Functions and apply et al.

Adam Kuczynski

What are functions?

A **function** is a chunk of R code that (typically) takes a set of inputs (i.e., **arguments**) and performs some sort of operation on them.

There are three components of a function:

- `body()`: the code inside the function
- `formals()`: the list of arguments which controls how you can call the function
- `environment()` the "map" of the location of the function's variables

```
formals(sample)
```

```
## $x
##
##
## $size
##
##
## $replace
## [1] FALSE
##
## $prob
## NULL
```

```
body(sample)
```

```
## {
##     if (length(x) == 1L && is.numeric(x)
##         1) {
##         if (missing(size))
##             size <- x
##         sample.int(x, size, replace, prob
##     }
##     else {
##         if (missing(size))
##             size <- length(x)
##         x[sample.int(length(x), size, rep
##     }
## }
```

```
environment(sample)
```

```
## <environment: namespace:base>
```

The `sample()` function randomly samples *size* draws from a given vector

Function Output

Often a function will **return** an object. For example:

- `mean()` returns a numeric vector of length 1
- `sample()` returns a numeric vector of length *size*
- `typeof()` returns a character vector of length 1
- `cor()` returns a matrix of size *rows* $\times$ *columns*
- `lm()` returns lists nested in a list of length 12

When a function returns an object, you'll often want to assign that object to save the output. For example:

```
shuffled_letters <- sample(letters, length(letters))
shuffled_letters
```

```
##  [1] "h" "u" "o" "z" "d" "g" "q" "a" "s" "y" "f" "n" "v" "p" "r"
## [20] "j" "w" "b" "i" "k" "x" "c"
```

Functions don't *always* return an object. Sometimes they print text to the console or perform some other operation on your computer

```
# Has no return value, just prints text!
message("This is a message!")
```

```
## This is a message!
```

```
# This just runs another R script!
source("My Script.R")
```

The data that you give (i.e., pass) a function are called **arguments**. Functions can have an unlimited amount of arguments. You can view the arguments of a function in the help pages: `?function_name`

When you call a function, you can specify the argument names or you can choose not to. If you do not specify argument names, you *must* pass them in the order the function is written in. Otherwise you have to specify the argument name.

```
# 300 random draws from normal distribution mean = 5, sd = 10
rnorm(300, 5, 10)

# Same
rnorm(sd = 10, n = 300, mean = 5)
```

Packages

What if there isn't a function in base R for something you want to do?

One of R's greatest features is the ability to extend the functionality of base R with **packages** (bundles of code that others have written to perform a set of tasks)

Anybody can write a package, which means that there are a lot of packages available - 17,911 at the time of this lecture - for you to use (for free!). If you need to do something base R cannot, chances are there's a package for it.

There's even an R package to generate ASCII art!

```
cowsay::say("Isn't R cool?!",
            by = "cow")
```

See `?cowsay::say` for other animals and options!

```
##
##  -----
## Isn't R cool?!
##  ------
##     \   ^__^
##      \  (oo)\ ________
##         (__)\         )\ /\
##              ||------w|
##              ||      ||
```

Installing Packages: A Warning

R packages are typically high quality and trustworthy. However, even the best packages contain bugs!

Also, because *anybody* can write an R package, you might find yourself using a package that is not well built or, even worse, contains malicious code

Bottom line: It is important to vet the packages you use prior to installing them!

Look for:

- Names/organizations of well-known developers
- Extensive documentation (including source code)
- Package citations[1]

[1] It is good practice to cite package names and versions in your manuscripts. See `?citation` and `?packageVersion` for help

User-defined Functions

What if you want to do something that can't be done with a pre-written function from base R *or* a package? Don't worry! One of the primary strengths of using R for your analyses is the ability to write code that does basically whatever you want.

Functions are just another type of object in R. To create a **function** object, use the `function()` function.

