Very helpful references

An introduction to R

FasteR: The fast lane to learning R

R for Data Science (and the first edition)

Advanced R (and the first edition)

Iteration

In my opinion, one of the best features of Stata is that you can write things like

```
local vars var1 var2 var3
foreach x of local `vars' {
  gen `x'sq = `x'^2
}
```

and it is as though you literally typed

```
gen var1sq = var1^2
gen var2sq = var2^2
gen var3sq = var3^2
```

The question is, how best to do this in R? It turns out that there are several ways to do this kind of iteration, but it works differently than in Stata.

What you *can't* do is type

```
vars <- c("var1", "var2", "var3")
# paste0("var", 1:3) for short
for (x in vars) {
  paste0(x, "sq") <- x^2
}
```

(or some variation on this) because R interprets this as the literal text

```
"var1sq" <- "var1"^2
```

and complains, because neither `"var1sq"` or "var1" are objects. In other words, R doesn't know what you want to create a *new* variable named `var1sq`, or that you want this variable to equal the square of `var1`.

Probably the simplest solution to this problem is to work with variables stored in a dataframe, because R allows you to refer to such variables using literal text, as in `df["var1"]`, which enables you to do things like

```
for (x in vars) {
  df[paste0(x, "sq")] <- df[x]^2
}
```

Once you are using this approach, the loop is actually unnecessary (although for more complicated things sometimes it's nice to spell things out in a loop), since you can use the vectorized version

```
df[paste0(vars, "sq")] <- df[vars]^2
```

The Tidyverse offers alternative ways to do this (although it requires you to remember the syntax). All of the following do the same thing (but in the last line I customized the names of the new variables so that they do not contain an underscore, which is the default):

```
df |> mutate(across(c(v1, v2, v3), .fns=list(sq = \(x) x^2)))
df |> mutate(across(vars, fns=list(sq = \(x) x^2)))
df |> mutate(across(num_range("var", 1:3), fns=list(sq = \(x) x^2)))
df |> mutate(across(starts_with("var"), fns=list(sq = \(x) x^2)))
df |> mutate(across(num_range("var", 1:3), .fns=list(sq = \(x) x^2),
  .names="{col}{fn}"))
```

You can replace the `.fns` syntax with the shorthand `.fns=list(sq = ~.x^2)`, although apparently this is no longer the recommended way of writing anonymous functions.

Unfortunately, none of these techniques work if the variables aren't saved in a dataframe. While it's probably a good idea to keep variables in a dataframe, sometimes you are working with, say, a number of matrices that can't be stored in that way.

In this case, one solution to the "literal text" problem is to use `assign` and `get`, as in

```
for (i in vars) {
  assign(paste0(i, "sq"), get(i)^2)
}
```

This uses two functions to do what Stata handles automatically: refer to "vars1sq" as a new variable and "vars1" as an existing one. You could use the following alternative approach, but it's much uglier:

```
for (i in vars) {
  assign(paste0(i, "sq"), eval(parse(text=i))^2)
}
```

Unfortunately, you can't use `assign(paste0("df$", i, "sq"))` to assign a variable inside the dataframe (R just saves it as the literal variable `df$isq`). Although it isn't necessary, you can use `within` to use `assign` inside dataframes:

```
within(d, {
  for (i in c("x1", "x2")) {
    assign(paste0(i, "sq"), get(i)^2)
  }
})
```

(This has the annoying side effect of saving `i` inside the dataframe, which you can remove using d$i <- NULL.)

Another way of working with objects that aren't in a dataframe is to put them inside a list. You

could either start with them in the list (and maybe keep them there for convenience), or combine them into a list using one of:

```
mylist <- mget(vars)
mylist <- lapply(vars, get)
```

(If for some reason you wanted to do this with variables stored in a dataframe, you could use `get("df")[vars]`.)

