01-Syntax, base, functions, loops and data types

Adi Sarid / adi@sarid-ins.co.il

March 2019

Goals of this exercise

Familiarize yourself with the basics of R. This exercise encompasses:

1. Setting variables
2. Various data types,
3. Working with vectors, data.frames
4. Installing and loading packages
5. Building functions and base-R iterations

Setting variables

A variables can be defined using the arrow notation (which we have already seen in the previous lesson `<-`). You can also do the same with `=` but its less common and should not be used (apart from a specific case, in function arguments, which we will discuss later).

```
a <- 1
b <- 2
c = 3 # just to show that this works

a
```

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

```
b
```

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

```
c
```

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

```
a*b
```

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

```
b*c
```

```
## [1] 6
```

Try to run the following code in the console. What is the problem with it, can you fix the code? ***Missing multiplication * sign***

```
d <- c(a + b)
```

Do not mix the assignment operator `=` (which I told you not to use), with the test equality operator `==`.

```
a == 1
```

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

```
b == a
```

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

```
b > a
```

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

Also note the use of logicals: Please explain what each of the following operators: `&` `|` `!` `!=`, you can use the following (and modify it in any way):

```
# TRUE FALSE and the likes
TRUE & FALSE
TRUE | FALSE
TRUE & TRUE
!TRUE
FALSE != TRUE
FALSE == FALSE
```

Try the following code. Bonus points if you can explain whats wrong with it (and why that is). ***Computer accuracy errors sqrt2^2 is not exactly 2***

```
sqrt2 <- sqrt(2)
sqrt2
```

```
## [1] 1.414214
```

```
2 == sqrt2^2
```

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

Data types

R has a number of "basic" data types:

- Integer
- Numeric (double)
- Date (posix)
- Factors
- Logicals

You can use `c()`, `rbind()`, `cbind()` to piece values together into vectors or more complex structures. Run the following code.

```
integer_example <- 10L
integer_example
```

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

```
numeric_example <- pi # pi is a reserved word...
numeric_example
```

```
## [1] 3.141593
```

```
character_examples <- "hello world"
character_examples
```

```
## [1] "hello world"
```

```
date_example <- as.Date("2018-10-01")
date_example
```

```
## [1] "2018-10-01"
```

```
factor_example <- as.factor(c("big", "big", "small", "medium", "small", "big", "bigger"))
factor_example
```

```
## [1] big    big    small  medium small  big    bigger
## Levels: big bigger medium small
```

```
summary(factor_example)
```

```
##    big bigger medium  small
##      3      1      1      2
```

```
logical_example <- c(TRUE, TRUE, FALSE, TRUE)
logical_example
```

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

Using the `c()` command try to piece together the `logical_example` with the `factor_example`, i.e. (replace the `???` with something else):

```
c(logical_example, factor_example)
```

```
##  [1] 1 1 0 1 1 1 4 3 4 1 2
```

What happend to the factor vector? does the resulting vector make sense? **factors tend to get mixed up when you combine them with other data types**

Do the same with the `date_example` and the `factor_example`. What happend now? What precautions would you take when working with factors? **Be very carfull when working with factors…**

---

data frames

Data frames are a more complex structure which contains mixed data. R comes bundled with a number of "classical" data frames. Try the following:

```
mtcars
iris
?mtcars
?iris
```

What types are the variables (columns) in each of these data sets? (double/factor/date/logical/integer/character)

Packages

An R package is a bundle of functions which share a common goal or vision. So far, we've been using base-r. The `tidyverse` packages is a package of packages. We will be working a lot with it. Let's try to load `tidyverse`.

```
library(tidyverse)
```

Did that work? if you got an error message you might need to install it. The following code will download and install the tidyverse. Be warned, this takes long.

```
install.packages("tidyverse")
```

