Best Coding Practices for R

Vikram Singh Rawat

2021-03-26

Contents

Part I

Introduction

```
knitr::opts_chunk$set(include = TRUE)

library(magrittr)
library(quanteda)
```

```
## Warning: package 'quanteda' was built under R version 4.0.3

## Package version: 2.1.2

## Parallel computing: 2 of 8 threads used.

## See https://quanteda.io for tutorials and examples.

##
## Attaching package: 'quanteda'

## The following object is masked from 'package:utils':
##
##     View
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

CoverPage

Know the rules well, so you can break them effectively. — The Dalai Lama

A COLLECTION OF TIPS FOR USING R AS A PROGRAMMING LANGUAGE

BEST CODING PRACTICES FOR R

The Most Damaging Phrase in any language is: ' IT'S ALWAYS BEEN DONE THAT WAY'.

BY GRACE HOPPER

Chapter 1

Introduction

- The most damaging phrase in the language is: 'It's always been done that way.*.

— Grace Hopper

Did you try to read the title from the cover of the book? I could have done a thousand things to make it easier for you to read it. But this cover reminds me of how often we overlook simple things which are very crucial from the reader's point of view.

R is an excellent programming language it's turing complete and doesn't lack anything for a production level code. It can be used in the entire data domain from API's to dashboards to apps and much more. Trust the language and trust in yourself. It's a journey everyone has gone through and everyone must go through.

R programmers have a bad reputation for not writing production level code. It stems from the fact that we mostly aren't trained programmers. We tend to overlook things that are crucial from a programming standpoint. As R programmers we are often less inclined to write the code for production. Mostly we try to write scripts and when we are asked to deploy the same we just wrap it in a function and provide it to the IT team. I have been at the receiving end of these issues where I had to maintain a poorly written code; **columns were referred by numbers, functions were dependent upon global environment variables, 50+ lines functions without arguement, poor naming conventions etc...**.

I too am a self taught programmer and have gone through these hiccups of code deployment, code reviews and speed issues. World going forward will all be code and data. The sooner you learn these skills the better it is for you to have trust in your own programming skills. R is a huge language and I would like to share the little knowledge I have in the subject. I don't claim to be an expert but this book will guide you in the right path wherever possible.

Most of the books about R programming language will tell you what are the possible ways to do one thing in R. This book will only tell you one way to do that thing correctly.

I will try to write it as a dictionary as succinctly as possible. So that you can use it for references. Let the journey begin...

Part II

Structure

Chapter 2

Folder Structure

2.1 Organizing files

The best way to organize your code is to write a **package**.

Organizing your code is the first and foremost thing you should learn. Because as the project grows and multiple files are put into a folder it gets harder to navigate the code. A proper folder structure definitely helps in these times. I Just couldn't emphasis it enough that best way to organize your code is to write a package. But even when you are not planning to write a package. There are best practices to make it readable and make a smooth navigation.

2.2 Create Projects

It's such a minor thing to say but I still till date see code like this:

```
setwd("c://myproject_name/")
```

It was a good practice like 5 - 6 years ago. Now Rstudio has a feature to create project.

Once you create a project it is easier to manage your files and folders and it's easier to give it somebody as well. It has virtually the same effect but then you can use Rstudio a little better. It's something I recommend to every user regardless of the skill level.

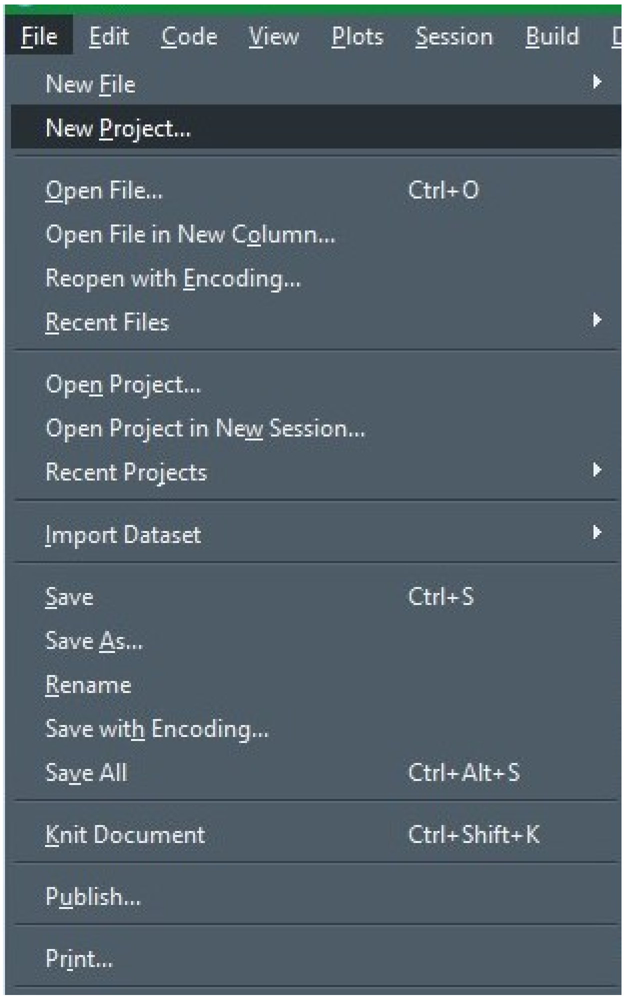


Figure 2.1: new project

2.3 Naming files

I data science most common problem is that we don't change the file names of excel or csv files provided by business people. And most of the time those file names are totally abbreviated with spaces in between and multiple cases like `Total Sales Mike 202002-AZ1P2R.csv`. This name is useful for the MIS or Business Analyst as they have a different way of organizing files then yours. They might do it because they have to keep a record of different people and have to provide it anytime asked. But as a Data Scientist your work is entirely different. You are not delivering files you are writing code. Let me reiterate this fact **YOU ARE WRITING CODE**. In most of the scenarios **Data Science is more like programming less like science**. Even though it has proportion of both of them. Using fundamentals of programming practices will help you out in long term. So change such file names to `sales_data_mike_feb2020.csv` or something similar. There are no right or wrong names just what makes more sense to a new user.

There is a trick about naming conventions:

1. use all lower case or upper case (helps you in never forgetting the cases)
2. use underscore in between (Because file names are mostly long Camel Or Pascal cases may confuse users)
3. make the name as general as possible (make sure a newcomer should be able to understand it without any problem)
4. In choosing a name there are no wrong answers only confusing ones

2.4 Folders Based on File-Type

A Very common practice is to keep different file types in different folder. One of the main mistake I see people writing code like this.

```
DBI::dbGetQuery(conn,
"
select
    count(*) as numbers,
    max(colname) as maxSome,
    min(colname) as minSome,
  from
    tablename
  group by
    col1,
    col2,
    col3
  order by
```

```
    numbers
")
```

or codes like this.

