUE)))
dat |> group_by(v1, v2) |>
  summarize(across(c(y1, y2),
  list(mean = \(x) mean(x, na.rm=TRUE),
  median = \(x) median(x, na.rm=TRUE))),
  .names="{col}_{fn}")
```

`\(x) mean(x, na.rm=TRUE)` defines an *anonymous* function named mean. The na.rm=TRUE option removes missing values before taking the mean

To immediately add the aggregated variables to your dataset, you can use

```
for (i in c("y1", "y2")) {
  dat[paste0(i,"_mean")] <-
  ave(dat[[i]], dat$z, FUN=mean)
}
```

We use [[i]] to ensure that the input to ave is a vector

A vectorized version is

```
dat[paste0(c("y1","y2"), "_mean")] <-
  lapply(dat[c("y1", "y2")],
  \(x) ave(x, dat$z, FUN=mean))
```

lapply (list apply) applies a function to every element of a list/vector (basically a compact loop)

Tidyverse:

```
dat |> group_by(z) |> mutate(across(c(y1, y2),
  list(mean = \(x) mean(x, na.rm=TRUE)),
  .names="{col}_{fn}"))
```

Stata

Get average of y1 by groups defined by the variables v1 and v2:

```
collapse (mean) y1 y2, by(v1 v2)
```

To retain the original data, use

```
preserve
collapse (mean) y1, by(v1 v2)
save newdata, replace
restore
```

which saves the means as newdata.dta and restores the original data

Alternatively, you can use

```
egen mean_y1 = mean(y1), by(v1 v2)
```

which will add the mean to the existing data under the name mean_y1

Instead of means, you can also produce standard deviations, observation counts, and other statistics

To get means and medians, you can use

```
collapse (mean) mean_y1=y1 mean_y2=y2 ///
  (median) med_y1=y1 med_y2=y2, by(v1 v2)
```

A little loop/macro trick saves you from having to name all of the variables manually:

```
local mean (mean)
local median (median)
local vars y1 y2
foreach x of local vars {
  local mean `mean' mean_`x'=`x'
  local median `median' median_`x'=`x'
}
collapse `mean' `median'
```

Basic statistics

R

Summary statistics (using modelsummary package):

```
modelsummary::datasummary_skim(dat)
```

The datasummary command provides more customizable tables, and there are several other packages that create summary statistics (see, e.g., vtable::st and psych::describe)

Summarize select variables:

```
datasummary_skim(dat[c("y1", "y2")])
```

Summary statistics by group:

```
datasummary_skim(by="z", data=dat)
```

Table of v2 by values of z (1 gives row percentages, 2 gives column percentages):

```
prop.table(xtabs(~ v2 + z, dat), 1)
prop.table(table(dat$v2, dat$z), 1)
```

Basic scatter plot:

```
plot(df$x1, df$y1)
```

Tidyverse:

```
ggplot(df, aes(x=x1, y=y1)) + geom_point()
```

Basic histogram:

```
hist(df$x1)
```

Tidyverse:

```
ggplot(df, aes(x=x1)) + geom_histogram()
```

Base R and ggplot graphics are highly customizable (see the resources at the end for more on graphics commands)

Stata

Summary statistics

```
summarize
```

Summarize select variables:

```
sum y1 y2
```

Summary statistics by group:

```
bysort z: sum
```

You can get more complex summary statistics using the table command

Get a table of v2 by values of z (row and col give row and column percentages):

```
tab v2 z, row col
```

Basic scatterplot:

```
scatter y1 x1
```

Basic histogram:

```
hist y1
```

An easy way to learn advanced graphing commands is to explore the "Graphics" menu, which will also tell you the do-file syntax

Basic econometrics

R

Run a regression:

```
model1 <- lm(y1 ~ x1 + x2, data=dat)
summary(model1)
```

Run a regression in the subsample where `z` is 1:

```
model2 <- lm(y1 ~ x1 + x2, data=dat[dat$z==1,])
```

Include dummies for every level of a categorical variable `z`:

```
model3 <- lm(y1 ~ x1 + x2 + as.factor(z),
  data=dat)
```

Include a polynomial term in `x1`:

```
model4 <- lm(y1 ~ x1 + I(x1^2) + x2, data=dat)
```

Robust standard errors:

```
library(sandwich)
library(lmtest)
coeftest(model1,
  vcov = vcovHC(model1, type="HC1"))
```

Clustered standard errors:

