Basics of Functions

STAT 133

Gaston Sanchez

github.com/ucb-stat133/stat133-fall-2016

Functions

R comes with many functions and packages that let us perform a wide variety of tasks. Sometimes, however, there's no function to do what we want to achieve. In these cases we need to create our own functions.
To talk about functions, we must first talk about **R expressions**

Expressions

Expressions

R code is composed of a series of **expressions**

- assignment statements
- arithmetic expressions
- function calls
- conditional statements
- etc

Simple Expressions

```r
# assignment statement
a <- 12345

# arithmetic expression
525 + 34 - 280

# function call
median(1:10)
```

Expressions

One way to separate expressions is with new lines:

```r
a <- 12345
525 + 34 - 280
median(1:10)
```

Grouping Expressions

Constructs for grouping together expressions

- semicolons
- curly braces

Separating Expressions

Simple expressions separated with new lines:

```r
a <- 10
b <- 20
d <- 30
```

Grouping Expressions

Grouping expressions with semicolons:

```
a <- 10; b <- 20; d <- 30
```

Although this is a perfectly valid expression, we recommend avoiding semicolons, since they make code harder to review.

Grouping Expressions

Grouping expressions with braces:

```
{
  a <- 10
  b <- 20
  d <- 30
}
```

Grouping Expressions

Multiple expressions in one line within braces:

```r
{a <- 10; b <- 20; d <- 30}
```

Again, this piece of code is just for illustration purposes (don't write code like this!)

Brackets and Braces in R

	Symbol	Use
[ ]	brackets	Objects
( )	parentheses	Functions
{ }	braces	Expressions

Brackets and Braces

```
# brackets for objects
dataset[1:10]
```

Brackets and Braces

```
# brackets for objects
dataset[1:10]
```

```
# parentheses for functions
some_function(dataset)
```

Brackets and Braces

```r
# brackets for objects
dataset[1:10]
```

```r
# parentheses for functions
some_function(dataset)
```

```r
# brackets for expressions
{
  1 + 1
  mean(1:5)
  tbl <- read.csv('datafile.csv')
}
```

Expressions

- A program is a set of instructions
- Programs are made up of expressions
- R expressions can be simple or compound
- **Every expression in R has a value**

Expressions

```r
# Expressions can be simple statements:
5 + 3

## [1] 8
```

```r
# Expressions can also be compound:
{5 + 3; 4 * 2; 1 + 1}

## [1] 2
```

Expressions

The value of an expression is the last evaluated statement:

```
# value of an expression
{5 + 3; 4 * 2; 1 + 1}

## [1] 2
```

The result has the visibility of the last evaluation

Simple Expressions

We use braces { } to group the statements of an expression:

```
# simple expression
{5 + 3}

## [1] 8
```

For simple expressions there is really no need to use braces.

Compound Expressions

- Compound expressions consist of multiple simple expressions
- Compound expressions require braces
- Simple expressions in a compound expression can be separated by semicolons or newlines

Compound Expressions

Simple expressions in a compound expression separated by semicolons:

```r
# compound expression (just for demo purposes)
# don't write code like this
{mean(1:10); '3'; print("hello"); c(1, 3, 4)}

## [1] "hello"
## [1] 1 3 4
```

Compound Expressions

Simple expressions in a compound expression separated by newlines:

```r
# compound expression
{
  mean(1:10)
  '3'
  print("hello")
  c(1, 3, 4)
}

## [1] "hello"
## [1] 1 3 4
```

You will use braces, but not like in this dummy example.