Now the elements of `vars` are stored in a list, and can be referred to by name. Thus you could add their squares to the list using lapply or a loop:

```
mylist[paste0(vars, "sq")] <- lapply(mylist[vars], \(x) x^2)

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

Note that, in the first case, we can't just use `mylist[vars]^2` because `mylist` is a list, which doesn't make sense to square. For the same reason, in the loop we need to index elements using double brackets, since `mylist[1]` is a list containing the first variable, while `mylist[[1]]` is the first variable itself.

For convenience, it probably makes sense to keep these variables stored in a list. If for whatever reason you need to add the to the global environment, you can use

```
list2env(mylist, env=globalenv())
```

Many regressions

Suppose that you want to run regressions in different subsamples. A natural starting point is the loop

```
for (i in unique(mtcars$cyl)) {
  print(summary(lm(mpg ~ wt, mtcars[mtcars$cyl==i,])))
}
```

This works fine, but there are a some additional considerations. First, you might want to store the results for future use (and the `print(summary())` construct is ugly):

```
models <- list()
for (i in unique(mtcars$cyl)) {
  models[[as.character(i)]] <- lm(mpg ~ wt, mtcars[mtcars$cyl==i,])
}
lapply(models, summary)
```

Here, we initialize an empty list called models to store the results in (in general, it's a bad idea to grow a list within a loop for speed reasons, but here this won't matter much). Next, we store the results using `models[[as.character(i)]]`. The `as.character` part is helpful because `cyl` takes the values 2, 4, and 6; if just used `models[[i]]`, R would automatically insert empty entries corresponding to 1, 3, and 5.

Another issue we might run into is whether the regression can be estimated for every subsample. If we extend our example to `cyl` and `gear`:

```
models <- list()
for (i in unique(mtcars$cyl)) {
  for (j in unique(mtcars$gear)) {
    models[[paste0(i, ", ", j)]] <- lm(mpg ~ wt,
      mtcars[mtcars$cyl==i & mtcars$gear==j,])
  }
}
lapply(models, summary)
```

we run into a problem because there aren't observations for all combinations of `cyl` and `gear`. Thus we need something to the effect of:

```
models <- list()
for (i in unique(mtcars$cyl)) {
  for (j in unique(mtcars$gear)) {
    if (dim(mtcars[mtcars$cyl==i & mtcars$gear==j,])[1]!=0) {
      models[[paste0(i, ", ", j)]] <- lm(mpg ~ wt,
        mtcars[mtcars$cyl==i & mtcars$gear==j,])
      }
  }
}
lapply(models, summary)
```

This works, but the loop is getting somewhat complicated (though this isn't the worst thing in the world). Instead, we could use `lapply`:

```
models <- lapply(unique(mtcars$cyl), \(x) lm(mpg ~ wt, mtcars[mtcars$cyl == i,]))
```

One thing that's nice about this is that it automatically names the list elements according to the values of `cyl`. For the double loop, we use `lapply` twice:

```
models <- lapply(unique(mtcars$cyl),
  \(i) lapply(unique(mtcars$gear),
    \(j) if (dim(mtcars[mtcars$cyl==i & mtcars$gear==j,])[1]!=0) {
      lm(mpg ~ wt, mtcars[mtcars$cyl==i & mtcars$gear==j,])
    }
  )
)
```

This isn't much cleaner. Alternatively, we can use `split` to split the dataframes into a list of dataframes for each element of the grouping variable:

```
mtcars.split <- split(mtcars, mtcars$cyl)
models <- list()
for (i in mtcars.split) {
  models(names(mtcars.split)[i]) <- lm(mpg ~ wt, i)
```

```
}
```

or

```
lapply(mtcars.split, \(x) lm(mpg ~ wt, x))
```

For two groups, we can split along two variables:

```
mtcars.split <- split(mtcars, list(mtcars$cyl, mtcars$gear))
```

then use a loop or `lapply`:

```
lapply(mtcars.split, \(x) if (dim(x)[1]!=0) {
    lm(mpg ~ wt, x)
  }
)
```

R also has some built-in functions that facilitate this kind of thing. Both of the following are equivalent to the above loop/`lapply` approaches:

```
tapply(mtcars, mtcars$cyl, \(x) lm(mpg ~ wt, x))
by(mtcars, mtcars$cyl, \(x) lm(mpg ~ wt, x))
```

We can also use these to group by two variables. In this case, by works for groups with insufficient observations, but `tapply` doesn't (also, `tapply` returns a list array, which is a bit harder to work with):

```
by(mtcars, list(mtcars$cyl, mtcars$gear), \(x) lm(mpg ~ wt, x))
tapply(mtcars, list(mtcars$cyl, mtcars$gear), \(x) lm(mpg ~ wt, x))
tapply(mtcars, list(mtcars$cyl, mtcars$gear),
  \(x) if (dim(x)[1]!=0) {lm(mpg ~ wt, x)})
```

The Tidyverse also provides some ways of doing this. A simple approach is:

```
mtcars |> split(mtcars$cyl) |>
  map(\(x) lm(mpg ~ wt, x)) |> map(summary)
```

This is almost a base-R approach, since `|>` is the native pipe, `split` and `summary` are base R functions, and `map` is very similar to `lapply`. Note that this has the same problem as a loop or `lapply` when splitting on `cyl` and `gear`.

Another approach is to use nested dataframes (also known as list columns). The idea is to create a column in a Tibble (the Tidyverse version of a dataframe) that consists of group-specific Tibbles (this is very similar to the `split` approach):

```
mtcars |> group_by(cyl) |> nest() |>
  mutate(models = map(data, \(x) lm(mpg ~ wt, x)))
```

Viewing the data shows that this results in a Tibble consisting of a dataset and the results of a model for each value of `cyl`.You could also use `nest(.by=cyl)` or `group_nest(cyl)` (though this is deprecated) instead of `group_by(cyl) |> nest()`.

Note the difference between the `map` syntax between the `split` and `nest` approaches. Each element of the split data is a dataframe, and these individual dataframes are the arguments to the `map` function. The nested dataframe has two columns, `cyl` and `data`, so in the `mutate` statement, we define an object named `models` that takes the `data` column as its argument.

One advantage of this approach is that it is easy to generalize to multiple grouping variables (just use `group_by(cyl, gear)`), and automatically handles the cases where there are no observations for certain combinations of groups.

As far as I know, this approach was introduced in the first edition of *R for Data Science*. There, the idea was to use `broom::tidy`, which creates cleaned-up versions of model output, to add the regression coefficients to the dataset (which is useful for, say, plotting the coefficients):

```
mtcars |> group_by(cyl) |> nest() |>
  mutate(models = map(data, \(x) lm(mpg ~ wt, x)),
         tidied = map(models, tidy) ) |>
         unnest(tidied)
```

We could achieve something similar in base R using `sapply`, which is a version of `lapply` that can simplify the output to a vector or matrix if possible:

```
coeffs <- sapply(split(mtcars, mtcars$cyl),
  \(x) lm(mpg ~ wt, x)$coefficients)
```

Instead of using `sapply`, we could also combine `lapply` with `do.call` or `reduce`, both of which are useful for functional programming in R:

```
coeffs <- lapply(split(mtcars, mtcars$cyl),
  \(x) lm(mpg ~ wt, x)$coefficients)
coeffs1 <- Reduce(rbind, coeffs)
coeffs2 <- do.call(rbind, coeffs)
```

`reduce` applies a function to the first two elements of a list, then applies the result to the third element, and so on. `do.call` applies a function to all elements of a list. In this case they have the same result, but not always. (If you type `reduce` instead of `Reduce`, you get the Tidyverse version, which requires the arguments to be reversed.)

Often, we are more interested in viewing the output. To do this in Tidyverse, we can modify our code to use