```
coeftest(model1, vcov=vcovCL(model1,
  cluster = ~z))
```

Run a fixed-effects regression (use the `clubSandwich` package for clustered SEs at other levels, or see below)

```
model5 <- plm(y ~ x, data=long1,
  model="within", index=c("id"))
coeftest(model5, vcov=vcovHC(model5,
  type="HC2", cluster="group"))
```

Potentially faster, with automatic clustering:

```
library(fixest)
model5b <- feols(y ~ x | id,
  cluster=~id, data=long2)
```

Run an IV regression, instrumenting for `x1` with `z`:

```
library(ivreg)
model6 <- ivreg(y1 ~ x1 + x2 | z + x2,
  data=dat)
```

Run a Probit model with (average) marginal effects (also see the `marginaleffects` package)

```
library(margins)
dat$bin <- (dat$y1 > 0)
model7 <- glm(bin ~ x1 + x2,
  family=binomial(link=probit),
  data=dat)
probit_margins <- margins(model7)
summary(probit_margins)
```

Replacing `link=probit` with `link=logit` produces a logit instead

Stata

Run a regression:

```
reg y1 x1 x2
```

Run a regression in the subsample where `z` is 1:

```
reg y1 x1 x2 if z==1
```

Include dummies for every level of a categorical variable `z`:

```
reg y1 x1 x2 i.z
```

Include a polynomial term in `x1`:

```
reg y1 c.x1##c.x1 x2
```

Robust standard errors:

```
reg y1 x1 x2, robust
```

Clustered standard errors:

```
reg y1 x1 x2, vce(cluster id)
```

Run a fixed-effects regression (assuming long form):

```
xtset id year
xtreg y x, fe vce(cluster id)
areg y x, absorb(id) vce(cluster id)
reghdfe y x, absorb(id) vce(cluster id)
```

`areg` doesn't require `xtset`ing the data, `reghdfe` (an external module) is potentially faster with many FEs

Run an IV regression, instrumenting for `x1` with `z`:

```
ivregress 2sls y1 (x1 = z) x2, first
```

The `first` option requests the first-stage regression. To get more regression diagnostics, use the `ivreg2` package

Run a Probit model with (average) marginal effects

```
gen bin = (y1>0)
probit bin x1 x2
margins, dydx(*)
```

Replacing `probit` with `logit` produces a logit instead

Post-estimation

R

Test linear hypotheses:

```
library(car)
linearHypothesis(model1,
  c("x1 = 0", "x2 = 0"))
linearHypothesis(model1, c("x1 = x2"))
```

Residuals:

```
model1$residuals
```

Predicted values:

```
model1$fitted.values
predict(model1)
```

Extract coefficients:

```
model1$coefficients
coef(model1)
```

Extract standard errors:

```
sqrt(diag(vcov(model1)))
summary(model1)$coefficients[,2]
```

`vcov(model1)` gets the variance-covariance matrix for the model and `diag` extracts the diagonal elements

Export a table of regressions:

```
library(modelsummary)
modelsummary(list(model1, model2, model3))
```

Can use `output=tex` to get LaTeX code and `output=data.frame` to save as a dataframe, which can be saved as a CSV file. There are several other packages that offer this functionality (see `texreg` and `stargazer`)

Plot regression coefficients (using `modelsummary` package)

```
modelplot(model3, coef_omit="Interc.")
```

Stata

Test linear hypotheses:

```
test x1 x2
test x1 = x2
```

Residuals:

```
predict theresiduals, res
```

Predictions:

```
predict thepredictions, xb
```

Coefficients and standard errors can be accessed using `_b[x1]` and `_se[x1]`, as in:

```
di _b[x1]
di _se[x1]
```

(`di` is short for `display`)

They are also stored as a matrix which can be retrieved using

```
matrix b = e(b)
matrix v = e(V)
```

You can see all the macros and matrices stored after an estimation command (such as `reg`) using `ereturn list` or after a non-estimation command (such as `summary`) using `return list`

Export a table of regressions:

```
reg y1 x1
outreg2 using output.txt, replace
reg y1 x1 x2
outreg2 using output.txt
```

By default, this produces a tab-delimited text file that can be pasted into spreadsheet programs. Adding the `tex` option produces LaTeX instead. There are other packages with this functionality (see `estout` for one)

Plot regression coefficients (using `coefplot` package):

```
reg y1 x1 x2
coefplot, drop(_cons)
```

"Programmery" stuff

R

Run a regression for every value of a variable:

```
models <- list()
for (i in as.factor(unique(dat$z))) {
  models[[i]] <- lm(y1 ~ x1 + x2,
    dat[dat$z==i,])
  print(summary(models[[i]]))
}
```

`unique` returns the possible values of `z` and `as.factor` indexes our list by name (rather than number). Note that we refer to elements of a list using `models[[i]]` instead of single brackets

We can also do this by defining a function and passing it to `lapply`:

```
subreg <- function(val) {
  lm(y1 ~ x1 + x2, dat[dat$z==val,])
}
models <- lapply(unique(dat$z), subreg)
lapply(models, summary)
```

We could call this function inside a loop instead of `lapply`

Alternatively, we can define an anonymous function:

```
models <- lapply(unique(dat$z),
  \(i) lm(y1 ~ x1 + x2, dat[dat$z==i,]))
```

For longer functions, we can enclose the contents in braces

We could also do this using the `by` function:

```
by(dat, dat$z, \(df) summary(
  lm(y1 ~ x1 + x2, df)))
```

This runs the function for all of the levels of the variable `dat$z` (we could also use `tapply` instead of `by`)

One Tidyverse-style approach (with its base-R analog) is:

```
dat |> split(dat$z) |>
  map(\(df) lm(y1 ~ x1, data=df) |> map(summary)
lapply(split(dat, dat$z),
  \(df) summary(lm(y1 ~ x1, data=df)))
```

Run regressions for different dependent variables (three ways ways)

```
deps <- c("y1", "y2")
models <- list()
for (i in seq_along(deps)) {
  models[[i]] <-
  lm(paste0(deps[i], "~ x1 + x2"), data=dat)
}
lapply(
  paste0(deps, "~ x1 + x2"), lm, data=dat
)
lm(cbind(y1, y2) ~ x1 + x2, dat)
```

Run regressions with different sets of independent variables:

```
indeps <- c("x1", "x1 + x2")
models <- lapply(
  paste0("y ~", indeps), lm, data=dat
)
```

Stata

Run a regression for every value of a variable:

```
bysort z: reg y1 x1
```

Can also do this with a loop (useful if you want to do something after each regression)

```
forvalues i=1/3 {
  reg y1 x1 if z==`i'
}
```

If we didn't know the values `z` took, we could use

```
levelsof z, local(zval)
foreach i of local zval {
  reg y1 x1 if z==`i'
}
```

You could also define a quick program (the `eclass` part is unnecessary but useful for simulation/bootstrapping; `1` denotes the first argument):

```
program define myreg, eclass
  reg y1 x1 if z==`1'
end
foreach i of numlist 1/3 {
  myreg `i'
}
```

Run regressions for different dependent variables:

```
local deps y1 y2
foreach y of varlist `deps' {
  reg `y' x1 x2
}
```

Run regressions with different sets of independent variables:

```
local ind1 x1
local ind2 x2
reg y1 `ind1'
reg y1 `ind1' `ind2'
```

A way of doing this with loops:

```
local regressors `" "x1"  "x1 x2" "'
local length: word count `regressors'
forvalues i=1/`length' {
  local current: word `i' of `regressors'
  reg y1 `current'
}
```

Here we use "compound quotes" `` `""' `` to allow quotation marks within our macro

Basic matrix algebra

R

Create a matrix:

```
A <- matrix(c(1,2,3,4), nrow=2, byrow=TRUE)
```

Subset a matrix:

```
A[1, 1:2]
A[, 2]
```

Transpose a matrix:

```
A.transpose <- t(A)
```

Invert a matrix:

```
A.inverse <- solve(A)
```

Multiply matrices

```
identity <- A.inverse %*% A
```

Turn a set of variables into a matrix:

```
X <- cbind(dat$x1, dat$x2)
```

Stata

Create a matrix:

```
mat A = (1, 2 \ 3, 4)
```

Subset a matrix:

```
mat B = A[1, 1..2]
mat C = A[1..., 2]
```

Transpose a matrix:

```
mat A_transpose = A'
```

Invert a matrix:

```
mat A_inverse = inv(A)
```

Multiply matrices:

```
mat identity = A_inverse * A
```

Print a matrix:

```
matrix list A
```

Can type `mat li` for short

Turn a set of variables into a matrix:

```
mkmat x1 x2, matrix(X)
```

Efficiently create the cross product $X'X$ between matrices or $y'X$ between a matrix and a vector (these automatically include constant terms):