```
shiny::HTML(
"
   <p>At Mozilla, we’re a global community of</p>

    <ul> <!-- changed to list in the tutorial -->
      <li>technologists</li>
      <li>thinkers</li>
      <li>builders</li>
    </ul>

    <p>working together to keep the Internet alive and accessible, so people worldwide

    <p>Read the <a href=\"https://www.mozilla.org/en-US/about/manifesto/\">Mozilla Man
"
)
```

This is a bad coding style. Every time I see this type of code I realize that the person doesn’t believe that either the code will change or It will be extended. ***There is nothing permanent in the programming*** neither code, nor frameworks and not even languages. If you keep this type of code in separate SQL files or html files you can easily edit them later, code will be more easier to read and there will be a separation of concern. Tomorrow if you need help in SQL or HTML, a UI designer or a Database designer can look into your code without getting bogged down in R code. It makes bringing more people to the team easier.

2.5 Creating Sub-folders

On bigger projects simple folder structure tend to become more confusing. This is the main concern I have with the data folder every data scientist create and put all the files he has in that single folder. In these scenarios it’s better to have a sub-folder for different file types or may be different roles.

Like you can create sub-folders based on file-types like CSV’s, json, rds etc.. or you can even create sub-folders based on roles or needs... Like all the data related to one tab or one functionality goes in one folder and so on... There has to be a logical consistency in the folder structure. It’s primarily for you to not get lost in your own folders that you created and secondary for people working with you to understand your code and help in places you need help.

2.6 Conclusion

You have to create folders and everything has to be arranged in. Keep everything as organized as you keep your house. There are a certain principles that will help you in it.

1. Create projects
2. Name the files properly
3. Create a file for different language
4. create sub-folders wherever you fill necessary.

Chapter 3

Code Structure

Once you have arranged the files and folders in a logical way then comes the fact that the code itself should be arranged in such a way that it feels easy to go through. Always remember **Code is read more often then it's written**. Your world should revolve around this line. If you delegate your work while leaving your firm to someone else than the person who is handling your code should be able to understand everything you were trying to do. You will be in that position someday and you would wish your colleagues must have done the same. Even if you aren't sharing your code to somebody one day when you will return back to the project after say 7 to 8 months you will be surprised to see the mess you created back then. With this in mind hope this will help you in your journey.

3.1 Create Sections

Rstudio gives you an ability to create section by pressing (ctrl + shift + R), or you can create one

by adding 4 dashes (-) after a comment or 4 hash symbol (#) after a comment.

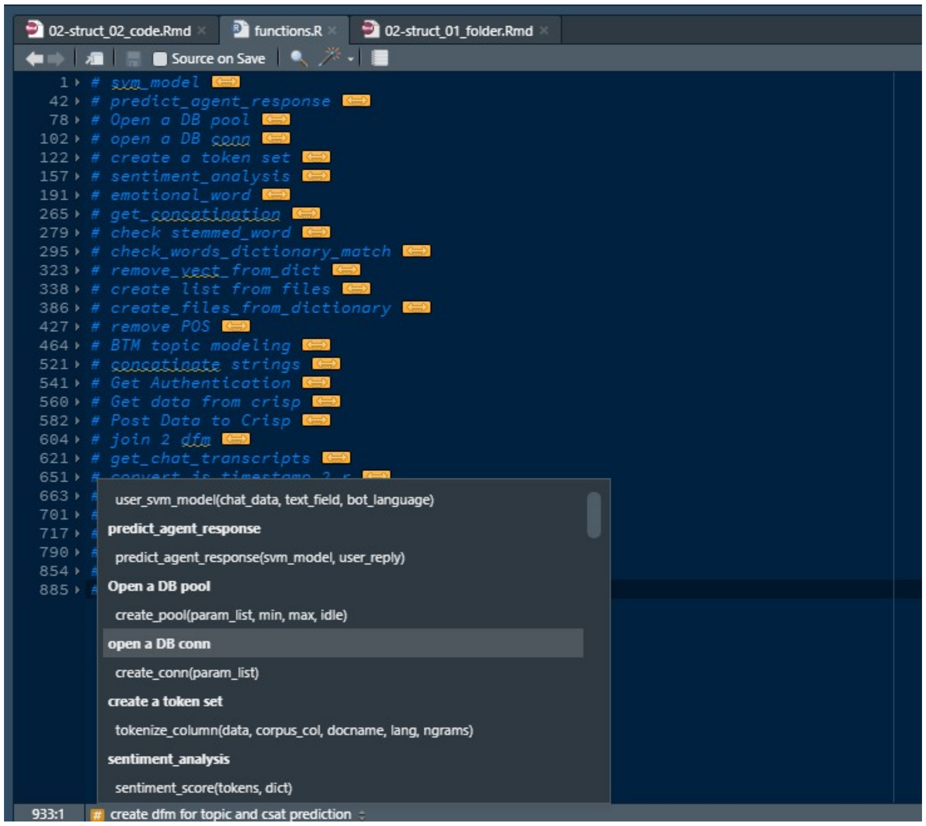
```
# some comment ----
# some comment ####
```

Both are valid syntax. you can name your section in the comment. Same rule applies for the Rmarkdown documents as well you should always name your code chunk.

```
# ```{r chunkname, cache=TRUE}
```

It also helps you jump between sections (Shift+Alt+J). You can easily switch between sections and fold them at will. It helps you not only in navigation but keeping a layout of the entire code as well.

A 800+ line files will look something like this.



```
# svm_model
# predict_agent_response
# Open a DB pool
# open a DB conn
# create a token set
# sentiment_analysis
# emotional_word
# get_concatination
# check stemmed_word
# check_words_dictionary_match
# remove_vect_from_dict
# create list from files
# create_files_from_dictionary
# remove POS
# BTM topic modeling
# concatinate strings
# Get Authentication
# Get data from crisp
# Post Data to Crisp
# join 2 dfm
# get_chat_transcripts
# convert js timestamp 2 r
#
#
#
#
#
#
```

```
user_svm_model(chat_data, text_field, bot_language)
predict_agent_response
  predict_agent_response(svm_model, user_reply)
Open a DB pool
  create_pool(param_list, min, max, idle)
open a DB conn
  create_conn(param_list)
create a token set
  tokenize_column(data, corpus_col, docname, lang, ngrams)
sentiment_analysis
  sentiment_score(tokens, dict)
```

```
933:1   # create dfm for topic and csat prediction
```

Figure 3.1: code chunk

It makes your code beautiful to look and makes it maintainable in long run.

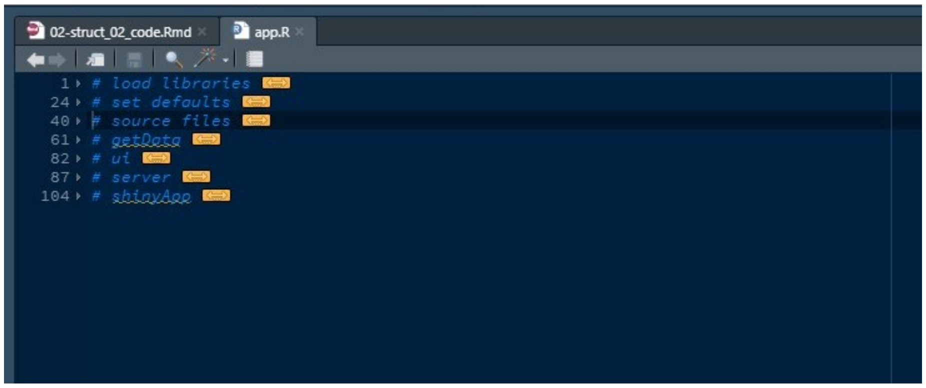
3.2 Order of Code

When you write code there are standard practices that are used across the domain and you should definitely use them. These are simple rules that most beginners aren't concerned about but the more experience you gain the more you start to realize the latent power of code organization. Here are a simple tip you should use.

1. Call your libraries on top of code

2. Set all default variables or global options and all the path variables at the top of the code.
3. Source all the code at the beginning
4. Call all the data-files at the top

In this exact order. This coherence keeps all your code easy to find. Most annoying thing in debugging someone else code is finding path variables laid out inside functions. Please don't ever do that. That is not a good practices. Take a look at one of my file



```
# load libraries
# set defaults
# source files
# getData
# ui
# server
# shinyApp
```

Figure 3.2: code order

If you think you will forget it. There is a golden rule you must remember. ***Put all the external dependencies on top of your code***. Everything that I mentioned above is external to the code in the file. In exact order.

1. Libraries are external to the file.
2. path variables
3. other files apart from one you are working is external as well.
4. databases and CSV

Just by doing this you will be able to navigate better in your code. There aren't any hard and fast rules for this only logical and sensible ones. Feel free to come up with your own layout that helps you in your analysis journey.

3.3 Indentation

It goes without saying that indentation makes your code readable. Python is not the only language who has the luxury of indentation. No matter what language you work in your code should be properly indented so that we can understand the nature of code written. There are a few things you can understand about indentation.

Maintain same number of spaces throughout your code. Your editor will help you out with it for sure but even if you are working on multiple editors. If you choose 2 spaces or 4 spaces as an equivalent of tabs you should stick to it. This is a golden rule you should never break.

Then maintain the same style in your code. Look at the code below.

