Making Your Own Functions

Recap

- Functions are the **verbs** of programming languages
- They perform actions *on* objects
- We can adjust how the function performs by modifying **arguments**
  - arguments are like adverbs
  - ex: `na.rm = TRUE`, `paired = TRUE`, `method = "spearman"`

Who -- Who Makes Functions?

- R is open source -- this means that anyone anywhere can make packages that contain functions and publish them
- **THAT MEANS YOU!! YOU CAN DO THIS, TOO!**

When You Should Make A Function?

- You need to do the same thing many times. *Make your life easier!*
- Sounds a lot like a `for loop`, right?
    - But `functions` are particularly helpful when you know you'll need to do this exact same thing again, *especially on a different dataset*!
- When you write a `for loop`, you'll write it specific to your dataset. When you write a `function`, it's better to write it in a **general** format so that you can fill in other datasets/columns/inputs that are not specific to your current dataset

The Process

- First you need to define your function
- Then you need to make sure your function is in your Global Environment
  - Usually people write their functions as `.R` script files
  - The file is saved somewhere they can access (in a project directory)
  - In your analysis file (could be `.R` or `.Rmd`), you call your function's `.R` file
- Then you can use it exactly like you would a function from any other package

Defining Your Function

```
nameOfFunction <- function(input) {
  do something
  return(whatever you want the output to be)
}
```

- Use the keyword `function`
- Curly brackets
- Use the `return()` function to make sure that the output contains what you want

Simple Example

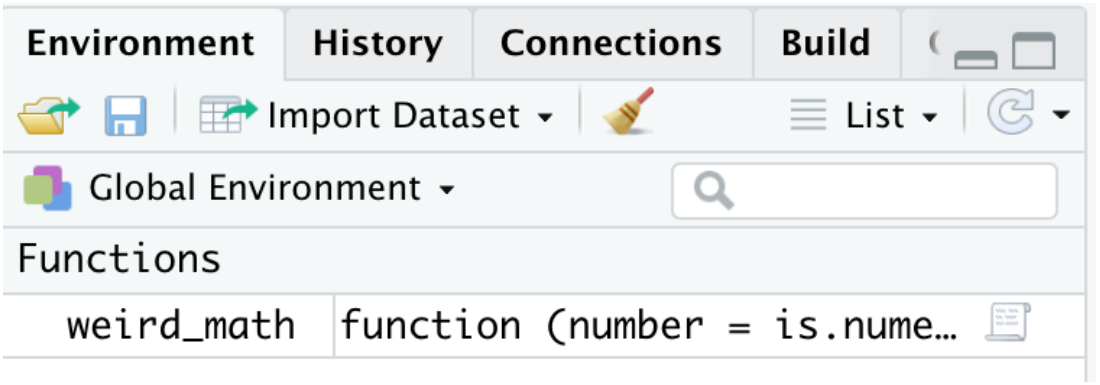
Let's say you want a function that takes a number, squares it, and then multiplies by 100.

```
weird_math <- function(number = is.numeric) {

  num_squared <- number^2
  num_squared_100 <- num_squared * 100

  return(num_squared_100)

}
```



Now let's use our function!

```
weird_math(number = 6)
```

```
## [1] 3600
```

If we want to store the output of our function...

```
newValue <- weird_math(number = 6)
newValue
```

```
## [1] 3600
```

If we wanted to store the output for a vector of numbers...

```
vectorExample <- weird_math(number = 1:10)
vectorExample
```

```
##  [1]   100   400   900  1600  2500  3600  4900  6400  8100 10000
```

Some Tips

1. For your input, I suggest being as specific as possible for the *names* of arguments. You want your users (aka you in 9 months) to easily figure out what goes in each argument.

2. You want to think carefully about the data class for the inputs. You don't technically need to specify it (with something like `is.numeric`), but you should think very carefully about what you want it to be. How do you anticipate someone using it?

3. Think carefully about what you want your output to be! Do you want it to be a single number? Do you want to put together a data.frame and return the data.frame? Do you want a list? Etc. You have to think through things!

4. Everything we've talked about in this class when it comes to code you can use in a `function`. For instance, you can put a `for loop` within a `function`!

Another Example

The AX-CPT is a cognitive control task. It has 4 different trial types:

- AX = the target trial (what we tell participants to be on the lookout for)
- AY = a challenging trial type; correct cue, incorrect probe
- BX = a challenging trial type; incorrect cue, correct probe
- BY = the baseline trial; incorrect cue, incorrect probe

Each participant has a reaction time and accuracy measure for each of these trial types. There are a number of various derived measures we can calculate that are more useful than just looking at these individual trial types. For example:

Proactive Behavioral Index

This computes a normalized difference score of performance distinctions between AY and BX trial types (i.e., [AY-BX]/[AY+BX]). It can be computed on both RT and accuracy measures as well as their sum. This measure has often been computed in the AX-CPT (Braver et al, 2009). The index measure is predicted to increase in the Proactive condition. However, it is not clear yet whether there is a prediction to be made for the Reactive condition.