```
matrix accum XX = x1 x2
matrix vecaccum yX = y x1 x2
```

These use Stata's basic matrix capabilities. There is also a more capable matrix language called Mata (that works differently)

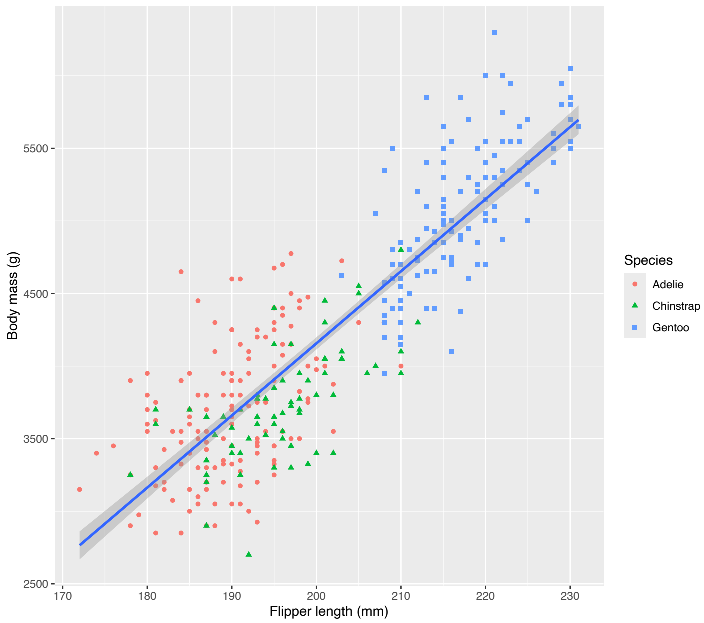
Data visualization

Data visualization just means nice graphs. All of the examples in this section are adapted from *R for Data Science*, and use the data from the `palmerpenguins` library. For the R examples, I used the default ggplot theme. For the Stata examples, I used the user-written `white_tableau` scheme, but otherwise stuck to the defaults (some of the following can be done more simply with more recent versions of Stata, but the techniques and schemes used here should work for somewhat older versions as well).

R

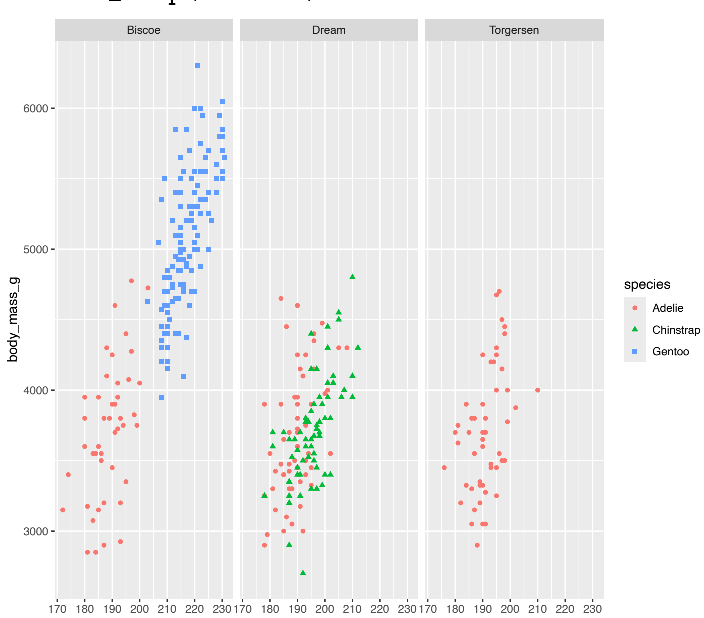
Graph 1:

```
ggplot(data = penguins,
  mapping = aes(x = flipper_length_mm,
    y = body_mass_g) +
  geom_point(aes(color = species,
    shape = species)) +
  geom_smooth(method = "lm") +
  labs(x = "Flipper length (mm)",
    y = "Body mass (g)", color = "Species",
    shape = "Species")
```



Graph 2:

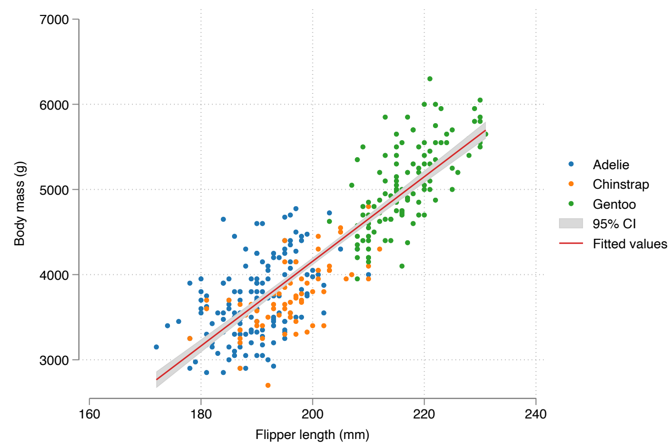
```
ggplot(penguins, aes(x = flipper_length_mm,
  y = body_mass_g)) +
  geom_point(aes(color = species,
    shape = species)) +
  facet_wrap(~island)
```



Stata

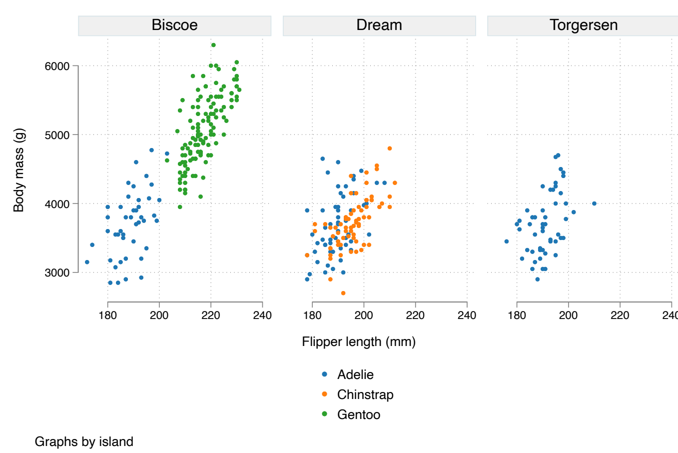
Graph 1:

```
insheet using penguins.csv, comma clear
destring body_mass_g flipper_length_mm, ///
  force replace
levelsof species, local(species)
foreach x of local species {
  gen bmg_‘x’ = body_mass_g if species=="‘x’"
  label variable bmg_‘x’ "‘x’"
}
tw (scatter bmg_* flipper_length_mm) ///
  (lfitci body_mass_g flipper_length_mm),  ///
  scheme(white_tableau) ///
  xtitle("Flipper length (mm)") ///
  ytitle("Body mass (g)")
```



Graph 2:

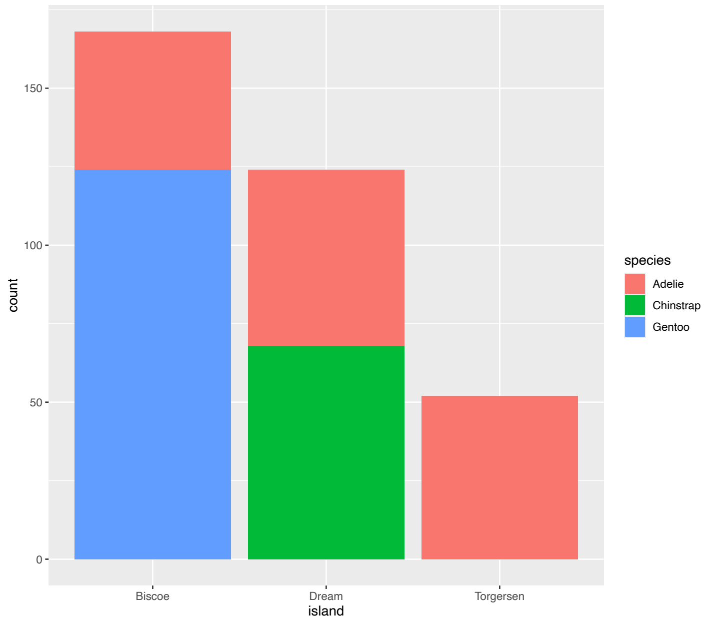
```
tw (scatter bmg_* flipper_length_mm),  ///
  scheme(white_tableau) ///
  xtitle("Flipper length (mm)") ///
  ytitle("Body mass (g)") by(island, rows(1))
```



R

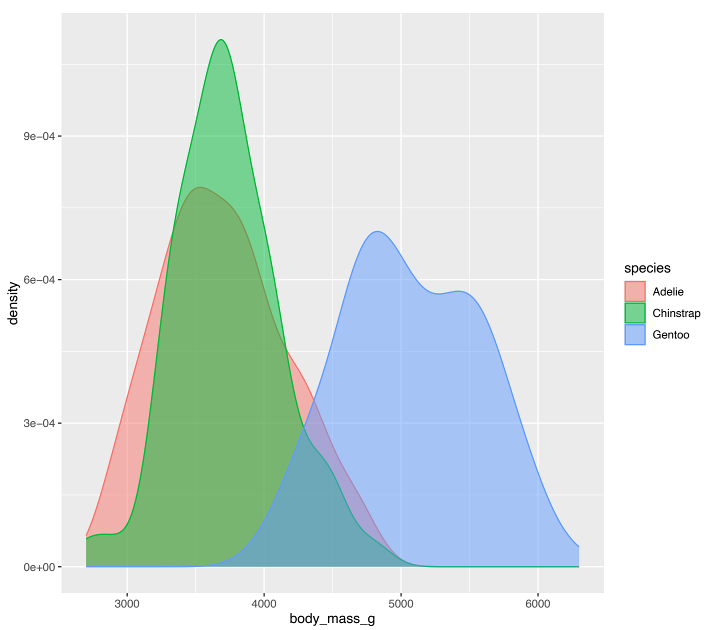
Graph 3:

