Notes on using R

John Gardner

Table of contents

Introduction

These notes are a tutorial on using R for empirical projects. I have tried to keep it short while still touching on the topics you need to know about to start using R for your own work. There

is much to know about R, and I encourage interested readers to consult the resources at the end of the notes for even more information.

Getting R (and R Studio)

You can download R from https://www.r-project.org. Technically this is all you need, but most people prefer to work in the RStudio editor, which makes it easy for you to run commands, view graphs and data, browse through help files, and more. You can download RStudio for free from https://posit.co (they also offer an editor called Positron that works with R and Python, but I don't have any experience with it).

Packages

Packages extend R's functionality. You can install them using `install.packages("packagename")` and load them for use using `library(packagename)` (note the you use quotes when installing but not loading). You can also use a command from a package without loading it using the syntax `packagename::command`.

Base R, Tidyverse, and data.table. The *Tidyverse* is a popular set of packages that provide different ways of working in R, and refine some of R's built-in capabilities. They can be loaded using `library(tidyverse)`. I recommend learning how to do things using R's built-in capabilities ("base R") as well as using the Tidyverse, since they both have their advantages. Another popular package for data manipulation which I don't discuss here is data.table, which is capable and fast, and may be of interest to you, particularly if you work with very large datasets.

Interacting with R

You can use R by directly typing commands into the *console* (the command line in the bottom window), but it is much better to create an R script (ending in `.R`), which contains a record of all of your commands. In RStudio, you can run the selected line by clicking the `Run` button, or by hitting `Command + Enter` on macOS or `Ctrl + Enter` on Windows. You can also run the entire file by selecting `Run all` from the dropdown menu on the `Run` button.

If you prefer, you can also run an entire R script using the command `source("filename.R")`. Before you do this, you will need to set the working directory to the path of your script file, which you can do using the command `setwd("path/to/file")`. If you prefer to use backslashes on Windows, you will need to *escape* them, so your command will look like `setwd("path\\to\\file")`.

It's a good idea to comment your R scripts with notes about the purpose of your code. You can use the `#` character to start a comment. In RStudio, you can also type `Command + Shift + R` to insert a nice-looking separator line.

You can view the documentation for any command by typing `help(commandname)` (this may take you to a page of search results if there are multiple similarly named commands). For me, the R documentation is hit or miss: sometimes it answers my questions and teaches me about new and useful options/commands, sometimes it's frustratingly terse. You get used to it, and you can always try a web search or LLM.

Types of data

The main data types that you will work with in R are *vectors*, *lists*, *matrices*, and *data frames* (there are also *arrays*, which generalize matrices and include matrices and data frames as a subset, but we won't be working with them).

Vectors

A vector is a list of entries that are all the same *type* (usually numeric or character). You can create a vector using the concatenate function `c()` and assign it a name using the assignment operator `<-`:[1]

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)
c <- c("a", "b", "c")
```

You can determine an object's type by using the `class` command. You can refer to the elements of a vector using `a[1]`, or for multiple elements, `a[1:2]`. You can add/subtract and perform elementwise multiplication/division on numeric vectors using `+`, `-`, `*`, and `/`. Transformations are *vectorized*, meaning that they will be applied to all elements of a vector:

```
a + b
a - b
a * b
a / b
exp(a)
b^2
```

If you have a vector of categorical data, you can turn it into a *factor* using the `factor` command:

[1] If you're feeling rebellious, you can go the other way: `c(1, 2, 3) -> a`.

```
fac.a <- factor(a)
fac.a
```

```
[1] 1 2 3
Levels: 1 2 3
```

You can also work with a vector as though it had been converted to a factor using the `as.factor` function. R provides facilities for working with factor variables (see `help(factor)`), although often you can simply leave them as numeric vectors.

Lists

A list is essentially a vector where not every element is of the same type. Lists can include vectors or other lists as elements. You can create a list using the `list()` command:

```
d <- list(1, 2, "a", "b")
e <- list(a, b, c)
```

You can view the structure of a list (or any object in R) using the `str` function. Referring to elements of a list works a little differently: `e[1]` returns a list that contains the vector `(1, 2, 3)`, while `e[[1]]` returns the vector `(1, 2, 3)` itself. Hence, `class(e[1])` returns `list` while `class(e[[1]])` returns `numeric`.

You can add names to the elements of a list using the `names` function, or when creating the list:

```
names(e) <- c("a", "b", "c")
f <- list(a = a, b = b, c = c)
```

(In the above, `a = a` means that the element named "a" takes the value `a`.) You can refer to the elements of a named list using `e[["a"]]` or `e$a`, both of which mean the same thing.

Matrices

A matrix is a collection of numeric vectors that all have the same length. You can create a matrix using the `matrix` function. By default, R creates matrices by filling in the columns (this can be reversed using the `by.row = TRUE` option). All of the following create the same matrix:

```
g <- matrix(c(1, 2, 3, 4), nrow = 2)
g <- matrix(1:4, ncol = 2)
g <- matrix(seq(1, 4, by=1), nrow = 2, ncol = 2)
g
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

You can create an $n \times k$ matrix of zeros using `matrix(0, n, k)` or an $n \times n$ identity matrix using `diag(n)`.[2] You can refer to the elements of a matrix using, e.g., `g[1, 2]` or a submatrix using `g[1:2, 2]`. You can name the rows of a matrix using `rownames` and the columns using `colnames`:

```
h <- matrix(1:9, nrow = 3)
rownames(h) <- c("r1", "r2", "r3")
colnames(h) <- c("x1", "x2", "x3")
h
```

```
   x1 x2 x3
r1  1  4  7
r2  2  5  8
r3  3  6  9
```

If the rows or columns of a matrix are named, you can subset the matrix using row or column names intead of numbers, as in `h[c("r1", "r2"), c("x1", "x3")]`. You can also index a matrix by another matrix. For example, the following returns a vector containing the third element of the first row of `h`, the second element of the second row, and the first element of the third row:

```
index <- cbind(1:3, c(3, 2, 1))
index
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
```

[2] The `diag` function contains multitudes. If you have a vector `a`, `diag(a)` will create a matrix with the elements of `a` on the diagonal, and zeros elsewhere. If you have a square matrix `A`, `diag(A)` will return a vector consisting of the diagonal elements of `A`. Somehow this all makes sense when you're actually using it.

```
h[index]
```

```
[1] 7 5 3
```

`cbind` stands for column bind, and allows you to append two columns side by side (there is also `rbind`, which operators analogously on rows). Note that `h[index]` returns a vector.

Matrix addition works as you might expect. The `%*%` operator performs matrix multiplication on conformable matrices, while `*` does element by element multiplication. You can invert a matrix using `solve` and transpose it using `t`.

```
i <- matrix(5:8, nrow = 2)
i
```

```
     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

```
g %*% i
```

```
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```

```
g * i
```

```
     [,1] [,2]
[1,]    5   21
[2,]   12   32
```

```
solve(g)
```

```
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

```
t(g)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

If you multiply a matrix by a vector elementwise (using `*`), or add a matrix and a vector, R will use the "recycling rule" to turn the vector into a conformable matrix. Since R fills matrices columnwise, this means that elementwise multiplication of the matrix `g` by the vector `j` in the following is the same as multiplying `g` by a matrix consisting of two copies of `j`, stacked side-by-side:

```
j <- c(1, 2)
g * j
```

```
     [,1] [,2]
[1,]    1    3
[2,]    4    8
```

```
g + j
```

```
     [,1] [,2]
[1,]    2    4
[2,]    4    6
```

R will interpret `j` as a row vector if you pre-multiply `g` by `j` (using matrix multiplication `%*%`), and as a column vector if you post-multiply `g` by `j` (i.e., `j %*% g` is equivalent to `matrix(j, nrow=1) %*% g`).

Data frames

A data frame is basically a list of vector that all have the same length (i.e., a dataset). You can create a data frame using the `data.frame` function:

```
dat <- data.frame(a, b, c)
```

Since a data frame is a list, you can subset it using the same rules as for lists. For example, `dat["a"]` returns a data frame consisting of only the variable `a`, while `dat[["a"]]` and `dat$a` return the variable (vector) `a` itself. You can also subset dataframes like matrices, so `dat[2:3, c("a", "c")]` returns a data frame consisting of the second and third observations on the variables `a` and `c`.]

You can preview a data frame (or vector) using `str(dat)`, `head(dat)`, or in the Tidyverse, `glimpse(dat)`. In RStudio, you can also view a data frame (or any data object) by typing `View(dat)`, or by double clicking on it in the *Environment* pane.

Programming tools

Functions

You can extend it by writing your own functions. For example, suppose that you are working with the `mtcars` dataset (which comes with R and contains characteristics of different cars), and you want to know how average milage changes with the number of cylinders a car has. While there are built-in functions to handle this kind of thing, you could can also write your own:

```
cyl.mean.mpg <- function(cylinders) {
  mean(mtcars[mtcars$cyl==cylinders, ]$mpg)
}
cyl.mean.mpg(4)
```

```
[1] 26.66364
```

```
cyl.mean.mpg(6)
```

```
[1] 19.74286
```

Note that we use the double equals sign "`==`" when checking whether a condition holds (when evaluating a character variable, we put the condition in quotes, as in `char.var == "a"`).

This illustrates the basic use of `function`, but there are a few ways that we can make our function more useful. R functions can accept multiple inputs. And while by default they return the last object created, we can also specify what we want the function to return, which can be a scalar, vector, matrix or list.