```
mtcars |> group_by(cyl) |> nest() |>
  mutate(models = map(data, \(x) lm(mpg ~ wt, x))) |>
  pluck("models") |> map(summary)
```

Here, the `pluck` statement selects only the `models` column of the nested dataframe, and sends this to the `map(summary)` statement (this is equivalent to saving `mtcars` with the estimated models, then using `mtcars$models |> map(summary)`.)

One drawback to this approach is that it doesn't print the corresponding values of `cyl` for each regression. Here is one approach to adding names to the models:

```
mtcars.models <- mtcars |> group_by(cyl) |> nest() |>
  mutate(models = map(data, \(x) lm(mpg ~ wt, x)))
names <- mtcars.models |> ungroup() |>
  mutate(name = paste0("cyl: ", cyl)) |>
  pluck("name")
mtcars.models <- mtcars.models |> pluck("models") |>
  set_names(names)
mtcars.models |> map(summary)
```

This estimates the models, creates a new variable called `names` based on the values of `cyl`, then sets the names of the `models` column equal to those names.

These approaches work well with packages for creating regression tables, most of which accept lists of models. For example, we can create a table of regressions using

```
models <- lapply(mtcars.split, \(x) lm(mpg ~ wt, x))
modelsummary::modelsummary(models)
```

or

```
mtcars |> group_by(cyl) |> nest() |>
  mutate(models = map(data, \(x) lm(mpg ~ wt, x))) |>
  pluck("models") |> modelsummary::modelsummary()
```

Recoding variables

Suppose we want to take the binary variable `am` from the `mtcars` dataset and recode 0 to 2 and 1 to 3. The simplest way to do this is:

```
d <- mtcars
d[d$am==0,]$am <- 2
d[d$am==1,]$am <- 3
```

(I'm setting `d` equal to `mtcars` so that I don't overwrite the original dataframe). We could also do this in base R using `d$am <- replace(d$am, d$am==0, 2)` and so on (of course, in this case we could also say `d$am <- d$am + 2`, but this is just an example).

In the Tidyverse, this can be accomplished using

```
d |> mutate(am = case_when(am==0 ~ 2, am==1 ~ 3))
```

One thing to watch out for is that if none of the conditions are met, all values of the variable get replaced with "NA"s unless we use `case_when(am==0 ~ 2, am==1 ~ 3, .default=am)`.

But what if we wanted to recode multiple variables the same way? One approach is to use a loop or `lapply`:

```
vars <- c("am", "vs")
# loop
for (i in vars) {
  d[i] <- 2*(d[i]==0) + 3*(d[i]==1)
}
```

```
# lapply
d[vars] <- lapply(d[vars], \(x) 2*(x==0) + 3*(x==1))
```

Alternatively, we could use

```
d[vars][d[vars]==0] <- 2
d[vars][d[vars]==1] <- 3
```

This latter approach uses clever R indexing: `d[vars]` is subset of `d` corresponding to the variables `am` and `vs`, and `d[vars]==0` is a matrix of "TRUE/FALSE" values. Then `d[vars][d[vars]==0]` is the vector of all values of `d[vars]` for which `d[vars]==0` is TRUE (note that this is one long vector of values from each column of `d[vars]`, which makes sense since the number of true elements likely differs for each variable, so it doesn't make sense for this to return a matrix).

In the Tidyverse, we can extend this to multiple variables using

```
d |> mutate(across(c(am, vs),
  \(x) case_when(x==0 ~ 2,
                 x==1 ~ 3)))
```

Merging and reshaping data

It turns out that it's pretty easy to merge data in base R. Let's create a dataset with different values for each combination of `cyl` and `am`:

```
df <- data.frame(cyl=sample(c(4,6,8), 6, replace=TRUE),
                 am=sample(0:1, 6, replace=TRUE),
                 y=rnorm(6))
```

We can merge this to `mtcars` by the values of `cyl` and `am` using:

```
merge(mtcars, df, by=c("cyl", "am"))
```

The Tidyverse equivalent is

```
mtcars |> left_join(df, join_by(cyl, am))
```

This is the type of merge that is used most often, where all observations in the original data are retained, but both base R and Tidyverse have options to change the type of merge/join (see `help(merge)` and `help(left_join)`).

I have always found reshaping in R much more of a pain than in Stata. Let's make a simple wide dataset (note that the `v` in the `lapply` statement doesn't do anything besides telling R to make 6 variables):

