

Stat 260, Lecture 10, Programming, Pipes and Functions

Brad McNeney

Load packages and datasets

```
library(tidyverse) # loads the %>% pipe
```

Reading

- ▶ Programming overview, Pipes and Functions: Intro to Part II, and Chapters 14 and 15 of the printed text, Chapters 17 to 19 of online text.
- ▶ Optional: Wickham (2014), Advanced R, Chapter 6, [<http://adv-r.had.co.nz/Functions.html>]

Programming in R

- ▶ We have already been programming in R, mostly by writing chunks of code that use tidyverse functions to do data visualization or wrangling.
- ▶ Now we discuss strategies for making our code easier to read and less prone to errors.
- ▶ One useful tool for this purpose is the forward pipe, `%>%`, which we'll discuss in more detail.
- ▶ A principle in programming is “Do not Repeat Yourself (DRY)”, and writing functions can help us stay DRY.

Pipes

- ▶ The forward pipe `%>%` is implemented in the `magrittr` package, and is loaded when we load the `tidyverse`.
- ▶ Forward pipes are useful for combining a linear sequence of data processing steps, when we won't need the intermediate steps.
 - ▶ Tidyverse functions are typically named as actions, or verbs, and a linear sequence of such actions reads like a sentence.

We can use pipes too much

- ▶ However, we can take the idea too far and make the code difficult to debug.
- ▶ **Exercise** Debug the following code

```
read_csv("API_ILO_country_YU.csv") %>%  
  gather(year, `Unemployment Rate`, `2010`:`2014`, convert=TRUE) %>%  
  filter(str_detect(`Country Name`, " \\(IDA.*\\)")) %>%  
  mutate(`Country Name` =  
    str_replace(`Country Name`, "", "\\(IDA.*\\)")) %>%  
  select(-`Country Code`) %>%  
  ggplot(aes(x=year, y=`Unemployment Rate`, color=`Country Name`)) %>%  
  + geom_point() + geom_line()
```

Pipes are not good when there are multiple inputs

- ▶ Code may involve parallel computations that are assembled at the end.
- ▶ For example, suppose you need to read in two tibbles, manipulate each with actions like filter/gather/mutate, and then join them together.
 - ▶ We should save the tibbles to intermediate objects before joining them.

Other Tools from magrittr

- ▶ See the text's section – I don't use them myself
- ▶ Note: `%%` is similar to the base R function `with()`

```
library(magrittr)
mtcars %$% cor(displacement,mpg)
```

```
## [1] -0.8475514
```

```
with(mtcars,cor(displacement,mpg))
```

```
## [1] -0.8475514
```

R functions – overview

- ▶ Encapsulating code in a function has several advantages:
 - ▶ can be used multiple times on different inputs
 - ▶ can compartmentalize computations and give them a name
 - ▶ can help you break a complicated task down into more manageable pieces.
- ▶ We will discuss:
 - ▶ when to write a function,
 - ▶ components of a function
 - ▶ writing pipeable functions,

When to write a function

- ▶ If you find yourself cutting and pasting the same code multiple times (more than twice, according to the text), then you should consider writing a function.
- ▶ See text for one example. Here is another.
- ▶ The Boston dataset in the MASS package includes data on house prices (`medv`) and characteristics of different neighborhoods in Boston [<https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/Boston.html>]
- ▶ Certain kinds of statistical analyses of the relationship between `medv` and the other variables require that the other variables be standardized, by subtracting the mean values and dividing by the SD.
- ▶ I have made the dataset available in our Notes folder with the following

```
library(MASS)
write_csv(Boston, file="Boston.csv")
```