Compound Expressions

It is possible to have assignments within compound expressions:

```
{
  x <- 4
  y <- x^2
  x + y
}

## [1] 20
```

Compound Expressions

The variables inside the braces can be used in later expressions

```
z <- {x <- 4; y <- x^2; x + y}
x

## [1] 4

y

## [1] 16

z

## [1] 20
```

Compound Expressions

```
# simple expressions in newlines
z <- {
  x <- 4
  y <- x^2
  x + y}
x

## [1] 4

y

## [1] 16

z

## [1] 20
```

Using Expressions

Expressions are typically used in

- Functions
- Flow control structures (e.g. `for` loop)

Compound Expressions

Do not confuse a function call (having arguments in multiple lines) with a compound expression

```r
# this is NOT a compound expression
plot(x = runif(10), y = rnorm(10),
     pch = 19, col = "#89F39A", cex = 2,
     main = "some plot",
     xlab = 'x', ylab = 'y')
```

Anatomy of a Function

Anatomy of a function

`function()` allows us to create a function. It has the following structure:

```r
function_name <- function(arg1, arg2, etc)
{
  expression_1
  expression_2
  ...
  expression_n
}
```

Anatomy of a function

- Generally we will give a name to a function
- A function takes one or more inputs (or none), known as as *arguments*
- The expressions forming the operations comprise the body of the function
- Simple expression doesn't require braces
- Compound expressions are surround by braces
- Functions return a single *value*

Function example

A function that squares its argument:

```r
square <- function(x) {
  x^2
}
```

Function example

A function that squares its argument:

```
square <- function(x) {
  x^2
}
```

- the function's name is `"square"`
- it has one argument `x`
- the function's body consists of one simple expression
- it returns the value `x * x`

Function example

It works like any other function in R:

```
square(10)

## [1] 100
```

In this case, `square()` is also vectorized

```
square(1:5)

## [1]  1  4  9 16 25
```

Why is `square()` vectorized?

Function example

Functions with a body consisting of a simple expression can be written with no braces (in one single line!):

```r
square <- function(x) x * x

square(10)

## [1] 100
```

If the body of a function is a compund expression we use braces:

```
sum_sqr <- function(x, y) {
  xy_sum <- x + y
  xy_ssqr <- (xy_sum)^2
  list(sum = xy_sum,
       sumsqr = xy_ssqr)
}

sum_sqr(3, 5)

## $sum
## [1] 8
##
## $sumsqr
## [1] 64
```

Function example

Once defined, functions can be used in other function definitions:

```
sum_of_squares <- function(x) {
  sum(square(x))
}

sum_of_squares(1:5)

## [1] 55
```

Area of a Rectangle

A function which, given the values l (length) and w (width) computes the value $l \times w$

```
area_rect <- function(l, w) {
  l * w
}
```

- The formal arguments of the function are `l` and `w`
- The body of the function consists of the simple expression `l * w`
- The function has been assigned the name `"area_rect"`

Nested Functions

We can also define a function inside another function:

```
getmax <- function(a) {
  maxpos <- function(u) {
    which.max(u)
  }
  list(position = maxpos(a),
       value = max(a))
}

getmax(c(2, -4, 6, 10, pi))

## $position
## [1] 4
##
## $value
## [1] 10
```

Function names

Different ways to name functions

- `squareroot()`
- `SquareRoot()`
- `squareRoot()`
- `square.root()`
- `square_root()`

Function names

Invalid names

- `5quareroot()`: cannot begin with a number
- `_sqrt()`: cannot begin with an underscore
- `square-root()`: cannot use hyphenated names

In addition, avoid using an already existing name, e.g. `sqrt()`

Function Output

Function output

- The body of a function is an expression
- Remember that every expression has a value
- Hence every function has a value

Function output

The value of a function can be established in two ways:

- As the last evaluated simple expression (in the body)
- An explicitly **returned** value via `return()`

The `return()` command

Sometimes the `return()` command is included to explicitly indicate the output of a function:

```
add <- function(x, y) {
  z <- x + y
  return(z)
}

add(2, 3)

## [1] 5
```

The `return()` command

If no `return()` is present, then R returns the last evaluated expression:

```
# output with return()
add <- function(x, y) {
  z <- x + y
  return(z)
}

add(2, 3)

## [1] 5
```

```
# output without return()
add <- function(x, y) {
  x + y
}

add(2, 3)

## [1] 5
```

The return() command

Depending on what's returned or what's the last evaluated expression, just calling a function might not print anything:

```r
# nothing is printed
add <- function(x, y) {
  z <- x + y
}

add(2, 3)
```

```r
# output printed
add <- function(x, y) {
  z <- x + y
  return(z)
}

add(2, 3)

## [1] 5
```

