R programming language: conceptual overview

Maxim Litvak

2016-06-10

Outline

R description

R is a **dynamic** language for **statistical computing** that combines **lazy functional** features and **object-oriented programming**.

R Properties

Properties:

- Dynamic
- Statistical computing
- Lazy functional
- OOP

  … R users usually focus on statistical computing, however, understanding the rest is crucial to boost productivity.

Statistical computing

- You already know how it works :-)

Functional - Basics I

- Functional programming (FP) is a paradigm that prescribes to break down the task into evaluation of (mathematical) functions
- FP is not about organizing code in subroutines (also called "functions" but in different sense)! (this is called procedural programming)
- It's about organizing the whole programm as function

Functional - Basics II

- Functions as first-class objects
  - can be passed as an argument
  - returned from a function
  - assigned to a variable
- Think of examples to the points above!

Functional - Scoping

Functional - Lazy

- “lazy” (or “call-by-need”) means evaluation is delayed until value is needed
- What do you think will the following piece of code work?

```
> f <- function(){g()}
```

Functional - Lazy

- It's valid even though we use function g() which isn't defined
- We kind of "promise" that it's gonna be defined to the time than f is called
- . . . but if we don't keep our promise

```
> f()
Error in f() : could not find function "g"
```

Functional - Lazy

- Now, let's define the function g() before calling the function f()

```
> g <- function() 0 # now g() is defined
> f()
[1] 0
```

- Now it works

Function - Referential transparency I

- Referential transparency - if an expression can be replaced with its value without changing the behaviour of the program (side effect)
- In R it's up to the developer, she/he should be however conscious if their code produce side effects
- Assume function **g** returns 0 and function **f** returns the only argument (f <- function(x) x). Is there a difference between
  - f(0)
  - f(g())

Function - Referential transparency IIa

- Which of the following 2 cases are referential transparent?

Function - Referential transparency IIb

- (cont.)
  - I

    ```
    > executed <- FALSE
    > g <- function(){
            executed <<- TRUE
            return(0)
            }
    > f(g())
    ```
  - II

    ```
    > executed <- TRUE
    > g <- function(){
            executed <- FALSE
            return(0)
            }
    > f(g())
    ```

Dynamic: Typing - I

- Types are optional and could be changed

Code

```
> var <- FALSE
> class(var)
[1] "logical"
> var
[1] FALSE
> var[3] <- 1
> class(var)
"numeric"
> var
[1] 0 NA 1
```

Dynamic: Typing - II

- What do you think would be the type of “var” variable after the following action?

```
> var <- "!"
> var[3] <- 1
```

Dynamic: Typing - III

- Types are implicitly there (assigned by compiler)
- Types could be changed (implicitly by compiler)

Dynamic: Evaluation (Language abstraction)

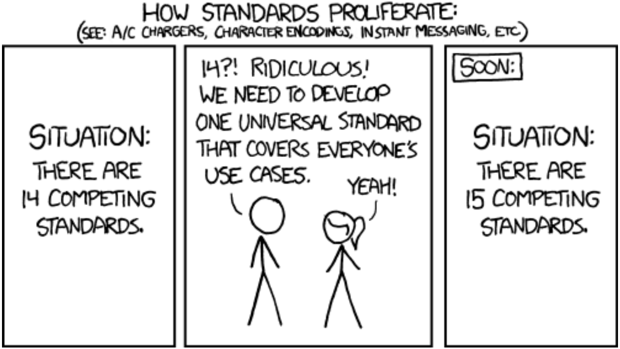
- With “eval” you can dynamically evaluate code, e.g. > eval(parse=text(“f <- function(x) x”))
- It allows to have more freedom in code manipulation (example will follow), beware performance!
- R allows to “abstract” the language itself

OOP - Basics

- Object-oriented programming is a paradigm in programming that prescribes to break down the task into objects with particular behaviour and data.

OOP in R

- Competing OOP standards in R: S3 (old), S4 (newer), reference classes, special libraries (R6, proto)
- xkcd:



OOP in R: S4

- Assume an object of class “Company” has 2 properties: headcount (HC) and earnings (EBIT)
- if you “add” (i.e. merge) 2 companies, then you add up their earnings +20% (synergy effects) and add up their headcount -20% (economies of scale)

OOP in R: S4

- Solution

```
> setClass("Company"
          , representation(HC = "numeric"
                          , EBIT = "numeric")
)

> setMethod("+"
          , signature("Company", "Company")
          , function(e1, e2){
    new("Company"
          , HC = (e1@HC + e2@HC)*0.8
          , EBIT = (e1@EBIT + e2@EBIT)*1.2
          )
})
> Microsoft <- new("Company"
          , HC = 50, EBIT = 95)
```

OOP in R: S4

- Result

```
> Microsoft + LinkedIn
An object of class "Company"
Slot "HC":
[1] 41.6

Slot "EBIT":
[1] 120
```

Comparison to other languages

- Python

```
class Company():
  def __init__(self, HC, EBIT):
    self.HC = HC
    self.EBIT = EBIT
  def __add__(self, other):
    return Company((self.HC+other.HC)*0.8
        ,(self.EBIT + other.EBIT)*1.2)
  def __repr__(self):
    out="HC:%s,EBIT:%s"%(self.HC,self.EBIT)
    return out

>>> Microsoft = Company(50, 95)
>>> LinkedIn = Company(2, 5)
```

Comparison to other languages

```
class Company
{
  private double HC;
  private double EBIT;
  public Company(double HC, double EBIT)
  {this.HC = HC;this.EBIT = EBIT;}
  public static operator +(Company A
                          , Company B)
  {
    double HC = (A.HC + B.HC)*0.8;
    double EBIT = (A.EBIT + B.EBIT)*1.2;
    return new Company(HC, EBIT)
  }
}
```

Statistical computing - revision

- Example: given X (e.g. "norm") distribution
    - pX is its probability function
    - dX is its density function
    - qX is its quantile function
- How to abstract X?
- Construct a function that takes name of the distribution with 2 parameters as an argument (e.g. "norm", "unif") and returns its quantile function parametrized with [0;1] (hint: use "eval")

Possible solution

- 1-st step: how could it look for a particular function eval(parse(text="function(x) qnorm(x,0,1)"))
- 2-nd step: separate distribution parameter

```
eval(
  parse(
    text=paste0(
      "function(x) q","norm","(x,0,1)"
        )
    )
)
```

```
function (x)
qnorm(x, 0, 1)
```

Possible solution (cont.)

- 3-rd step: abstract distribution as an argument and return as function

```
F <- function(dist){
  eval(parse(
    text=paste0(
      "function(x) q", dist ,"(x,0,1)"
    )
))
}
```

- Now you can get quantiles for different distributions
  - Log-normal

    > F("lnorm")(0.5) "1"
  - Uniform

    > F("unif")(0.8) "0.8

Last remark

- Further it can be generalize to distributions with different number of parameters and pass parameters as an argument

References

- Morandat, Floréal, et al. “Evaluating the design of the R language.” ECOOP 2012–Object-oriented programming. Springer Berlin Heidelberg, 2012. 104-131.

Repository

- You can find the latest version of this presentation here:
- github.com/maxlit/workshops/tree/master/R/r-advanced-overview