From `help("function")`:

```
function( arglist ) expr
return(value)
\( arglist ) expr # we'll talk about this later
```

- `arglist` is a comma separated list of the arguments that your function will take
  - for example: `sample()` takes 4 arguements: `x`, `size`, `replace`, and `prob`
  - there's no limit to how many arguments you can take
  - arguments can have default values such as `replace=` in `sample()`
- `expr` is the R code that the function will execute each time it is called
- `return` is a special function that denotes the **return value** (i.e., the output of the function)

Example: `firstlast()`

Let's write a function that takes a vector as input and outputs a named vector of the first and last elements:

```
firstlast <- function(x) {
    first <- x[1]
    last  <- x[length(x)]
    return(c("first" = first, "last" = last))
}
```

Test it out:

```
firstlast(c(4, 3, 1, 8))
```

```
## first  last
##     4     8
```

```
set.seed(1)
firstlast(sample(1:99999999))
```

```
##    first     last
## 66608964 39153287
```

Testing `firstlast()`

When writing your own function, it is important that you think about how the function will be used. This is particularly important if you are writing a function that others will use, and you can't anticipate exactly what the input will be.

For example: What happens if I pass `firstlast()` the following objects?

```
firstlast(10)
```

```
## first  last
##    10    10
```

```
firstlast(numeric())
```

```
## first
##    NA
```

```
firstlast(mtcars)
```

```
## $first.mpg
##  [1] 21.0 21.0 22.8 21.4 18.7
## [16] 10.4 14.7 32.4 30.4 33.9
## [31] 15.0 21.4
##
## $last.carb
##  [1] 4 4 1 1 2 1 4 2 2 4 4 3
```

Checking Inputs

Using a combination of `if/else` statements and `stop()` (which stops the function from running and prints the text inside as an error message), you can make sure valid information is passed to your function

```
firstlast <- function(x){

  # Check that x is valid
  if(!is.vector(x) || length(x) < 2){
    stop("`x` needs to be vector of length 2 or larger")
  }

  first <- x[1]
  last  <- x[length(x)]
  return(c("first" = first, "last" = last))
}

firstlast(10)
```

```
## Error in firstlast(10): `x` needs to be vector of length 2 or larger
```

```
firstlast(mtcars)
```

```
## Error in firstlast(mtcars): `x` needs to be vector of length 2 or larger
```

Example: Reporting Quantiles

```
reportQuantiles <- function(x, na.rm = FALSE){
  quants <- quantile(x = x,
                     na.rm = na.rm,
                     probs = c(0.01, 0.05, 0.10, 0.25, 0.50,
                               0.75, 0.90, 0.95, 0.99))

  names(quants) <- c("Bottom 1%", "Bottom 5%", "Bottom 10%", "Bottom 25%",
                     "Median", "Top 25%", "Top 10%", "Top 5%", "Top 1%")

  return(quants)
}

reportQuantiles(rnorm(100000))
```

```
##    Bottom 1%    Bottom 5%   Bottom 10%   Bottom 25%       Median       Top 25%
## -2.317959618 -1.642248613 -1.283682278 -0.674976854 -0.006112585  0.668551795
##       Top 10%       Top 5%       Top 1%
##  1.276930110  1.639988060  2.339787800
```

Notice the `na.rm = FALSE` argument. This is a default, which means that if the user does not specify what they want `na.rm` to be, it will automatically be `FALSE

You can call `reportQuantiles()` as many times as you need without writing the internal code over and over

Functions Without Arguments

Sometimes you may want to write a function that takes no arguments, but still does something useful

For example, let's write a function that simulates a flip of a nickel. There's a 49.99% chance the nickel lands on heads, a 49.99% chance it lands on tails, and a 1 in 6000 (0.02%) chance it lands on it's edge

```
flipNickel <- function(){
  sideup <- sample(x = c("heads", "tails", "edge"),
                   size = 1,
                   prob = c(.5-1/6000, .5-1/6000, 1/6000))

  return(sideup)
}

flipNickel()
```

```
## [1] "tails"
```

Note. You can effectively achieve a 50/50 coinflip using the `rbinom()` function, which is a vectorized function that randomly draws values from the binomial distribution

Lexical Scoping

When you call an object in R (referring to a variable, calling a function), R has to know where to look for it. R first looks in the **environment** that the object was created in. If it doesn't find it there, it goes up one level to the **parent environment**, etc.

Consider the following examples. What do you think the output will be?

