Introduction to R Programming

geshun

2019-08-05

Contents

Introduction to R Programming

Documenting my understanding of R as a programming language and tool for data analysis.

Chapter 1

Introduction

R is a powerful programming language used for data analysis and visualization.

Chapter 2

Binding

We can read `x <- c(1, 2, 3)` as first create a vector (object) containing values `1, 2, 3` and bind it to (*to used here should not be interpreted as a movement from a point to another, like arrow*) the name `x`. Thus, the assignment operator `<-` binds from the name `x` (LHS) to the the object `c(1, 2, 3)` (RHS).

This way (2 steps, create then bind) of reading `x <- c(1, 2, 3)` is a natural flow from the idea that every call in R is a function. `c` is a function and thus, it has been called on `1, 2, 3` to create a vector. That function would have to first execute before the assignment operator which itself is a function can do the binding. Realize that we could have written the same expression as `` `<-`(x, c(1, 2, 3)) ``, of which we expect the inner function to execute before the outer function.

Since `c(1, 2, 3)` gets created prior to being assigned to a variable (or name), where does the object go after creation? It finds a place in memory (thus memory is allocated for it, check with `lobstr::obj_addr(c(1, 2, 3))`) waiting to be used by another function (since it's an argument of the outer function). What the assignment operator then do is to find a way to reference that object in memory by binding it to a name.

Binding an object to a name just the object reusuable. There are few words thrown around here, *object*, *value*, *name*, and *variable*. When we think of an object, we think of a data structure/type. The object `c(1, 2, 3)` is an atomic vector, containing numeric values 1, 2, and 3. It can be referenced in memory using the name or variable `x`. Think of the assigment operator as a way of creating binding from what is on the RHS to what's on the LHS. To create binding from a name to an object (or value), the object must first exist. Realize that the memory address or location of an object changes at each run even if you *think* you're creating the same object. Run `lobstr::obj_addr(c(1, 2, 3))` couple of times in the console to see this.

Consider the following piece of R code:

```
x <- c(1, 2, 3)
y <- x
obj_addr(x) == obj_addr(x)
```

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

We want to explain why both x and y have the same memory address. Note that the object was created at the first line of the code and a binding was created to reference the object with a name `x`. In the second line of the code, only binding is occuring and no object creation. But, we know that before binding can occur, the object must first exist in memory. In this case the object that the name `x` references already exists in memory. The second line is not creating a copy of `x` but binding `y` to the same object as `x`. In this case, both `x` and `y` will have the same memory address. Both `x` and `y` are pointing to the same object. Does that mean when `x` is modified, `y` automatically gets modified?

At least here we see multiple names referencing the same object. The trick used is first create object and bind to a name and assign the name to other names. Why should someone care about given multiple names to the same object, any usefulness or applications? How does that enlighten our programming knowledge or ideas?

2.1 Non-syntactic Names

All you got to know about these terms, is that R has rules of what constitutes a valid name. If you stick to the rules, you get a syntactic name and if you override the rules, you get non syntactic name. A syntactic name can be a combination of letters, numbers, period and underscore and can begin with a letter or a dot (and not followed by underscore). It cannot be any of the reserved words in R. I don't like names with periods in them since they sometimes conflict with S3 objects (Give example).

To see why a name starting with "." cannot be followed by a number, ask if R treats `0.25` and `.25` the same. R is making sure a name is not treated like a value.

Non-syntactic names are created using backtick or quotation. For instance `` `for` <- 4 `` or `"for" <- 4`. I think there is nothing exotic or fancy about using non-syntactic names. It's just a bad habbit and thus I stay away from such temptations. Besides, one needs to learn a new rule for using those names; you cannot assign the value `4` to `` `for` `` and reuse it like `for ^ 2`. There will be an error.

The `read.csv()` function does checsk on column names to make sure they are syntactically valid using the default argument `check.names = TRUE`. It

uses `base::make.names()` function which translates all invalid characters (like space) to ".", missing values are translated to string "NA", and duplicated values are altered by `base::make.unique()` function. To supress checking for syntactically valid names, set `check.names = FALSE`.

```
df <- data.frame(sample(5), letters[1:5])
names(df) <- c('x million, $', 'y project')
write.csv(df, 'df.csv', row.names = FALSE)
```

Now, explore reading the .csv file with the following:

```
df_utils <- utils::read.csv('df.csv')
df_readr <- readr::read_csv('df.csv')
df_dtable <- data.table::fread('df.csv')
```

I choose these three packages for reading files because they are what people usually use. Each by default, read the column names differently. Sometimes, the column names can carry extra information for understanding the data. Data familiarization is key in doing meaningful analysis. Allowing for name check in `read.csv` can replace useful information with not-so useful one (like replacing the dollar sign with a "."). We lose the information that the currency is US Dollars. Also, there are a lot of periods to remember when referring to the column names. Though `fread` returns the name unaltered, `read_csv` adds backtick to indicate that the names are non-syntactic, which is precisely how you would refer to the names when interacting with the dataframe, for instance selecting roles.

```
df_dtable[`x million, $` == 5]
```

```
##    x million, $ y project
## 1:            5         c
```

```
dplyr::filter(df_readr, `x million, $` == 5)
```

```
## # A tibble: 1 x 2
##   `x million, $` `y project`
##            <dbl> <chr>
## 1              5 c
```

