Week 4: Functions in R

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

3 October 2022

Module website: tinyurl.com/POP77001

Overview

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

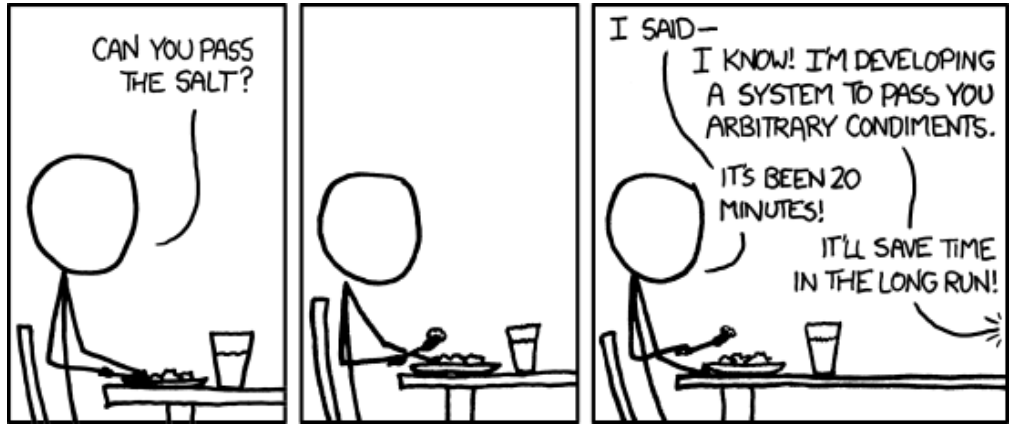
Decomposition and abstraction

Source: IKEA

Decomposition and abstraction

- So far: built-in types, assignments, branching and looping constructs
- In principle, any problem can be solved just with those
- But a solution would be non-modual and hard-to-maintain
- Functions provide *decomposition* and *abstraction*

Functions



Source: xkcd

Functions in R

- Function call is the centerpiece of computation in R
- It involves function object and objects that are supplied as arguments
- Functions in R do not have side-effects (nonlocal modifications of input objects)
- In R we use function `function()` to create a function object
- Functions are also referred to as *closures* in some R documentation

```
<function_name> <- function(<arg_1>, <arg_2>, ..., <arg_n>) {
  <function_body>
}
```

Functions in R

- Function call is the centerpiece of computation in R
- It involves function object and objects that are supplied as arguments
- Functions in R do not have side-effects (nonlocal modifications of input objects)
- In R we use function `function()` to create a function object
- Functions are also referred to as *closures* in some R documentation

```
<function_name> <- function(<arg_1>, <arg_2>, ..., <arg_n>) {
  <function_body>
}
```

In [2]:

```
foo <- function(arg) {
  # <function_body>
}
```

Function components

- Body (`body()`) - code inside the function
- List of arguments (`formals()`) - controls how function is called
- Environment/scope/namespace (`environment()`) - location of function's definition and variables

Function components example

Function components example

In [3]:

```
is_positive <- function(num) {
  if (num > 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

Function components example

In [3]:
```
is_positive <- function(num) {
  if (num > 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

In [4]:
```
body(is_positive)
```

```
{
    if (num > 0) {
        return(TRUE)
    }
    else {
        return(FALSE)
    }
}
```

Function components example

In [3]:
```
is_positive <- function(num) {
  if (num > 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

In [4]:
```
body(is_positive)
```

```
{
    if (num > 0) {
        return(TRUE)
    }
    else {
        return(FALSE)
    }
}
```

In [5]:
```
formals(is_positive)
```

```
$num
```

Function components example

In [3]:
```
is_positive <- function(num) {
  if (num > 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

In [4]:
```
body(is_positive)
```

```
{
    if (num > 0) {
        return(TRUE)
    }
    else {
        return(FALSE)
    }
}
```