Example 1

```
varA <- "A"
varB <- "B"


myFunc <- function(){
  varA <- "X"
  varB <- "Y"
  return(paste(varA, varB))
}

myFunc()
paste(varA, varB))
```

Example 2

```
varA <- "A"
varB <- "B"
varC <- "C"

myFunc <- function(){
  varA <- "X"
  varB <- "Y"
  return(paste(varA, varB, varC))
}

myFunc()
paste(varA, varB, varC))
```

Example 1

```
varA <- "A"
varB <- "B"


myFunc <- function(){
  varA <- "X"
  varB <- "Y"
  return(paste(varA, varB))
}

myFunc()
```

```
## [1] "X Y"
```

```
paste(c(varA, varB))
```

```
## [1] "A" "B"
```

Example 2

```
varA <- "A"
varB <- "B"
varC <- "C"

myFunc <- function(){
  varA <- "X"
  varB <- "Y"
  return(paste(varA, varB, varC))
}

myFunc()
```

```
## [1] "X Y C"
```

```
paste(c(varA, varB, varC))
```

```
## [1] "A" "B" "C"
```

When an object is called inside a function, R looks within that function for the object (Example 1). If R cannot find that object in the function, it searches its parent environment (the global environment; Example 2).

R Only Searches Upwards

R will search as many parent environments as it needs to find a object, but it will *never* search downwards (i.e., into child environments)

```
myFunc <- function(){
  myVector <- 1:5
  myOtherVector <- letters[1:5]
  return(paste0(myVector, myOtherVector))
}

myFunc()
```

```
## [1] "1a" "2b" "3c" "4d" "5e"
```

```
exists("myVector")
```

```
## [1] FALSE
```

```
exists("myOtherVector")
```

```
## [1] FALSE
```

<<-

The `<<-` assignment operator works similar to the `<-` operator except it forces R to make the assignment in the global environment. Unless you have a good reason not to do so, you should stick with the `<-` operator for assignment (it keeps your code clean and bug free).

```
myFunc <- function(){
  myVector <<- 1:5
  myOtherVector <<- letters[1:5]
  return(paste0(myVector, myOtherVector))
}

myFunc()
```

```
## [1] "1a" "2b" "3c" "4d" "5e"
```

```
exists("myVector")
```

```
## [1] TRUE
```

```
exists("myOtherVector")
```

```
## [1] TRUE
```

dot-dot-dot (`...`)

Many functions in R take an arbitrary number of targets but still work. These functions rely on the special 'dot-dot-dot' argument: `...`

Let's create a function that takes an arbitrary number of strings and pastes them together as one comma-separated string:

```
csString <- function(...){
  args <- c(...)
  string <- paste0(args, collapse = ", ")
  return(string)
}
```

```
csString("a", "b", "c")
```

```
## [1] "a, b, c"
```

```
csString("Apples", "Bananas", "Carrots", "Dates")
```

```
## [1] "Apples, Bananas, Carrots, Dates"
```

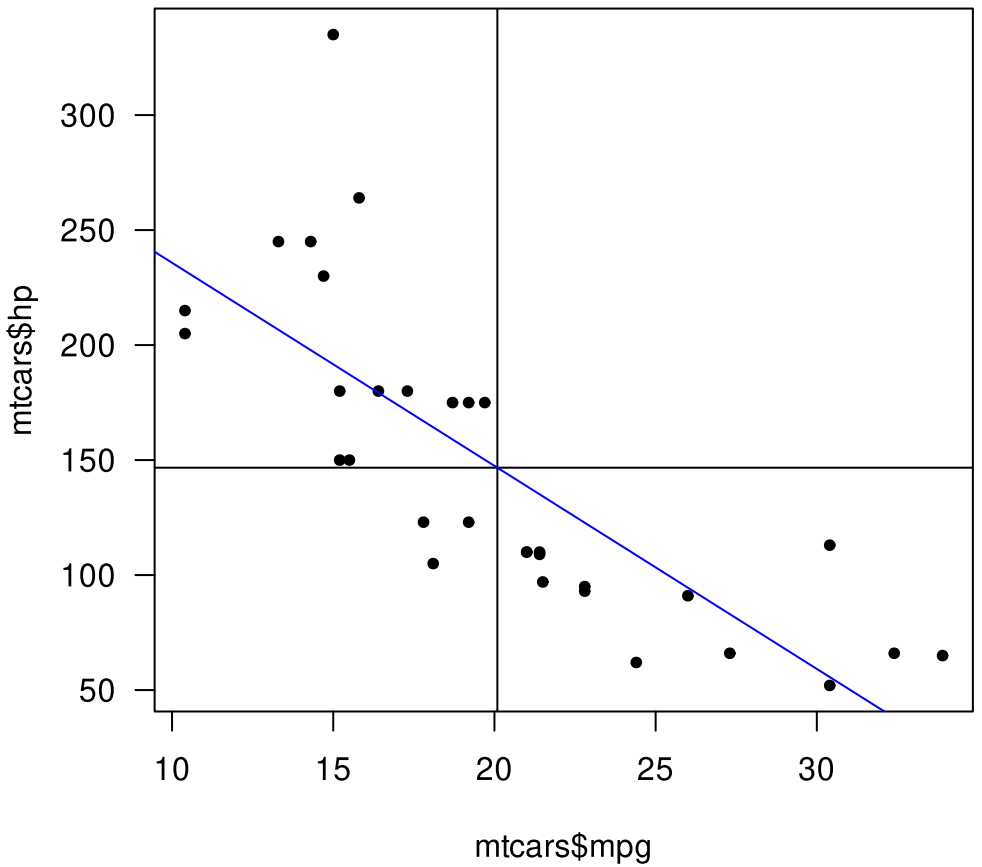
. . .