Boston dataset

```
Boston <- read_csv("Boston.csv")  
dim(Boston)
```

```
## [1] 506 14
```

```
Boston
```

```
## # A tibble: 506 x 14
```

```
##      crim    zn indus  chas   nox    rm   age   dis   rad   tax ptratio  
##      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0.00632   18    2.31    0 0.538  6.58  65.2  4.09    1   296   15.3  
## 2 0.0273    0    7.07    0 0.469  6.42  78.9  4.97    2   242   17.8  
## 3 0.0273    0    7.07    0 0.469  7.18  61.1  4.97    2   242   17.8  
## 4 0.0324    0    2.18    0 0.458  7.00  45.8  6.06    3   222   18.7  
## 5 0.0690    0    2.18    0 0.458  7.15  54.2  6.06    3   222   18.7  
## 6 0.0298    0    2.18    0 0.458  6.43  58.7  6.06    3   222   18.7  
## 7 0.0883   12.5   7.87    0 0.524  6.01  66.6  5.56    5   311   15.2  
## 8 0.145    12.5   7.87    0 0.524  6.17  96.1  5.95    5   311   15.2  
## 9 0.211    12.5   7.87    0 0.524  5.63  100   6.08    5   311   15.2  
## 10 0.170   12.5   7.87    0 0.524  6.00  85.9  6.59    5   311   15.2  
## # ... with 496 more rows, and 3 more variables: black <dbl>, lstat <dbl>,  
## #   medv <dbl>
```

Standardize columns of Boston

- ▶ You can standardize the first column of Boston with

```
Boston$crim <- (Boston$crim - mean(Boston$crim, na.rm=TRUE))/  
sd(Boston$crim, na.rm=TRUE)
```

- ▶ Now cut-and-paste 12 times to standardize the remaining predictors of medv

```
Boston$zn <- (Boston$zn - mean(Boston$zn, na.rm=TRUE))/  
sd(Boston$zn, na.rm=TRUE)  
Boston$indus <- (Boston$indus - mean(Boston$indus, na.rm=TRUE))/  
sd(Boston$indus, na.rm=TRUE)  
# Etc.
```

- ▶ Not only is this tedious, it is error-prone. Plus, we will need to do the same operation on other datasets.

A standardization function

- ▶ The following function standardizes a vector.
 - ▶ We'll learn more about the components of a function in the slides to follow.

```
standardize <- function(x) {  
  (x - mean(x,na.rm=TRUE))/sd(x,na.rm=TRUE)  
}  
Boston$crim <- standardize(Boston$crim)  
Boston$zn <- standardize(Boston$zn)  
# Etc
```

- ▶ This has reduced the amount of code and chances for cut-and-paste errors.

Components of a function

- ▶ In R, functions are objects with three essential components:
 - ▶ the code inside the function, or `body`,
 - ▶ the list of arguments to the function, and
 - ▶ a data structure called an `environment` which is like a map to the memory locations of all objects defined in the function.

Example function

```
f <- function(x) {  
  x^2  
}  
f
```

```
## function(x) {  
##   x^2  
## }
```


The function arguments

- ▶ These are the arguments to the function.
- ▶ Function arguments can have default values, as in:

```
f <- function(x=0) { x^2 }
```

- ▶ **Exercise** Re-write our `standardize()` function to take an argument `na.rm`. Set this argument to `TRUE` by default.

Argument defaults

- ▶ Argument defaults can be defined in terms of other arguments:

```
f <- function(x=0,y=3*x) { x^2 + y^2 }  
f()
```

```
## [1] 0
```

```
f(x=1)
```

```
## [1] 10
```

```
f(y=1)
```

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

Argument matching

- ▶ When you call a function, the arguments are matched first by name, then by “prefix” matching and finally by position:

```
f <- function(firstarg,secondarg) {  
  firstarg^2 + secondarg  
}  
f(firstarg=1,secondarg=2)
```

```
## [1] 3
```

```
f(s=2,f=1)
```

```
## [1] 3
```

```
f(2,f=1)
```

```
## [1] 3
```

```
f(1,2)
```

```
## [1] 3
```