Here is a somewhat more complex version of our function:

```
cyl.stats <- function(cylinders, variable) {
  mean <- mean(mtcars[mtcars$cyl==cylinders, ][[variable]])
  sd <- sd(mtcars[mtcars$cyl==cylinders, ][[variable]])
  return(list(mean = mean, sd = sd))
}
cyl.stats(4, "mpg")
```

```
$mean
[1] 26.66364

$sd
[1] 4.509828
```

```
cyl.stats(6, "wt")
```

```
$mean
[1] 3.117143

$sd
[1] 0.3563455
```

This function takes two arguments, `cylinders` and `variable`, and returns a list containing the mean and standard deviation of `variable` among all cars with the specified number of cylinders.

Loops

Loops allow you to perform actions iteratively. As a simple example, we could use a loop to run our basic `cyl.mean.mpg` function for cars with 4, 6, or 8 cylinders:

```
means <- rep(0, 3)
values <- c(4, 6, 8)
for (i in 1:3) {
  means[i] <- cyl.mean.mpg(values[i])
}
means
```

```
[1] 26.66364 19.74286 15.10000
```

A few notes on this: We initially set `means` to a vector of zeros with the correct final length (an alternative would be to build the vector `means` by joining the result of each iteration with the previous one). On bigger jobs, this makes the loop faster because R doesn't have to make a new copy of the vector each time. Also, we use `cyl.mean.mpg(values[i])` to make sure that the first entry in `means` consists of the *value* of the first element of `values`, and so on.

One limitation of the loop above is that we had to specify the values of cylinders. We can avoid this by having R determine the possible values of `cyl` and the length of `means`:

```
values <- sort(unique(mtcars$cyl))
means <- rep(0, length(values))
for (i in seq_along(values)) {
  means[i] <- mean(mtcars[mtcars$cyl==values[i], ]$mpg)
}
means
```

```
[1] 26.66364 19.74286 15.10000
```

Here, `sort(unique(mtcars$cyl))` gives us the unique values of `cyl` (in increasing order, as opposed to the order in which they occur, which `unique` provides by default), and `seq_along(cyls)` tells the loop to iterate over those values (this is basically the same as saying `for (i in 1:length(values))`, but won't throw an error if the length is zero). Note that we could have also called our `cyl.mean.mpg` function inside the loop instead of defining the value of `means` "by hand".

Loops can be nested. If we wanted to know how the means of both `mpg` and `wt` change with `cyl`, we could use:

```
values <- sort(unique(mtcars$cyl))
vars <- c("mpg", "wt")
means <- matrix(0, nrow=length(values), ncol=2)
for (i in seq_along(values)) {
  for (j in seq_along(vars)) {
    means[i, j] <- cyl.stats(values[i], vars[j])[[1]]
  }
}
rownames(means) <- paste0("cyl = ", values)
colnames(means) <- vars
means
```

```
             mpg       wt
cyl = 4 26.66364 2.285727
cyl = 6 19.74286 3.117143
cyl = 8 15.10000 3.999214
```

As we will see below, loops are useful for modifying a data frame. They can also be used to modify objects that aren't elements of an existing data frame. For example, suppose that we wanted the exponential of three matrices, `x1`, `x2`, and `x3`. One way we could create these matrices using a loop is with the `assign` function:

```
set.seed(12345)
for (i in 1:3) {
  assign(paste0("x", i), matrix(runif(4), ncol = 2))
}
```

Here, we use the `paste0` function to create the strings "x1", "x2", and "x3" (there is also `paste`, which allows you to specify a separator like "_"). If we tried to say `paste0("x", 1) <- matrix(...)`, we'd get an error message because R would think we were trying to assign

something to the literal string "x1". Instead, we use the `assign` function to tell R that we are creating an object whose name is that string. We also use the `runif` function to populate the matrices with pseudo-random draws from a uniform distribution (and seed the random number generator to make sure that we get the same values every time).

We could also use a loop to define transformations of these matrices. To do this, we use the `get` function (which is the inverse of `assign`) to retrieve the objects whose names are given by the strings "x1", "x2", and "x3":

```
for (i in 1:3) {
  assign(paste0("x", i, "_exp"), exp(get(paste0("x", i))))
}
```

Alternatively, we could collect our matrices into a list, then work with the list. The `mget` function allows us `get` a list of multiple objects, all at once. Once we have a list, we can iterate through the elements of the list:

```
matrices <- mget(paste0("x", 1:3))
for (i in 1:3) {
  matrices[[i]] <- exp(matrices[[i]])
}
```

The first time I tried this, I accidentally typed `exp(matrices[i])` and got an error, because `matrices[1]` is a list containing `x1` (which can't be exponentiated), while `matrices[[1]]` is the matrix `x1` itself, which can. Now, we could return to the `assign` function to add these transformed matrices as objects in our environments. Alternatively, we can use the `list2env` function, which handles this for us (in this case, we have to tell R to save the matrices in the *global environment*, which is the main workspace where objects are stored):

```
names(matrices) <- paste0("x", 1:3, "_exp")
list2env(matrices, envir = .GlobalEnv)
```

Conditionals and while loops

Instead of looping over a set index, we can use `while` to continue looping while a condition is met. Similarly, we can use `if`, `else if` and `else` to execute commands based on whether a condition is met.

In the following example, we illustrate these commands by using the bisection method to find the positive root of $f(x) = x^2 - 16$.[3]

[3]Here is the bisection algorithm. Pick a point L where $f(L) < 0$ and a point H where $f(H) > 0$, then calculate the midpoint M. If $|f(M)| < \epsilon$, stop. Otherwise, if $f(L) * f(M) < 0$, the root is between L and M, so start over, replacing H with M. If $f(L) * f(M) > 0$, the root is between M and H, so start over, replacing L with M.

```
f <- function(x) x^2 - 16
# initial values
eps <- 1
low <- 0
high <- 5
i <- 1 # no. of iterations
while (eps > 10^(-5)) {
  mid <- .5*(low + high)
  eps <- abs(f(mid))
  if (f(low) * f(mid) < 0) {
    high <- mid
  } else {
    low <- mid
  }
  i <- i + 1
}
eps
```

```
[1] 7.629395e-06
```

```
mid
```

```
[1] 4.000001
```

```
i
```

```
[1] 21
```

Apply functions

Loops are useful programming tools, but sometimes it is more convenient to use one of the "apply" family of functions, which are abstractions of loops known as *functionals* (programming with them is called *functional programming*).

The `apply` function itself applies a function to every row or column of an array (e.g., a matrix or data frame). For example, if we wanted the mean of every column of `mtcars`, we could use

```
apply(mtcars, 2, mean)
```

```
       mpg        cyl       disp         hp       drat         wt       qsec
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
        vs         am       gear       carb
  0.437500   0.406250   3.687500   2.812500
```

Or, if we wanted the sum of every row of the matrix `x1`, we could use

```
apply(x1, 1, sum)
```

```
[1] 1.481886 1.761898
```

The `lapply` function applies a function to every element of a list (I think of this as a general-purpose substitute for loops) and returns the results as a list.[4] For example, instead of using a loop to get the mean mileage for every value of cylinder, we can use:

```
lapply(unique(mtcars$cyl), cyl.mean.mpg)
```

```
[[1]]
[1] 19.74286

[[2]]
[1] 26.66364

[[3]]
[1] 15.1
```

Note that, here, we don't need to use constructs like `seq_along` – `lapply` automatically sets the first element of the output list to the value of `cyl.mean.mpg` applied to the first element of the input list, and so on.

We can also define functions within the `lapply` statement. Both of the following produce the same result:

```
lapply(unique(mtcars$cyl),
       function(x) mean(mtcars[mtcars$cyl==x, ]$mpg))
lapply(unique(mtcars$cyl), \(x) mean(mtcars[mtcars$cyl==x, ]$mpg))
```

[4] The *purrr* package from the Tidyverse has a modified version of `lapply` called `map`, which works essentially the same way, but offers a few additional conveniences (see *R for Data Science* for more).

The `\(x)` syntax is called an *anonymous function*, but it's just a shorthand for the usual function syntax. Also note that for short functions, we can dispense with the braces and just type `function(x) thing_to_do`.

You can also use `lapply` with functions that take multiple arguments by supplying additional arguments after the function:

```
lapply(unique(mtcars$cyl), cyl.stats, "mpg")
```

```
[[1]]
[[1]]$mean
[1] 19.74286

[[1]]$sd
[1] 1.453567


[[2]]
[[2]]$mean
[1] 26.66364

[[2]]$sd
[1] 4.509828


[[3]]
[[3]]$mean
[1] 15.1

[[3]]$sd
[1] 2.560048
```

There is a variant of `lapply` called `sapply` that tries to simplify the output as a vector or matrix, if possible (the "s" stands for simplify):

```
sapply(unique(mtcars$cyl), \(x) mean(mtcars[mtcars$cyl==x,]$mpg))
```

```
[1] 19.74286 26.66364 15.10000
```

The `tapply` command applies a function to a vector within groups defined by levels of another vector. We could have been using `tapply` instead of our `cyl.mpg.mean` function all along:

```
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
       4        6        8
26.66364 19.74286 15.10000
```

This is equivalent to using the `split` command to make a list of vectors, each containing the values of `mpg` for a particular level of `cyl`, then using `sapply` on that list:

```
mpg.split <- split(mtcars$mpg, mtcars$cyl)
sapply(mpg.split, mean)
```

```
       4        6        8
26.66364 19.74286 15.10000
```

We can also use `lapply` instead of loops to work with lists of data objects. Let's use the `rm` command to remove our modified matrices, then recreate then using `lapply`:

```
rm(list=paste0("x", 1:3, "_exp"), matrices)
matrices <- lapply(paste0("x", 1:3), get)
matrices_exp <- lapply(matrices, exp)
names(matrices_exp) <- paste0("x", 1:3, "_exp")
lapply(1:3, \(i) assign(paste0("x", i, "_exp"),
                        matrices_exp[[i]],
                        envir = .GlobalEnv))
```

```
[[1]]
         [,1]     [,2]
[1,] 2.056291 2.140378
[2,] 2.400731 2.425711

[[2]]
         [,1]     [,2]
[1,] 1.578509 1.384163
[2,] 1.181012 1.664000

[[3]]
         [,1]     [,2]
[1,] 2.070324 1.035139
[2,] 2.690527 1.164595
```

The above illustrates how `lapply` can be used in conjunction with `get` and `assign` to replicate the functionality provided by the `mget` and `list2env` commands. Note that, when we use `assign` inside a function, we have to specify the environment (as we did when using `list2env`), which we didn't have to do when using a loop.