The `...` argument is particularly useful when you are writing your own function that calls another function, and you want to allow the user to specify as many (or as few) arguments for that function as they would like.

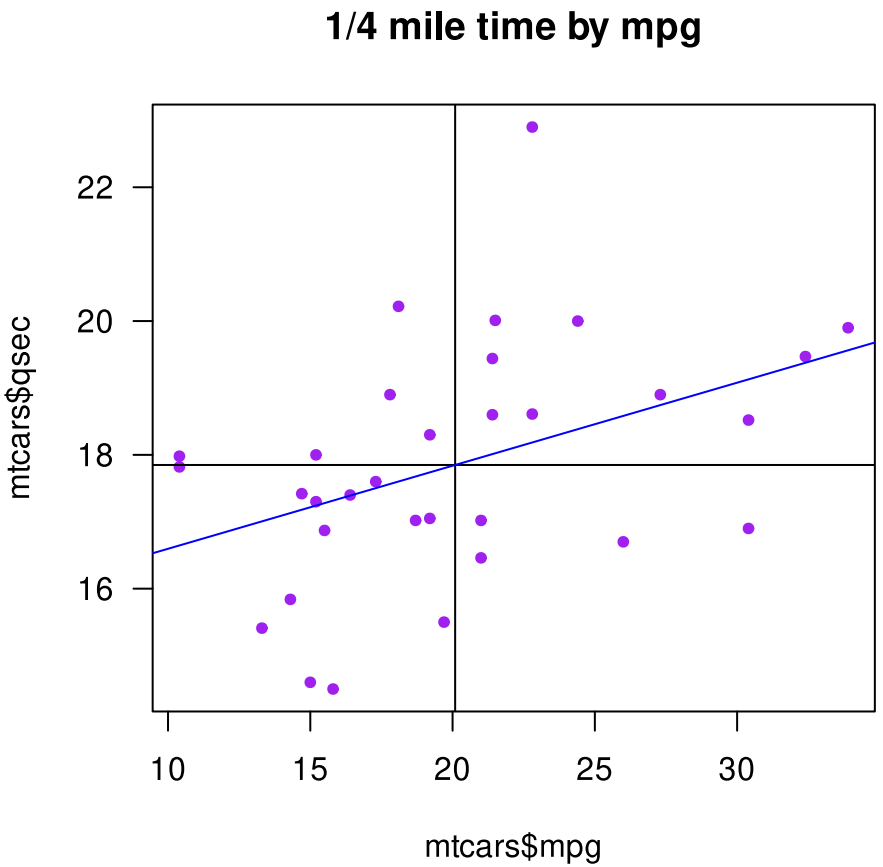
For example, it's often useful to write your own functions to create the same plots from different data. Because `plot()` takes so many arguments, you wouldn't (and couldn't because `plot()` also has the `...` argument!) want to write *all* the argument names and their defaults in your own function.

```
myPlot <- function(...){
  arglist <- list(...)
  plot(axes = FALSE, pch = 20, ...)
  box()
  axis(side = 1, lwd = 0, lwd.ticks = 1)
  axis(side = 2, las = 1, lwd = 0, lwd.ticks = 1)
  abline(v = mean(arglist$x),
         h = mean(arglist$y))
  abline(lm(arglist$y ~ arglist$x), col = "blue")
}
```

```
myPlot(x = mtcars$mpg, y = mtcars$hp)
```



```
myPlot(x = mtcars$mpg, y = mtcars$qsec, col = "purple",
       main = "1/4 mile time by mpg")
```