The function body

- ▶ This is the code we want to execute.
- ▶ When the end of a function is reached, the value of the last line is returned.
 - ▶ If you prefer, you can end a function with `return()` to signal the function's returned value.

```
f <- function(x=0) { x^2}  
f <- function(x=0) { return(x^2)}
```

Control Flow

- ▶ Code within a function is not always executed linearly from start to end.
- ▶ We may need to execute different code chunks depending on the function inputs.
- ▶ We may need to repeat certain calculations, or loop.
- ▶ Such constructs are called control flow.
- ▶ We'll touch on some of the basics.

if and if-else

- ▶ if tests a condition and executes code if the condition is true. Optionally, can couple with an else to specify code to execute when condition is false.

```
if("cat" == "dog") {  
  print("cat is dog")  
} else {  
  print("cat is not dog")  
}
```

```
## [1] "cat is not dog"
```

Conditions require TRUE or FALSE

- If not, you will get a warning or error.

```
if(c(TRUE,FALSE)) {} # Throws the following warning:  
#Warning message:  
#In if (c(TRUE, FALSE)) { :  
# the condition has length > 1 and only the first element will be used  
if(NA) {} # Throws the following error:  
#Error in if (NA) { : missing value where TRUE/FALSE needed
```

for loops

- ▶ Example: In the following, `seq_along(nreps)` is the “index set”.

```
n <- 10; nreps <- 100; x <- vector(mode="numeric",length=nreps)
for(i in seq_along(nreps)) {
  x[i] <- mean(rnorm(n))
}
summary(x)
```

```
##      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
## -0.200456  0.000000  0.000000 -0.002005  0.000000  0.000000
```

```
print(i)
```

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

- ▶ **Exercise** Write a function `standardize_tibble()` that loops through the columns of a tibble and standardizes each with your `standardize()` function. Hints: If `tt` is a tibble, `ncol(tt)` is the number of columns, and `seq_along(ncol(tt))` is an appropriate index set. If `tt` is a tibble, `tt[[1]]` is the first column.

for loop index set

- ▶ Index sets of the form `seq_along(n)` are most common, but can be almost any vector.

```
ind <- c("cat", "dog", "mouse")
for(i in ind) {
  print(paste("There is a", i, "in my house"))
}
```

```
## [1] "There is a cat in my house"
## [1] "There is a dog in my house"
## [1] "There is a mouse in my house"
```

- ▶ We'll learn more about this kind of iterating in the chapter **Iteration with purrr**.

while loops

- ▶ Use a while loop when you want to continue until some logical condition is met.

```
set.seed(1)
# Number of coin tosses until first success (geometric distn)
p <- 0.1; counter <- 0; success <- FALSE
while(!success) {
  success <- as.logical(rbinom(n=1,size=1,prob=p))
  counter <- counter + 1
}
counter
```

```
## [1] 4
```

break

- ▶ break can be used to break out of a for or while loop.

```
for(i in seq_along(100)) {  
  if(i>3) break  
  print(i)  
}
```

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

The function environment

- ▶ The environment within a function is like a map to the memory locations of all its variables.
- ▶ The function arguments are “passed by value”, meaning that a copy is made and stored in the function’s environment.
- ▶ Variables created within the function are also store in its environment

```
f <- function(x) {  
  y <- x^2  
  ee <- environment() # Returns ID of environment w/in f  
  print(ls(ee)) # list objects in ee  
  ee  
}  
f(1) # function call
```

```
## [1] "ee" "x"  "y"
```

```
## <environment: 0x7fb7bdebd7e8>
```