```
v <- c(paste0("x", 1:3), paste0("y", 1:3))
df <- data.frame(lapply(v, \(x) rnorm(10)))
names(df) <- v
df$id <- 1:10
df$gp <- sample(1:5, 10, replace=TRUE)
```

It isn't too hard to use base R's `reshape` to turn this into a long dataset (the real issue is that the documentation isn't very clear). Any of the following will work:

```
df.l <- reshape(df, idvar="id",
  varying=c("x1", "x2", "x3", "y1", "y2", "y3"),
  sep="", direction="long")
df.l <- reshape(df, idvar="id", varying=v, sep="", direction="long")
df.l <- reshape(df, idvar="id", varying=c(paste0("x", 1:3),
  paste0("y", 1:3)), sep="", direction="long")
df.l <- reshape(df, idvar="id", varying=grep("^x|^y", names(df)),
  sep="", direction="long")
df.l <- reshape(df, idvar="id",
  varying=grep("([A-Za-z]+)(\\d+)", names(df)),
  sep="", direction="long")
```

Note that R makes a new variable called `time`, although the name of this variable can be customized. The last two lines uses regular expressions for names that start with `x` or `y` or names that have any letters followed by any numbers. I avoid regular expressions because I can never remember them, but *R for Data Science* has a nice chapter on them.

To convert this to a long dataset, we can use

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

In the Tidyverse, we can reshape to long form using any of

```
df.l2 <- df |> pivot_longer(
  cols=c("x1", "x2", "x3", "y1", "y2", "y3"),
  names_to=c(".value", "year"),
  names_pattern = "([A-Za-z]+)(\\d+)")
df.l2 <- df |> pivot_longer(cols=starts_with(c("x", "y")),
  names_to=c(".value", "year"),
  names_pattern = "([A-Za-z]+)(\\d+)")
df.l2 <- df |> pivot_longer(cols=c(num_range("x", 1:3), num_range("y", 1:3)),
  names_to=c(".value", "year"),
  names_pattern = "([A-Za-z]+)(\\d+)")
```

The `names_to` syntax takes the second character of the variable name and saves it as the value of a new variable `year`. The `names_pattern` syntax looks for characters followed by numbers. If the variables were named `x_1`, `x_2`, etc. we could replace this with the simpler syntax `names_sep="_"`, but unfortunately `names_sep=""` doesn't work, so we have to use the difficult-to-remember regular expression syntax.

The long datasets produced by `reshape` and `pivot_longer` are, thankfully, the same, but they look different because they are in a different order. In base R, we can change the order using `df.l[order(c(df.l$id, df.l$time)),]`, or in the Tidyverse we can use `df.l2 |> arrange(year, id)`.

To move back to wide form, we can use:

```
df.l2 |> pivot_wider(names_from = "year",
```

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

Incidentally, we can use `pivot_wider` to create dummies for every level of a categorical variable:

```
mtcars |> mutate(d=1) |> pivot_wider(names_from = "cyl",
  values_from="d", values_fill=0, names_glue="cyl{cyl}")
```

This is best understood in reverse: If we had variables `cyl4`, `cyl6`, and `cyl8` in a wide form, we could pivot to a long form, and we'd have three observations per variable (one for each value of `cyl`). If we deleted the rows with `cyl==0`, we'd be back to our original dataset. The preceding code reverses this, using `values_fill=0` to restore the deleted rows, and `names_glue="cyl{cyl}"` to name the new variables.

In my opinion, this is convoluted compared to the base-R approach (here, the multiplication by one is to coerce the logical vectors to numeric):

