Functions

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat/stat133
Course web: gastonsanchez.com/teaching/stat133

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.

Anatomy of a Function

Anatomy of a function

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

```
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 don'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 * x
}
```

- 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:

```r
square(10)
```

```
## [1] 100
```

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

```r
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!):

```
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_square <- function(x) sum(square(x))

sum_square(1:5)

## [1] 55
```

Area of a Circle

A function which, given the value r computes the value πr^2

```
area_circle <- function(r) pi * r^2
```

- The formal argument of the function is `r`
- The body of the function consists of the simple expression `pi * r^2`
- The function has been assigned the name `"area_circle"`

```
area_circle(5)

## [1] 78.53982
```

Evaluation of Functions

Function evaluation involves:

- A set of variables associated to the arguments is temporarily created
- The variable definitions are used to evaluate the body function
- Temporary variables are removed at the end
- The computed values are returned

Evaluation Example

Evaluating the function call `area_circle(5)` takes place as follows:

- Temporarily create a variable `r` with value 5
- Use that value 5 to compute `pi * 5^2`
- Remove the temporary variable definition
- Return the value `78.53982`

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:

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

add(2, 3)

## [1] 5
```

```r
# 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:

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

add(2, 3)
```

```
# 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:

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

a1 <- add(2, 3)
a1

## [1] 5
```

```r
# 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:

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

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

```
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
```

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:

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

```
abc(1)

## [1] 5

abc(1, 4)

## [1] 7
```

Arguments with no default values

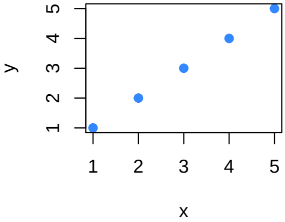
You can also set an argument value to `NULL` if you don't want to specify a default value:

```
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)
```



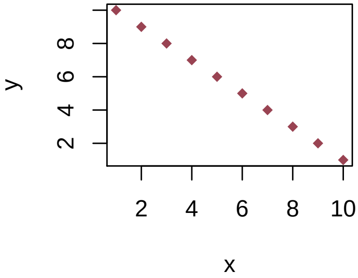
More arguments

```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

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



Argument Matching

- Arguments with default values are known as **named** arguments
- Arguments with no default values are referred to as **positional** arguments
- Arguments can be matched positionally or by name

Argument Matching

```r
values <- seq(-2, 1, length.out = 20)

# equivalent calls
mean(values)
mean(x = values)
mean(x = values, na.rm = FALSE)
mean(na.rm = FALSE, x = values)
mean(na.rm = FALSE, values)
```

Partial Matching

Named arguments can also be partially matched:

```
# equivalent calls
seq(from = 1, to = 2, length.out = 5)
seq(from = 1, to = 2, length = 5)
seq(from = 1, to = 2, len = 5)
```

`length.out` is partially matched with `length` and `len`

Matching Order

Order of argument matching operations:

- Check for exact match for a named argument
- Check for a partial match
- Check for a positional match

Exercise

Write a function that checks if a number is positive (output `TRUE`) or negative (output `FALSE`)

Exercise

Write a function that checks if a number is positive (output `TRUE`) or negative (output `FALSE`)

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

is_positive(2)

## [1] TRUE

is_positive(-1)

## [1] FALSE
```

Exercise

What happens in these cases?

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

is_positive(0)
is_positive(NA)
is_positive(TRUE)
is_positive("positive")
is_positive(1:5)
```

Using arguments for other functions

Theere are various functions that include the argument `na.rm` to indicate if missing values should be removed. One of them is `mean()`:

```
# deafult na.rm = FALSE
mean(c(1, 2, 3, NA, 5))

## [1] NA

# deafult na.rm = TRUE
mean(c(1, 2, 3, NA, 5), na.rm = TRUE)

## [1] 2.75
```

Using arguments for other functions

If we create a function that uses other functions containing `na.rm`, it is wise to include that argument:

```
meansd <- function(x, na.rm = FALSE) {
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}

meansd(c(1, 2, 3, NA, 5), na.rm = TRUE)

##     mean       sd
## 2.750000 1.707825
```

Dots argument

If you check functions like `c()`, `paste()`, `plot()`, you'll notice the use of a special argument: `...`

- it matches zero, one or more actual arguments
- it allows us to pass arguments to other functions inside the function
- `...` allows us to "cascade" arguments to other functions without inluding them in the definition

Dots argument

Using ...

```
fplot <- function(y, ...) {
  x <- 1:length(y)
  plot(x, y, type = 'n', ylim = c(0, 1), ...)
  points(x, y, col = "#93a8f2", pch = 19)
  lines(x, y, col = "#dd93f2", lwd = 3)
}

fplot(runif(10), bty = 'n')
fplot(runif(10), bty = 'n', main = "some title")
fplot(runif(10), bty = 'n', xlab = '')
```