`function(function(function()))`

Functions can be supplied as arguments to other functions and R will evaluate the functions inside out

```
dim(cbind(mtcars, mtcars))
```

```
## [1] 32 22
```

same as 👉

```
dat <- cbind(mtcars, mtcars)
dim(dat)
```

```
## [1] 32 22
```

There's no limit to how nested your functions can get and, as we will see next week, it is quite common to have several layers of nested functions.

For example, technically this is a deeply nested function:

```
1 + 2 * 3 / 4^5
```

```
## [1] 1.005859
```

```
`+`(1, `/`(`*`(2, 3), `^`(4, 5)))
```

```
## [1] 1.005859
```

Another Nested Function Example

Suppose you want to summarize the `mtcars` dataframe by the numbers of cylinders in each car (4, 6, or 8), you only want to do this for the cars with more than 100 horsepower, and you want to add a column that converts miles per gallon (mpg) to kilometers per liter (kpl):

```
transform(aggregate(formula = . ~ cyl,
                    data = subset(mtcars, hp > 100),
                    FUN = function(x) round(mean(x), 2)),
          "kpl" = mpg*0.425144)
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.
```

Although this code works, it takes a lot of energy to read it and understand what's going on

magrittr

The `magrittr` package in R introduced the **pipe** operator (`%>%`) to make nested function calls more intuitive to write and easier to read

```
lhs %>% rhs
```

`lhs` (**l**eft **h**and **s**ide) is the function that you are *piping* to the `rhs` (**r**ight **h**and **s**ide)

For example, let's compute a correlation matrix for columns 1 through 5 of the *swiss* dataframe for those rows where `Infant.Mortality` is greater than its mean:

```
swiss %>%
  subset(Infant.Mortality > mean(Infant.Mortality),
         select = 1:5) %>%
  cor()
```

```
##             Fertility Agriculture Examination  Education   Catholic
## Fertility    1.0000000   0.2913229  -0.6813507 -0.4716997  0.7231541
## Agriculture  0.2913229   1.0000000  -0.4729718 -0.6269116  0.2675015
## Examination -0.6813507  -0.4729718   1.0000000  0.7462636 -0.5308847
## Education   -0.4716997  -0.6269116   0.7462636  1.0000000 -0.2594915
## Catholic     0.7231541   0.2675015  -0.5308847 -0.2594915  1.0000000
```

Revisiting Our Nested Function

You'll recall:

```
transform(aggregate(formula = . ~ cyl,
                    data = subset(mtcars, hp > 100),
                    FUN = function(x) round(mean(x), 2)),
          "kpl" = mpg*0.425144)
```

```
mtcars %>%
  subset(hp > 100) %>%
  aggregate(. ~ cyl, data = ., FUN = . %>% mean %>% round(2)) %>%
  transform(kpl = mpg*0.425144)
```

This is much easier to read and write!

Other magrittr Pipes

The `magrittr` package has several other pipes that can be useful (but that are less commonly used):

- `%T>%` 'tee' pipe: works like `%>%` but returns the lhs (good for rhs functions with no return value - like plots)

```
1:5 %>% sum()
```

```
## [1] 15
```

```
1:5 %T>% sum()
```

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

- `%$%` 'exposition' pipe: exposes the names of the lhs object to the rhs object

```
mtcars %$% cor(disp, qsec)
```

```
## [1] -0.4336979
```

- `%<>%` 'assignment' pipe: the result of the pipe is assigned to the lhs object (equivalent to `lhs <- lhs %>% rhs`). It is generally discouraged to use this pipe.

```
swiss$Catholic %<>% sqrt()
```

Base R Pipe: `|>`

Recently (May 18, 2021) R version 4.1.0 was released with a base R pipe operator: `|>`

`|>` works in much the same way as `magrittr::%>%`, except that `.` cannot be used to reference the `lhs` object:

```
10 %>% sample(1:5, ., TRUE)
```

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

```
10 |> sample(1:5, ., TRUE)
```

```
## Error in sample.int(x, size, replace, prob): object '.' not found
```

Instead, you have to create an **anonymous function**

```
10 |> {function(x) sample(1:5, x, TRUE)}()
```

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

The Anonymous Function

An **anonymous function** (also known as a **lambda** function) differs from all other functions in that it does not have a name . Anonymous functions are usually arguments to a higher-order (parent) function.

