R4DS

Hennie Bezuidenhout

03 October 2016

# Programming in R Studio - Pipes

## The point of the **Pipe** is to help us read our code in an easy to understand way!!

### magrittr package - but all packages in tidyverse automatiically make

When are we better off not using the *Pipe* ?

1. When **Pipes** are longer than ten steps (Rather create intermediate objects with meaningful names)
2. We have multiple *inputs* or *outputs* (the same solution as above applies)
3. When we are starting to think about a *directed graph* with a complex dependency structure

Remember that the%T>%can assist us with more complex *Pipes* -Effectively, the **T-Pipe** returns the left-hand side instead of the right-hand side incase we want to plot or print

# Introducing Functions

## Allows us to automate common tasks in a more powerful and general way than copy-and-pasting



We do not copy and paste

What are the advantages of using *functions* ?

* You can give a function an evocative name that makes your code easier to understand.
* As requirements change, you only need to update code in one place, instead of many.
* You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another)

##### We should seriously consider writing a function whenever we have copied and pasted a block of code more than **TWICE**

There are 3 steps to creating a new *function* ?

* You need to pick a name for the function
* You list the inputs, or arguments, to the function inside function
* You place the code you have developed in **body** of the function

rescale01 <- function(x) { rng <- range(x, na.rm = TRUE) (x - rng[1]) / (rng[2] - rng[1]) }

Most important attribute of functions: \*\* WE DO NOT WANT TO REPEAT OURSELVES\*\*

## Conditional Execution

An if statement allows us to conditionally execute code

if (condition) {  
# code executed when the condition is \*\*TRUE\*\*  
}  
else {  
# code executed when the condition is \*\*FALSE\*\*  
}

The condition must evaluate to TRUE or FALSE

Both if and function should (almost) always be followed up by squiggly brackets, except for very short easy to understand if statements

## Function Arguments

The arguments to a function typically fall into two broad sets: one set supplies the **DATA** to compute on, and the other supplies arguments that control the ***details*** of the computation

Generally, data arguments should come first, and detail arguments should go on the end *and usually should have default arguments*

##### Simple Example

mean\_ci <- function(x, conf = 0.95) { >se <- sd(x) / sqrt(length(x)) **(This is DATA)** alpha <- 1 - conf **(This is DATA)** mean(x) + se \* qnorm(c(alpha / 2, 1 - alpha / 2)) (This is detailed arguments to manupilate the DATA) }

# Vectors

There are 2 main types of Vectors

* **Atomic Vectors** (Logical, Integer, Double, Character, Complex and Raw)
* **Lists**

*I feel that in dlyr we will work primarily with Tibbles which automatically recognizes types of Vectors upon reading the Files*

# Iteration

Iteration addresses duplication.Why ?

* It is easier to see the intend of your code
* It is easier to respond to changes in requirements
* We are likely to have fewer bugs because each line of code is used in more places

**Iteration** helps us to do the same thing to multiple inputs: Repeating the same operation on different columns, or on different datasets.

*How will we be able to get the mean for each of the below columns ?*

library(knitr)  
df <- tibble::tibble(  
 a = rnorm(10),  
 b = rnorm(10),  
 c = rnorm(10),  
 d = rnorm(10)  
)  
kable(df)

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | c | d |
| -1.6383542 | 0.1217940 | -0.0419457 | 0.3658106 |
| 0.9917541 | 0.3576189 | -1.1547951 | 1.2888813 |
| 0.1054715 | -0.2619808 | 1.7871795 | -0.1022025 |
| -0.9578962 | -1.7925068 | -0.5559031 | -1.3615441 |
| -0.4201849 | -2.0460951 | -0.9218160 | -0.8816309 |
| 0.2659615 | -0.0382365 | -2.4799948 | -0.8622424 |
| 1.5848295 | -1.0804535 | 0.7862585 | 1.5468027 |
| 0.1364809 | -0.3064728 | 0.8409765 | -0.1167895 |
| -1.4169326 | 1.4038942 | -0.8092626 | -2.5429623 |
| 0.6868489 | -0.4678301 | -0.3828218 | 0.9965809 |

output <- vector("double", ncol(df)) # 1. output  
for (i in seq\_along(df)) { # 2. sequence  
 output[[i]] <- median(df[[i]]) # 3. body  
}  
output  
#> [1] -0.2458 -0.2873 -0.0567 0.1443

Every loop has 3 components:

* The OUTPUT:output <- vector("double", ncol(df)) Before we start the loop, we must always allocate sufficient space for the output (This increases the speed at which the loop will be able to operate)
* The SEQUENCE: for (i in seq\_along(df)). This determines what to loop over. Each rum of the for loop will assign *i* to a different value from seq\_along(df)
* the BODY: output[[i]] <- median(df[[i] This is the code that does the work. It runs repeatedly, each time with a different value for *i*. The first iteration will run output[[1]] <- median(df[[1]]), the second will run output[[2]] <- median(df[[2]]), and so on.

##### Exercise Exercise Exercise Exercise

*mtcars is a preloaded dataset in R*

* Let us compute the mean for this dataset via looping across the respective columns

View(mtcars)  
  
firstever <- vector("double", ncol(mtcars))  
for (i in seq\_along(mtcars)) {  
   
 firstever[[i]] <- median(mtcars[[i]])  
}  
  
## It would be nice if I manage to show the Results in a Table format