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 |
| 0.1576175 | 1.5280674 | -0.3392919 | -0.7797182 |
| -0.9308839 | 0.1896681 | 0.0471822 | -0.5003695 |
| -1.1076444 | 0.4516419 | -0.9142371 | -0.0775462 |
| 1.5329450 | 0.4816699 | 1.8539797 | 1.1898770 |
| 0.2291775 | 1.1230929 | -0.3802082 | -0.6512980 |
| 0.5493024 | -0.8338630 | -0.0845228 | 0.5314601 |
| 0.6376600 | -1.2235029 | 0.9364001 | 0.1883663 |
| 0.9596642 | -0.1679399 | -0.4437989 | -0.1874724 |
| -0.2903132 | 0.9122210 | 1.9036251 | 1.9919491 |
| -1.2546871 | 0.0573408 | 1.0396013 | -0.6012974 |

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]])  
}  
  
firstever

## [1] 19.200 6.000 196.300 123.000 3.695 3.325 17.710 0.000  
## [9] 0.000 4.000 2.000

## It would be nice if I manage to show the Results in a Table format

*Iris is a preloaded dataset in R*

-Let us compete the number of unique observations in every column

View(iris)  
  
uniquely <- vector("integer",ncol(iris))  
for (i in seq\_along(iris)) {  
   
 uniquely[[i]] <- length(unique(iris[[i]]))  
   
}  
  
uniquely

## [1] 35 23 43 22 3

## I like this

There are four variations to the basic for loop

* Modifying an existing object, instead of creating a new object
* Looping over names or values, instead of indices
* Handling outputs of unknown length
* Handling sequences of unknown length

We need to analyse the while loop in more detail. I am not entirely sure how often we will be using it, but the R4DS only briefly touches on it. **I've openend up an ISSUE for this**

### For loops vs. functionals

For loops are not as important in R as they are in other languages because R is a functional programming language. *This means that it is possible to wrap up for loops in a function, and CALL that function instead of using the for loop directly*

One of the behaviours that makes R a functional programming language is the idea of passing a function to another function.

f1 <- function(x) abs(x - mean(x)) ^ 1  
f2 <- function(x) abs(x - mean(x)) ^ 2  
f3 <- function(x) abs(x - mean(x)) ^ 3

##### This can be rewritten as:

f <- function(x, i) abs(x - mean(x)) ^ i

##### Similarly:

col\_mean <- function(df) {  
 output <- vector("double", length(df))  
 for (i in seq\_along(df)) {  
 output[i] <- mean(df[[i]])  
 }  
 output  
}  
  
col\_median <- function(df) {  
 output <- vector("double", length(df))  
 for (i in seq\_along(df)) {  
 output[i] <- median(df[[i]])  
 }  
 output  
}  
  
  
col\_sd <- function(df) {  
 output <- vector("double", length(df))  
 for (i in seq\_along(df)) {  
 output[i] <- sd(df[[i]])  
 }  
 output  
}

##### This can be rewritten as:

col\_summary <- function(df, fun) {  
 out <- vector("double", length(df))  
 for (i in seq\_along(df)) {  
 out[i] <- fun(df[[i]])  
 }  
 out  
}

#### In which case we can simply call the function and specify the "calculation" needed going forward:

col\_summary(df, median)  
#> [1] 0.237 -0.218 0.254 -0.133  
col\_summary(df, mean)  
#> [1] 0.2026 -0.2068 0.1275 -0.0917

Let us next discuss purr package which provides functions that eliminate the need for many common loops. The apply family of functions in base R solve a similar problem, but purr is more consistent and thus easier to learn.

The goal of using purrr functions instead of for loops is to allow you break common list manipulation challenges into independent pieces:

* How can you solve the problem for a single element of the list? Once you’ve solved that problem, purrr takes care of generalising your solution to every element in the list.
* If you’re solving a complex problem, how can you break it down into bite-sized pieces that allow you to advance one small step towards a solution? With purrr, you get lots of small pieces that you can compose together with the pipe.

### The MAP Functions

-map() makes a list -map\_lgl() makes a logical vector -map\_int() makes an integer vector -map\_dbl() makes a double vector -map\_chr() makes a character vector

Each function takes a vector as input, applies a function to each piece, and then returns a *new vector* that’s the same length (and has the same names) as the input. The type of the vector is determined by the **suffix** to the map function.

*Once you master these functions, you’ll find it takes much less time to solve iteration problems. But you should never feel bad about using a for loop instead of a map function. The map functions are a step up a tower of abstraction, and it can take a long time to get your head around how they work. The important thing is that you solve the problem that you’re working on, not write the most concise and elegant code (although that’s definitely something you want to strive towards!).*

### This is incredible!!!

library(purrr)  
  
  
map\_dbl(mtcars, mean)

## mpg cyl disp hp drat wt   
## 20.090625 6.187500 230.721875 146.687500 3.596563 3.217250   
## qsec vs am gear carb   
## 17.848750 0.437500 0.406250 3.687500 2.812500

MyIllusiveTable <- map\_dbl(mtcars, mean)  
  
print(MyIllusiveTable)

## mpg cyl disp hp drat wt   
## 20.090625 6.187500 230.721875 146.687500 3.596563 3.217250   
## qsec vs am gear carb   
## 17.848750 0.437500 0.406250 3.687500 2.812500