# Functions - part 2

## 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.

# 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:

```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:

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

This is better

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

# Meaningful Names to Arguments

Avoid this:

```
# 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

```
# 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

```
# 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)
}
```

# 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)
- Binary operators are actually functions
- These functions take two inputs—hence the term *binary*
- It is possible to define your own binary operators

# Binary Operators

Example:

```
# addition operator
2 + 3

# equivalent to
'+'(2, 3)
```

# 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:

```r
# 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

```
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: 0x7fb63242d750>
## [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

```r
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

```
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

# One more thing ...

Let's look at another exmaple

```
mean(1:5)

## [1] 3

mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fb630b8cb48>
## <environment: namespace:base>
```

# One more thing ...

You can do things like this

```
# confusing but it works
mean <- 1:5
mean(mean)

## [1] 3
```

# Some issues

You can also do things like this

```
# not a good idea but you can do it
mean <- function(x) 2*x + 5

mean(1:5)

## [1]  7  9 11 13 15
```

It seems we've lost the original `mean()` function

# The `::` Operator

`::` operator to the rescue

```
# my mean
mean(1:5)

## [1]  7  9 11 13 15

# base mean
base::mean(1:5)

## [1] 3
```

Here we use the name espace `base` of the R package `"base"` to access the original `mean()`

# 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  7.00000  9.00000 11.00000 13.00000 15.
##  [9] 19.00000 21.00000 23.00000 25.00000  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     <NA>     <NA>     <NA>
##  1.00000 10.00000  7.00000  9.00000 11.00000 13.00000 15.00000
##     <NA>     <NA>     <NA>     <NA>     <NA>
## 19.00000 21.00000 23.00000 25.00000  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]  7  9 11 13 15 17 19 2
##
## $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 $\mu$ and $\sigma$) 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

```
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
```