PBI Example

```
axcpt
```

```
##    Participant ax_acc ay_acc bx_acc by_acc
## 1          ID1    0.8    0.6    0.8    0.9
## 2          ID2    0.6    0.5    0.5    0.8
## 3          ID3    0.4    0.2    0.7    0.8
## 4          ID4    0.8    0.9    0.5    1.0
## 5          ID5    0.7    0.7    0.6    0.9
## 6          ID6    0.9    0.8    0.7    1.0
## 7          ID7    0.4    0.5    0.5    0.7
## 8          ID8    0.5    0.5    0.6    0.8
## 9          ID9    0.6    0.6    0.7    0.8
## 10        ID10    0.9    0.7    0.8    1.0
```

PBI Example

So let's make a function to calculate the PBI so that if you run the AX-CPT on a new dataset (maybe you collect data on a different population?), you don't need to re-code everything.

```
pbi <- function(ay,bx){
  pbi <- (ay-bx)/(ay+bx)
  pbi <- round(x = pbi, digits = 2)
  return(pbi)
}

# now run the function!
pbiValues <- pbi(ay = axcpt$ay_acc, bx = axcpt$bx_acc)
pbiValues
```

```
##  [1] -0.14  0.00 -0.56  0.29  0.08  0.07  0.00 -0.09 -0.08 -0.07
```

One of my actual functions

This is the code I use to calculate grades when I teach Biological Psychology (yay neuro things!)

```
##     Student Exam1 Exam2 Exam3 Exam4  Exam5 participationFinal
## 1 Student 1  96.5 105.0  97.5  98.5  97.50          104.16667
## 2 Student 2  75.5  78.0  90.5  92.0  96.00          104.16667
## 3 Student 3  49.5  74.0  90.5  43.0  92.50           79.16667
## 4 Student 4  92.5  88.0  99.0  97.0  98.00          104.16667
## 5 Student 5  90.0  91.0  99.0  87.0  94.00          104.16667
## 6 Student 6  81.5  80.0  98.5  97.0  97.00          104.16667
## 7 Student 7  85.0  63.5  91.5  83.0  90.00          104.16667
## 8 Student 8  92.0 105.0  98.0 100.0 100.00           95.83333
## 9 Student 9  84.5  88.0  97.5  97.0  91.75           46.66667
```

First I'm going to read in the `.R` file that has just the function in it. We use `source()` instead of `read.csv()`

```
source(here::here("R", "gradingFunction.R"))
```

Note you should ignore the `here::here` part of this. In your own work, you'll simply point a working directory to wherever the function is located

One of my actual functions

```
gradeFx = function(classScores) {
  # get the participation grade, store for later
  participation = unlist(classScores[i,7], use.names = FALSE)

  # Sort
  currentGrades = sort(unlist(classScores[i,2:6], use.names = FALSE))

  # now add in participation to end of vector
  currentGrades = c(currentGrades, participation)

  # lowest exam grade = 10%
  # 4 other exams = 20%
  # participation = 10%
  weights = c(.1, .2, .2, .2, .2, .1)

  # get their grade!
  grade = round(sum(weights*currentGrades), digits = 1)

  return(grade)
}
```

Notice how there is an `i` involved?

One of my actual functions

Now I'm going to use this function *within* a `for loop`!

"For each row in `biopsych`, apply the `gradeFx` function that I made"

```
for (i in 1:nrow(biopsych)) {
  biopsych$finalGrade[i] = gradeFx(classScores = biopsych)
}

biopsych
```

```
##      Student Exam1 Exam2 Exam3 Exam4  Exam5 participationFinal finalGrade
## 1 Student 1  96.5 105.0  97.5  98.5  97.50          104.16667       99.8
## 2 Student 2  75.5  78.0  90.5  92.0  96.00          104.16667       89.3
## 3 Student 3  49.5  74.0  90.5  43.0  92.50           79.16667       73.5
## 4 Student 4  92.5  88.0  99.0  97.0  98.00          104.16667       96.5
## 5 Student 5  90.0  91.0  99.0  87.0  94.00          104.16667       93.9
## 6 Student 6  81.5  80.0  98.5  97.0  97.00          104.16667       93.2
## 7 Student 7  85.0  63.5  91.5  83.0  90.00          104.16667       86.7
## 8 Student 8  92.0 105.0  98.0 100.0 100.00           95.83333       99.4
## 9 Student 9  84.5  88.0  97.5  97.0  91.75           46.66667       88.0
```

What to make of all of this

You don't actually need to know what is happening in my grading function exactly. Here are the points to take home:

- `functions` and `for loops` should both make your life easier. They are a pain to get up and running, but once they are, they save you a TON of time.
- You can put a `function` that you made within a `for loop`. You can also put a `for loop` within a function. For example, you can make a function just for bootstrapping confidence intervals if you wanted.
- `functions` are especially helpful when you know you'll want to do the same thing on multiple datasets.
- Your flexibility is endless. The most important thing to do is to really consider what it is you have and what it is you *want* to do. Thinking through these problems is, **by far**, the hardest part of coding.