Up until May 18, 2021, anonymous functions were created the same way all other functions are created: the `function()` function. However, many people thought these was too verbose and, following the lead of other programming languages, made a shorthand for `function()`: `\()`

Technically `\()` can be used to write all your functions. For example:

```
# These are equivalent
addOne <- function(x) return(x + 1)
addOne <- \(x) return(x + 1)
```

In practice, however, `\()` will only be used to write anonymous functions, and you should follow that custom (for code readability)

Note. If you are worried about your code being backwards compatible (e.g., because you are collaborating with a large group), avoid using `|>` and `\()` for a little while

Debugging Functions

```
errorFunction <- function(){
  a <- "a"
  b <- "b"
  stop("The error occurs here.")
  c <- "c"
}

errorFunction()
```

```
## Error in errorFunction(): The error occurs here.
```

Often it won't be clear what is causing a certain bug (unlike the above example). Sometimes it's a typo in your code, sometimes its an invalid argument.

To debug a function use the `debug()` function.

```
debug(errorFunction)
```

When you are finished debugging, use `undebug()` so you won't go into debug mode every time the function is called

```
undebug(errorFunction)
```

Recursive functions

A **recursive function** is a function that calls itself. Recursive functions are useful in situations where problems can be broken down into smaller, repetitive problems, or when you need to iterate over arbitrarily nested objects.

```
myFactorial <- function(number){
  if(number == 0){
    return(1)
  } else{
    return(number * myFactorial(number-1))
  }
}

myFactorial(5)
```

```
## [1] 120
```

This evaluated as:

1. `1`
2. `2 * myFactorial(1)`
3. `3 * myFactorial(2)`
4. `4 * myFactorial(3)`
5. `5 * myFactorial(4)`

Making an Operator

Now that we know how to make our own functions we can make our own operators. These operators are known as **infix operators** because they are placed *between* arguments. `+`, `-`, `*`, `/`, `%*%`, `%in%`, etc. are all infix operators.

An example:

Many programming languages have shorthand operators for incrementing and decrementing variables:

- `+=` (add the rhs to the lhs: `lhs <- lhs + rhs`)
- `-=` (subtracts the rhs from the lhs: `lhs <- lhs - rhs`)
- `++` (adds one to a variable: `lhs <- lhs + 1`)
- `--` (subtracts one from a variable: `lhs <- lhs - 1`)

These are very useful in loops:

```
count <- 0
while(count < 10){
  count++ # instead of count <- count + 1
  print(count)
}
```

Unfortunately R doesn't come with these operators. But we can make our own very easily!

```
`%+=%` <- function(lhs, rhs){
  # Evaluates the expression in the parent frame
  # `substitute()` needed so the expression does
  # not run in the eval.parent() call
  eval.parent(substitute(lhs <- lhs + rhs))
}

`%-=%` <- function(lhs, rhs){
  eval.parent(substitute(lhs <- lhs - rhs))
}
```

```
value <- 0
value %+=% 5
print(value)
```

```
## [1] 5
```

```
value %+=% 5
print(value)
```

```
## [1] 10
```

```
value <- 20
value %-=% 5
print(value)
```

```
## [1] 15
```

```
value %-=% 5
print(value)
```

```
## [1] 10
```

What about not `%in%`?

Recall that the `%in%` operator returns a vector of the positions of the lhs vector that are in the rhs vector:

```
1:5 %in% 1:3
```

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

We can inverse this to get the opposite, but it is a bit hard to read:

```
!1:5 %in% 1:3
```

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

We can *invert* or **negate**[1] `%in%` to get a "not in" operator:

```
`%!in%` <- Negate(`%in%`)
1:5 %!in% 1:3
```

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

[1] `Negate()` produces logical negations of *functions*, inverting their output. For example: `is.not.numeric <- Negate(is.numeric)`

Classes and Methods

Classes

Objects in R are **instances** of one or more **classes**. A class defines the behavior of an object.