There are two other functions that are useful for functional programming. Suppose that you wanted to use `cbind` to join a list of matrices or dataframes. It's straightforward to do this with a loop (`NULL` below is a convenient way to define an object which is initially empty):

```
big.matrix <- NULL
for (i in seq_along(matrices)) {
  big.matrix <- cbind(big.matrix, matrices[[i]])
}
big.matrix
```

```
          [,1]      [,2]      [,3]      [,4]      [,5]       [,6]
[1,] 0.7209039 0.7609823 0.4564810 0.3250954 0.7277053 0.03453544
[2,] 0.8757732 0.8861246 0.1663718 0.5092243 0.9897369 0.15237349
```

What if you are using `apply` functions instead? `do.call` applies a function with the elements of a list as its arguments. For example, the following is equivalent to the loop above:

```
do.call(cbind, matrices)
```

```
          [,1]      [,2]      [,3]      [,4]      [,5]       [,6]
[1,] 0.7209039 0.7609823 0.4564810 0.3250954 0.7277053 0.03453544
[2,] 0.8757732 0.8861246 0.1663718 0.5092243 0.9897369 0.15237349
```

`Reduce` applies a function to the first to elements of a list, then applies the function to the result of the first step and the third element of the list, and so on. In this case, `Reduce(cbind, matrices)` produces the same result as `do.call`. However, `Reduce` also allows you to view the intermediate steps (we could replace `` `+` `` below with `sum`; the former shows how we can refer to symbolic mathematical operators):

```
Reduce(`+`, 1:5, accumulate=TRUE)
```

```
[1]  1  3  6 10 15
```

Loops vs. functionals. In R, a vectorized command like `v <- c(1:5)^2` is much faster than the loop `for (i in 1:5) v[i] <- i^2`. However, apply-style functions are really just a more concise way of writing a loop (and sometimes more concise means harder to code), so you should use whatever works best for the task at hand.

Working with data

Reading and writing data

R can read many different types of data files, but I am going to focus on the most common types. One frequently used format that is easy to work with is CSV (comma seperated value). If you have a csv file named `data.csv` saved on your desktop (on macOS), you can read it and store it as a data frame using

```
setwd("~/Desktop")
library(tidyverse)
d <- read_csv("data.csv")
```

By default, the entries in the first row will be interpreted as column headings. The `read_csv` function is part of the readr package from the Tidyverse. Alternatively, you can use the built-in function `read.csv`. The advantages of `read_csv` are that it can be faster and simpler to work with. It stores the data as a "Tibble," which is a modified data frame (see *R for Data Science* for a discussion of the differences), while some packages expect a traditional data frame. If you think you've run into this problem, you can convert a Tibble to a data frame using `d <- data.frame(d)`.

You can save a dataframe (or matrix) as a csv using the command `write_csv(d, "filename.csv")`. Alternatively, you can use the built-in command `write.csv`.

You can read Excel files using the `read_excel` command from the readxl package. The basic syntax is

```
d <- readxl::read_excel("data.xlsx", sheet="Sheet1")
```

The haven package from the Tidyverse can import data from several other statistical programs In particular, you can use `read_dta` to import a Stata file. The foreign package also reads data from other statistical software.

Once you've imported some data, you might want to clean up the formatting a little. You can rename one or more variables using

```
names(d)[names(d)=="y1"] <- c("why1")
names(d)[names(d) %in% paste0("y", 1:3)] <- paste0("why", 1:3)
```

or in the Tidyverse using

```
d |> rename(why1 = y1)
```

You can also change the case of variables names using either of the following:

```
names(d) <- tolower(names(d))
d |> rename_with(tolower)
```

You can save the image of your entire workspace (all data, functions, etc.) as an RData file using

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

and load an existing image using

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

Aside. Here is the code I used to generate these data:

```
d <- data.frame(replicate(6, rnorm(50)))
names(d) <- c(paste0("x", 1:3), paste0("y", 1:3))
d[c("z1", "z2")] <- as.factor(replicate(2, sample(1:5, 50,
                                                  replace=TRUE)))
write.csv(d, file="data.csv", row.names = FALSE)
```

`replicate` is a member of the apply family (it is equivalent to `sapply(1:2, \(x) sample(1:5, 50, replace=TRUE))`). `rnorm` creates draws from a normal distribution.

Subsetting data

There are several ways to prune a data frame so that it only contains certain variables. For example, if you want to drop the `y` variables from `d`, you can use any of the following:

```
d[c("x1", "x2", "x3", "z1", "z2")]
d[ , c("x1", "x2", "x3", "z1", "z2")]
d[c(paste0("x", 1:3), paste0("z", 1:2))]
d[names(d) %in% c(paste0("x", 1:3), paste0("z", 1:2))]
d[startsWith(names(d), "x") | startsWith(names(d), "y")]
d[!startsWith(names(d), "y")]
d[grep("^[xz]", names(d))]
subset(d, select=c(paste0("x", 1:3), paste0("z", 1:2)))
subset(d, select=-c(y1, y2, y3))
d[-c(4, 5, 6)]
```

Here are a few notes on these commands: `!startsWith(names(d), "y")` sepcifices the names in `d` that *don't* start with "y" (`!` is the logical negation operator). `grep("^[xz]", names(d))` is a *regular expression* that returns the names of `d` that start with either "x" or "z" (regular expressions are useful for working with strings, but I avoid them because I can never remember the syntax). `select=-c(y1, y2, y3)` returns the subset of `d` excluding the "y" variables, and `d[-c(4, 5, 6)]` returns the subset that excludes columns 4-6.

You can also drop a variable using `d$z1 <- NULL`.

In the Tidyverse, this can be accomplished using the `select` command from the dplyr package. All of the following do the same:

```
d |> select(x1, x2, x3, z1, z2)
d |> select(starts_with(c("x", "z")))
d |> select(num_range("x", 1:3), num_range("z", 1:2))
d |> select(!starts_with("y"))
```

This syntax deserves a bit of explanation. "`|>`" is the "pipe." It takes data and "pipes" it into the first argument of the next command (and so only works if the next command takes data as its first argument). The pipe is part of base R, but is most often used with Tidyverse commands. Note that since we didn't use `d <-` to modify the data frame, running these commands will simply print the data frame (actually, it will print a preview, which is one of the ways that Tibbles differ from data frames).

Now suppose that you only wanted to retain observations where `z1` is 4 or 5 and `z2` is 2. Any of the following would work:

```
d[(d$z1==4 | d$z1==5) & d$z2==2,]
d[d$z1 %in% c(4, 5) & d$z2==2, ]
subset(d, d$z1 %in% c(4, 5) & d$z2==2)
```

The syntax `|` means "or". Sometimes it's useful to have a detailed understanding of how this kind of subsetting works. The first command above is equivalent to

```
condition <- (d$z1==4 | d$z1==5) & d$z2==2
d[condition, ]
```

In the above, `condition` is a logical vector (`TRUE`/`FALSE` values) indicating whether the condition holds, and `d[condition, ]` is the subset of `d` for observations that satisfy the condition (i.e., where `condition == TRUE`).

We can also using dplyr's `filter` command:

```
d |> filter((z1==4 | z1==5) & z2==2)
d |> filter(z1 %in% c(4, 5) & z2==2)
```

Note. You can use the pipe to "chain" commands together (this is more common with Tidyverse commands, but it works in base R, too):

```
d |> select(z1, z2) |> filter(z1 %in% c(4, 5) & z2==2)
d |> subset(select=c(z1, z2)) |> subset((z1==4 | z1==5) & z2==2)
```

We could sort our data according to the variable `z1` using either of

```
d[order(d$z1), ]
d |> arrange(z1)
```

If we wanted to sort according to `z1`, and then the value of `z2` within levels of `z1`, we could use `order(d$z1, d$z2)` or `arranage(z1, z2)`.

