# Functions - part 1

## STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat/stat133
Course web: gastonsanchez.com/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 doesn't require braces
- Compound expressions are surround by braces
- Functions return a single *value*

# Function example

A function that squares its argument:

```
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!):

```
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"`

```
area_rect(5, 3)

## [1] 15
```

# 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_rect(5, 3)` takes place as follows:

- Temporarily create a variable `l` with value 5, and `w` with value 3
- Use those values to compute `l * w`
- Remove the temporary variable definition
- Return the value `15`

# Test yourself

```
ft2 <- function(from, to) from:to^2
```

What does `ft2(1, 3)` return?

A) `1 2 3 4 5 6 7 8 9`

B) `1 4 9`

C) `1 4 9 16 25 36 49 64 81`

D) `1 2 3`

E) `1 2 3 1 2 3`

# 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:

```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:

```
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

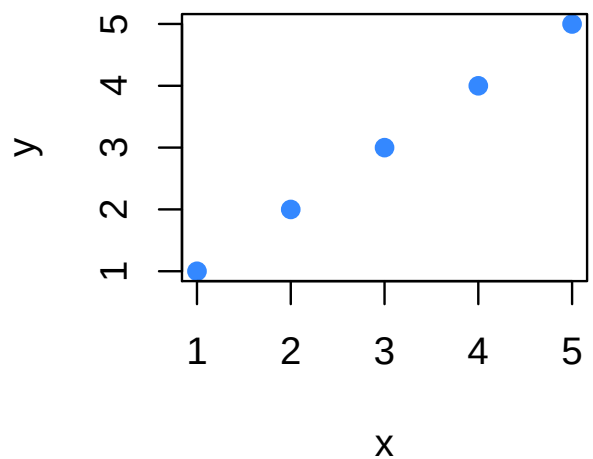
You can also set an argument value to `NULL` if you don't want to specify a default value:

```
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)
```



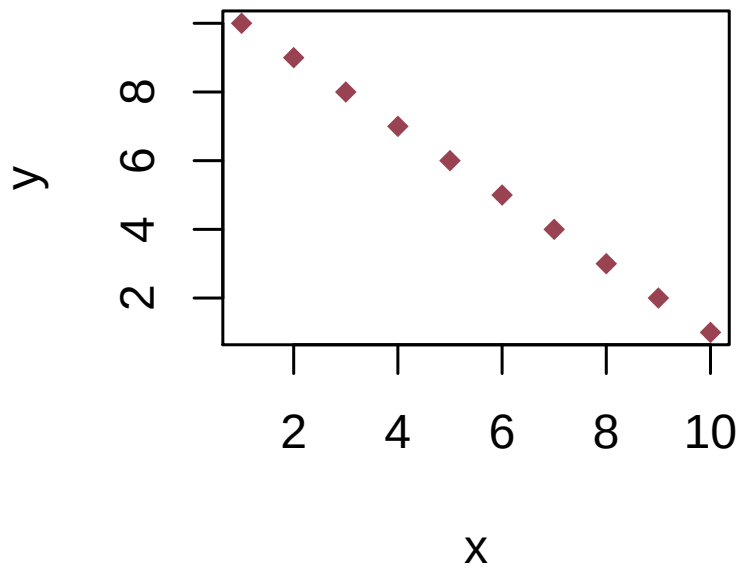
# More arguments

```
# 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)
```



# Positional and Named Arguments

```
output <- some_function(pos1, pos2, name1 = 1, name2 = "yes", nam
```

- `pos1` positional argument
- `pos2` positional argument
- `name1` named argument
- `name2` named argument
- `name3` named argument

# 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

# Example

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

# Example

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
```

# Example

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

```
# a simpler way
is_positive <- function(x) {
  x > 0
}
```

# Example

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

```
# no need to do this
is_positive <- function(x) {
  if (x > 0) print('TRUE') else print('FALSE')
}
```

Remember that every function has a value: the last statement that is evaluated (or an output from `return()`)

# Example

What happens in these cases?

```
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 = '')
```