The return() command

Here we call the function and assign it to an object. The last evaluated expression has the same value in both cases:

```
# nothing is printed
add <- function(x, y) {
  z <- x + y
}

a1 <- add(2, 3)
a1

## [1] 5
```

```
# output printed
add <- function(x, y) {
  z <- x + y
  return(z)
}

a2 <- add(2, 3)
a2

## [1] 5
```

The `return()` command

If no `return()` is present, then R returns the last evaluated expression:

```r
add1 <- function(x, y) {
  x + y
}

add2 <- function(x, y) {
  z <- x + y
  z
}
```

```r
add3 <- function(x, y) {
  z <- x + y
}

add4 <- function(x, y) {
  z <- x + y
  return(z)
}
```

The return() command

`return()` can be useful when the output may be obtained in the middle of the function's body

```
f <- function(x, y, add = TRUE) {
  if (add) {
    return(x + y)
  } else {
    return(x - y)
  }
}
```

```
f(2, 3, add = TRUE)

## [1] 5

f(2, 3, add = FALSE)

## [1] -1
```

return() versus print()

Often, R beginners use the function print() to indicate the output of a function:

```
add <- function(x, y) {
  z <- x + y
  print(z)
}

add(2, 3)

## [1] 5
```

`return()` versus `print()`

- The function `print()` is a **generic** method in R.
- This meeans that `print()` has a different behavior depending on its input
- Unless you want to print intermediate results while the function is being executed, there is no need to return the ouput via `print()`

```
add <- function(x, y) {
  z <- x + y
  z  # no need to use print()
}

add(2, 3)

## [1] 5
```

Function Arguments

Function arguments

Functions can have any number of arguments (even zero arguments)

```
# function with 2 arguments
add <- function(x, y) x + y

# function with no arguments
hi <- function() print("Hi there!")

hi()

## [1] "Hi there!"
```

Arguments

Arguments can have default values

```
hey <- function(x = "") {
  cat("Hey", x, "\nHow is it going?")
}

hey()

## Hey
## How is it going?

hey("Gaston")

## Hey Gaston
## How is it going?
```

Arguments with no default values

If you specify an argument with no default value, you must give it a value everytime you call the function, otherwise you'll get an error:

```
sqr <- function(x) {
  x^2
}

sqr()

## Error in sqr():  argument "x" is missing, with no
default
```

Arguments with no default values

Sometimes we don't want to give default values, but we also don't want to cause an error. We can use `missing()` to see if an argument is missing:

```r
abc <- function(a, b, c = 3) {
  if (missing(b)) {
    result <- a * 2 + c
  } else {
    result <- a * b + c
  }
  result
}
```

```r
abc(1)

## [1] 5

abc(1, 4)

## [1] 7
```

Arguments with no default values

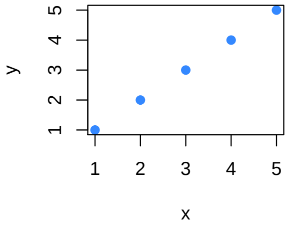
You can also set an argument value to NULL if you don't want to specify a default value:

```r
abcd <- function(a, b = 2, c = 3, d = NULL) {
  if (is.null(d)) {
    result <- a * b + c
  } else {
    result <- a * b + c * d
  }
  result
}
```

More arguments

```r
# arguments with and without default values
myplot <- function(x, y, col = "#3488ff", pch = 19) {
  plot(x, y, col = col, pch = pch)
}

myplot(1:5, 1:5)
```



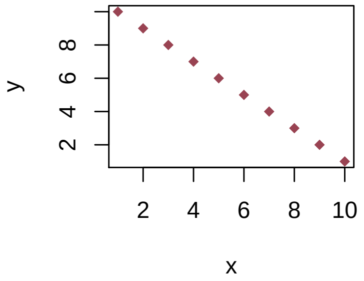
More arguments

```r
# arguments with and without default values
myplot <- function(x, y, col = "#3488ff", pch = 19) {
  plot(x, y, col = col, pch = pch)
}
```

- `x` and `y` have no default values
- `col` and `pch` have default values (but they can be changed)

More arguments

```r
# changing default arguments
myplot(1:10, 10:1, col = "#994352", pch = 18)
```



Writing Functions

Writing Functions

How to write functions?