In [5]:
```
formals(is_positive)
```

```
$num
```

In [6]:
```
environment(is_positive)
```

```
<environment: R_GlobalEnv>
```

Function call

- Function is executed until:
    - Either `return()` function is encountered
    - There are no more expressions to evaluate
- Function call always returns a value:
    - Argument of `return()` function call
    - Value of last expression if no `return()` (implicit return)
- Function can return only one object
    - But you can combine multiple R objects in a list

Function call example

Function call example

In [7]:

```r
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
  return(res)
}
```

Function call example

In [7]:
```r
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
  return(res)
}
```

In [8]:
```r
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
```

Function call example

In [7]:
```
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
  return(res)
}
```

In [8]:
```
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
```

In [9]:
```
print(res_1)
print(res_2)
```

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

Implicit return example

Implicit return example

In [10]:

```r
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
  res
}
```

Implicit return example

In [10]:
```
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
  res
}
```

In [11]:
```
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
```

Implicit return example

In [10]:
```
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
  res
}
```

In [11]:
```
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
```

In [12]:
```
print(res_1)
print(res_2)
```

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

Implicit return example continued

Implicit return example continued

In [13]:

```r
# While this function provides the same functionality as the two versio
# This is an example of a bad programming style, return value is very u
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
}
```

Implicit return example continued

In [13]:
```r
# While this function provides the same functionality as the two versio
# This is an example of a bad programming style, return value is very u
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
}
```

In [14]:
```r
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
```

Implicit return example continued

In [13]:
```
# While this function provides the same functionality as the two versio
# This is an example of a bad programming style, return value is very u
is_positive <- function(num) {
  if (num > 0) {
    res <- TRUE
  } else {
    res <- FALSE
  }
}
```

In [14]:
```
res_1 <- is_positive(5)
res_2 <- is_positive(-7)
```

In [15]:
```
print(res_1)
print(res_2)
```

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

Function arguments

- *Arguments* provide a way of giving input to a function
- Arguments in function definition are *formal arguments*
- Arguments in function invocations are *actual arguments*
- When a function is invoked (called) arguments are matched and bound to local variable names
- R matches arguments in 3 ways:
    1. by *exact name*
    2. by *partial name*
    3. by *position*
- It is a good idea to only use unnamed (positional) for the main (first one or two) arguments

Function arguments example

Function arguments example

In [16]:

```
format_date <- function(day, month, year, reverse = TRUE) {
  if (isTRUE(reverse)) {
    formatted <- paste(
      as.character(year), as.character(month), as.character(day), sep =
    )
  } else {
    formatted <- paste(
      as.character(day), as.character(month), as.character(year), sep =
    )
  }
  return(formatted)
}
```

Function arguments example

In [16]:
```
format_date <- function(day, month, year, reverse = TRUE) {
  if (isTRUE(reverse)) {
    formatted <- paste(
      as.character(year), as.character(month), as.character(day), sep =
    )
  } else {
    formatted <- paste(
      as.character(day), as.character(month), as.character(year), sep =
    )
  }
  return(formatted)
}
```

In [17]:
```
format_date(4, 10, 2021)
```

```
[1] "2021-10-4"
```

Function arguments example

In [16]:
```
format_date <- function(day, month, year, reverse = TRUE) {
  if (isTRUE(reverse)) {
    formatted <- paste(
      as.character(year), as.character(month), as.character(day), sep =
    )
  } else {
    formatted <- paste(
      as.character(day), as.character(month), as.character(year), sep =
    )
  }
  return(formatted)
}
```

In [17]:
```
format_date(4, 10, 2021)
```

```
[1] "2021-10-4"
```

In [18]:
```
format_date(y = 2021, m = 10, d = 4) # Technically correct, but rather
```

```
[1] "2021-10-4"
```

Function arguments example

In [16]:
```r
format_date <- function(day, month, year, reverse = TRUE) {
  if (isTRUE(reverse)) {
    formatted <- paste(
      as.character(year), as.character(month), as.character(day), sep =
    )
  } else {
    formatted <- paste(
      as.character(day), as.character(month), as.character(year), sep =
    )
  }
  return(formatted)
}
```

