Session 8
Functions 2

R for Social Data Science

Jeffrey Ziegler, PhD

Assistant Professor in Political Science & Data Science
Trinity College Dublin

Fall 2022

Road map for today

Last time:

- Decomposition and abstraction
- Function definition and function call

This time:

- Anonymous functions
- Functionals
- Scoping in R

Anonymous functions

- While R has no special syntax for creating anonymous (aka lambda in Python) function
- Note that the result of ‘function()‘ does not have to be assigned to a variable
- Thus function ‘function()‘ can be easily incorporate into other function calls

Anonymous functions

```
add_five <- function(){
  return(function(x) x + 5)
}
af <- add_five()
af # 'af' is just a function, which is yet to be invoked (called)
```

```
function(x) x + 5
<environment: 0x55d78232a7d8>
```

```
af(10) # Here we call a function and supply 10 as an argument
```

```
[1] 15
```

```
# Due to vectorized functions in R this example is an obvious
    overkill (seq(10) ^ 2 would do just fine)
# but it shows a general approach when we might need to apply a
    non-vectorized functions
sapply(seq(10), function(x) x ^ 2)
```

Functionals

- *Functionals* are functions that take other functions as one of their inputs
- Due to R's functional nature, functionals are frequently used for many tasks
- 'apply()' family of base R functionals is the most ubiquitous example
- Their most common use case is an alternative of *for* loops
- Loops in R have a reputation of being slow (not always warranted)
- Functionals also allow to keep code more concise

Functional example

```
# Applies a supplied function to a random draw
# from the normal distribution with mean 0 and sd 1
functional <- function(f) { f(rnorm(10)) }
functional(mean)
```

```
[1] -0.09413735
```

```
functional(median)
```

```
[1] -0.1556706
```

```
functional(sum)
```

```
[1] -2.926588
```

Summary of common 'apply()' functions

Function	Description	Input Object	Output Object	Simplified
'apply()'	Apply a given function to margins (rows/columns) of input object	matrix/array/data.frame	vector/matrix/array/list	Yes
'lapply()'	Apply a given function to each element of input object	vector/list	list	No
'sapply()'	Same as 'lapply()', but output is **s**implified	vector/list	vector/matrix	Yes
'vapply()'	Same as 'sapply()', but data type of output is specified	vector/list	vector	No
'mapply()'	**M**ultivariate version of 'sapply()', takes multiple objects as input	vectors/lists	vector/matrix	Yes

Extra: Using apply, sapply, lapply in R

‘lapply()‘ function

- Takes a function and a vector or list as input
- Applies the input function to each element in the list
- Returns list as an onput

```
lapply(<input_object>, <function_name>, <arg_1>, ..., <arg_n>)}
```

'LAPPLY()' EXAMPLES

```r
l <- list(a = 1:2, b = 3:4, c = 5:6, d = 7:8, e = 9:10)
# Apply sum() to each element of list 'l'
lapply(l, sum)
```

```
 $a
[1] 3
$b
[1] 7
$c
[1] 11
$d
[1] 15
$e
[1] 19
```

‘LAPPLY()‘ EXAMPLES

```r
# We can exploit the fact that basic operators are function calls
# Here, each subsetting operator ‘[‘ with argument 2 is applied to each element
# Which gives us second element within each element of the list
lapply(l, ‘[‘, 2)
```

```
 $a
[1] 2
$b
[1] 4
$c
[1] 6
$d
[1] 8
$e
[1] 10
```

‘apply()‘ function

- Works with higher-dimensional (> 1d) input objects (matrices, arrays, data frames)
- Is a common tool for calculating summaries of rows/columns
- ‘<margin>‘ argument indicates whether function is applied across rows (1) or columns (2)

```
apply(<input_object>, <margin>, <function_name>, <arg_1>, ..., <arg_n>)}
```

'APPLY()' EXAMPLES

```r
m <- matrix(1:12, nrow=3, ncol=4)
m
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```r
# Sum up rows (can also be achieved with rowSums() function)
apply(m, 1, sum)
```

```
[1] 22 26 30
```

```r
# Calculate averages across columns (also available in colMeans())
apply(m, 2, mean)
```

```
[1]  2  5  8 11
```

```r
# Find maximum value in each column
apply(m, 2, max)
```

```
[1]  3  6  9 12
```