- Always start simple with test toy-values
- Work first on what will be the body of the function
- Check out each step of the way
- Don't try to do much at once
- Create the function (i.e. encapsulate the body) once everything works
- Don't write long functions: write short / small functions (preferably less than 10 lines of code)

Variance Function Example

The sample variance is given by the following formula:

$$var(x) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

Variance Function Example

```
# start simple
x <- 1:10

# get working code
sum((x - mean(x))^2) / (length(x) - 1)

## [1] 9.166667

# test it: compare it to var()
var(1:10)

## [1] 9.166667
```

Variance Function Example

```
# encapsulate your code
variance <- function(x) {
  sum((x - mean(x))^2) / (length(x) - 1)
}

# check that it works
var(1:10)

## [1] 9.166667
```

Variance Function Example

```
# then consider less simple cases
variance (runif(10))

## [1] 0.162643

var(rep(0, 10))

## [1] 0

var(c(1:9, NA))

## [1] NA
```

Variance Function Example

```r
# adapt it gradually
variance <- function(x, na.rm = FALSE) {
  if (na.rm) {
    x <- x[!is.na(x)]
  }
  sum((x - mean(x))^2) / (length(x) - 1)
}

variance(c(1:9, NA), na.rm = TRUE)

## [1] 7.5
```

Don't worry about `if()` conditionals, we'll talk about them later.

Writing Functions

When writing functions:

- Choose meaningful names of functions
- Preferably a verb
- Choose meaningful names of arguments
- Think about the users (who will use the function)
- Think about extreme cases
- If a function is too long, maybe you need to split it

Names of functions

Avoid this:

```r
f <- function(x, y) {
  x + y
}
```

This is better

```r
add <- function(x, y) {
  x + y
}
```

Names of arguments

Give meaningful names to arguments:

```r
# Avoid this
area_rect <- function(x, y) {
  x * y
}
```

This is better

```r
area_rect <- function(length, width) {
  length * width
}
```

Names of arguments

Even better: give default values (whenever possible)

```r
area_rect <- function(length = 1, width = 1) {
  length * width
}

# default
area_rect()

# specifying argument values
area_rect(length = 10, width = 2)
```

Meaningful Names to Arguments

Avoid this:

```r
# what does this function do?
ci <- function(p, r, n, ti) {
  p * (1 + r/p)^(ti * p)
}
```

Meaningful Names to Arguments

Avoid this:

```
# what does this function do?
ci <- function(p, r, n, ti) {
  p * (1 + r/p)^(ti * p)
}
```

This is better:

```
# OK
compound_interest <- function(principal, rate, periods, time) {
  principal * (1 + rate/periods)^(time * periods)
}
```

Meaningful Names to Arguments

```r
# names of arguments
compound_interest <- function(principal = 1, rate = 0.01,
                              periods = 1, time = 1)
{
  principal * (1 + rate/periods)^(time * periods)
}

compound_interest(principal = 100, rate = 0.05,
                  periods = 5, time = 1)

compound_interest(rate = 0.05, periods = 5,
                  time = 1, principal = 100)

compound_interest(rate = 0.05, time = 1,
                  periods = 5, principal = 100)
```

Describing functions

Also add a short description of what the arguments should be like. In this case, the description is outside the function

```r
# function for adding two numbers
# x: number
# y: number
add <- function(x, y) {
  x + y
}
```

Describing functions

In this case, the description is inside the function

```r
add <- function(x, y) {
  # function for adding two numbers
  # x: number
  # y: number
  x + y
}
```

Describing functions

```r
# description of arguments
compound_interest <- function(principal = 1, rate = 0.01,
                              periods = 1, time = 1)
{
  # principal = Principal Amount
  # rate = Annual Nominal Interest Rate as a decimal
  # time = Time Involved in years
  # periods = number of compounding periods per unit time
  principal * (1 + rate/periods)^(time * periods)
}
```