Transforming data

The variable `z1` in the data frame `d` takes the values 1-5. Suppose we want to recode 4s and 5s to 1 and all other values to zero. We could do this using

```
d[d$z1==3 | d$z1==2 | d$z1==1, ]$z1 <- 0
d[d$z1==4 | d$z1==5 , ]$z1 <- 1
```

If `d` is saved as a data frame instead of a Tibble, this will give us an error if no observations satisfy the condition in brackets, because then we're trying to replace something that has length zero with something that has positive length (this is another way that Tibbles differ from data frames). We can avoid this by using the following alternative syntax, which always works:

```
d[d$z1==3 | d$z1==2 | d$z1==1, "z1"] <- 0
d[d$z1==4 | d$z1==5, "z1"] <- 1
```

Missing values. R stores missing values as `NA`. Our data don't have any missing values, but if they did, R would return an error using either of these approaches. How missing values are handled differs across commands, so it's a good idea to be vigilant about potential `NA`s. If we have missing values (or to be safe if we're not sure), we can use the following:

```
d[(d$z1==3 | d$z1==2 | d$z1==1) & !is.na(d$z1), "z1"] <- 0
d[(d$z1==4 | d$z1==5) & !is.na(d$z1), "z1"] <- 1
```

We could simplify this using the `%in%` operator (which also handles `NA`s correctly):

```
d[d$z1 %in% c(1, 2, 3), "z1"] <- 0
d[d$z1 %in% c(4, 5), "z1"] <- 1
```

We could also do this in a single line using *indicator function* notation, in which `(expression)` evaluates to `TRUE` if the expression is true and `FALSE` otherwise. In numeric settings, R will interpret `TRUE` as 1 and `FALSE` as 0, and we can use this to force the result to be numeric. Since the indicator function will return zeros for `NA`s, to be safe we should use:

```
d[!is.na(d$z1),]$z1 <- 1*(d[!is.na(d$z1),]$z1 %in% c(4, 5))
```

If we wanted to do this for both `z1` and `z2`, we could use a loop:

```
vars <- paste0("z", 1:2)
for (i in vars) {
  d[!is.na(d[i]), ][i] <- 1*(d[!is.na(d[i]), ][i] %in% c(4, 5))
}
```

Note here that we need to use `d[[i]] %in% ...` because `%in%` expects a vector, not a data frame.

To avoid the problems with these approaches, it might be easier to use the built-in `replace` function:

```
d$z1 <- replace(d$z1, d$z1 %in% c(1, 2, 3), 0)
d$z1 <- replace(d$z1, d$z1 %in% c(4, 5), 1)
```

which we could automate using a loop or `lapply`:

```
d[paste0("z", 1:2)] <- lapply(d[paste0("z", 1:2)],
                              \(x) replace(x, x %in% c(1,2,3), 0))
d[paste0("z", 1:2)] <- lapply(d[paste0("z", 1:2)],
                              \(x) replace(x, x %in% c(4,5), 1))
```

The dplyr package from the Tidyverse has convenient functions for doing this sort of thing. To recode `z1`, we can use:

```
d <- d |> mutate(z1 = case_when(z1 %in% c(1, 2, 3) ~ 0,
                                z1 %in% c(4, 5) ~ 1,
                                .default = z1))
```

`mutate` is dplyr's command for transforming and adding variables. The `case_when` syntax tells R to replace `z1` with 0 when `z1=5`, and to use the existing value of `z1` otherwise.

If we wanted to transform both `z1` and `z2` at the same time, we could use:

```
d <- d |> mutate(across(c(z1, z2),
                        \(x) case_when(x %in% c(1, 2, 3) ~ 0,
                                       x %in% c(4, 5) ~ 1,
                                       .default = x)))
```

In the above, we could replace `c(z1, z2)` with `all_of(vars)`, `num_range("z", 1:2)`, or `starts_with("z")`.

If we wanted to create a new variable `x1_sq` equal to the square of `x1`, we could use

```
d$x1_sq <- d$x1^2
```

If we wanted to do this for all of the `x` variables, we could use either

```
for (i in 1:3) {
  d[paste0("x", i, "_sq")] <- d[paste0("x", i)]^2
}
```

or the vectorized approach

```
d[paste0("x", 1:3, "_sq")] <- d[paste0("x", 1:3)]^2
```

In the Tidyverse, we could use

```
d <- d |> mutate(x1_sq = x1^2)
```

for one variable, or for multple variables,

```
d <- d |> mutate(x1_sq = x1^2, x2_sq = x2^2)
d <- d |> mutate(across(num_range("x", 1:3),
                        .fns = list(sq = \(x) x^2),
                        .names = "{col}_{fn}"))
```

We could add additional functions to the list if we wanted to add multiple transformations of these variables.

Merging and reshaping data

Imagine that our (pre re-coding) variables `z1` and `z2` represent "state" and "year". Suppose that we have another dataset that records the values of some different variables for each state and year. How could we merge this dataset to our original dataset `d`?

First, let's make such a dataset. We want all combinations of `z1` and `z2`, each of which take the values 1-5. We can create all of these combinations using the `expand.grid` function, then make some fake data for each combination:

```
temp <- expand.grid(z1=1:5, z2=1:5)
w1 <- runif(25)
w2 <- runif(25)
d.new <- data.frame(cbind(temp, w1, w2))
```

We can use the `merge` function to combine this new data frame with `d`:

```
d.merged <- merge(d, d.new, by=c("z1", "z2"))
```

There are some options to the merge function. For example, you might want the merged data to include values from `d.new` that don't appear in `d` (which doesn't occur in our example), see `help(merge)` for details.

You can also do this using the `left_join` command from dplyr:

```
d.merged2 <- d |> left_join(d.new, join_by(z1, z2))
```

A left join keeps all rows in the original dataset (we might throw away some rows from the new dataset if they don't correspond to any values in the original). There are also right joins (keep all rows from the new dataset), full joins (keep all rows from both), and inner joins (only keep rows that appear in both). These correspond to the options in the base-R `merge` function.

Now suppose that the variables `x1-x3` and `y1-y3` represent observations on the same variable for different units in different time periods (i.e., panel data). It is often easier to work with panel data in *long form*, which has one column per variable but multiple rows per unit, one for each time period.

It isn't too hard to reshape a dataset into long form in base R, but I always found the documentation a little confusing. First, we need to add an "id" variable that identifies each unit (we can do this using the `dim` function, which returns a vector containing the dimensions of an array). Then we can use the `reshape` command:

```
d <- read.csv("data.csv")
d$id <- 1:dim(d)[1]
d.long <- reshape(d, idvar="id",
                  varying=c(paste0("x", 1:3), paste0("y", 1:3)),
                  sep="", direction="long")
```

We could use regular expressions to replace the `varying` command with `varying=grep("^x|^y", names(d))`, which selects all variables that start with "x" or "y" (or we could just type everything manually `varing=c("x1", "x2"...)`).

In the Tidyverse, this can be accomplished using the `pivot_longer` command:

```
d.long <- d |> pivot_longer(cols = c(paste0("x", 1:3),
                                     paste0("y", 1:3)),
                            names_to = c(".value", "time"),
                            names_pattern = "([A-Za-z]+)(\\d+)")
```

Here, `cols` determines which columns get reshaped to long form, `names_to` determines what we do with the different parts of variable names like `x1` (we use the first part to name the variable and the second to index the time period), and `names_pattern` determines how we split names like `x1` into parts. In this case, `names_pattern` uses a regular expression that looks for letters followed by numbers. Unfortunately, this is the only syntax that reliably works in all cases (if our variables were named like "`x_1`", we could replace this with the simpler `names_sep="_"`). We could, however, replace the `cols` statement with the simpler `cols = starts_with(c("x", "y"))` or `cols = c(num_range("x", 1:3), num_range("y", 1:3))`.

To reshape back to "wide" form, we can use

```
d.wide <- reshape(d.long, idvar="id", timevar="time",
                  direction="wide", v.names=c("x", "y"))
```

or, in the Tidyverse,

```
d.wide <- d.long |> pivot_wider(names_from="time",
                                values_from=c("x", "y"),
                                names_glue="{.value}{time}")
```

Analysis

Summary statistics

R doesn't have a great built-in way to easily get descriptive statistics. You can get some descriptives for a data frame (or a subset of one) using:

```
summary(d)
```

```
       x1                 x2                 x3                 y1          
 Min.   :-1.94396   Min.   :-2.0193   Min.   :-2.395694   Min.   :-2.15245  
 1st Qu.:-0.62954   1st Qu.:-0.8372   1st Qu.:-0.620201   1st Qu.:-0.81119  
 Median : 0.07309   Median :-0.2028   Median : 0.006369   Median : 0.13922  
 Mean   : 0.10972   Mean   :-0.1018   Mean   : 0.018148   Mean   :-0.02562  
 3rd Qu.: 0.61040   3rd Qu.: 0.4488   3rd Qu.: 0.593582   3rd Qu.: 0.55230  
 Max.   : 2.72821   Max.   : 2.1323   Max.   : 1.918224   Max.   : 1.90431  
       y2                 y3                 z1              z2        
 Min.   :-1.8943   Min.   :-2.7330   Min.   :1.00   Min.   :1.00  
 1st Qu.:-0.4492   1st Qu.:-0.8780   1st Qu.:2.00   1st Qu.:2.00  
 Median : 0.2419   Median :-0.4337   Median :3.00   Median :3.00  
 Mean   : 0.1493   Mean   :-0.2758   Mean   :2.92   Mean   :3.06  
 3rd Qu.: 0.7228   3rd Qu.: 0.6060   3rd Qu.:4.00   3rd Qu.:4.00  
 Max.   : 2.4996   Max.   : 2.0274   Max.   :5.00   Max.   :5.00  
       id        
 Min.   : 1.00  
 1st Qu.:13.25  
 Median :25.50  
 Mean   :25.50  
 3rd Qu.:37.75  
 Max.   :50.00  
```

but chances are you want to know the sample size, standard deviation, etc. as well.