```
foo <- function(
  first_arg, second_arg, third_arg
){
  create_file <- readxl::read_excel(path = first_arg, sheet = second_arg,
                                    range = third_arg)
}

bar <- function(
  first_arg,
  second_arg,
  third_arg
){
  create_file <- readxl::read_excel(
    path = first_arg,
    sheet = second_arg,
    range = third_arg
    )
}
```

function foo is written horizontal and bar is written vertical. I would prefer styling of function bar but you may choose one and stick to it for entire project. Mixing styles is not considered good and might create problem in code review.

There is a package by name `grkstyle` which implements vertical arrangement of code as mentioned above. You can look into it as well.

3.4 Conclusion’

In this chapter we discussed how to structure your code to make it more meaningful to read and easier to debug. The key takeaways from this chapter is:

1. Create sections to write beautiful and navigable code
2. Put those sections in a logical order
3. Don’t

Chapter 4

Functions

I can not over emphasize the importance of functions. As a data scientist most of the time you will be writing functions. Only in couple of cases where you have to write complicated classes there too methods are nothing more than functions. Having solid grasp of best practices in functions is a must for everybody working in any language what-so-ever. Hopefully this chapter will help you in best coding practices for functions.

4.1 Metadata or Information header

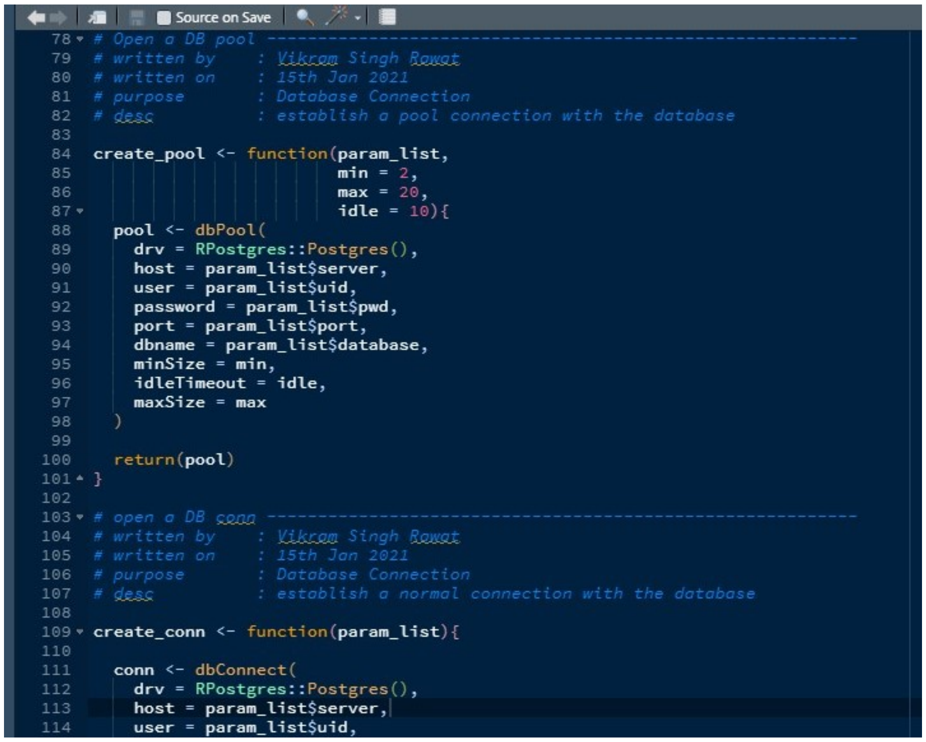
As I mentioned in the previous chapter it is a good practice to create sections for everything you do in R. functions are no exception to the rule. But along with that there are a couple of information you should write along with the function.

I worked in a few MNC where we had to write metadata of every function before writing it down. It makes it easier for code-reviewer to understand you code and for the entire team to collaborate in the project. It's good for personal projects too… Let me give you an example of what I mean by this.

You can see that if you are working on large teams or may be in big corporate settings where anybody can be reassigned to a different project. This data helps by identifying who wrote what and why.

Examples of some important tags can be :

1. written by
2. written on
3. modified by
4. modified on



```
# Open a DB pool -------------------------------------------------------------
# written by    : Vikram Singh Rawat
# written on    : 15th Jan 2021
# purpose       : Database Connection
# desc          : establish a pool connection with the database

create_pool <- function(param_list,
                        min = 2,
                        max = 20,
                        idle = 10){
  pool <- dbPool(
    drv = RPostgres::Postgres(),
    host = param_list$server,
    user = param_list$uid,
    password = param_list$pwd,
    port = param_list$port,
    dbname = param_list$database,
    minSize = min,
    idleTimeout = idle,
    maxSize = max
  )

  return(pool)
}

# open a DB conn -------------------------------------------------------------
# written by    : Vikram Singh Rawat
# written on    : 15th Jan 2021
# purpose       : Database Connection
# desc          : establish a normal connection with the database

create_conn <- function(param_list){

  conn <- dbConnect(
    drv = RPostgres::Postgres(),
    host = param_list$server,
    user = param_list$uid,
```

Figure 4.1: functions metadata

5. purpose
6. descriptions

You can create your own tags based on usecases and information needed for further scenarios.

4.2 Pass everything through parameters

I have seen people writing functions with calling things from global environments. Take a look at the code below.

```
foo <- function(x){
   return( x + y)
}

y <- 10

foo(5)
```

```
## [1] 15
```

Here the value of *foo* is based on y which is not a part of the function instead it's in global environment and function always have to search global environment for the object. consider these scenarios:

```
bar <- function(x, y){
  y <- y
  return(
    foo(x)
  )
}

bar(5, 20)
```

```
## [1] 15
```

you would assume that the answer is 25 but it's 15 because foo was created in the global environment and it will always look up value in global environment before anything else. This is called **Lexical Scoping** it's okay if you don't know it. It is very confusing and could mess up your code at any point in time. I am an experienced R programmer I too have trouble getting my head around it.

We can avoid all these situations by following the best coding practices that have been used in software industries for years. **Function should be a self contained code** which shouldn't be impacted by the outer world. Only is certain scenarios you allow to deviate from these rules but it's a good coding practice none the less. now in the above example instead of relying on the global variable if I just had created a parameter for Y, my code would be simpler to write and easier to understand and I would not have to think about lexical scoping on every step.