```
d <- mtcars
vals <- seq(4, 8, by=2) # or unique(d$cyl)
d[paste0("cyl", vals)] <- lapply(vals, \(x) 1*(d$cyl==x))
```

This same basic idea can be implemented using Tidyverse commands (I don't know of a way to do this that has a `mutate` "feel"):

```
vals |> map(\(x) 1*(d$cyl == x)) |>
  set_names(paste0("cyl", vals)) |>
  bind_cols(d) |>
  relocate(num_range("cyl", vals), .after=last_col())
```

Regression tricks

Specifications

You can get all interactions between two categorical variables using:

```
lm(mpg ~ as.factor(am):as.factor(gear), mtcars)
```

You can add transformations using `I()`:

```
lm(mpg ~ wt + I(wt^2), mtcars)
```

If you want to run the same regression on multiple dependent variables, you can use either

```
lm(cbind(mpg, disp) ~ wt, mtcars)
lapply(c("mpg", "disp"), \(x) lm(paste0(x, "~ wt"), mtcars))
```

The latter approach can also be used to loop through different specification.

Predictions

After estimating a model, you can obtain the predictions using `model$fitted.values`, but you can also use `predict(model, data)`, which is useful if you want to obtain out-of-sample predictions:

```
model <- lm(mpg ~ wt, mtcars)
predict(model, mtcars[mtcars$am==0,])
```

Standard errors

The `sandwich` and `lmtest` packages are the best way that I know to get different kinds of standard errors. For example,

```
library(sandwich)
library(coeftest)
m <- lm(mpg ~ wt, mtcars)
coeftest(m, vcovHC)
coeftest(m, vcovCL(m, cluster = ~mtcars$cyl))
```

will give robust and cluster-robust standard errors (the exact form used can be changed using the `type` option).

Fixed effects

The `plm` package can be used to obtain pooled, fixed effects, and random effects estimates:

```
m.pool <- plm(y~x, data=df.l, model="pooling", index=c("id", "time"))
m.fix <- plm(y~x, data=df.l, model="within", index=c("id", "time"))
m.rand <- plm(y~x, data=df.l, model="random", index=c("id", "time"))
```

You can avoid specifying the indices by transforming the dataframe into `plm`'s format (see the help file).

`plm` works with `sandwich`, but only supports clustering on the group (or group and time):

```
coeftest(m.pool, vcovHC(m.pool, method="white1"))
coeftest(m.pool, vcovHC(m.pool, method="white1", cluster="group"))
```

If you want clustered standard errors on a higher level with `plm`, you can use the `clubSandwich` package:

```
library(clubSandwich)
coeftest(m.fix, vcov=vcovCR(m.fix, type="CR1S", cluster=df.l$gp))
```

`CR1S` reproduces Stata's standard errors.

You can also obtain fixed effects estimates using the `fixest` package, which automatically supports clustered standard errors:

```
library(fixest)
m <- feols(y ~ x | id, df.l, cluster = ~ gp)
```

`fixest` is fast, supports multiple fixed effects, and has several other useful features (type `help(fixest)` for more info).

Hypothesis tests

By default, `summary` doesn't display confidence intervals, but you can obtain them using `confint(m)`.

The `car` package can be used to test linear hypotheses:

```
m <- lm(mpg ~ wt + cyl, mtcars)
car::linearHypothesis(m, c("wt = 0", "cyl = 0"))
```

```
car::linearHypothesis(m, c("wt = cyl"))
```

Both `car` and `marginaleffects` can test nonlinear hypotheses:

```
car::deltaMethod(m, "wt^2 - cyl")
marginaleffects::hypotheses(m, "wt^2 - cyl = 0")
```

For all of the above, you can specify a variance-covariance matrix using the `vcov` option.

Instrumental variables

`ivreg` is now its own package. The basic syntax is

```
m <- ivreg(mpg ~ wt | disp, data=mtcars)
```

Note that you have to actually include the `data=` part.

Marginal effects

Both the `margins` and `marginaleffects` package can estimate marginal effects:

```
probit <- glm(am ~ gear + cyl, family=binomial(link=probit), mtcars)