However, there are several packages that can do this. One easy approach is using `describe` from the `psych` package:

```
psych::describe(d, fast=TRUE)
```

```
   vars  n  mean    sd median   min   max range  skew kurtosis   se
x1    1 50  0.11  1.02   0.07 -1.94  2.73  4.67  0.52     0.09 0.14
x2    2 50 -0.10  0.96  -0.20 -2.02  2.13  4.15  0.24    -0.44 0.14
```

```
x3    3 50  0.02  0.91   0.01 -2.40  1.92  4.31 -0.05     0.22 0.13
y1    4 50 -0.03  1.04   0.14 -2.15  1.90  4.06 -0.25    -0.78 0.15
y2    5 50  0.15  0.87   0.24 -1.89  2.50  4.39 -0.08    -0.06 0.12
y3    6 50 -0.28  0.99  -0.43 -2.73  2.03  4.76 -0.22    -0.01 0.14
z1    7 50  2.92  1.35   3.00  1.00  5.00  4.00  0.00    -1.27 0.19
z2    8 50  3.06  1.41   3.00  1.00  5.00  4.00 -0.15    -1.32 0.20
id    9 50 25.50 14.58  25.50  1.00 50.00 49.00  0.00    -1.27 2.06
```

Here, we used the `fast=TRUE` option to get a more limited set of statistics. You can also use `describeBy` to get descriptives within groups.

You can also calculate your own descriptive statistics. For example, to get the sample size, mean and standard deviation, you could use

```
apply(d[paste0("y", 1:3)], 2, \(x) c(n=length(x), mean=mean(x), sd=sd(x)))
```

```
              y1        y2         y3
n    50.00000000 50.000000 50.0000000
mean -0.02562224  0.149286 -0.2757643
sd    1.04306947  0.868742  0.9878856
```

Our sample data don't have any missing values, but if they did, our statistics would also be `NA`. We could use `is.na` to get the sample size, and add the `na.rm=TRUE` option to the `mean` and `sd` functions:

```
apply(d[paste0("y", 1:3)], 2, \(x) c(n=sum(is.na(x)),
                                     mean=mean(x, na.rm=TRUE),
                                     sd=sd(x, na.rm=TRUE)))
```

You can calculate statistics within groups using `aggregate` (which omits missing values by default). To get the sample size, mean and median of `y1`-`y3` by `z1` and `z2` you could use:

```
head(aggregate(cbind(y1, y2, y3) ~ z1 + z2, d,
  \(x) c(n = length(x), mean=mean(x), sd=sd(x))))
```

```
  z1 z2       y1.n     y1.mean       y1.sd       y2.n    y2.mean      y2.sd
1  2  1 3.00000000 -0.15625681  1.59970107  3.0000000 -0.1648793  1.5722075
2  3  1 2.00000000 -0.02454541  0.20067036  2.0000000  0.2116156  1.0023726
3  4  1 3.00000000  0.08727342  0.64314253  3.0000000  0.1789289  0.7663552
4  5  1 2.00000000  1.02753392  0.14723381  2.0000000  0.9686784  0.3591143
5  1  2 2.00000000 -0.20859931  2.67238968  2.0000000  0.9521382  0.9171654
```

```
6  2  2  1.00000000 -1.02335712          NA  1.0000000 -0.2331576         NA
       y3.n    y3.mean      y3.sd
1 3.0000000 -0.7569015  0.1534366
2 2.0000000  0.1597525  0.6511911
3 3.0000000 -1.3442514  1.2223017
4 2.0000000 -0.2588588  0.9739089
5 2.0000000 -0.3362057  0.4113071
6 1.0000000 -0.7457210         NA
```

Here, we used `cbind` to apply `aggregate` to multiple variables (this trick often works with commands that take "~" formulas, including the regression command `lm` introduced below). We also used the `head` function to only print the first few rows of output. From here, you could also store the results as a data frame in order to export them to a spreadsheet program for further editing.

The dplyr package is particularly convenient for descriptive statistics. To get the sample size, mean and standard deviation, we can use `summarize`:

```
d |> summarize(y1_mean = mean(y1, na.rm=TRUE), y1_sd = sd(y1, na.rm=TRUE),
               n=n())
```

```
     y1_mean    y1_sd  n
1 -0.02562224 1.043069 50
```

```
d |> summarize(across(num_range("y", 1:3),
                      list(mean = \(x) mean(x), sd = \(x) sd(x))),
               n=n())
```

```
     y1_mean    y1_sd  y2_mean    y2_sd    y3_mean     y3_sd  n
1 -0.02562224 1.043069 0.149286 0.868742 -0.2757643 0.9878856 50
```

the sample size `n` gets special treatment here because it applies to the entire dataset, not any specific variable (alternatively, we could use `n=\(x) sum(is.na)` as above).

We could do this by `z1` and `z2`:

```
d |> group_by(z1, z2) |> summarize(across(num_range("y", 1:3),
                                    list(mean = \(x) mean(x, na.rm=TRUE),
                                         sd = \(x) sd(x, na.rm=TRUE))),
                                    n=n())
```

```
`summarise()` has grouped output by 'z1'. You can override using the `.groups`
argument.

# A tibble: 23 x 9
# Groups:   z1 [5]
      z1    z2 y1_mean  y1_sd y2_mean  y2_sd y3_mean  y3_sd     n
   <int> <int>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>  <dbl> <int>
 1     1     2 -0.209   2.67    0.952  0.917  -0.336  0.411     2
 2     1     3 -0.388  NA      -0.155 NA       0.896 NA         1
 3     1     4  0.691   1.57    0.162  0.953  -0.160  0.675     4
 4     1     5  0.641   0.288   0.193  0.726  -0.346  1.36      3
 5     2     1 -0.156   1.60   -0.165  1.57   -0.757  0.153     3
 6     2     2 -1.02   NA      -0.233 NA      -0.746 NA         1
 7     2     3  0.199   0.798   0.303  0.444  -0.106  1.84      4
 8     2     4 -0.995  NA       0.275 NA       0.723 NA         1
 9     2     5 -0.716  NA      -1.35  NA      -0.806 NA         1
10     3     1 -0.0245  0.201   0.212  1.00    0.160  0.651     2
# i 13 more rows
```

If you wanted to add these summaries back to your dataset, you could save them as a data frame, then do a merge. Alternatively, you can use base-R's `ave` function:

```
d[paste0("y", 1:3, "_mean")] <- lapply(d[paste0("y", 1:3)],
                                       \(x) ave(x, list(d$z1, d$z2),
                                                FUN=mean))
```

(it might be simpler to do this using a loop), or obtain the statistics using dplyr's `mutate`:

```
d <- d |> group_by(z1, z2) |> mutate(across(num_range("y", 1:3),
                                            list(mean = \(x) mean(x),
                                                 sd = \(x) sd(x)),
                                            .names = "{col}_{fn}"),
                                     n=n())
```

You can create tables for discrete variables using the `table` command. The basic syntax simply produces counts, but it can be used with `prop.table` to get proportions:

```
table(d$z1)
```

```
 1  2  3  4  5
10 10 11 12  7
```

```
prop.table(table(d$z1))
```

```
   1    2    3    4    5
0.20 0.20 0.22 0.24 0.14
```

To perform crosstabs, you can use either `table` or the somewhat more convenient `xtabs` command. When we use `prop.table` with a two-dimensional table, we have to specify whether we want row proportions (with a `1`) or column proportions (with a `2`).

```
prop.table(table(d$z1, d$z2), 1)
```

```
            1          2          3          4          5
  1 0.00000000 0.20000000 0.10000000 0.40000000 0.30000000
  2 0.30000000 0.10000000 0.40000000 0.10000000 0.10000000
  3 0.18181818 0.18181818 0.00000000 0.36363636 0.27272727
  4 0.25000000 0.08333333 0.33333333 0.25000000 0.08333333
  5 0.28571429 0.28571429 0.14285714 0.14285714 0.14285714
```

```
prop.table(xtabs( ~ z1 + z2, d), 2)
```

```
   z2
z1           1          2          3          4          5
  1 0.00000000 0.25000000 0.10000000 0.30769231 0.33333333
  2 0.30000000 0.12500000 0.40000000 0.07692308 0.11111111
  3 0.20000000 0.25000000 0.00000000 0.30769231 0.33333333
  4 0.30000000 0.12500000 0.40000000 0.23076923 0.11111111
  5 0.20000000 0.25000000 0.10000000 0.07692308 0.11111111
```

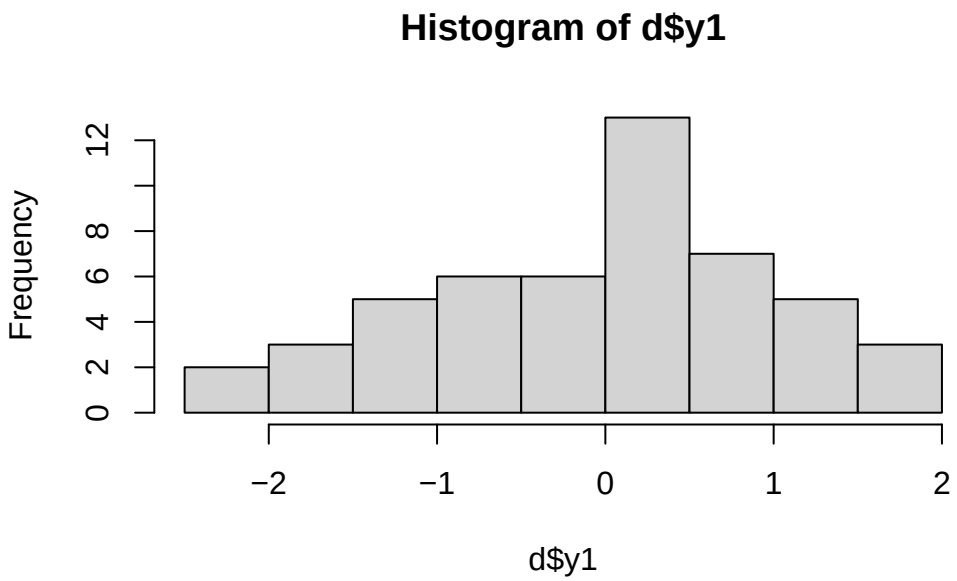
We could use, e.g., `round(prop.table(xtabs( ~ z1 + z2, d), 2), 2)`, to round the decimals to two places.