```
ggplot(penguins,
  aes(x = island, fill = species)) +
  geom_bar()
```



Graph 4:

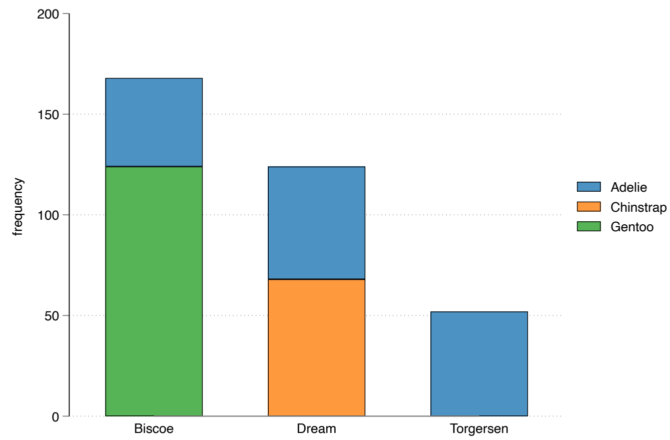
```
ggplot(penguins,
  aes(x = body_mass_g, color = species,
    fill = species)) +
  geom_density(alpha = 0.5)
```



Stata

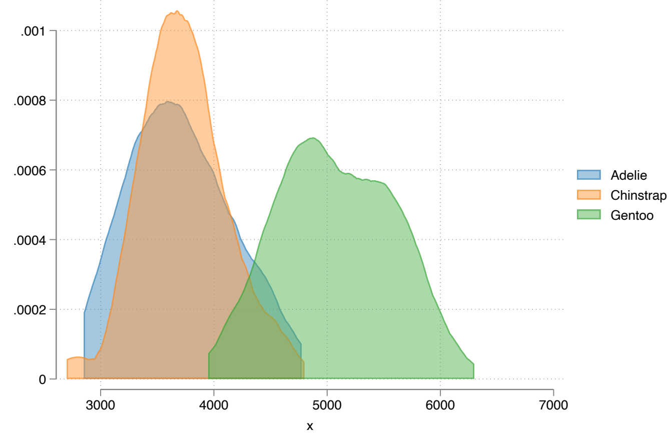
Graph 3:

```
graph bar (count), ///
  over(species, sort(1) descending) ///
  over(island) stack asyvars scheme(white_tableau)
```



Graph 4:

```
levelsof species, local(species_local)
local i = 1
foreach x of local species_local {
  local plottext ‘" ‘plottext’ ///
  (kdensity body_mass_g if species=="‘x’", ///
    recast(area) color(%50)) "’
  local legendtext  ‘" ‘legendtext’ ‘i’ "‘x’" "’
  local i = ‘i’ + 1
}
graph twoway ‘plottext’, ///
scheme(white_tableau) legend(order(‘legendtext’))
```



Resources for more

R

An introduction to R by Venebales, Smith, et al.

FasteR: Fast Lane to Learning R by Matloff

R for Data Science by Wickham, Cetinkaya-Rundel and Grolemund

A First Course in Statistical Programming with R by Braun and Murdoch

The Art of R Programming by Matloff

Stata

German Rodriguez's Stata tutorial

A little Stata programming goes a long way by Baum

The official Stata documentation

An Introduction to Stata Programming by Baum