Writing Functions

Writing Functions

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:

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

This is better

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

Describing functions

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

```
# 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
}
```

Binary Operators

Binary Operators

- One type of functions very common in R are **binary operators**, eg:
  - `2 + 5` (sum)
  - `3 / 2` (exponent)
  - `a %in% b` (value matching)
  - `X %*% Y` (matrix multiplication)
- These functions take two inputs—hence the term *binary*
- It is possible to define your own binary operators

Binary Operators

```r
# binary operator
"%p%" <- function(x, y) {
  paste(x, y, sep = " ")
}

'good' %p% 'morning'

## [1] "good morning"
```

How to create a binary operator?

- A binary operator is defined as one or more characters surrounded by percent symbols %
- When defining the function, the entire name must be quoted
- Include two arguments
- As usual, avoid using names of existing operators:
  - `"%%"`, `%*%`, `%/%`, `%o%`, `%in%`

Another example

Here's another example:

```
# binary operator
"%u%" <- function(x, y) {
  union(x, y)
}

1:5 %u% c(1, 3, 5, 7, 9)

## [1] 1 2 3 4 5 7 9
```

Lazy Evaluation

Lazy Evaluation

Arguments to functions are evaluated lazily, that is, they are evaluated only as needed:

```
g <- function(a, b) {
  a * a * a
}

g(2)

## [1] 8
```

`g()` never uses the argument `b`, so calling `g(2)` does not produce an error

Lazy Evaluation

Another example

```
g <- function(a, b) {
  print(a)
  print(b)
}

g(2)

## [1] 2

## Error in print(b):  argument "b" is missing, with no
default
```

Notice that 2 got printed before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`

Messages

There are two main functions for generating warnings and errors:

- `stop()`
- `warning()`

There's also the `stopifnot()` function

Stop Execution

Use `stop()` to stop the execution of a function (this will raise an error)

```
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    stop("x is not numeric")
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

Stop Execution

Use `stop()` to stop the execution of a function (this will raise an error)

```
# ok
meansd(c(4, 5, 3, 1, 2))

##     mean       sd
## 3.000000 1.581139

# this causes an error
meansd(c('a', 'b', 'c'))

## Error in meansd(c("a", "b", "c")):  x is not numeric
```

Warning Messages

Use `warning()` to show a warning message

```
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    warning("non-numeric input coerced to numeric")
    x <- as.numeric(x)
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

A warning is useful when we don't want to stop the execution, but we still want to show potential problems

Warning Messages

Use `warning()` to show a warning message

```
# ok
meansd(c(4, 5, 3, 1, 2))

##     mean       sd
## 3.000000 1.581139

# this causes a warning
meansd(c(TRUE, FALSE, TRUE, FALSE))

## Warning in meansd(c(TRUE, FALSE, TRUE, FALSE)):
non-numeric input coerced to numeric

##      mean        sd
## 0.5000000 0.5773503
```

Stop Execution

`stopifnot()` ensures the truth of expressions:

```
meansd <- function(x, na.rm = FALSE) {
  stopifnot(is.numeric(x))
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}

meansd('hello')

## Error:  is.numeric(x) is not TRUE
```

Environments and Functions

Consider this example

```r
w <- 10

f <- function(y) {
  d <- 5
  h <- function() {
    d * (w + y)
  }
  return(h())
}

f(2)

## [1] 60
```

How / Why does `f()` work?

Consider this other example

```r
w <- 10

f <- function(y) {
  d <- 5
  return(h())
}

f(2)

## Error in f(2):  could not find function "h"
```

Why `f()` does not work?

Environments

- All the variables that we create need to be stored somewhere
- The place where they are stored is called an **environment**
- R works with enviroments, all of which are in (virtual) memory
- Usually, we don't need to explicitly deal with environments
- Environments are nested

Global Environment

- The user workspace is the **global environment**
- The global environment is the **top level** environment
- It is formally referred to as `R_GlobalEnv`
- Variables defined in the global environment can be seen from anywhere
- The contents of the global environment are listed with `ls()`

```
# top level environment
environment()

## <environment: R_GlobalEnv>
```

Searching objects

- When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value
- To retrieve the value of an object the order is:
- Search the current environment
- Search the global environment for a symbol name matching the one requested
- Search the namespaces of each of the packages on the search list: `search()`

Environments and Functions

- A function consists not only of its arguments and body but also of its *environment*
- An environment is made up of the collection of objects present at the time the function comes into existence
- When a function is created by evaluating the corresponding expression, the current environment is recorded as a property of the function

Let's go back to our first example

```
w <- 10

f <- function(y) {
  d <- 5
  h <- function() {
    d * (w + y)
  }
  return(h())
}

f(2)

## [1] 60
```

How does `f()` work?