```
foo <- function( x, y ){
   return( x + y )
}


bar <- function( x, y ){
  return(
    foo( x, y )
  )
}

bar(5, 20)
```

```
## [1] 25
```

Now this code returns 25 as we all expected and trust me the Y is still available in global environment but that doesn't impact the foo or bar at all. Now you can nest this function under multiple other functions and it will behave exactly like it should.

There is a golden rule you should take away from this section. **Avoid Global Variables at all costs**. As much as possible pass everything through the parameters. That what they are for right !!!

4.3 Use Return Statement

It is a very simple thing yet most of the R users never worry about it because R takes care of finer details for you. But return statements actually make your code easier to read.

Suppose you have to review code return statement makes it easier to glance at the code and understand what is it doing. Almost all the programming languages are habitual with it. There are no good advantage I can tell you for a return statement other than readability. But just by following these practices R community as a whole could get more respect in programming community.

So please use Return statements wherever possible. In Big MNC your code will never pass reviewer unless it has return statements.

It also is good for functions that don't return anything you can just return true or false depending on the fact that the function ran without producing any error. Functions where you modify a data.table or where you change something in the database etc... It's a standard practice in old programming languages like C++ and it's a good practice indeed. We as a community should embrace these practices which will help us down the road.

4.4 Keep a consistency in Return Type

Return type of a function should be consistent regardless of what happens in a code. You may assume this is so simple that it goes without saying who would in their sane mind return character vector instead of a numerical one and you would be right. But Things get complicated when people start to work in composite data types like **Lists and Dataframes**.

Working with lists people get confused and forget this basic principle. I have seen function returning list of 2 elements on some conditions and 3 on other and 4 on some more. It makes it harder for users to work on those return values.

Don't even get me started on dataframes. People write functions that do some magic stuff on dataframes and it sometimes return a dataframe of 10 columns, sometime 11 and sometime 8. It's such a common mistake to make. I understand if you are fetching a table from database and returning that same table via functions but during manipulations you must add empty columns or delete existing ones to make it consistent for the end user regardless of the conditions you have in the functions.

4.5 Use Sensible Names for parameters

Yet another simple thing but because most of us including me come from non computer science background we have a tendency to use names like **x, y, z, beta, theta, gamma, string etc...** in our function parameters. I too am guilty of doing it in above code for foo and bar functions and in general. Many good and well established libraries in R are guilty of this sin too... But in long run these words don't make much sense. It's hard to maintain that code and it's hard for user as well. Let's take an example :

```
join <- function(x, y) x + y

join(x = 12, y = 12)
```

```
## [1] 24
```

do you see that as a user who hasn't written or even looked at the code it's already hard for him to understand what does x and y stands for. Only to get an error like this.

```
join(x = "mtcars", y = "iris")
```

```
## Error in x + y: non-numeric argument to binary operator
```

I know it is a stupid example but I see it every time in real code. When you only need numeric values why not include that information in the parameter name. something like:

```
join <- function(num_x, num_y) num_x + num_y
```

It may not seem like much but this small change makes the life of the user so much better where he doesn't need to consult the documentation again and again. Their are other ways you can come up with sensible names in your code just to avoid this issue. It's a standard practice during code review to check the names and these names are never allowed in production environment. We will discuss more about names in another chapter but for now understand that parameter names are just as important as the name of the function and it should be meaningful and easier to understand. There should be some information buried in the name.

4.6 use tryCatch

During deployment we would not like the shiny app or rest api or the chron job to fail. It's not a good experience to have for either the developer or the client. Best way to avoid it is wrap every function in a tryCatch block and log the errors. This way if you app has some bugs (which every app does). It will not crash and not destroy the experience of all the other people using it.

Let's bring back the foo function :

```
foo <- function( x, y ){
   tryCatch(
     expr = {
       return( x + y )
     },
     error = function(e){
       print(
         sprintf("An error occurred in foo at %s : %s",
                 Sys.time(),
                 e)
         )
     })
}

foo("mtcars", "iris")
```

```
## [1] "An error occurred in foo at 2021-03-26 17:07:09 : Error in x + y: non-numeric argument to
```

Now imagine this line to be printed in a json file or inserted in a database with time stamp and other information instead of crashing the entire code only a particular functionality will not run which is huge. This is the difference between staying late on Saturday night to fix a bug vs telling them that I will fix it on Monday. To me that is big enough.

4.7 Write simple and unique functions

Task of one function should be to do one thing and one thing only. There are numerous times when people assume they have written excellent code because everything is in a function.

Purpose of a function is to reduce one unique task in a single line. If your function does multiple things then it's a good Idea to ***Break your function into multiple one and then create a function which uses all of them***.

```
average_func <- function( mult_params ){
  tryCatch(
    expr = {
      ###
      # code to do stuff 1
      ###

      ###
      # code to do stuff 2
      ###
    },
    error = function(e){
      ###
      # code to log errors
      ###
    })
}
```

Now imagine if today you are logging on a json file and tomorrow client wants to log it into a database. Changing it on every function is not only time consuming but dangerous in terms that now you can break the code.

Now compare that to this code.

```
stuff_1 <- function(params_1){
  ###
  # code to do stuff 1
  ###
}

stuff_2 <- function(params_2){
  ###
  # code to do stuff 1
  ###
}

log_func <- function( log_params){
  ###
  # code to log errors
  ###
}

best_func <- function( mult_params ){
  tryCatch(
    expr = {
    stuff_1()
    stuff_2()
    },
    error = function(e){
    log_func()
    })
}
```

Here in this code every function has a clear responsibility and the main function is just a composite of multiple unique functions and it will be very easy to debug this code or change the functionality entirely.

4.8 Use Package::Function() approach

R classes work differently than the traditional oops we all are aware of. Instead of `Class_Object.Method` syntax like other programmings have, we in R use `method( Class_Object )` syntax. Where just by changing name collision is a pretty common thing.

It's a pretty common thing in R that 2 packages use same function name for different operations. So It's always better to use `qualified imports` fancy name for mentioning which package does the function comes from.

4.8.1 You should always load libraries in the order of their usage

```
# library("not_used_much")
# library("least_used")
# library("fairly_used")
# library("most_used")
# library("cant_do_without_it")
```

R uses the loading sequence to identify which function to give preference. It's usually to the last package loaded. It's called masking and it's not a reliable technique but it's better to arrange your code in that order for sake of simplicity.

and Yes do not forget to mention the package name clearly. like prefer writing this always:

```
# dplyr::filter()
# stats::filter()
#
# ## instead of
#
# filter()
```

NOTE:

In R you can view any function definition by running the function without `()` round brackets. like this

```
quanteda::tokens
```

```
## function (x, what = "word", remove_punct = FALSE, remove_symbols = FALSE,
##     remove_numbers = FALSE, remove_url = FALSE, remove_separators = TRUE,
##     split_hyphens = FALSE, include_docvars = TRUE, padding = FALSE,
##     verbose = quanteda_options("verbose"), ...)
## {
##     tokens_env$START_TIME <- proc.time()
##     object_class <- class(x)[1]
##     if (verbose)
##         catm("Creating a tokens object from a", object_class,
##             "input...\n")
##     UseMethod("tokens")
## }
## <bytecode: 0x0000000021eacc00>
## <environment: namespace:quanteda>
```

And you can check which methods are available for which classes by using

```
methods(class = "dfm")
```

```
##   [1] -                     !                     $
##   [4] $<-                   %%                    %*%
##   [7] %/%                   &                     *
##  [10] /                     [                     [[
##  [13] [<-                   ^                     +
##  [16] all                   any                   anyNA
##  [19] Arith                 as.data.frame         as.dfm
##  [22] as.DocumentTermMatrix as.logical            as.matrix
##  [25] as.numeric            as.wfm                bootstrap_dfm
##  [28] cbind                 cbind2                coerce
##  [31] coerce<-              colMeans              colSums
##  [34] Compare               convert               dfm
##  [37] dfm_compress          dfm_group             dfm_lookup
##  [40] dfm_replace           dfm_sample            dfm_select
##  [43] dfm_smooth            dfm_sort              dfm_subset
##  [46] dfm_tfidf             dfm_tolower           dfm_toupper
##  [49] dfm_trim              dfm_weight            dfm_wordstem
##  [52] dim                   dim<-                 dimnames
##  [55] dimnames<-            docfreq               docnames
##  [58] docnames<-            docvars               docvars<-
##  [61] fcm                   featfreq              featnames
##  [64] head                  initialize            is.finite
##  [67] is.infinite           is.na                 kronecker
##  [70] length                log                   Logic
##  [73] Math                  Math2                 meta
##  [76] meta<-                metadoc               metadoc<-
##  [79] ndoc                  nfeat                 ntoken
##  [82] ntype                 Ops                   print
##  [85] rbind                 rbind2                rep
##  [88] rowMeans              rownames<-            rowSums
##  [91] show                  sparsity              Summary
##  [94] t                     tail                  textplot_network
##  [97] textplot_wordcloud    textstat_dist         textstat_entropy
## [100] textstat_frequency    textstat_keyness      textstat_lexdiv
## [103] textstat_simil        textstat_summary      topfeatures
## [106] View
## see '?methods' for accessing help and source code
```

Or you can check how many classes have a method by same name with.

