Jenny Bryan: "Object type type 'closure' is not subsettable"

Link to the talk

0. General notes
   - She is going to refer to variables as **objects** - so are/will we! R is an *object-oriented language*.
   - Here is a mostly unrelated twitter thread about why we use `->` and/or `<-` preferentially over `=` for variable aka object assignment
1. Restart R
   - **8:13**: "…the package loaded into memory being the one that is installed on disk…". What on earth does this?! When we install, or save!, something on our computers, it gets stored permanently on the *harddrive* aka "disk" aka HD. That said, most newer computers do not have physical disks for harddrives, instead something referred to as "Solid State Drive" or SSD - it's much much faster than are disks, but we often continue to use the older terminology to refer to harddrives as disks generally speaking. Your HD is your long term memory. There is also RAM ("random access memory") which we refer to shorthand as "memory" - this is your short-term memory. (Yes both HD and RAM are memory, but whenever someone says memory, think RAM. Whenever someone says storage or disk, think HD). Generally (and loosely) speaking, whenever you open a file on your computer, the computer locates the file in hard drive, makes a copy into the memory, and then you access what's in memory. Because it can take a while for your computer to grab stuff out of the harddrive (but faster now with SSD!), your computer keeps the stuff you use most often and/or recently used in the memory (RAM) for faster access. Sometimes, what's in short-term memory is not actually the same as what's in long-term HD, so conflicts and weird behaviors emerge. This is what can happen with R packages that are installed DURING an R session - it changes what's in HD, but not what's in RAM!.
   - **8:40**: Whenever you launch a project for a second time, you probably see your work in the history (up arrow in console reveals last time's commands), and all data you had made remains in the environment. **This is R's default behavior and it can/will cause you grief**: What if something you did LAST TIME in R is sneakily causing a bug THIS TIME in R? Secretly, R stores two "hidden files": These are files whose name begins with a period, and unless your Finder settings are set to show hidden files, they normally don't appear (RStudio is set up automatically to show these files - we will talk about this more when we learn GitHub). They are `.Rhistory` (stores all commands you ran last time) and `.RData` (stores all data you defined last time). By default, when you launch a new R session, these files are loaded and you are greeted with all your old commands and your old data. Turning off the storing of `.RData` as Jenny describes will save you one day, I promise - why would you ever want data defined yesterday, a day whose work and efforts are completely gone from your brain, affecting what you are doing today?!
   - **9:10**: When Jenny brings up `R --no-save --no-restore-data`… Note: We are using R within RStudio, but it is possible to use R in other ways, like from the command line (aka terminal - "scary" black screen you see on programmers' monitors). If you have installed R locally AND you're on a UNIX system (aka Mac), you can do this too! But probably don't, it's much less pleasant.
   - **9:25**: `rm(list = ls()`. What does that mean? `rm()` is a function to REMOVE defined things from the environment. `ls()` is a function that lists ALL defined variables in your R session (try it out in console!). The entire `rm(list = ls())` is a common trick people use to remove all defined variables from the environment, but as Jenny explains, it is NO MATCH for other tricks like restarting without stored data. It doesn't get rid of everything you naively think it does! **That broom in the Environment pane actually just calls this command under the hood!**
     - **10:46**. In the slide "Which persist after `rm(list = ls()`", take some time to udnerstand what EACH OF THOSE LINES ACTUALLY DOES! This may involve some googling. Some should be familiar (A, B [does dumb bad stuff], E) and some may not be (C, D, F)!
2. Reprex
   - **Background**: What is a minimum reproducible example?
   - **More Background** here
   - **17:40** There is a package called `praise` that you should all look into. You just ask for praise and it gives you encouragement. This is all it does. It's lovely. Read more here. This site will also help to understand Jenny's code example (what are those curly braces doing?)
     - The code bug examples she is showing now are *exactly* the type of errors that prevent your code from being reproducible - i.e. running on someone else's machine, or in a later R session. Restarting R (without saving R data!) and re-running code will help you ensure your code is reproducible. (her **19:30** conclusion!)
   - **19:46**: This figure is *everything*

- **20:25**: You'll see some familiar things in this code (`dplyr`!) and some unfamiliar things (`purrr`? It's another `tidyverse` package, I'll tell you in class sometime, just remind me please.). That's cool, just go with it. But most importantly *check out those comments - they are GREAT*! Those are excellent examples of how leaving comments in code can improve anyone's understanding of what the code does at a "higher level."

3. Debug
   - **Background**: A very comprehensive look at `browser` here
   - **26:40**: For what it's worth, my answer before watching this whole talk was **D**, with some bonus components of **A** thrown in!
   - **28:00** (and **32:30**): What's a "call stack"? Think of it as, a record that R (in this case) has saved of what it did under the hood, one step at a time. I.e., what are the ordered steps that R took when executing the code? The call stack stores, in order, what R did. Error messages often show you glimpes into the stack, an overwhelming cascading list of stuff that went wrong and in what order. In a couple minutes she will introduce `traceback()` - that's this!!
   - **28:12** There is no actual function in base R called `fruit_avg` - Jenny wrote this function herself, but she isn't showing that part to us, but *you should be able to have a sense of what it does - make sure you do!!* . We will one day learn about writing our own functions too!
   - **33:11**: `rlang` is a package that is used within `tidyverse` (it is automatically loaded when you load any `tidyverse` library). It basically allowed `tidyverse` to do the magic it does. If you want to really learn more (caution: advanced!), see here.
   - **36:45**: The code that makes up the custom function `fruit_avg()` is finally revealed! Much of it is in base R (base R is actually not evil, but it's way uglier. It's also sometimes/often more powerful, just not necessarily for what we are doing in this class). That entire function could have been written using `tidyverse` code, but she chose not to. Doesn't make *any* difference! There are always many different ways to write code that accomplishes the same thing, and it often depends on personal preference.
4. Deter
   - This section of the talk introduces some **Best Practices** in coding.
   - **Tests at 41:34**: small pieces of code that SHOULD WORK and you KNOW THE ANSWER TO, and you can AUTOMATE (in package development) to run and make sure the output from that code is AS IS SHOULD BE.
   - **Assertions at 42:00**: little "checkpoints" you can put into your code that say "at this point in the code, something should be true. if it's not true, kill the code because a fatal flaw has occurred." For example, let's say a data frame needs a certain column in it for a plot you make to work properly. Before you plot, there is code you can write that will confirm: Is that column there? And if the column is missing, you will get an informative error message that YOU HAVE WRITTEN! that says "that column is missing, something went wrong! stop here and go fix it!"
   - **42:50 - 47:30 or so** gets into important concepts in package development and/or in writing large programming pipelines within version control - watch at your leisure but it won't be terribly relevant to our class. *THE PUNCHLINE YOU NEED THOUGH:* **Writing fancy code for the sake of 'it's fancy and I'm proud' is NOT ALWAYS A GOOD IDEA.** If you must, be sure to leave yourself an outrageous amount of comments.