In [17]:
```r
format_date(4, 10, 2021)
```

```
[1] "2021-10-4"
```

In [18]:
```r
format_date(y = 2021, m = 10, d = 4) # Technically correct, but rather
```

```
[1] "2021-10-4"
```

In [19]:
```r
format_date(y = 2021, m = 10, d = 4, FALSE) # Technically correct, but
```

```
[1] "4-10-2021"
```

Function arguments example

In [16]:
```r
format_date <- function(day, month, year, reverse = TRUE) {
  if (isTRUE(reverse)) {
    formatted <- paste(
      as.character(year), as.character(month), as.character(day), sep =
    )
  } else {
    formatted <- paste(
      as.character(day), as.character(month), as.character(year), sep =
    )
  }
  return(formatted)
}
```

In [17]:
```r
format_date(4, 10, 2021)
```

```
[1] "2021-10-4"
```

In [18]:
```r
format_date(y = 2021, m = 10, d = 4) # Technically correct, but rather
```

```
[1] "2021-10-4"
```

In [19]:
```r
format_date(y = 2021, m = 10, d = 4, FALSE) # Technically correct, but
```

```
[1] "4-10-2021"
```

In [20]:
```
format_date(day = 4, month = 10, year = 2021, FALSE)
```

```
[1] "4-10-2021"
```

Nested functions

Nested functions

In [21]:

```
which_integer <- function(num) {
  even_or_odd <- function(num) {
    if (num %% 2 == 0) {
      return("even")
    } else {
      return("odd")
    }
  }
  eo <- even_or_odd(num)
  if (num > 0) {
    return(paste0("positive ", eo))
  } else if (num < 0) {
    return(paste0("negative ", eo))
  } else {
    return("zero")
  }
}
```

Nested functions

In [21]:
```r
which_integer <- function(num) {
  even_or_odd <- function(num) {
    if (num %% 2 == 0) {
      return("even")
    } else {
      return("odd")
    }
  }
  eo <- even_or_odd(num)
  if (num > 0) {
    return(paste0("positive ", eo))
  } else if (num < 0) {
    return(paste0("negative ", eo))
  } else {
    return("zero")
  }
}
```

In [22]:
```r
which_integer(-43)
```

```
[1] "negative odd"
```

Nested functions

In [21]:
```
which_integer <- function(num) {
  even_or_odd <- function(num) {
    if (num %% 2 == 0) {
      return("even")
    } else {
      return("odd")
    }
  }
  eo <- even_or_odd(num)
  if (num > 0) {
    return(paste0("positive ", eo))
  } else if (num < 0) {
    return(paste0("negative ", eo))
  } else {
    return("zero")
  }
}
```

In [22]:
```
which_integer(-43)
```

```
[1] "negative odd"
```

In [23]:
```
even_or_odd(-43)
```

```
Error in even_or_odd(-43): could not find function "even_or_od
```

```
d"
Traceback:
```

R environment basics

- Variables (aka names) exist in an *environment* (aka namespace/scope in Python)
- The same R object can have different names
- Binding of objects to names (assignment) happens within a specific environment
- Most environments get created by function calls
- Approximate hierarchy of environments:
    - *Execution* environment of a function
    - *Global* environment of a script
    - *Package* environment of any loaded packages
    - *Base* environment of base R objects

R environment example

R environment example

In [24]:

```r
x <- 42
# is equivalent to:
# Binding R object '42', double vector of length 1, to name 'x' in the
assign("x", 42, envir = .GlobalEnv)
x
```

```
[1] 42
```

R environment example

In [24]:
```r
x <- 42
# is equivalent to:
# Binding R object '42', double vector of length 1, to name 'x' in the
assign("x", 42, envir = .GlobalEnv)
x
```

```
[1] 42
```