```
methods(generic.function = "textstat_lexdiv")
```

```
## [1] textstat_lexdiv.default* textstat_lexdiv.dfm*     textstat_lexdiv.tokens*
## see '?methods' for accessing help and source code
```

now most of these methods are hiden from general usage so you might not be abel to view them.

```
# textstat_lexdiv.dfm
# will not work will produce an error
# Error: object 'textstat_lexdiv.dfm' not found

textstat_lexdiv
```

```
## function (x, measure = c("TTR", "C", "R", "CTTR", "U", "S", "K",
##     "I", "D", "Vm", "Maas", "MATTR", "MSTTR", "all"), remove_numbers = TRUE,
##     remove_punct = TRUE, remove_symbols = TRUE, remove_hyphens = FALSE,
##     log.base = 10, MATTR_window = 100L, MSTTR_segment = 100L,
##     ...)
## {
##     UseMethod("textstat_lexdiv")
## }
## <bytecode: 0x0000000021e71470>
## <environment: namespace:quanteda>
```

```
# works but the implementation is still hidden because method will be decided based on the class
```

But If you still want to know how to know the definition of a method of the class. Just use this code.

```
getAnywhere("textstat_lexdiv.dfm")
```

```
## A single object matching 'textstat_lexdiv.dfm' was found
## It was found in the following places
##   registered S3 method for textstat_lexdiv from namespace quanteda
##   namespace:quanteda
## with value
##
```

```
## function (x, measure = c("TTR", "C", "R", "CTTR", "U", "S", "K",
##     "I", "D", "Vm", "Maas", "all"), remove_numbers = TRUE, remove_punct = TRUE,
##     remove_symbols = TRUE, remove_hyphens = FALSE, log.base = 10,
##     ...)
## {
##     unused_dots(...)
##     tokens_only_measures <- c("MATTR", "MSTTR")
##     x <- as.dfm(x)
##     if (!sum(x))
##         stop(message_error("dfm_empty"))
##     if (remove_hyphens)
##         x <- dfm_split_hyphenated_features(x)
##     removals <- removals_regex(separators = FALSE, punct = remove_punct,
##         symbols = remove_symbols, numbers = remove_numbers, url = TRUE)
##     if (length(removals)) {
##         x <- dfm_remove(x, paste(unlist(removals), collapse = "|"),
##             valuetype = "regex")
##     }
##     if (!sum(x))
##         stop(message_error("dfm_empty after removal of numbers, symbols, punctuatio
##     if (any(tokens_only_measures %in% measure))
##         stop("average-based measures are only available for tokens inputs")
##     available_measures <- as.character(formals()$measure)[-1]
##     measure <- match.arg(measure, choices = available_measures,
##         several.ok = !missing(measure))
##     if ("all" %in% measure)
##         measure <- available_measures[!available_measures %in%
##             "all"]
##     compute_lexdiv_dfm_stats(x, measure = measure, log.base = log.base)
## }
## <bytecode: 0x000000001f2b8510>
## <environment: namespace:quanteda>
```

If you want to understand more of this learn OOPS in R. R has multiple oops system and R is a highly Object oriented programming but the style is different from other languages. This book is all about best practices in R and thus we are not going to go deep into fundamentals of R programming here but this trick is worth knowing.

4.9 Conclusion

In this chapter we discussed the best practices for writing functions in R. Here are the key takeaways from the chapter.

1. write information about the function at top of it.
2. avoid global variable and pass everything through parameters
3. use return statement to end your function
4. keep consistency in return types of a function
5. use logical names for parameter
6. use tryCatch in every function
7. functions are supposed to do one thing and one thing only
8. Use qualified imports with syntax package::functions everytime possible

Chapter 5

Naming Conventions

This chapter is crucial only for people to understand what are the bad naming practices we the R users have acquired over the years because of flexibility in the language. These names we give to the data or variables are not valid outside or R community and thus are subject to code reviews. You may even be asked to change name before deploying the code in production. The more bad naming practices the more time it takes you to fix them. It's a good practice to know the best practices for naming things in general.

5.1 Popular naming conventions

There are 3 most famous naming conventions in programming community. They are used throughout the code in big projects to smaller ones. These are :

5.1.1 camelCase

These names start with small letter and every subsequent word will start with upperCase letter like my name in camelCase would be written as ***vikramS-inghRawat***. All the functions in **SHINY** are camelCase. It's a great example of camelCase naming conventions.

5.1.2 PascalCase

PascalCase is just like camel case but the only difference is the first letter is also UpperCase. My name would be written as ***VikramSinghRawat***.

5.1.3 snake_case

These names are all lower case with underscore between the name. My name in snake_case would be ***vikram_singh_rawat***. **TIDYVERSE** is a great example of snake_cases. I really like the naming conventions of packages like **stringi** and **quanteda**.

whenever you start a project you should choose one of the naming conventions for the entire team. So that no matter who writes the code there is a logical consistency in the names and anybody can predict the next letter.

In many projects that I have worked camelCase were chosen for naming variables and PascalCase for methods or functions. I came to know later that this is a style many programming languages choose. Infact in langauges like golang if you write snake_cases linter will ask you to correct the name. But for **SQL** and **R** I would highly recommend snake_cases as many databases like postgres don't allow capital cases in column names you have to surround names in quotes if you need to use uppercase letters. In R tidyverse has gained huge momentum and now all the packages are following suite. Apart from that if your package can even tell what datatype are you working on that is a huge add on. Packages like **stringi** and **quanteda** are excellent example of this.

And I would like to add no matter what you choose **Please never include dot in any name**. That's a practice valid for only R code and it too is not accepted anywhere apart from R programming language.

Overall choose a naming convention for a project and stick to it or ask your client if they have a preference on it. This saves you from trouble of code reviews.

5.2 Informative Names

I may sound like a tidyverse fanboy (I am not) but classes and data types in R are quite opaque so names of functions and objects should reflect precisely what they represent. There is no harm in using names with data-types before them

```
# int_currency <- 1:10
# chr_letters <- letters
# dt_mtcars <- data.table::data.table(mtcars)
# tbl_mtcars <- tibble::tibble(mtcars)
```

Above advice may be more useful for package developers but it can be used in broad scenarios even on a project where there are multiple working on a same project. If I know what datatype I am dealing with I don't have to go through the entire code and working on top of it becomes that much easier.

You can use more descriptive names without data types in the beginning for your projects. Names like **data, mainData, dummyVar, tempList** etc.. should never be used in a project. Use more descriptive names like sales_data_2020, api_token, rate_of_interest etc...

5.3 Conclusions

Proper naming conventions will help collaboration in big teams and it makes the code easier to maintain. We should all strive for better names in the code. It's the hardest job to come up with new and unique names for a variable everytime you create one but this is the difference between an average programmer and a good one.

1. Choose a naming convention and stick to it
2. Don't include dots (.) in names
3. Use informative names

Chapter 6

Environment Management

If you create a product today be it an API or Shiny App or Even a normal R-script. One thing you can't be sure of is to update the packages or the version of R. There are companies where you can not access different version of a package because multiple projects are relying on the same copy of the package. It's hard to update your package in these companies and you will need to get permissions from top admins to do so. Thus it's better to rely on as less packages as possible and that too on the popular ones.

But even after you have created a code you would want to keep a record of all the packages and their version as it is for that particular project. This is where environments come in handy.

6.1 renv for package management

There was a package called Packrat a few years ago I would have suggested you to use that always. But currently there is a package I have been using for over a year now by name renv. It does everything that you need to recreate your environment anywhere else.

Basically you need to activate the package in your project. By using this command.