To get the class of an object, use the `class()` function:

```
class(1:10)
```

```
## [1] "integer"
```

```
class(letters)
```

```
## [1] "character"
```

```
class(mean)
```

```
## [1] "function"
```

```
class(mtcars)
```

```
## [1] "data.frame"
```

Methods

A **method** is a function associated with a specific class. There are many **generic functions** in R which change their behavior depending on the class of the object which it is passed.

Methods are denoted by `.classname` after the generic function name. For example, let's take a look at the `summary` generic function, which has 34 methods:

```
head(methods(summary))
```

```
## [1] "summary.aov"                    "summary.aovlist"
## [3] "summary.aspell"                 "summary.check_packages_in_di
## [5] "summary.connection"             "summary.data.frame"
```

This means that when summary is passed an object of class `aov` (`print.aov`) it works differently than when it is passed a `data.frame` (`print.data.frame`)

Making Our Own Method

Making your own method is just like making your own function, except you need to name the function accordingly: `generic.class()`

To assign an object a class, use the `class()` function

```
string <- "Please print me!"
class(string)
```

```
## [1] "character"
```

```
print(string)
```

```
## [1] "Please print me!"
```

```
class(string) <- "refuseprint"
class(string)
```

```
## [1] "refuseprint"
```

```r
string <- "Please print me!"
```

```r
# print() method for objects of r class `refuseprint`
print.refuseprint <- function(x){
  print("I refuse to print!!!")
}

# Notice that I don't need to call `print.refuseprint()`
# R knows what to do!
print(string)
```

```
## [1] "I refuse to print!!!"
```

Side Note: This is why it is generally frowned upon to name objects using dot notation (e.g., `day.one`, `participant.ID`). The `.` actually means something, so it's best to reserve it for its purpose!

Revisiting Loops

`apply()` et al.

- `apply()`
- `lapply()`
- `sapply()`
- `mapply()`
- `tapply()`
- `replicate()`
- `sweep()`

Disclaimer

The `apply` family of functions (`*apply`) offer a different way to loop in R

Some people argue that these functions are faster (to write and also to execute) than `for` loops.

1. `*apply` is not faster to execute than a `for` loop, generally speaking
2. `*apply` may be faster to write (but may also not be)

Advantages of `*apply`

- You do not need to pre-allocate
- In some cases they *may* be faster than for loops (and in other cases they may not be)
- In some case they're easier to read (and sometimes they are not)

Bottom line: Use the tool that (a) makes most sense for your problem, (b) works for you and your collaborators, and (c) that you feel confident with

`lapply()`

```
lapply(X, FUN, ...)
```

`lapply()` iterates over `X` (a vector, list, or columns of a data frame), applies the `FUN` function to each element, and returns a list. The `...` argument allows you to pass additional arguments into `FUN`

```
# Vector    👇 anonymous function
lapply(1:3, function(x) x^2)
```

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

You can also use a named function 👇

```
square <- function(x) x^2
lapply(1:3, square)
```

```
# List
myList <- list("One" = 1:10,
               "Two" = lm(qsec ~ hp, m
               "Three" = mtcars)
lapply(myList, function(x) class(x))
```

```
## $One
## [1] "integer"
##
## $Two
## [1] "lm"
##
## $Three
## [1] "data.frame"
```

`...` in `lapply`

The `...` in `lapply` allow you to supply additional arguments to `FUN`.

For example, let's take the mean across each column of mtcars:[1]

```
lapply(mtcars, mean)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
```

```
lapply(mtcars, mean, na.rm = T)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
```

```
# Equivalent to:
lapply(mtcars, function(x) mean(x, na.rm = T))
```

[1] I introduced some `NA`s into `mtcars` for this example

`sapply()`: Simple `lapply()`

A downside of `lapply()` is that lists can be hard to work with and are also less common than other data types (vectors, dataframes, matrices). `sapply()` simplifies the output by returning a vector or a matrix

`sapply()` is a **wrapper** for `lapply()`, which means that it calls `lapply()` itself, then does some extra work for you

```
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = FALSE)
```

- `X`, `FUN`, and `...` are the same as in `lapply()`
- `simplify`: if `TRUE`, returns a vector or matrix (whichever is most appropriate), if `FALSE` returns a list
- `USE.NAMES`: if `TRUE` and `X` is character, use `X` as names for result

