Herramientas Computacionales para la Investigación Interdisciplinaria Reproducible

- Profesor: Dr. José Manuel Magallanes, PhD (jmagallanes@pucp.edu.pe)Profesor del **Departamento de Ciencias Sociales, Pontificia Universidad Católica del Peru**. Senior Data Scientist del **eScience Institute** and Visiting Professor at **Evans School of Public Policy and Governance, University of Washington**. Fellow Catalyst, **Berkeley Initiative for Transparency in Social Sciences, UC Berkeley**.

Sesión 2: Introducción al R

Part C: Funciones

Las funciones nos ayudan a organizar código que cumple un 'rol' específico. Así, se facilita la lectura y corrección del mismo, al crear elementos independientes. Así mismo, usando funciones verás como integrar lo visto en las secciones anteriores.

Veremos tres elementos organizadores: input, transformación, output. Por ejemplo, si necesitas convertir un valor numérico de Fahrenheit en Celsius, el input is el valor en Fahrenheit, la transformación es la formula, y la output es el resultado en Celsius (u otro que corresponda).

```
# ejecutar
converterToCelsius=function(valueInFarenheit){ #input
  #transformacion
  resultInCelsius= (valueInFarenheit-32)*5/9
  #output
  return (resultInCelsius)
  }
```

Del ejemple anterior, veamos la sintaxis de una función. El primer element es el nombre (*converterToCelsius*) a la izquierda de =; luego el *input*, que es un detalla de lo que la función espera se ingrese usando la palabra reservada **function**; de ahí, la *transformación* se escribirá dentro de los **{}**; la última fila de la transformación tiene devuelve la *output* (for example: *resultInCelsius*), para lo cual se usa la palabra clave **return**. Si escribes sólo la *output* sin usar el comando *return* la función aun funciona, pero disminuye su *readability*. El código ejemplo le ha dado una nueva función a R:

```
converterToCelsius(100)
```

He organizado los materiales de la siguiente manera:

1. Input.
2. Output.
3. Aplicando funciones
   - a estructuras simples.
   - a estructuras compuestas.

El Input

El input no tiene que ser sólamente *uno*:

```
# 2 inputs:
XsumY=function(valueX,valueY){
  ###
  resultSum=valueX+valueY
  ###
```

```
  return (resultSum)
}
```

Veamos cómo funciona:

```
XsumY(3,10)
```

Algún input puede tener un valor *default*:

```
riseToPower=function(base,exponent=2){
  #####
  result=1
  if (exponent > 0){
    for (time in 1:exponent){
      result=result*base
    }
  }
  #####
  return(result)
}
```

Veamos cómo funciona:

```
riseToPower(9)
riseToPower(9,3)
riseToPower(9,0)
```

Nota aqui unas diferencias:

```
# quizas esto es mas claro:
riseToPower(base=9,exponent=0)
# con argumentos explicitos no se require orden
riseToPower(exponent=0,base=9)
```

Siempre hay que estar listo para saber qué hacer ante posibles cálculos; para evitar ello:

```
# Then
divRounded=function(numerator,denominator,precision=2){
    if (denominator==0){
      return('0 en el denominador')
    }else{
    result = numerator/denominator
    return (round(result, precision))}
}
```

Testing this function:

```
n=13
d=12
p=3
divRounded(n,d,p)
```

Podemos preparar una lista como input e ingresarla a la función con la ayuda de **do.call**:

```
inputArgs=list(precision=3,numerator=13,denominator=12)
do.call(divRounded,inputArgs)
```

Ir al inicio

La Output

El output no tiene que ser un único valor:

```
factors=function(number){
  vectorOfAnswers=c() # para guardar respuestas
  for (i in 1:number){
    #si el residuo de 'number'/'i' es igual a cero...
    if ((number %% i) == 0){
      # ...añade 'i' a la respuesta!
      vectorOfAnswers=c(vectorOfAnswers,i)
    }
  }
  return (vectorOfAnswers)
}
```

Testing:

```
factors(20)
```

Ir al inicio

Aplicando funciones a estructuras simples

Creemos una función:

```
double=function(x){
    return (2*x)}
```

Creemos un vector:

```
myVector=c(1,2,3)
```

¿Qué sale de aquí?

```
double(myVector)
```

La función aplicada a un vector, aplica la transformación a cada elemento.

Pero esto te da error:

```
myList=list(1,2,3)
double(myList)
```

Aquí lo importante es saber aplicar la función según el tipo de input. Usemos **Map**, con vectores:

```
Map(double,myVector) # devuelte lista
```

con Lista:

```
Map(double,myList) # devuelte lista
```

Tenemos también **mapply**. Aquí con vector:

```
mapply(double,myVector)
```

Aquí con lista:

```
mapply(double,myList)
```

Nota que la estructura de la output es distinta.

Ir al inicio

Aplicando funciones a estructuras compuestas

Creemos un data frame:

```
numberA=c(10,20,30,40,50)
numberB=c(6,7,8,9,10)
dataDF=data.frame(numberA,numberB)
dataDF
```

Let's *double* each value.Let me here make a copy of my data frame:

```
# use a copy of the orginal
dataDF2=dataDF

# see the copy
dataDF2
```

Now that I got a copy, let me use loops in that copy:

```
for (column in 1:ncol(dataDF2)){
 for (row in 1:nrow(dataDF2[column])){
   dataDF2[row,column]=double(dataDF2[row,column])
 }
}

# updated:
dataDF2
```

The copy has new values, but my original stays the same:

```
dataDF
```

Now, let me apply the function *directly* to the data frame:

```
double(dataDF)
```

I *see* the result of 'doubling' the data frame, but the data frame is still unchanged:

```
dataDF
```

If you need the data frame to change, you need to do this:

```
dataDF=double(dataDF)
# now see it:
dataDF

# recreating the data frame:
dataDF=data.frame(numberA,numberB)
```

As you saw above, the function *double* was designed to receive as input a simple value (a number). Then, without effort from your side, R itself decided to apply it to each element in the data frame. You can put more effort on your side, and be more explicit; for that you need to use **lapply**:

```
lapply(dataDF,double)
# remember you are SEEING the result, but no changes to the orginal data frame!
```

This last function is extremely important. We got lists as output, but we can easily get the data frame:

```
as.data.frame(lapply(dataDF,double))
```

The function **lapply** will apply the function to *each column* of the data frame.

There are functions that could be applied to columns or rows, keep in mind that **lapply** applies a function to column.

```
# you are adding the column values here:
as.data.frame(lapply(dataDF,sum))
```

If you need to apply a function by row or by column, the right option is **apply**:

```
# you are adding by row:
apply(dataDF,1,sum) # 1 to apply by row (2 for column).
```

- Go to page beginning
- Go to PART A: Data Structures in R
- Go to PART B: Control of Execution in R
- Go to Course schedule

AUSPICIO:

El desarrollo de estos contenidos ha sido posible gracias al grant del Berkeley Initiative for Transparency in the Social Sciences (BITSS) at the Center for Effective Global Action (CEGA) at the University of California, Berkeley

RECONOCIMIENTO

El autor reconoce el apoyo que el eScience Institute de la Universidad de Washington le ha brindado desde el 2015 para desarrollar su investigación en Ciencia de Datos.