```
renv::activate()
```

Then take a snapshot of current project where it will record a list of all the packages used in your project by this command.

```
renv::snapshot()
```

and When you want to reproduce it on a docker container or a remote machine or any place else. You would simple need to run.

```
renv::restore()
```

and it generates a lock file with all the information about a project including the version or R and the versions of the packages used so at any time you can recreate the entire environment again.

I could give you multiple ways of tackling the same problem. But this book is about the best possible one so this is it. You just need to use this package to solve almost all of your problems.

6.2 config for external dependencies

There is a package called config that allows you to read yaml format in R. That is a standard practice to keep all the **Credentials, tokens, API keys etc..** in a config file. There are many other ways you can secure credentials and everything but config is easiest amongst them all and you can use it for storing all the parameters and external path variables that your code requires. It could be an address to external file storage or anything else.

It's good to keep all the variable your code requires outside the main code so that when you need to update them you don't need to change the entire code itself. Below is a snippet of config file from one of my project.

```
default:
  datawarehouse:
    driver: Postgres
    server: localhost
    uid: postgres
    pwd: postgres
    port: 5432
    database: master
  dockerdatabase:
    driver: Postgres
    server: postgres_plum
    uid: postgres
    pwd: postgres
    port: 5432
    database: master
  filestructure:
    logfile: "logs/logs.csv"
```

as you can see I haven't only kept the passwords and user names but external files as well. Tomorrow if I have to change the logging file I will just have to update it here without opening any R code. It removes so much burden on reading the code again and again.

Use it whenever possible.

6.3 Conclusion

This chapter doesn't discuss much on concepts but the takeaways from the chapter are:

1. Use **renv** for all the project you plan to maintain or keep for long term
2. Use **config** to manage all the external dependencies your project have or might have

Chapter 7

data Management

7.1 Don't use numbers

7.2 Use Databases

7.3 Keep a Copy

7.4 Use Efficient Packages

Part III

Memory

Chapter 8

Type System

> ***With Great power comes great responsibility***
>
> – (Uncle Ben) *Man who raised spider-man*

Despite what most people believe R too has data types. Every language tries to consume the memory space as efficiently as possible and for that they have pre-specified memory layouts that work almost all the same in every language. If you have worked on SQL databases the role of data types are exactly the same across all the languages. R just makes it easier to infer the data-type from your code so that you don't have to declare it specifically.

primal data types for vectors in R are :

- logical
- numeric
- integer
- complex
- character
- raw

then there are composite data types like date, posixct, even a dataframe is a list with some rules.

8.1 Things you should know

There are a certain things you should know about data types in R.

8.1.1 R don't have scalar data types

```
x <- 10L
x
```

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

There is a reason `[1]` is written before the number 10. It's because unlike other languages R don't have scalar values. Everything in R is a vector. It may be a vector of length 1 or 1 billion but it's all still a vector. This is one of the primary reason R works more like **SQL** (*as most data guy love*) and less like **JAVA** (*as most programmers love*).

It gives huge speed to data manipulation as all the operations are more like ***In Memory Columnar Table***. But in return when you are creating a webpage or an API or something where you need a scalar value to be updated again and again. R consumes more resources to do that kind of thing.

```
microbenchmark::microbenchmark(
vectorized = {
  x <- rnorm(1e3)
},
scalar = {
  y <- numeric(1L)
  for(i in 1:1e3){
  y[[i]] <- rnorm(1)
  }
},
times = 10
)
```

```
## Unit: microseconds
##        expr    min     lq    mean  median     uq    max neval cld
##  vectorized   53.6   55.1   64.02   55.60   58.9  133.3    10  a
##      scalar 2890.2 3037.7 3289.75 3302.85 3337.7 3924.9    10   b
```

Even then I would ask you to go ahead with R because the difference will most probably in 1-2 milliseconds which will never impact your performance any serious way. But this is something you should remember that vectorized R is way faster than even python but scalar manipulations in R are a bit slower. Choose vectorized version of the code whenever possible even if you do a bit more steps in it, it will still be faster than the scalar versions.

8.1.2 Dates are basically integers under the hood.

```
x <- Sys.Date()
class(x)
```

```
## [1] "Date"
```

```
# "Date"
as.integer(x)
```

```
## [1] 18712
```

This number means it has been 18 thousand 7 hundred days since 1970 which is roughly (365 * 50)

8.1.3 POSIXlt are basically lists under the hood

```
unclass(x)
```

```
## [1] 18712
```

```
y <- Sys.time()
y <- as.POSIXlt(y)
class(y)
```

```
## [1] "POSIXlt" "POSIXt"
```

```
# "POSIXlt"

unclass(y)
```

```
## $sec
## [1] 9.624757
##
## $min
## [1] 7
##
## $hour
## [1] 17
```

```
##
## $mday
## [1] 26
##
## $mon
## [1] 2
##
## $year
## [1] 121
##
## $wday
## [1] 5
##
## $yday
## [1] 84
##
## $isdst
## [1] 0
##
## $zone
## [1] "IST"
##
## $gmtoff
## [1] 19800
##
## attr(,"tzone")
## [1] ""       "IST"    "+0630"
```

because it stores metadata along with it, use posixct whenever possible.

8.1.4 Integers are smaller than numeric

```
x <- sample(1:1e3, 1e8, replace = TRUE)
class(x)
```

```
## [1] "integer"
```

```
# [1] "integer"
y <- as.numeric(x)
class(y)
```

```
## [1] "numeric"
```

```
# [1] "numeric"

object.size(x)
```

```
## 400000048 bytes
```

```
# 400000048 bytes
object.size(y)
```

```
## 800000048 bytes
```

```
# 800000048 bytes
```

See the difference yourself. It's about twice the size of the original integer vector. It's all because of datatypes. You should use integer only when you need one. There is a cool trick to let R know that you are creating an integer. Just add L at the end.

```
x <- 1
class(x)
```

```
## [1] "numeric"
```

```
#[1] "numeric"

y <- 1L
class(y)
```

```
## [1] "integer"
```

```
# [1] "integer"
```

Letter L at the end after a number will tell R that you want an integer. Please use integers when you need one.

8.1.5 define your datatypes before the variable

```
i <- integer(1e3)
class(i)
```

```
## [1] "integer"
```

```
length(i)
```

```
## [1] 1000
```

```
l <- logical(1e3)
class(l)
```

```
## [1] "logical"
```

```
length(l)
```

```
## [1] 1000
```

```
n <- numeric(1e3)
class(n)
```

```
## [1] "numeric"
```

```
length(n)
```

```
## [1] 1000
```

```
c <- character(1e3)
class(c)
```

```
## [1] "character"
```

```
length(c)
```

```
## [1] 1000
```

Just like any other language even in R you can create an empty vector of a predefined length which are initialized at 0 or ”” or FALSE based on the data types. Use this functionality when you want to create a column or vector you know nothing about except data type.

Defining data-types beforehand is an excellent programming practice and we as R user should use it more often. It also removes burden on the compiler to try to guess the data-type.

8.1.6 lists are better than dataframe under a loop