Visualizing data

R's graphing capabilities extend well beyond what can be covered in a short introduction, so we'll only sketch the basics, give a sense of the possibilities, and point in the direction of further resources.

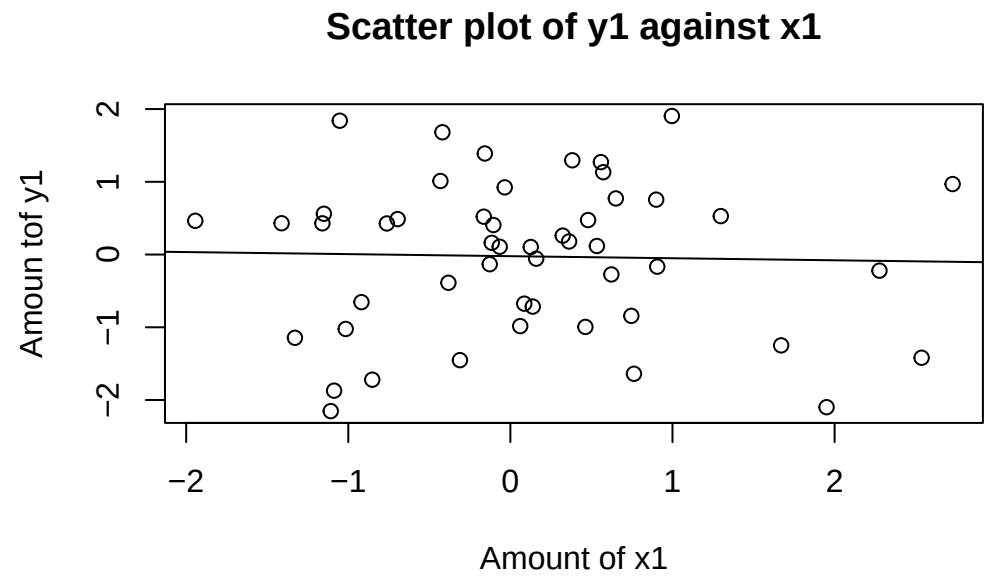
The basics of R's built-in plotting functions are easy. For example, you can obtain histogram using:

```
hist(d$y1)
```



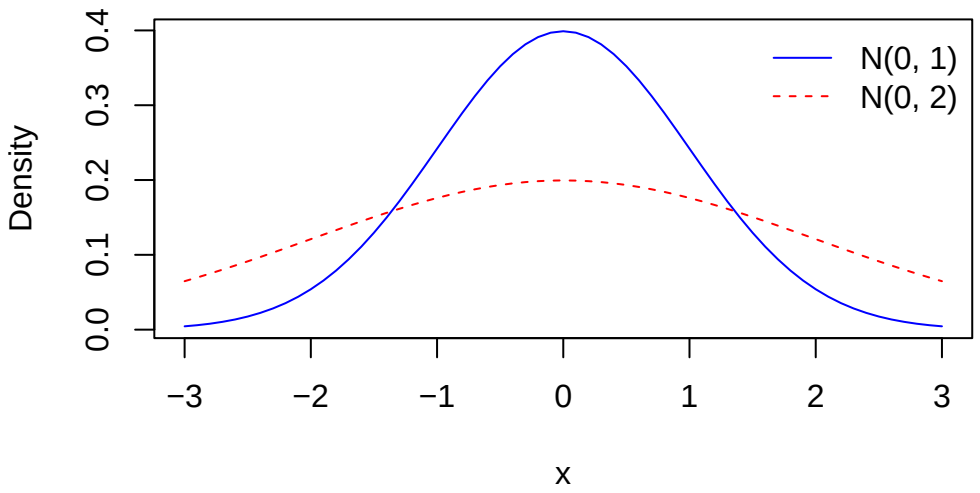
Here is a scatter plot, with a title and custom axis labels and a regression line (via the `abline` command):

```
plot(d$x1, d$y1, xlab="Amount of x1", ylab="Amoun tof y1",
     main="Scatter plot of y1 against x1")
abline(lm(y1 ~ x1, d))
```



I often use Base-R graphics to create simple line diagrams. Here's how we can plot the densities for two normal distributions with different variances:

```
x <- seq(-3, 3, by=.1)
d1 <- dnorm(x, 0, 1)
d2 <- dnorm(x, 0, 2)
plot(x, d1, type="l", col="blue", ylab="Density")
lines(x, d2, type="l", lty=2, col="red")
legend("topright", bty="n", c("N(0, 1)", "N(0, 2)"),
       col=c("blue", "red"), lty=1:2)
```



Here, `dnorm` returns the normal density (there is also `pnorm` for the distribution function and `qnorm` for the quantiles, useful for computing critical values), `plot` displays the initial curve, `lines` adds the second, `legend` adds the legend, `type="l"` gives a line (rather than underlying points), `col` controls the colors, `lty` controls the line type (solid or dashed), and `bty="n"` stops R from putting a box around the legend.

As you can see, base-R plots have a lot of options. You can read about them by typing `help(plot)`, or just search the web.

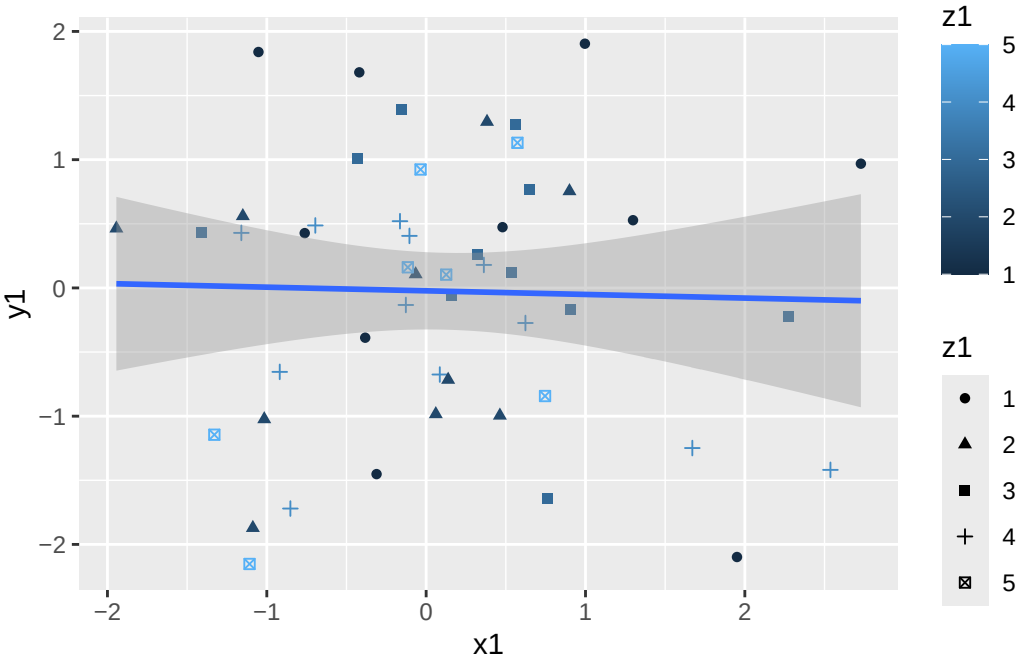
There is another plotting system called ggplot2 which is part of the Tidyverse, is very versatile, and has much more accessible documentation. Roughly speaking, you specify the data, then the geometric objects, describing the aesthetics of the plot along the way. For example, we could make a basic histogram and scatterplot using, saving the latter as a PDF, using

```
ggplot(dadta=d, aes(x=x1)) + geom_histogram()
ggplot(d, aes(x=x1, y=y1)) + geom_point()
ggsave("scatter.pdf")
```

Here are some more complicated examples, adapted from *R for Data Science*. The following produces a scatterplot of `y1` against `x1`, with the color and shape of the points determined by the value of `z1`, along with a linear fit and labelled axes (note that this requires that we treat `z1` as a factor variable):

