Data Science with R

Part I: Assignments and Primitive Data Structures

Raphael Schleutker

Its not at all important to get it right the first time. Its vitally important to get it right the last time. — Andrew Hunt and David Thomas

Table of contents

R as a Calculator

R as a Calculator

In the simplest case, R can be used just as a calculator.

```
1 + 1
```

```
# [1] 2
```

```
5 - 1
```

```
# [1] 4
```

```
3 * 4
```

```
# [1] 12
```

```
6/2
```

```
# [1] 3
```

R as a Calculator

R knows most common math operations.

```
2**5
```

```
# [1] 32
```

```
2^5
```

```
# [1] 32
```

Does R also know rules like multiplication and division first, then addition and subtraction? **Try it!**

```
2 + 3 * 5
```

R as a Calculator

We can use parantheses to define the precedence of operations.

```
2 + 3 * 5
```

```
# [1] 17
```

```
(2 + 3) * 5
```

```
# [1] 25
```

R as a Calculator

R even knows some constants like π.

```
pi

# [1] 3.141593
```

Other numbers like e can be produced as the result of a function call (we will come to that later).

```
exp(1)  # exp(1) = e^1 = e

# [1] 2.718282
```

R as a Calculator

R also knows a lot of functions for everyday math.

```
sin(pi)

# [1] 0.00000000000000001224606

cos(pi)

# [1] -1

tan(pi)

# [1] -0.00000000000000001224647
```

R as a Calculator

What about some experiments?

```
1 / 0

# [1] Inf

0 / 0

# [1] NaN

1 / Inf

# [1] 0
```

R as a Calculator – Summary

Meaning	Operator
Basic operators	`+, -, *, /, ^, **`
Integer division, modulo	`%/%, %%`
Extreme values	`min(), max()`
Absolute value	`abs()`
Square root	`sqrt()`
Rounding	`round(), floor(), ceiling()`
Logarithm	`log(), log10(), log2(), exp()`
Sum, Product	`sum(), prod()`

R as a Calculator – Summary

Meaning	Operator
Basic operators	`+, -, *, /, ^, **`
Integer division, modulo	`%/%, %%`
Extreme values	`min(), max()`
Absolute value	`abs()`
Square root	`sqrt()`
Rounding	`round(), floor(), ceiling()`
Logarithm	`log(), log10(), log2(), exp()`
Sum, Product	`sum(), prod()`

R does'nt provide functions for everything. Sometimes, you need a basic mathematical knowledge to get the desired result, e.g. $\sqrt[3]{8} = 8^{\frac{1}{3}}$:

```
8^(1/3)


# [1] 2
```

Assignments in R

Assignments in R

What, if we want to store the result of a calculation for later use? We can create a variable and store the information on the computer. The variable is then the name for the computer to look up for the correct information.

```
a <- 4
a


# [1] 4
```

The assignment operator in R

When coming from other programming languages you might use the equal sign = to create variables. Eventhough it is possible to use this assignment operator in R (and you sometimes see other people using it) it is everything but advisible! The reason is beyond our scope for now. So always stick with <-.

Assignments in R

We can now use this stored value for further calculations:

```
a + 3

# [1] 7

a * a

# [1] 16

b <- 3 * a
b

# [1] 12
```

Assignments in R

We will learn later that everything that does something in R is a function and that every function has a return value. We can conclude from this that <- is a function (as it does something) and therefor it has to return something (as it is a function). Indeed it returns the right-hand value, which could be used for just another assignment:

```
d <- c <- 5
c

# [1] 5

d

# [1] 5
```

First 5 is assigned to the variable c. This returns the value 5 again (we could imagine that c <- 5 is replaced by 5) and consequentially 5 is assigned to the variable d as well.

Assignments in R

There is another assignment operator -> and it should not be hard to determine what it does in contrast to <-.

```
5 -> e
e


# [1] 5
```

Please avoid this! It's confusing and I am not aware of any particulary useful application of ->.

Assignments in R

Values stored in variables are not persistent! If you close R and re-open it, all the created variables are gone.

Why is that? When a program needs storage space it asks the operating system (Windows, iOS, Linux, ...) for space. The OS grants some space and the software can do what it want with it, for instance storing a value there. The name of the variable is then a name that the software uses to look up the address on the hard drive. If the software is closed, all the granted storage space goes back to the OS. The value is still stored on the hard drive but as soon as the OS needs the space for something else, it will overwrite the value. Additionally, you can not reconstruct the address of the value when you re-start the software in the future.