```
# x_dataframe <- data.frame(x = 1:1e3L)
#
# for(i in 1:1e4L){
#   x_dataframe$x[[i]] <- i
# }
# This code will produce an error because you can't increase the row count of a dataframe like th

x_list <- list(x = 1:1e3L)

for(i in 1:1e4L){
  x_list$x[[i]] <- i
}

x_dataframe <- as.data.frame(x_list)
class(x_dataframe)
```

```
## [1] "data.frame"
```

you can not create additional rows easily in data.frame. but all dataframes are lists under the hood with some additional rules. you can convert them to list run a loop and convert back to data.frame. It's not only efficient but it's faster too…

8.1.7 use lists whenever possible

Other languages have structs to handle multiple object types. R have lists and lists are most versatile piece of data-type you will find across any language. There are tons of example like the one I provided above where lists are more efficient because they don't have any restrictions.

In my personal use case I have seen people trying to put a square peg in round hole by using data.frames at places where a simple list will be more efficient and appropriate. Please use list as frequently as possible and remember, ***always opt for lower level data type for better memory management***.

8.2 Choose data types carefully

As we saw in examples above choosing the right data-type can mean a lot in

```
x <- seq.Date(
  from = as.Date("2000-01-01"),
  to = Sys.Date(),
  length.out = 1e4)

x <- sample(x, 1e8, TRUE)

y <- data.table::as.IDate(x)

length(x) == length(y)
```

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

```
object.size(x)
```

```
## 800000272 bytes
```

```
object.size(y)
```

```
## 400000336 bytes
```

as you can see the base date type consumes around twice the memory compared to **IDate** data type from data.table. It may be because one is using numeric data types under the hood and other is using integers under the hood and it makes a huge difference when you are working on big data, to understand the data types in R and properly map them to save more space on your RAM.

Despite what most people say RAM and CPU are not cheap, throwing more processor on something should only be done when the code is properly optimized. I don't want you to be hyper optimize your code on every sentence you write but being aware of some best practices will surely help you along the way. Go to next section for speed optimization as well.

There are many packages in R we will talk about that provide speed ups to the code and saves memory too... We will talk about them later in the book. At this point all I can say is if R is slow or it crashes may be the data-type you have chosen is not right fit for the job. Try changing it and it will work just fine.

8.3 don't change datatypes

R gives you an option to do this.

```
x <- "Hello world"

print(x)
```

```
## [1] "Hello world"
```

```
x <- 1L

print(x)
```

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

Now you just assigned x as a character vector and then replaced x as an integer vector. This is something you can do but it's something you should never do. changing the datatype of a vector is not recommended in any programming language unless you are trying to convert from one data-type to another. like :

```
x <- "2020-01-01"
class(x)
```

```
## [1] "character"
```

```
x <- as.Date(x)
class(x)
```

```
## [1] "Date"
```

```
y <- c("1", "2", "3")
class(y)
```

```
## [1] "character"
```

```
y <- as.integer(y)
class(y)
```

```
## [1] "integer"
```

This and many operations like this where you know beforehand that you need to change the data-type is a good example of cases where you must change the data-types. So apart from cases where data-conversion is needed you should never change the data-types ever. It's a bad practice to do so.

This is one of the scenario when you have the power but you mustn't use it.

8.4 Future of type-system in R

Type system is important when you really want to save memory. It's specially true when you are dealing with huge volume of data and you want to save RAM more efficiently as possible, which is what an R users bread and butter. It's more like when you need it you really can't do without it. Every programming language is understanding this now. R is no exception to the rule. people are coming up with excellent theories on how to integrate a type system in R. Sooner or Later we will have a proper type system.

Currently, There is a package called **Typed** by **Antoine Fabri** on CRAN. You can install it directly from cran. It will not give you speed benefits though because it doesn't talk to compiler directly but it surely will restrict people to using wrong data-types where you don't need them. It's helpful when you write functions where you only need vector of certain lengths and a certain type so that further operations can be optimized.

Then there are packages like contractr by By Alexi Turcotte, Aviral Goel, Filip Křikava, Jan Vitek which can talk to compilers directly. Package is still in early development have no claims or documents available at their homepage at https://prl-prg.github.io/contractr/. But they are trying to insert type system through roxygen arguements above a function. I think this will be useful for package developers. It has a long way to go but we are thinking in right direction at least. For more information you should watch this video

These packages are no substitute for an inbuilt type system and we may ignore it but we really need a type system going forward. Lets hope for the best and have our fingers crossed for the moment.

8.5 Conclusion

In This chapter we focused on multiple data-types in R and how to save memory and CPU time by utilizing the best one in it. There are only a few but critical takeaways from this chapter :

1. remember:
    1. R don't have scalars
    2. dates are integers
    3. POSIXlt should rarely be used
    4. use integers when you can
    5. define data-types beforehand
    6. use lists wherever possible

2. choose data-types carefully

3. Don't change data-types unless necessary

4. In Future we will have a type-system and we should learn to love types early on.

Chapter 9

Pass By Value-Reference

In programming we have a concept of how to pass a value to a function. If we can do away with modification of the object inside the function then it's okay to pass the original object and let it change else we can create a copy of it and let the function modify it at will without effecting the object itself.

understanding this concept is very crucial if you want to write efficient code. Let's dive deeper into it.

9.1 Understanding the system

There are mostly 2 systems available for passing the objects from one function to another. Let's understand both of them.

9.1.1 Pass by Value

This is when you create a copy of the original object and pass it to the function. Because you are actually passing just a copy to the function whatever you do to the object doesn't impact the original one. Let's check it by an example.

```
x <- list(y = 1:10)

pass_by_value <- function(x){
  x$y <- 10:1
}

pass_by_value(x)
x$y
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

now x was passed to the function and modified yet it remains same because only copy of the object was passed to the function (Well, not precisely but this is what we will discuss later).

9.1.2 Pass by reference

This is when you pass the entire object as is. Basically you pass the pointer to the original object and now if you change the object you change the original copy of it. Let's check the same example again.

```
x <- new.env()
x$y <- 1:10

pass_by_value <- function(x){
  x$y <- 10:1
}

pass_by_value(x)
x$y
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

Now x was passed by reference and no copy was assigned to the function. So when you changed the object inside the function original object was changed.

Hope you now understand practically what does the word mean.

9.2 Copy on modify

R has no effective means to specify when to pass with value and when to pass with reference. And because there are only 2 ways to deal with this problem everybody assumes that R does create a copy of the object every time it passes the object through a function. But R has a different way of doing things which is called **copy of modify**. There are better blogs written over it and nuances are very peculiar which while writing code you shouldn't worry about much. I will try to simplify the concept from the practical point of you view so that you can use it in real life without much thought to it.

R basically passes an object by references until you modify it. Let's check it live:

```
mt_tbl <- tibble::as_tibble(mtcars)

tracemem(mt_tbl)
```

```
## [1] "<00000000276B8438>"
```

```
dummy_tbl <- mt_tbl
## No tracemem yet

mpg_col <- as.character(mt_tbl$mpg)
## No tracemem yet

mt_tbl %>%
  filter(
    cyl == 6,
    hp > 90
  ) %>%
  group_by(gear) %>%
  summarise(n()) %>%
  select(gear)
```

```
## tracemem[0x00000000276b8438 -> 0x00000000277fc248]: initialize <Anonymous> filter_rows filter.
## tracemem[0x00000000277fc248 -> 0x00000000278e97a8]: names<-.tbl_df names<- initialize <Anonymo
```

```
## # A tibble: 3 x 1
##    gear
##   <dbl>
## 1     3
## 2     4
## 3     5
```

`tracemem` is a function that will return memory address every time the object is copied. So far it didn't return anything even though it passed through 4 functions `filter`, `group_by`, `summarise`, `select` and each of those functions must be using multiple functions internally. Yet no copy of the object was made. Because **So Far we haven't modified anything**. now look at the code below.