Now if you installed the package, try to load it again `library(tidyverse)`. To use a function after you loaded a packages you can call `function_name(arg1 = ..., arg2 = ..., ...)`. Use `glimpse` to verify your answers for the previous questions (what types are the variables in mtcars and iris):

```r
library(tidyverse)
```

```
## -- Attaching packages ------------------------------------------------------------------------------

## v ggplot2 3.1.0     v purrr   0.2.5
## v tibble  2.0.1     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.3.1     v forcats 0.3.0

## -- Conflicts ---------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
glimpse(iris)
```

```
## Observations: 150
## Variables: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9,...
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1,...
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5,...
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1,...
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, s...
```

```
glimpse(mtcars)
```

```
## Observations: 32
## Variables: 11
## $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19....
## $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, ...
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 1...
## $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, ...
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.9...
## $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3...
## $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 2...
## $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, ...
## $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, ...
```

Use the function `count` to answer:

How many flower-types are there in `iris`? **Three: setosa, versicolor, virginica** How many cylinder values are there in `mtcars`? **Also three: 4, 6, 8**

```
count(iris, Species)
count(???, cyl)
```

Later on, we will learn some more convinient ways to answer such questions.

Functions and iteratons - intermediate exercise

We will discuss some base-R iterations, however, **in real situations you should do all in your power to avoid base-r loops!**.

In the following exercise you will build a function which computes the Fibonacci series (0, 1, 1, 2, 3, 5, 8, 13, 21,. . .), and a loop which does the same. You will compare their runtime using `bench::mark()`.

WARNING: This is a **relatively complex** exercise if you're not fluent in programming, and it's not directly related to data analysis. The reasone I am giving you this exercise is that it is a great exercise to reherse elements we were discussing, in one single exercise.

1. Functions and recursion (a function calling itself)
2. Base-r loops
3. Conditionals (`if...else if...else`).

First, if you don't know what the Fibonacci series is, go to wikipedia, Fibonacci number and read about it (just the intro, should suffice).

Complete the following function so that a call to the function will generate the nth Fibonnacci number. Replace the `???`. Rows which start with the hash sign `#` are comments and will be ignored.

```
fibonnacci <- function(n){
  if (n == ???) {
    # starting condition for F_0
    return(0)
  } else if (??? == 1) {
    # starting condition for F_1
    return(1)
  } else {
    # use recursion to calculate the number
    return(fibonnacci(n-1) + ???(n-2))
  }
}

fibonnacci(30)
```

An alternative way to compute the Fibonnacci is via loops. Complete the following code:

```
fib_loop <- function(n){
  f_n_minus_1 <- 1
  f_n_minus_2 <- 0
  for (i in 1:n){
    f_n_new <- f_n_minus_1 + ???
    f_n_minus_1 <- ???
    f_n_minus_2 <- ???
  }
  f_n_new
}
```

Check that your are getting consistent results. Now, compare the two functions using:

```
install.packages(bench) # if it is not installed
bench::mark(fibonacci(30), fib_loop(30))
```

Which method is quicker?

Note the use of `::`, I didn't mention this earlier, but instead of loading the package entirely `library(bench)` we're just calling the function `mark` from packages`bench` directly, using the double `::`.

```
fibonacci <- function(n){
  if (n == 0) {
    # starting condition for F_0
    return(0)
  } else if (n == 1) {
    # starting condition for F_1
    return(1)
  } else {
    # use recursion to calculate the number
    return(fibonacci(n-1) + fibonacci(n-2))
  }
}

fib_loop <- function(n){
  f_n_minus_1 <- 1
  f_n_minus_2 <- 0
  n <- 30
  for (i in 1:n){
    f_n_new <- f_n_minus_1 + f_n_minus_2
    f_n_minus_1 <- f_n_minus_2
    f_n_minus_2 <- f_n_new
  }
  f_n_new
}

fibonacci(30)
```

```
## [1] 832040
```

```
fib_loop(30)
```

```
## [1] 832040
```

```
bench::mark(fibonacci(30), fib_loop(30))
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is
## disabled.
```

```
## # A tibble: 2 x 10
##   expression   min   mean median     max `itr/sec` mem_alloc  n_gc n_itr
##   <chr>      <bch> <bch:> <bch:> <bch:t>     <dbl> <bch:byt> <dbl> <int>
## 1 fibonacci~ 1.09s  1.09s  1.09s   1.09s   9.14e-1        0B    20     1
## 2 fib_loop(~ 1.6us 2.99us  1.8us 138.8us   3.35e+5        0B     0 10000
## # ... with 1 more variable: total_time <bch:tm>
```