‘MAPPLY()‘ FUNCTION

- Takes a function and multiple vectors or lists as input
- Applies function to each corresponding element of input sequences
- Simplifies output into vector (if possible)

```
mapply(<function_name>, <input_object_1>, ..., <input_object_n>, <arg_1>, ..., <arg_n>)}
```

‘MAPPLY()‘ EXAMPLES

```r
means <- -2:2
sds <- 1:5
# Generate one draw from a normal distribution where
# each mean is an element of vector 'means'
# and each standard deivation is an element of vector 'sds'
# rnorm(n, mean, sd) takes 3 arguments: n, mean, sd
mapply(rnorm, 1, means, sds)
```

```
[1] -2.3877425 -3.8041251  1.2425808  4.2079390  0.2520243
```

‘MAPPLY()‘ EXAMPLES

```r
# While simplification of output
# (attempt to collapse it in fewer dimensions)
# makes hard to predict the object returned
# by apply() functions that have simplified = TRUE by default
mapply(rnorm, 5, means, sds)
```

```
          [,1]      [,2]       [,3]       [,4]       [,5]
[1,] -1.676801 -3.0455835  0.8957769  0.5118888 -6.4469782
[2,] -2.690624 -1.5524074 -1.4870650 -4.4084040  2.4245422
[3,] -1.664708 -0.9970396  0.9591408 -1.7019869  0.7672098
[4,] -1.400437 -1.9529977  1.0721986 -0.2210901  8.5994742
[5,] -1.958179  2.6664414  0.4189656 -1.5375013  8.7470140
```

Packages

- Program can access functionality of a package using ‘library()‘ function
- Every package has its own namespace (which can accessed with ‘::‘)

```
 library(<package_name>)
<package_name>::<object_name>
```

Package loading example

```r
# Package 'Matrix' is part of the standard R library and doesn't
    have to be installed separately
library("Matrix")
```

```
  Warning message:
“package ‘Matrix’ was built under R version 4.1.3
```

```r
# While it is possible to just use function sparseVector() after
    loading the library,
# it is good practice to state explicitly which package the object
     is coming from
sv <- Matrix::sparseVector(x = c(1, 2, 3), i = c(3, 6, 9), length
    = 10)
sv
```

```
  sparse vector (nnz/length = 3/10) of class "dsparseVector"
[1] . . 1 . . 2 . . 3 .
```

Tutorial: Exercise 1 - Functionals

- As R is a functional language, many of iteration routines can be avoided
- For example, instead of creating a loop for calculating standard deviations
- We are more likely to run a function ‘apply(<object_name>, 2, <function_name>)‘ to calculate desired summary statistic for each of variables
- Apply this function to matrix from the code on next slide (and in "Tutorial08.R")
- Now, change 2 in the function call to 1
- What do you see? What do the current numbers show? Does this summary make sense and why?

Tutorial: Exercise 1 - Functionals

```r
# Remember to make your code replicable by setting seed
set.seed(2022)
# Here we create a matrix of 30 observations of 5 variables
# where each variable is a random draw from a normal distribution with mean 0
# and standard deviation drawn from a uniform distribution between 0 and 10
mat <- mapply(
  function(x) cbind(rnorm(n = 30, mean = 0, sd = x)),
  runif(n = 5, min = 0, max = 10)
)
```

Tutorial: Exercise 2 - Functions

- Let's turn to a more complicated case
- Below you can see another matrix object, but this time it's interspersed with letters
- What is the type of this matrix?Write a function that can take this matrix as an input and return a list, where each element is a column of the input matrix
- Internally, you can re-use loop from previous exercise
- In addition to that while building iteratively your list try checking whether a column is coercible into numeric

Tutorial: Exercise 2 - Functions

```r
set.seed(2022)
mat2 <- cbind(
  letters[sample.int(26, 30, replace = TRUE)],
  mapply(
    function(x) cbind(rnorm(n = 30, mean = 0, sd = x)),
    runif(n = 3, min = 0, max = 10)
  ),
  letters[sample.int(26, 30, replace = TRUE)]
)
```

Overview

This week:

- Decomposition and abstraction
- Function definition and function call
- Functionals
- Scoping in R

Next week:

- Debugging
- Testing
- Performance and complexity