In [25]:
```r
x <- 5
foo <- function() {
  x <- 12
  return(x)
}
y <- foo()
print(y)
print(x)
```

```
[1] 12
[1] 5
```

```
x <- 42
```

```
`<-`(x, 42)
```

```
assign(
  “x”, 42,
  envir = .GlobalEnv
)
```

Every operation is a function call

Matthew Kay
@mjskay

remember kids everything in #rstats is a function

```
y = 9

for (x in 1:y) {
  if (x > 2) {
    xy = x * y
    cat(x, "*", y, "is", xy, "\n")
  }
}
#> 3 * 9 is 27
#> 4 * 9 is 36
#> 5 * 9 is 45
#> 6 * 9 is 54
#> 7 * 9 is 63
#> 8 * 9 is 72
#> 9 * 9 is 81
```

```
`=`(y, 9)

`for`(x, `:`(1, y), `{`(
  `if`(`>`(x, 2), `{`(
    `=`(xy, `*`(x, y)),
    cat(x, "*", y, "is", xy, "\n")
  ))
))
#> 3 * 9 is 27
#> 4 * 9 is 36
#> 5 * 9 is 45
#> 6 * 9 is 54
#> 7 * 9 is 63
#> 8 * 9 is 72
#> 9 * 9 is 81
```

11:23 pm · 31 Oct 2021 · Twitter Web App

39 Retweets **4** Quote Tweets **389** Likes

Examples of operators as function calls

Examples of operators as function calls

In [26]:
```
`+`(3, 2) # Equivalent to: 3 + 2
```

```
[1] 5
```

Examples of operators as function calls

In [26]:
```
`+`(3, 2) # Equivalent to: 3 + 2
```

```
[1] 5
```

In [27]:
```
`<-`(x, c(10, 12, 14)) # x <- c(10, 12, 14)
x
```

```
[1] 10 12 14
```

Examples of operators as function calls

In [26]:
```
`+`(3, 2) # Equivalent to: 3 + 2
```

```
[1] 5
```

In [27]:
```
`<-`(x, c(10, 12, 14)) # x <- c(10, 12, 14)
x
```

```
[1] 10 12 14
```

In [28]:
```
`[`(x, 3) # x[3]
```

```
[1] 14
```

Examples of operators as function calls

In [26]:
```
`+`(3, 2) # Equivalent to: 3 + 2
```

```
[1] 5
```

In [27]:
```
`<-`(x, c(10, 12, 14)) # x <- c(10, 12, 14)
x
```

```
[1] 10 12 14
```

In [28]:
```
`[`(x, 3) # x[3]
```

```
[1] 14
```

In [29]:
```
`>`(x, 10) # x > 10
```

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

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

- 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

In [30]:
```
add_five <- function() {
  return(function(x) x + 5)
}
af <- add_five()
```

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

In [30]:
```
add_five <- function() {
  return(function(x) x + 5)
}
af <- add_five()
```

In [31]:
```
af # 'af' is just a function, which is yet to be invoked (called)
```

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

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

In [30]:
```
add_five <- function() {
  return(function(x) x + 5)
}
af <- add_five()
```

In [31]:
```
af # 'af' is just a function, which is yet to be invoked (called)
```

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

In [32]:
```
af(10) # Here we call a function and supply 10 as an argument
```

```
[1] 15
```

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

In [30]:
```
add_five <- function() {
  return(function(x) x + 5)
}
af <- add_five()
```

In [31]:
```
af # 'af' is just a function, which is yet to be invoked (called)
```

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

In [32]:
```
af(10) # Here we call a function and supply 10 as an argument
```

```
[1] 15
```

In [33]:
```
# Due to vectorized functions in R this example is an obvious overkill
# but it shows a general approach when we might need to apply a non-vec
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

Functional example

In [34]:
```r
# 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 example

In [34]:
```
# Applies a supplied function to a random draw
# from the normal distribution with mean 0 and sd 1
functional <- function(f) { f(rnorm(10)) }
```