Let's see the environments

```
w <- 10   # variable (in global environment)

# a function (in global environment)
f <- function(y) {
  d <- 5      # local variable
  h <- function() {     # subfunction
    d * (w + y)      # w is a free variable
  }
  return(h())
}

environment(f)

## <environment: R_GlobalEnv>
```

Function Environment

- `w` is a global variable (in global environment)
- `f()` is a function in the global environment
- `d` is a local variable—local to `f()`
- `h()` is a subfunction—local to `f()`
- `w` is not an argument but a free variable

Let's see the environments

```
f <- function(y) {
  d <- 5
  h <- function() {
    d * (w + y)
  }
  print(environment(h))   # h()'s environment
  return(h())
}

environment(f)

## <environment: R_GlobalEnv>

f(2)

## <environment: 0x7f9a0e1af180>
## [1] 60
```

Variable's Scope

- A variable's **scope** is the set of places from which you can see the variable
- R will try to find a variable in the current environment
- If it doesn't find them it will look in the parent environment
- And then that environment's parent
- And so on until it reaches the global environment

Variable's Scope

- A variable's **scope** is the set of places from which you can see the variable
- R will try to find a variable in the current environment
- If it doesn't find them it will look in the parent environment
- And then that environment's parent
- And so on until it reaches the global environment

Variable Scope

- When we define a variable inside a function, the rest of the statements in that function will have access to that variable

Variable Scope

```
f <- function(x) {
  y <- 1
  g <- function(x) {
    (x + y) / 2
  }
  g(x)
}

f(5)

## [1] 3
```

`g()` is a subfunction that have access to `y` in `f`'s environment.

Variable Scope

```r
f <- function(x) {
  y <- 1
  g(x)
}

g <- function(x) {
  (x + y) / 2
}

f(5)

## Error in g(x):  object 'y' not found
```

`g()` is a function that doesn't have access to `y`; `g()` can only see things in the global environment

Your Turn

Exercise

R has a function `summary()` that when applied on a numeric vector provides something like this:

```
summary(1:10)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.25    5.50    5.50    7.75   10.00
```

Create a `describe()` function that takes a numeric vector and returns: minimum, maximum, mean, and standard deviation

Exercise

First attempt

```
describe <- function(x) {
  x_min <- min(x)
  x_max <- max(x)
  x_mean <- mean(x)
  x_sd <- sd(x)
  return(c(x_min, x_max, x_mean, x_sd))
}

describe(1:10)

## [1]  1.00000 10.00000  5.50000  3.02765
```

Exercise

Second attempt (adding names)

```
describe <- function(x) {
  x_min <- min(x)
  x_max <- max(x)
  x_mean <- mean(x)
  x_sd <- sd(x)
  values <- c(x_min, x_max, x_mean, x_sd)
  names(values) <- c("min", "max", "mean", "sd")
  return(values)
}

describe(1:10)

##      min      max     mean       sd
##  1.00000 10.00000  5.50000  3.02765
```

Exercise

Third attempt (using a list as output)

```
describe <- function(x) {
  list(
    min = min(x),
    max = max(x),
    mean = mean(x),
    sd = sd(x)
  )
}
```

```
describe(1:10)

## $min
## [1] 1
##
## $max
## [1] 10
##
## $mean
## [1] 5.5
##
## $sd
## [1] 3.02765
```

Exercise

Probability Density of the Normal Distribution:

$$f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Write a function that takes a value x (with parameters μ and σ) which computes the probability density distribution of the normal distribution

Exercise

Normal Distribution:

```
normal_dist <- function(x, mu = 0, sigma = 1) {
  constant <- 1 / (sigma * sqrt(2*pi))
  constant * exp(-((x - mu)^2) / (2 * sigma^2))
}

normal_dist(2)

## [1] 0.05399097
```

Argument Matching

```r
normal_dist <- function(x, mu = 0, sigma = 1) {
  constant <- 1 / (sigma * sqrt(2*pi))
  constant * exp(-((x - mu)^2) / (2 * sigma^2))
}

normal_dist(2)
normal_dist(2, sigma = 3, mu = 1)
normal_dist(mu = 1, sigma = 3, 2)
normal_dist(mu = 1, 2, sigma = 3)
```

Argument Matching

R is “smart” enough in doing pattern matching with arguments’ names (not recommended though)

```
normal_dist(2)

## [1] 0.05399097

normal_dist(2, m = 0, s = 1)

## [1] 0.05399097

normal_dist(2, sig = 1, m = 0)

## [1] 0.05399097
```