Enclosing environments

- ▶ Our function `f` was defined in the global environment, `.GlobalEnv`, which “encloses” the environment within `f`.
- ▶ If `f` needs a variable and can't find it within `f`'s environment, it will look for it in the enclosing environment, and then the enclosing environment of `.GlobalEnv`, and so on.
- ▶ The `search()` function lists the hierarchy of environments that enclose `.GlobalEnv`.

```
search()
```

```
## [1] ".GlobalEnv"      "package:magrittr" "package:forcats"
## [4] "package:stringr" "package:dplyr"    "package:purrr"
## [7] "package:readr"   "package:tidyr"    "package:tibble"
## [10] "package:ggplot2" "package:tidyverse" "package:stats"
## [13] "package:graphics" "package:grDevices" "package:utils"
## [16] "package:datasets" "package:methods"   "Autoloads"
## [19] "package:base"
```

- ▶ To facilitate this search, each environment includes a pointer to its enclosing environment.

Exercise

- The following two functions are defined in a user's global environment. Where do they look for the object `y`?

```
f1 <- function(x) {  
  y <- 15  
  x*y  
}  
f2 <- function(x) {  
  x*y  
}
```

Writing readable code

- ▶ The more your code does, the harder it is for others to read.
 - ▶ Here “others” includes you some time in the future.
 - ▶ Wickham says we should write code that future-you can understand, because past-you doesn't answer emails.
- ▶ See the **Functions are for Humans and Computers** section of text for tips on writing readable code.

Other reasons to write functions

- ▶ Functions can be used to prevent repetition, but even if used only once they can improve code readability.
 - ▶ For example, you are writing a function `func()` that computes a statistic `mystat` that takes 10 lines of code to calculate.
 - ▶ The rest of your function is only 5 lines.
 - ▶ Write a function called `mystat()` and call it from `func()`.
 - ▶ If you define `func()` first, it will be easier to document `mystat()`.
- ▶ Writing code in a top-down way is like writing an outline for an essay and then filling in the details.
 - ▶ The main function is the outline.
 - ▶ the sub-functions are the details of each topic.

Exercise

- ▶ Create an R script that first defines `standardize_tibble()` and then `standardize()`. In `standardize()`, replace `mean()` by a function `center()` and `sd()` by a function `spread()`, where `center()` and `spread()` are functions that you write to compute the mean and SD using only the `sum()` function. `center()` and `spread()` should remove missing values by default.

R packages

- ▶ Use the `library()` command to load packages.
- ▶ When we load a package it is inserted in position 2 of the search list, just after `.GlobalEnv`.

```
# install.packages("hapassoc")  
library(hapassoc)  
search()
```

```
## [1] ".GlobalEnv"      "package:hapassoc"  "package:magrittr"  
## [4] "package:forcats"  "package:stringr"   "package:dplyr"  
## [7] "package:purrr"    "package:readr"     "package:tidyr"  
## [10] "package:tibble"   "package:ggplot2"   "package:tidyverse"  
## [13] "package:stats"    "package:graphics"  "package:grDevices"  
## [16] "package:utils"    "package:datasets"  "package:methods"  
## [19] "Autoloads"        "package:base"
```

Detaching packages

- ▶ Detach a package from the search list with `detach()`

```
detach("package:hapassoc")  
search()
```

```
## [1] ".GlobalEnv"          "package:magrittr"    "package:forcats"  
## [4] "package:stringr"     "package:dplyr"       "package:purrr"  
## [7] "package:readr"       "package:tidyr"       "package:tibble"  
## [10] "package:ggplot2"     "package:tidyverse"   "package:stats"  
## [13] "package:graphics"    "package:grDevices"   "package:utils"  
## [16] "package:datasets"    "package:methods"     "Autoloads"  
## [19] "package:base"
```

Package namespaces

- ▶ Package authors create a list of objects that will be visible to users when the package is loaded. This list is called the package namespace.
- ▶ You can access functions in a package's namespace without loading the package using the `::` operator.

```
set.seed(321)
n<-30; x<-(1:n)/n; y<-rnorm(n,mean=x); ff<-lm(y~x)
car::sigmaHat(ff)
```

```
## [1] 0.926726
```

- ▶ Doing so does not add the package to the search list.