Reproducible Computation

E. F. Haghish

May 8, 2016

Defensive Programming

Defensive Programming

Defensive programming is a form of *defensive design* intended to ensure the continuing function of a piece of software **under unforeseen circumstances**. Defensive programming techniques are used especially when a function could be misused. A difference between defensive programming and non defensive programming is that **few assumptions** are made by the programmer, who attempts to handle all possible error states.

- Reliability of the function
- Making the source code comprehensible
- Making the function behave in a **predictable manner despite unexpected inputs**

Finding the balance

- Defensive programming means
  - Extra care
  - Extra runtime
  - Extra programming cost
  - Extra maintenance cost
  - Increased chance of mistakes through dealing with too many exceptions
  - Reliable functionality
- Lack of defensive programming
  - Confusing errors
  - False results
  - Crash

What really concerns the class?

The computation can be - reproduced on another machine - By independent researcher, without requiring any assistance - Version control

```
version
```

```
##                _
## platform       x86_64-apple-darwin13.4.0
## arch           x86_64
## os             darwin13.4.0
## system         x86_64, darwin13.4.0
## status
## major          3
## minor          2.4
## year           2016
## month          03
## day            10
## svn rev        70301
## language       R
## version.string R version 3.2.4 (2016-03-10)
## nickname       Very Secure Dishes
```

Input

- Secure input handling
  - Validating the input prior to execution
  - Terminate on inappropriate input
  - Use white list (define what is allowed)
  - Use black list (define what is not allowed)

R Objects

- vectors (one-dimensional arrays)
- arrays
- lists
- matrices
- tables
- data frames

Evaluating objects

Examples of object evaluation

```
missing()
is.vector()
is.array()
is.matrix()
is.list()
is.na()
is.null()
is.logical()
is.integer()
is.numeric()
is.character()
```

Unspecified arguments

Unspecified arguments are **missing** arguments. If a default value is specified, they are yet **missing** although they can be evaluated.

```
a <- function (x = 0) {
    if (missing(x)) warning("y is missing")
    print(x)
}

a()
```

```
## Warning in a(): y is missing
```

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

Example

Write a function that take 3 arguments, value of `x` which is a **numerical input**, `n` which is an intiger, and `df` degrees of freedom which is also an integer. call the function `chi.probability`.
Since we know what object types are allowed in the function, we an proceed to evaluate them using a **whitelist**.

Object type

R creates "double" type for numeric objects which includes both integers and real numbers.

```r
a <- 10
typeof(a)
```

```
## [1] "double"
```

to check if a number is integer, we have to compare it with its rounded value.

```r
all.equal(a, as.integer(a))
```

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

Possible messages

- message() # only print a message
- warning() # only print a warning
- stop() # stops execution

The `stop` function also indicates in which function the error has happened. This can be suppressed by adding `call. = FALSE` argument to the `stop` function

Create a R package project in Rstudio

R package structure

- functions
- **help files**
- package description
- **namespace**
- Add a REDMEA.md file and upload the directory to GitHub

Installing from GitHub

after pushing the package to GitHub, we can install the package by using `devtools`

```
install.packages("devtoops")
install.packages("roxygen2") #dynamic documentation
library(devtools)
```

Install the package and load it

```
install_github("haghish/temp")
library(temp)
```

Documenting the R package

- How does the function work?
- What are its arguments?
- What are the inputs?
- What does it return?
- WHo is the author?
- **What functions or packages does it rely on**?
    - If it relies on a package, what version of it?
- Any example how it works?

Using Roxygen2 package for dynamic documentation

- The documentation can be written within the functions (R script)
  - Infer on the code
  - Easier to update and maintain
  - Maintained the same time a change is made in the function

Roxygen commands

```
@title       #Title
@usage       #Syntax
@keywords    #Keywords
@description #description of the function
@param       #defines the arguments
@author      #author information
@examples    #usage examples
@export      #makes the function available in the global envi
@import      #import a package (apart from base)
@importFrom pkg fname #import a function from a package
```