2.2 The mean question

Consider the following and decide if they are point to the same underlying function object.

```
mean
base::mean
get('mean')
evalq(mean)
match.fun('mean')
```

First of all, they all have the same output. The output looks like:

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000018ee6d20>
## <environment: namespace:base>
```

However, having the same output does not imply the same underlying object (in terms of address in memory). `m <- 1:4` and `n <- 1:4` have the same output but different underlying objects as can be seen using `obj_addr(m) == obj_addr(n)` which produces `FALSE` though `identical(m, n)` results to `TRUE`.

To answer the question, we could do `obj_addrs(list(mean, base::mean, get('mean'), evalq(mean), match.fun('mean')))` and notice that the memory addresses are the same and hence they all point to the same underlying object.

A more intuitive way to look at this is to realize that a memory is located when an object is created. All the above are ways of accessing the same function object with the name `mean` and not create objects. The function object which computes the mean of numbers has been assigned the name `mean` and it's located in the base package. Base package/library is loaded at each session of R thus memory is created to store the function object and assignment operator does the binding of the function object to the name `mean`. None of the constructs above is creating another object but rather access an object via it's name which was loaded in memory once at the beginning of R session.

> If `obj_addr(a) == obj_addr(b)` is `TRUE` then `identical(a, b)` is also `TRUE`. However, the reverse is not necessarily true.

2.3 Copy-on-modify

Restating our findings so far:

- a memory is alloted when an object is created (or modified)
- for binding to occur an object first has to exist (in memory)
- binding to a name is done via the assignment operator (binding from RHS - object to LHS - name)

- every call in R is by a function

Consider this

```
x <- c(1, 2, 3)
y <- x
y[[3]] <- 4
y
```

```
## [1] 1 2 4
```

```
x
```

```
## [1] 1 2 3
```

We have stated earlier that both `x` and `y` are pointing to the same object. Can we inteprete line 3 as we are modifying `y`? And does modifying `y` automatically modify `x` since they are bound to the same object? Why is `x` different from new `y`? The answer is that R did not modify the original object but a copy of the object and assigned the resulting object to `y`. Why should R create a copy instead of using the original object? Why should R work on a copy instead of the original? Is there something fundamental (could be a principle) in R that will be compromised if the original is modified instead? I think it's the fact that every call is by a function and the fact that memory is created at each object creation and we don't want to mix memories.

First, we rewrite the infix form of line 3 in prefix form as `` `<-`(`[[`(y, 3), 4) ``. When `[[` is used in conjunction with assignment operator it works as a replacer instead of extracter. Thus, we replace (or modify) an object or value of an object with another object or value.

When data is modified, a new memory is allocated. That allocation first takes time to process (thus allocation takes processing time) and occupies memory. Thus, we lose in terms of processing time and memory. It's a good thing that a copy of the object is created instead of the original object being modified, however, since a new memory is allocated for each new modification, memory becomes used up very quickly (that is filled with both the original and copies) and hence creates a slow application. With high-volume data or large datasets, memory is already an issue, so manipulating large objects with R can create problems with memory. NB: Modifying data frame using `data.table` (converted to data.table object) is more efficient that ordinary data frame.

Also, the right hand side is a creation of an object and thus calls for memory allocation. In addition, there is an object creation at the LHS when we use `[[` since doing `y[[3]]` without the assignment creats an object (in this case would return the third element of `y` thus 3). This is a clue that the resulting object has a different memory location from the original and also, we are not modifying the original object but a copy which later get assigned to a name `y`.

2.3.1 tracemem()

The `tracemem()` function is helpful to see when an object gets copied. One can know the number of times an object get copied in a single operation for instance.

```
x <- 1:5
tracemem(x)
```

```
## [1] "<000000001CA2C790>"
```

```
y <- x
y[[3]] <- 12L
```

```
## tracemem[0x000000001ca2c790 -> 0x000000002379cb30]: eval eval withVisible withCallin
```

```
untracemem(x)
```

We see that only one copy of `x` occurs. It would not have made any difference even if we had traced the memory of `y` since they both point to the same object. In this case tracing `x` will be the same as tracing `y`. Realize that there is no modification and hence no copy when we did `y <- x`. The modification occurred when we did `y[[3]] <- 12L`.

```
x <- 1:5
y <- x
tracemem(y)
```

```
## [1] "<000000001D0D4A70>"
```

```
y[[3]] <- 12L
```

```
## tracemem[0x000000001d0d4a70 -> 0x00000000233147e8]: eval eval withVisible withCallin
```

```
untracemem(y)
```

Showing that only one copy would have been created either tracing `x` or `y`. Let's slightly change what we are assigning to the third element of the vector from integer to double.

```
x <- 1:5
y <- x
tracemem(y)
```

```
## [1] "<000000001946F518>"
```

```
y[[3]] <- 12
```

```
## tracemem[0x0000000001946f518 -> 0x000000001c787990]: eval eval withVisible withCallingHandlers
## tracemem[0x000000001c787990 -> 0x000000001e6516e8]: eval eval withVisible withCallingHandlers
```

```
untracemem(y)
```

Now we have two instances of different copies. Why?