In [35]:
```
functional(mean)
```

```
[1] -0.09413735
```

Functional example

In [34]:
```
# Applies a supplied function to a random draw
# from the normal distribution with mean 0 and sd 1
functional <- function(f) { f(rnorm(10)) }
```

In [35]:
```
functional(mean)
```

```
[1] -0.09413735
```

In [36]:
```
functional(median)
```

```
[1] -0.1556706
```

Functional example

In [34]:
```
# Applies a supplied function to a random draw
# from the normal distribution with mean 0 and sd 1
functional <- function(f) { f(rnorm(10)) }
```

In [35]:
```
functional(mean)
```

```
 [1] -0.09413735
```

In [36]:
```
functional(median)
```

```
 [1] -0.1556706
```

In [37]:
```
functional(sum)
```

```
 [1] -2.926588
```

Summary of common `apply()` functions

Function	Description	Input Object	Output Object	Simplifie
`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

Function	Description	Input Object	Output Object	Simplifie
`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

`lapply()` examples

In [38]:
```r
l <- list(a = 1:2, b = 3:4, c = 5:6, d = 7:8, e = 9:10)
```

`lapply()` examples

In [38]:
```r
l <- list(a = 1:2, b = 3:4, c = 5:6, d = 7:8, e = 9:10)
```

In [39]:
```r
# 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

In [38]:
```r
l <- list(a = 1:2, b = 3:4, c = 5:6, d = 7:8, e = 9:10)
```

In [39]:
```r
# 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
```

In [40]:
```r
# We can exploit the fact that basic operators are function calls
# Here, each subsetting operator `[` with argument 2 is applied to each
# 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

`apply()` examples

In [41]:
```
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
```

`apply()` examples

In [41]:
```
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
```

In [42]:
```
# Sum up rows (can also be achieved with rowSums() function)
apply(m, 1, sum)
```

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

`apply()` examples

In [41]:
```
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
```

In [42]:
```
# Sum up rows (can also be achieved with rowSums() function)
apply(m, 1, sum)
```

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

In [43]:
```
# Calculate averages across columns (also available in colMeans())
apply(m, 2, mean)
```

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

`apply()` examples

In [41]:
```
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
```

In [42]:
```
# Sum up rows (can also be achieved with rowSums() function)
apply(m, 1, sum)
```

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

In [43]:
```
# Calculate averages across columns (also available in colMeans())
apply(m, 2, mean)
```

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

In [44]:
```
# 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 the 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

`mapply()` examples

In [45]:
```
means <- -2:2
sds <- 1:5
```

`mapply()` examples

In [45]:
```
means <- -2:2
sds <- 1:5
```

In [46]:
```
# 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

In [45]:
```
means <- -2:2
sds <- 1:5
```

In [46]:
```
# 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
```

In [47]:
```
# 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
```

```
          [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -1.400437 -1.9529977  1.0721986 -0.2210901  8.5994742
[2,] -1.400437 -1.9529977  1.0721986 -0.2210901  8.5994742
[3,] -1.400437 -1.9529977  1.0721986 -0.2210901  8.5994742
[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

Package loading example

In [48]:
```
# Package 'Matrix' is part of the standard R library and doesn't have t
library("Matrix")
```

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

Package loading example

In [48]:
```
# Package 'Matrix' is part of the standard R library and doesn't have t
library("Matrix")
```

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

In [49]:
```
# While it is possible to just use function sparseVector() after loadir
# it is good practice to state explicitly which package the object is c
sv <- Matrix::sparseVector(x = c(1, 2, 3), i = c(3, 6, 9), length = 10)
```

Package loading example

In [48]:
```
# Package 'Matrix' is part of the standard R library and doesn't have t
library("Matrix")
```

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

In [49]:
```
# While it is possible to just use function sparseVector() after loadir
# it is good practice to state explicitly which package the object is c
sv <- Matrix::sparseVector(x = c(1, 2, 3), i = c(3, 6, 9), length = 10)
```

In [50]:
```
sv
```

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

Next

- Tutorial: Implementing functions
- Next week: Debugging and Testing in R