```
new_tbl <- mt_tbl %>%
  filter(cyl == 6,
         hp > 90) %>%
  group_by(gear) %>%
  summarise(n()) %>%
  select(gear)
```

```
## tracemem[0x00000000276b8438 -> 0x0000000023229218]: initialize <Anonymous> filter_r
## tracemem[0x0000000023229218 -> 0x00000000232292c8]: names<-.tbl_df names<- initiali
```

now we are modifying the results somewhere and thus a copy is created. The actual rules are very very complicated. But in simple term as long as you don't modify any thing R doesn't create a copy and everything is passed down by reference.

It impacts speed too… Let check it by an example

```
foo <- function(x){
  sum(x)
}

bar <- function(x){
  x[[1]] <- 1
  sum(x)
}
```

As you can see both the functions are identical the only difference is that in `bar` I am modifying the object while in `foo` I am not changing the object. Let's run a speed test…

```
x <- rnorm(1e7)

microbenchmark::microbenchmark(
  foo = foo(x),
  bar = bar(x),
  times = 10
)
```

```
## Unit: milliseconds
##  expr     min      lq     mean   median      uq     max neval cld
##   foo  8.1866  8.2571  8.62935  8.47075  8.8821  9.8689    10  a
##   bar 22.8338 23.0881 24.28595 23.64840 24.7094 28.8780    10   b
```

As you can see the difference in time is because `bar` is creating a copy of the object. And you may assume that it will create a copy at every time you change a object and you will be dead wrong as R is smart enough to understand that It can get away with only single copy of the object. Lets create a function that changes more things in x and see the difference.

```
bar_new <- function(x){
  x[[1]] <- 1
  x[[10]] <- 10
  x[[1e3]] <- 1e3
  sum(x)
}

microbenchmark::microbenchmark(
  foo = foo(x),
  bar = bar(x),
  bar_new = bar_new(x),
  times = 10
)
```

```
## Unit: milliseconds
##     expr     min      lq     mean   median      uq      max neval  cld
##      foo  8.2169  8.2651  8.44543  8.34205  8.5265   8.9672    10  a
##      bar 22.8238 23.1383 32.09560 23.27455 25.8790 106.3200    10   b
##  bar_new 23.0490 23.4477 24.82441 24.54035 25.9817  27.1775    10  ab
```

Now as you can see that while the function foo and bar have significant differences in performance, same is not true for bar and bar_new. Because bar_new too creates a copy but maintains that copy for the entire function.

So R is smart enough to understand when to create a copy and when not to create a copy. Once a copy is created it is retained in R and R uses it smartly. We can gain speed and memory benefits by making sure all the modification is done inside a single function. So that R doesn't create much copies.

Instead of using bar 3 times it's better to use bar_new once. So that you don't copy it multiple times. See the difference for yourself. And thus **try to keep all the modifications close and in as less functions as possible**.

```
microbenchmark::microbenchmark(
  bar = {
    bar(x)
    bar(x)
    bar(x)
    },
  bar_new = bar_new(x),
  times = 10
)
```

```
## Unit: milliseconds
##     expr     min      lq      mean   median       uq      max neval  cld
##      bar 68.7984 68.9156 108.74168 73.17875 134.8133 261.3985    10   b
##  bar_new 22.7925 23.0976  24.91976 24.71115  25.9266  27.7143    10  a
```

best is to group these modifications together.

So the gist of the matter is:

1. R passes everything by reference until you modify it
2. R creates a copy when you modify the object
3. You should always keep all the Object modifications in same function

9.3 for pass by reference

As I told you before R has no way of specifying when the object will be pass by reference and when it will be passed by value. And there are certainly times you wish you had passed it by value and certainly times when you wish you passed it by reference.

When you modify something inside a function you create a copy of it. So take example of a loop inside and outside a function

```
x <- numeric(10)
for(i in 1:10){
  x[[i]] <- rnorm(1)
}
x
```

```
##  [1] -0.62699314 -1.16810425  1.23961013  0.48383496  0.61872715  0.06284149
##  [7] -0.63595299  1.10870584  0.10175430  0.67721856
```

It modifies the object in place. Now lets wrap it in a function and see what happens.

```
x <- numeric(10)

foo <- function(x){
  for(i in 1:10){
    x[[i]] <- rnorm(1)
  }
  return(x)
}

foo(x)
```

```
##  [1] -1.3557271 -0.4499918 -0.6147331 -1.0472309 -0.3556228 -0.4305612
##  [7] -0.1251895 -0.7638334 -0.2739515 -1.8029968
```

```
x
```

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

Now x is not modified because it is being modified inside a function. This is crucial at times when you are running a long job that might take hours to complete just to find an error in the middle. You might want to start the loop from the exact position you left off. With this sort of code you will not reach that. Let's generate an error in the code and uses bigger number.

```
total_length <- 1e2
```

```
set.seed(1)

x <- numeric(total_length)

foo <- function(number){
  y <- sample(1:total_length,1)
  for(i in 1:total_length){
    number[[i]] <- i
    if(y == i){
      stop(sprintf("there is an error at %s", y))
    }
  }
  return(number)
}

foo(x) ## You will get an Error
```

```
## Error in foo(x): there is an error at 68
```

If you run this code you will get an error at some number and x will still be the same. All the processing of code till that moment is lost for everybody. Which is not what you want if each iteration took just 2 minutes to run. This difference could mean hours in some scenarios.

R has 4 datatypes that provide mutable objects or pass by reference semantics.

1. R6 Classes
2. environments
3. data.table
4. listenv

I wouldn't recommend writing an R6 class just to run a simple loop, however if your use case is pretty complex R6 would be a valid solution for it. We already saw how environments can be used for pass by reference. But passing around environments is not a good idea it requires you to know too much about the language and be very careful with what you are doing hence I only prefer 2 approaches. One with data.table and other with listenv package.

But their usecase is very different. One should be used where you are comfortable with lists are more suited while other should be used where data.frame or vectors are more suited for the task. Doing it for listenv is very easy. It's the same code with just the new listenv object.

```
foo_list <- function(list){
  y <- sample(1:total_length,1)
  for(i in 1:total_length){
    list$x[[i]] <- i
    if(y == i){
      stop(sprintf("there is an error at %s", y))
    }
  }
}

list_env <- listenv::listenv()
list_env$x <- numeric(total_length)

foo_list(list_env)
```

```
## Error in foo_list(list_env): there is an error at 39
```

Now again we got an errors but this time all the other changes have been saved in x.

```
list_env$x
```

```
##   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##  [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39  0  0  0  0  0  0  0  0  0  0  0
##  [51]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
##  [76]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Same thing could be done in data.table as well. let's write a new function for doing it. data.table has 2 ways of looping through the vectors.

1. With `:=` operator which is slow but useful for more data insertion than one by one

2. with `set` function which is faster where you need to insert data one by one.

Let's use the second approach to write a function.

```
x_dt <- data.table::data.table(x = numeric(total_length))

foo_dt <- function(dt){
  y <- sample(1:total_length,1)
  for(i in 1:total_length){
    data.table::set(
      x = dt,
      i = i,
      j = "x",
      value = i)

    if(y == i){
      stop(sprintf("there is an error at %s", y))
    }
  }
}

foo_dt(x_dt)
```

```
## Error in foo_dt(x_dt): there is an error at 1
```

Now just like again even though we got errors we can still check the ones that have been completed during the loop.

```
x_dt$x
```

```
##   [1] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

So let me make things simpler. When you want to modify objects in place you need to use 1 of the 2 approach. **When you are working on data.frames and vectors use data.table while when you are working on anything else, anything in general, use listenv approach**.

9.4 Conclusion

This chapter focused on how to save memory of your R program by using objects through reference and avoid creating copies of the object. Let's summarize what we have read so far.

1. keep all the modifications of objects in a single function
2. use pass by reference through listenv and data.table for saving memory
3. avoid creating multiple copies of an object at all costs

Part IV

Speed

Chapter 10

For Loops

Chapter 11

Multithreading

Chapter 12

Vectorize

Chapter 13

Benchmarking

Chapter 14

packages

Part V

Production Tools

Chapter 15

Docker

Chapter 16

Proxy Server

Chapter 17

Cloud Services

Part VI

Shiny Tips

Chapter 18

Speed

Chapter 19

Memory