# margins
probit_margins <- margins::margins(probit, vcov=vcovHC(probit))
summary(probit_margins)

# marginal effects
marginaleffects::avg_slopes(probit)
marginaleffects::slopes(probit, newdata="mean", vcov=vcovHC(probit))
```

`margins` primarily produces average marginal effects, although it can be coerced into doing marginal effects at averages. For `marginaleffects`, `avg_slopes` produces average marginal effects, and the `slopes` command can be used to obtain marginal effects at averages (as above). For both, the `vcov` option can be omitted for default standard errors.

Whether a particular estimation command is supported is hit or miss, but overall I've had more luck with `marginaleffects`.

Miscellany

- Both the `haven` and `foreign` packages can import files from other packages, including Stata. `haven` produces Tibbles, which sometimes don't interact well with other packages. You can work around this by turning a Tibble `d` into a traditional dataframe: `d <- data.frame(d)`.
- You can change variable names to lower case using `d <- rename_with(d, tolower)`.
- Suppose that for each row of a matrix, you want to extract the entry corresponding to a particular column. You can do this by indexing the first matrix by another $(n \times 2)$ matrix:

  ```
  a <- matrix(1:4, nrow=2, byrow=T)
  b <- matrix(c(1, 1, 2, 2), nrow=2, byrow=T)
  a[b]
  ```

- `%in%` allows you to select elements that meet one of a list of criteria, as in `mtcars[mtcars$cyl %in% c(2, 4),]`
- `subset` allows you to subset a dataframe by row or column, as in

  ```
  subset(mtcars, cyl==4)
  subset(mtcars, select = (c(mpg, wt)))
  ```

  One thing to note: In bracket notation, you can use `-` to *exlude* an element by its number, as in `mtcars[, -1]`, but not by its name. `subset` works with names as well, as in `subset(mtcars, select = -c(mpg))`.
- Just like you can use `within` to modify elements within a dataframe, you can use `with` to run commands using a particular dataframe (this saves you the hassle of typing `df$`):

  ```
  with(mtcars, lm(mpg ~ wt + cyl))
  ```
- `aggregate` allows you to return a scalar function by groups:

  ```
  aggregate(cbind(mpg, wt) ~ cyl, mtcars, mean)
  ```

  this is equivalent to the Tidyverse command

  ```
  mtcars |> group_by(cyl) |> summarize(across(c(mpg, wt), mean))
  ```
- `ave` allows you to add a group-specific statistic to your data, as in:

  ```
  d <- mtcars
  d$avg_mpg <- ave(d$mpg, d$cyl, FUN=mean)
  ```

  Note that it's necessary to actually type `FUN=` for some reason.

  This is equivalent to the Tidyverse command:

  ```
  mtcars |> group_by(cyl) |> mutate(avg_mpg = mean(mpg))
  ```
- `round` is useful for changing the number of digits, as in:

  ```
  round(aggregate(cbind(mpg, wt) ~ cyl, mtcars, mean), 2)
  ```
- `map("name")` will extract all elements of a nested list that are named `name`, as in:

  ```
  mtcars |> split(mtcars$cyl) |> map("mpg")
  ```

  This is equivalent to the base-R commands:

  ```
  lapply(split(mtcars, mtcars$cyl), `[[`, "mpg")
  lapply(split(mtcars, mtcars$cyl), \(x) x[["mpg"]])
  ```

  Note that `[[` refers to the operation of taking an element of a list (as in `list[["mpg"]]`. Almost any R operator can be viewed as a function (try `lapply(1:3,+, 1)`).
- `unlist` will turn a list into a vector (with the `recursive=FALSE` option, it will turn a list into a simpler list by moving sub-elements up on "level").

- R doesn't have a great built-in way to easily get descriptive statistics, but `psych::describe` and `modelsummary::datasummary_skim` are pretty good.
- The `textreg`, `stargazer`, and `modelsummary` packages are all useful for exporting estimation commands to other formats, including LaTeX.