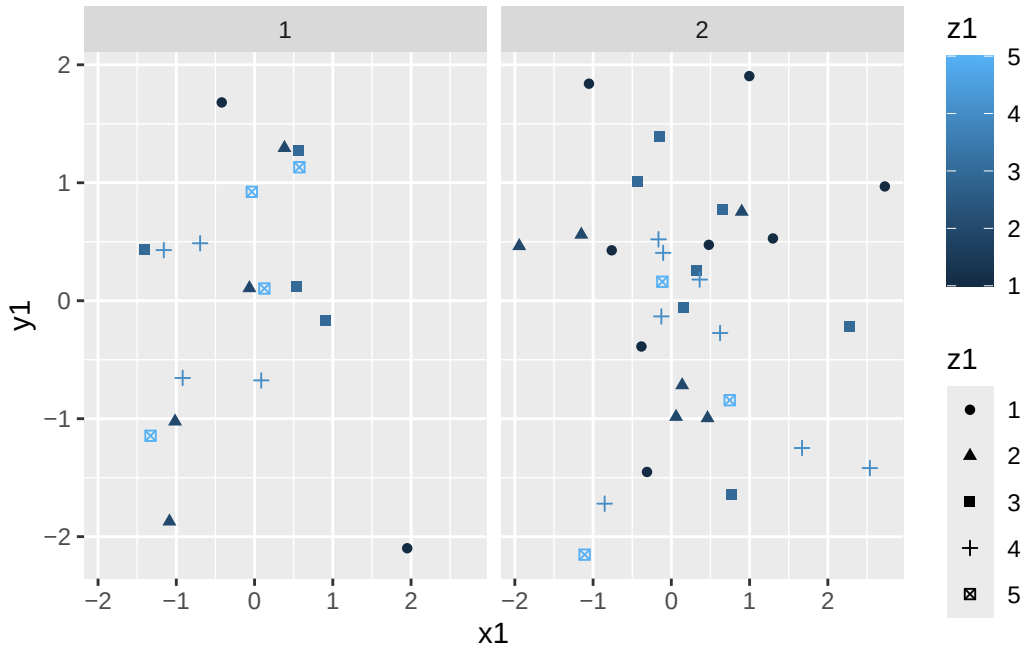
```
ggplot(d, aes(x1, y1)) + geom_point(aes(color=z1, shape=as.factor(z1))) +
  geom_smooth(method="lm") +
  labs(x="x1", y="y1", color="z1", shape="z1")
```

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



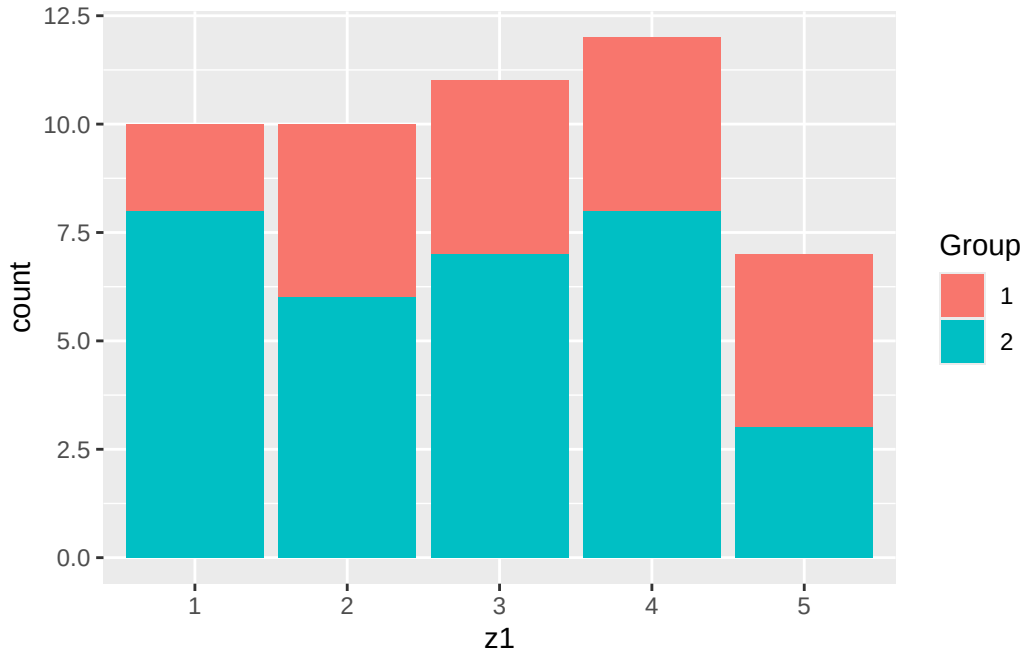
We can also use `facet_wrap` (also see `facet_grid`) to make subplots for different values of a categorical variable:

```
# make a binary group variable based on z2
d <- d |> mutate(gp = factor(1*(z2<3) + 2*(z2>=3)))
ggplot(d, aes(x1, y1)) + geom_point(aes(color=z1, shape=as.factor(z1))) +
  facet_wrap(~ gp) + labs(x="x1", y="y1", color="z1", shape="z1")
```



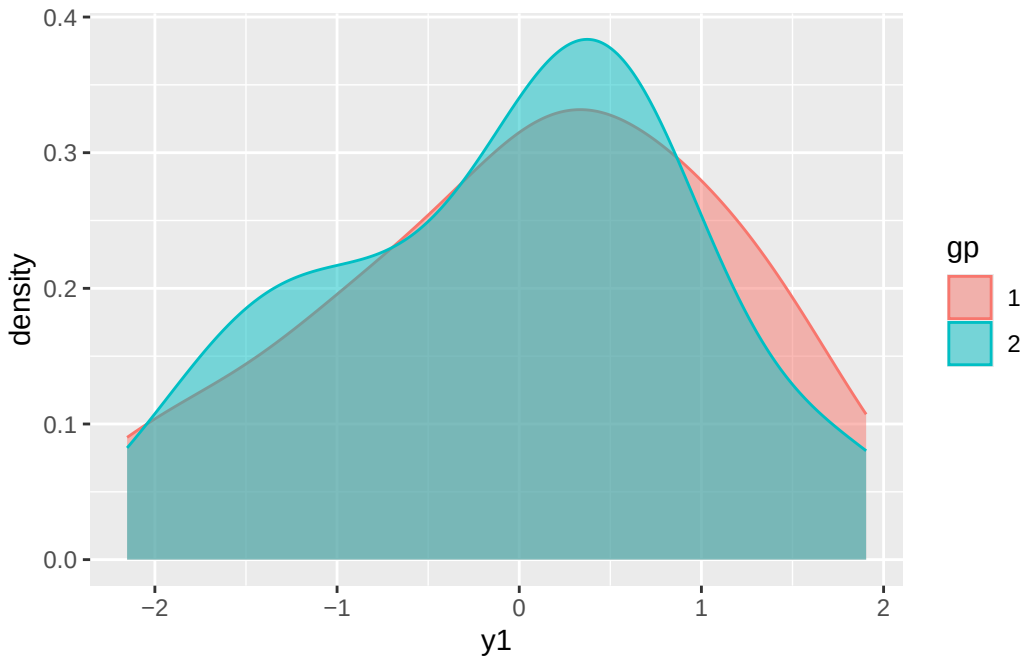
Here is a bar graph of `z2`, with the bars broken down by `group` (note that we use `as.factor` because `z1` is not saved as a factor variable):

```
ggplot(d, aes(as.factor(z1), fill=gp)) + geom_bar() +
  labs(x="z1", fill="Group")
```



And here are separate kernel densities for `y1` for each value of `group`:

```
ggplot(d, aes(y1, color=gp, fill=gp)) + geom_density(alpha=.5)
```



Regression basics

An introduction like this one is also not the place to go into detail on statistical or econometric estimation commands, but we can highlight some techniques that come up frequently.

In R, estimated models are most usefully stored as objects which can be accessed later. For example, to run an OLS regression and view the results, you can use

```
model1 <- lm(y1 ~ x1 + x2 + x3, d)
summary(model1)
```

```
Call:
lm(formula = y1 ~ x1 + x2 + x3, data = d)

Residuals:
    Min      1Q  Median      3Q     Max
-2.0277 -0.7679  0.1993  0.7355  2.1677
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.01206    0.14987  -0.080    0.936
x1          -0.06640    0.15017  -0.442    0.660
x2           0.01543    0.15725   0.098    0.922
x3          -0.25943    0.16790  -1.545    0.129

Residual standard error: 1.049 on 46 degrees of freedom
Multiple R-squared:  0.05134,   Adjusted R-squared:  -0.01053
F-statistic: 0.8298 on 3 and 46 DF,  p-value: 0.4843
```

Since the model has been saved, we can refer back to it and everything saved along with it (you can see everything saved with the model using `str(model1)` or `names(model1)`). For example, we can extract the coefficients using either of the following `model$coefficients` or `coef(model1)`.

We can extract the variance-covariance matrix using

```
vcov(model1)
```

```
              (Intercept)           x1          x2            x3
(Intercept)  0.0224618504 -0.002226572 0.002111048 -0.0007521723
x1          -0.0022265716  0.022551337 0.003262413  0.0046395898
x2           0.0021110485  0.003262413 0.024728827  0.0026378130
x3          -0.0007521723  0.004639590 0.002637813  0.0281889529
```

and the residuals and predicted values using `model1$residuals` and `model1$fitted.values`. If we wanted to obtain the predicted values from applying our model to a different dataset, we could use `predict(model1, dataset)`.

R has useful syntax for adding terms to a model. We can use `:` to get the interaction between two variables, `*` to get interactions and main effects, `I()` for transformations of variables, and `as.factor` to include indicators for every level of a categorical variable:

```
lm(y1 ~ x1*x2 + x1:x3 + I(x2^2) + as.factor(z1), d)
```

```
Call:
lm(formula = y1 ~ x1 * x2 + x1:x3 + I(x2^2) + as.factor(z1),
    data = d)
```

```
Coefficients:
   (Intercept)              x1              x2         I(x2^2)  as.factor(z1)2
       0.38126        -0.09714         0.06716         0.11604        -0.71730
as.factor(z1)3  as.factor(z1)4  as.factor(z1)5           x1:x2           x1:x3
      -0.15187        -0.86856        -0.80041         0.02989        -0.03123
```

Note that when we use `lm` without saving it, R simply prints the coefficients (without standard errors or other statistics).