The practical consequence from this is that you have to write your code in a way so that you just have to re-run the script and everything is as before. This is not always possible. Sometimes, running a script takes a lot of time. In that case, we can store a persistent copy of the result as a file on the computer.

Primitive Data Structures

Primitive Data Structures

Key for successfull programming is understanding the data structures. There are dozens of different data structures in different programming languages. In R, fortunately, we are mainly dealing with 2 data structures: vectors and lists (well and some structures building on top of them ...). Vectors are the most primitive ones. They are just series of values, separated by a comma.

```
c(1, 2, 3, 4, 5)

# [1] 1 2 3 4 5
```

Here, we call the function `c()` (short for combine) to create a vector.

Of course, we can assign vectors to variable names as well.

```
x <- c(1, 2, 3, 4, 5)
x


# [1] 1 2 3 4 5
```

Indeed, R created vectors automatically before, when we entered only one value (a vector of length 1).

Primitive Data Structures

Vectors can have different types to store different kinds of information.

```
v <- c(TRUE, TRUE, FALSE)       # Logical values
x <- c(1, 2, 3, 4, 5)           # Numeric values
y <- c("a", "b", "c")           # Characters.
z <- c(1+2i, 3-4i)              # Complex values.
```

Primitive Data Structures

```
v

# [1]  TRUE  TRUE FALSE

x

# [1] 1 2 3 4 5

y

# [1] "a" "b" "c"

z

# [1] 1+2i 3-4i
```

If we don't know the type of value a variable holds, we can ask for it.

```
typeof(v)

# [1] "logical"

typeof(x)

# [1] "double"

typeof(y)

# [1] "character"

typeof(z)

# [1] "complex"
```

The type of a vector

One vector can only have one type. There is no exception to this rule. If you try to create a vector with different types, R will automatically coerce all values to the most general type.

```
t <- c(1, "a", TRUE, 6-3i)
t


# [1] "1"    "a"    "TRUE" "6-3i"
```

All values were coerced to characters.

Primitive Data Structures – Logical Values

Logical values can have two different values: `TRUE` or `FALSE`. They are usually the result of comparisons:

```
4 == 3

# [1] FALSE

"A" == "A"

# [1] TRUE
```

Primitive Data Structures – Logical Values

Logical values are very useful for control structures later. They allow us to do different things for different data.

```
a <- 58 # The result of a calculation that we don't know in advance.

if(a == 58) {
        b <- 4
} else {
        b <- 10
}

b


# [1] 4
```

Primitive Data Structures – Logical Values

Logical values can be used in math operations. In this case `TRUE` stands for `1` while `FALSE` stands for `0`.

```
TRUE + TRUE

# [1] 2

FALSE * 40

# [1] 0
```

Primitive Data Structures – Numeric Values

R knows different kinds of numeric values.

- Integer are self-explanatory. They are created by tailoring values with an `L`.
- Doubles are decimal numbers. Double refers to the precision with which a value is stored (double usually means 64bit).

```
1L
```

```
# [1] 1
```

```
1
```

```
# [1] 1
```

Primitive Data Structures – Characters

Characters are used to store textual information. Several characters are called a string.

```
a <- "Hello World"
```

Characters are not numeric, even if they consist of digit characters only.

```
"1" + "2"

# Error in "1" + "2":  non-numeric argument to binary operator
```

Primitive Data Structures – Characters

Of course, R has some useful functions to work with textual information.

```
paste("1", "2")

# [1] "1 2"

strsplit("Hello world!", split = " ")

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

We will see later how to work with strings and how we can use a combination of numeric values and strings to represent categorical data.

Forget it. You will never use it. Never...

At the beginning, it's hard to get your head around vectors but they turn out to be quite handy.

```
a <- c( 1, 2, 3, 4, 5)
a + 1

# [1] 2 3 4 5 6

a + c(1, 2)

# [1] 2 4 4 6 6
```

The operation happens elementwise and shorter vectors are recycled.

Many functions expect vectors.

```
sum(a)

# [1] 15

prod(a)

# [1] 120
```

There are some useful functions that create vectors.

```
1:10

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

seq(1, 20, 2)

#  [1]  1  3  5  7  9 11 13 15 17 19

rep(c(1, 2), each = 2)

# [1] 1 1 2 2
```

This way, you can create virtually all patterns. And sometimes, you need them...