When we use `as.factor` to include all levels of `z1`, R automatically chooses an omitted category. We can specify the omitted category by incuding `z1` in the formula using `relevel(as.factor(z1), ref=2)`. If we are going to be treating `z1` as a factor in many commands, we can permanently convert and set its baseline category using

```
d$z1 <- as.factor(d$z1)
d$z1 <- relevel(d$z1, ref=2)
```

We can use the `linearHypothesis` function from the car package to test linear hypotheses, and the `deltaMethod` function to test nonlinear ones:

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

```
Linear hypothesis test:
x1  + x2 = 0

Model 1: restricted model
Model 2: y1 ~ x1 + x2 + x3

  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     47 50.628
2     46 50.575  1  0.053104 0.0483  0.827
```

```
deltaMethod(model1, c("x1^2 + x2/2"))
```

```
             Estimate        SE     2.5 % 97.5 %
x1^2 + x2/2  0.012123  0.078401 -0.141540 0.1658
```

By default, R doesn't report confidence intervals, but you can obtain them using `confint(model1)`.

The sandwich package supports robust and clustered standard errors. The easiest way to report them is using the `coeftest` function from the lmtest package:

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

```
t test of coefficients:

             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.012058   0.159304 -0.0757   0.9400
x1          -0.066404   0.177596 -0.3739   0.7102
x2           0.015426   0.151925  0.1015   0.9196
x3          -0.259434   0.205761 -1.2609   0.2137
```

```
coeftest(model1, vcov=vcovCL(model1, cluster = ~ z2))
```

```
t test of coefficients:

             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.012058   0.099796 -0.1208  0.90436
x1          -0.066404   0.076382 -0.8694  0.38916
x2           0.015426   0.171451  0.0900  0.92870
x3          -0.259434   0.139238 -1.8632  0.06882 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that there are several variants on the robust and cluster-robust variance estimators, and you can use the `type` option to specify which one you want (see `help(vcovHC)` and `help(vcovCL)` for some details).

You can estimate binary choice models using the generalized linear models function `glm`. The syntax for a probit is

```
d$y1.bin <- (d$y1 >= 0)
probit <- glm(y1.bin ~ x1 + I(x1^2) + as.factor(z1),
              family=binomial(link=probit), d)
summary(probit)
```

```
Call:
glm(formula = y1.bin ~ x1 + I(x1^2) + as.factor(z1), family = binomial(link = probit),
    data = d)

Coefficients:
               Estimate Std. Error z value Pr(>|z|)
(Intercept)     0.01550    0.42642   0.036    0.971
x1             -0.16422    0.21318  -0.770    0.441
I(x1^2)        -0.08172    0.13168  -0.621    0.535
as.factor(z1)1  0.73018    0.60978   1.197    0.231
as.factor(z1)3  0.48036    0.57754   0.832    0.406
as.factor(z1)4 -0.14324    0.54733  -0.262    0.794
as.factor(z1)5  0.18197    0.62399   0.292    0.771

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 68.593  on 49  degrees of freedom
Residual deviance: 64.627  on 43  degrees of freedom
AIC: 78.627

Number of Fisher Scoring iterations: 4
```

We could get a logit by using `link=logit` instead.

Both the marginaleffects and margins package can be used to obtain marginal effects. For example, we can obtain average marginal effects using

```
library(marginaleffects)
avg_slopes(probit)
```

```
 Term Contrast Estimate Std. Error      z Pr(>|z|)   S  2.5 % 97.5 %
   x1    dY/dX  -0.0666      0.074 -0.899    0.368 1.4 -0.212 0.0785
   z1    1 - 2   0.2663      0.210  1.266    0.205 2.3 -0.146 0.6785
   z1    3 - 2   0.1819      0.214  0.850    0.395 1.3 -0.237 0.6012
   z1    4 - 2  -0.0549      0.210 -0.262    0.794 0.3 -0.466 0.3565
   z1    5 - 2   0.0704      0.241  0.293    0.770 0.4 -0.401 0.5419

Type: response
```

Since we used `I()` to include the quadratic term and `as.factor` to include `z1`, `marginaleffects` knows to treat `x1` and its square as a single expression, and to determine the marginal effects of `z1` using discrete differences.

There are several packages that format regression tables and export them to other formats (e.g., LaTeX or HTML), including texreg, stargazer, and modelsummary, all of which allow you to customize the contents of the tables.[5]

For example, we can run regressions for each level of `z1` (as we saw in the *Programming tools* section, there are multiple ways to do this), then summarize the results in a single table, and export that table (the following example uses texreg, but the alternative packages work similarly):

```
library(texreg)
models.z1 <- lapply(sort(unique(d$z1)), \(i) lm(y1 ~ x1 + x2 + x3,
                                                d[d$z1==i, ]))
names(models.z1) <- paste0("z1 = ", sort(unique(d$z1)))
screenreg(models.z1)
```

```
=====================================================
             z1 = 2  z1 = 1  z1 = 3  z1 = 4   z1 = 5
-----------------------------------------------------
(Intercept)  -0.20    0.57    0.45   -0.16    -0.04
             (0.48)  (0.60)  (0.30)  (0.17)   (0.27)
x1            0.17   -0.30   -0.46   -0.44 *   1.11
             (0.47)  (0.46)  (0.32)  (0.17)   (0.36)
x2            0.02   -0.03   -0.12   -0.07    -0.43
             (0.61)  (0.81)  (0.29)  (0.15)   (0.35)
x3           -0.02   -0.35   -0.27   -0.59 *  -1.68
             (0.49)  (0.52)  (0.35)  (0.20)   (0.65)
-----------------------------------------------------
R^2           0.02    0.10    0.25    0.62     0.83
Adj. R^2     -0.46   -0.35   -0.07    0.48     0.67
Num. obs.    10      10      11      12        7
=====================================================
*** p < 0.001; ** p < 0.01; * p < 0.05
```

[5] Although these commands export LaTeX tables, I often find that they need further editing. Instead, I export to HTML (which can be opened in Excel), edit them, then paste them into LyX or use Excel2LaTeX to get LaTeX-formatted tables.

```
htmlreg(models.z1, file="models_z1.html")
```

If we want to use robust or clustered standard errors, we need to do a little more work to add them to the table. Here's how we can display robust standard errors (and significance stars based on them) using stargazer:

```
library(texreg)
ses.z1 <- lapply(models.z1, \(x) coeftest(x, vcovHC)[, 2])
ps.z1 <- lapply(models.z1, \(x) coeftest(x, vcovHC)[, 4])
screenreg(models.z1, override.se = ses.z1, override.pvalues = ps.z1)
```

```
=====================================================
             z1 = 2  z1 = 1  z1 = 3  z1 = 4    z1 = 5
-----------------------------------------------------
(Intercept)  -0.20    0.57    0.45   -0.16     -0.04
             (0.62)  (0.81)  (0.43)  (0.19)    (0.39)
x1            0.17   -0.30   -0.46   -0.44 **   1.11
             (1.00)  (0.66)  (0.50)  (0.09)    (0.62)
x2            0.02   -0.03   -0.12   -0.07     -0.43
             (1.30)  (1.34)  (0.34)  (0.21)    (0.56)
x3           -0.02   -0.35   -0.27   -0.59     -1.68
             (0.67)  (0.68)  (0.55)  (0.37)    (0.75)
-----------------------------------------------------
R^2           0.02    0.10    0.25    0.62      0.83
Adj. R^2     -0.46   -0.35   -0.07    0.48      0.67
Num. obs.    10      10      11      12         7
=====================================================
*** p < 0.001; ** p < 0.01; * p < 0.05
```

As another example, we could create a table consisting of different model specifications:

```
formulas <- c("x1", "x1 + x2", "x1 + x2 + x3")
specifications <- lapply(formulas, \(x) lm(paste0("y1 ~", x), d))
screenreg(specifications)
```

```
=====================================
             Model 1  Model 2  Model 3
-------------------------------------
```

```
(Intercept)  -0.02    -0.02    -0.01
             (0.15)   (0.15)   (0.15)
x1           -0.03    -0.02    -0.07
             (0.15)   (0.15)   (0.15)
x2                     0.04     0.02
                      (0.16)   (0.16)
x3                             -0.26
                               (0.17)
---------------------------------------
R^2           0.00     0.00     0.05
Adj. R^2     -0.02    -0.04    -0.01
Num. obs.    50       50       50
=======================================
*** p < 0.001; ** p < 0.01; * p < 0.05
```

Here are a few additional packages that are useful for regressions and econometrics:

- The plm package estimates panel data models, including fixed and random effects
- The fixest package estimates fixed effects models efficiently, works with multiple dimensions of fixed effects, and has built-in clustering and graphing capabilities
- The ivreg packages handles instrumental variables and two-stage least squares

Resources for more

- An introduction to R by Venebales, Smith, et al. is a classic introduction to R, and also comes bundled with R itself
- FasteR: Fast Lane to Learning R by Matloff is a detailed tutorial, with a focus on base R
- R for Data Science by Wickham, Cetinkaya-Rundel and Grolemund is an excellent introduction to using R via the Tidyverse (the first edition touches on a few useful topics omitted from the second)
- Advanced R by Wickham delves into some of the more technical details of working with R (also see the first edition)
- Ggplot2: Elegant Graphics for Data Analysis by Wickham is a deep dive into ggplot2
- *The Art of R Programming* by Matloff provides a lot of details on R programming, with a focus on base R