---

```
lapply(1:3, function(x) x^2)
```

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

```
sapply(1:3, function(x) x^2)
```

```
## [1] 1 4 9
```

`apply()`

`apply()` iterates over the margins of an array[1] (or matrix or dataframe)

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

- `X`: an array (or matrix or dataframe)
- `MARGIN`: a vector specifying the subscripts that the function will be applied over
  - `1` = rows
  - `2` = columns
  - `c(1, 2)` = rows and columns
- `...`: additional arguments to `FUN`
- `simplify`: if `TRUE` results are simplified to a vector, matrix, or dataframe (whichever is appropriate), if `FALSE` a list is returned

[1] an `array` is an object that can store data in more than 2 dimensions. We aren't talking about them in this class, but see `?array` for more info

`apply()`: Examples

```
# Take the mean down all rows, across all columns
apply(mtcars, 1, mean)
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710
##            29.90727            29.98136            23.59818
##   Hornet Sportabout             Valiant          Duster 360
##            53.66455            35.04909            59.72000
##            Merc 230            Merc 280           Merc 280C
##            27.23364            31.86000            31.78727
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Linco
##            46.50000            46.35000            66.23273
##   Chrysler Imperial            Fiat 128         Honda Civic
##            65.97227            19.44091            17.74227
##       Toyota Corona    Dodge Challenger          AMC Javelin
##            24.88864            47.24091            46.00773
##    Pontiac Firebird            Fiat X1-9       Porsche 914-2
##            57.37955            18.92864            24.77909
##      Ford Pantera L        Ferrari Dino       Maserati Bora
##            60.97182            34.50818            63.15545
```

`apply()`: Examples

```
# Take the mean down all rows, across all specified columns
apply(mtcars[, c("cyl", "drat", "wt")], 1, mean)
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710
##            4.173333            4.258333            3.390000
##   Hornet Sportabout             Valiant          Duster 360
##            4.863333            4.073333            4.926667
##            Merc 230            Merc 280           Merc 280C
##            3.690000            4.453333            4.453333
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Linco
##            4.933333            4.950000            5.393333
##   Chrysler Imperial             Fiat 128          Honda Civic
##            5.525000            3.426667            3.515000
##       Toyota Corona    Dodge Challenger          AMC Javelin
##            3.388333            4.760000            4.861667
##    Pontiac Firebird            Fiat X1-9       Porsche 914-2
##            4.975000            3.338333            3.523333
##      Ford Pantera L        Ferrari Dino       Maserati Bora
##            5.130000            4.130000            5.036667
```

`apply()`: Examples

```
# Take the sum across all columns
apply(mtcars, 2, sum)
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec
##  642.900  198.000 7383.100 4694.000  115.090  102.952  571.160
##       am     gear     carb
##   13.000  118.000   90.000
```

```
# Add 3 to all values
mtcars_p3 <- apply(mtcars, 1:2, function(x) x + 3)
head(mtcars_p3)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear c
## Mazda RX4          24.0   9  163 113 6.90 5.620 19.46  3  4    7
## Mazda RX4 Wag      24.0   9  163 113 6.90 5.875 20.02  3  4    7
## Datsun 710         25.8   7  111  96 6.85 5.320 21.61  4  4    7
## Hornet 4 Drive     24.4   9  261 113 6.08 6.215 22.44  4  3    6
## Hornet Sportabout 21.7  11  363 178 6.15 6.440 20.02  3  3    6
## Valiant            21.1   9  228 108 5.76 6.460 23.22  4  3    6
```

`mapply()`

`mapply()` is a multivariate version of `sapply()`

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

- `...` is a list of arguments to iterate over
- `MoreArgs` is a `list` of other arguments to pass to `FUN`

```
mapply(rep, 1:3, 7:5)
```

```
## [[1]]
## [1] 1 1 1 1 1 1 1
##
## [[2]]
## [1] 2 2 2 2 2 2
##
## [[3]]
## [1] 3 3 3 3 3
```

You can have has many `...` as you want!

```
mapply(sum, 1:3, 4:6, 7:9)
```

```
## [1] 12 15 18
```

`replicate()`

`replicate()` is a wrapper for a special case of `sapply()` where a single expression is **replicated** repeatedly

```
replicate(n, expr, simplify = "array")
```

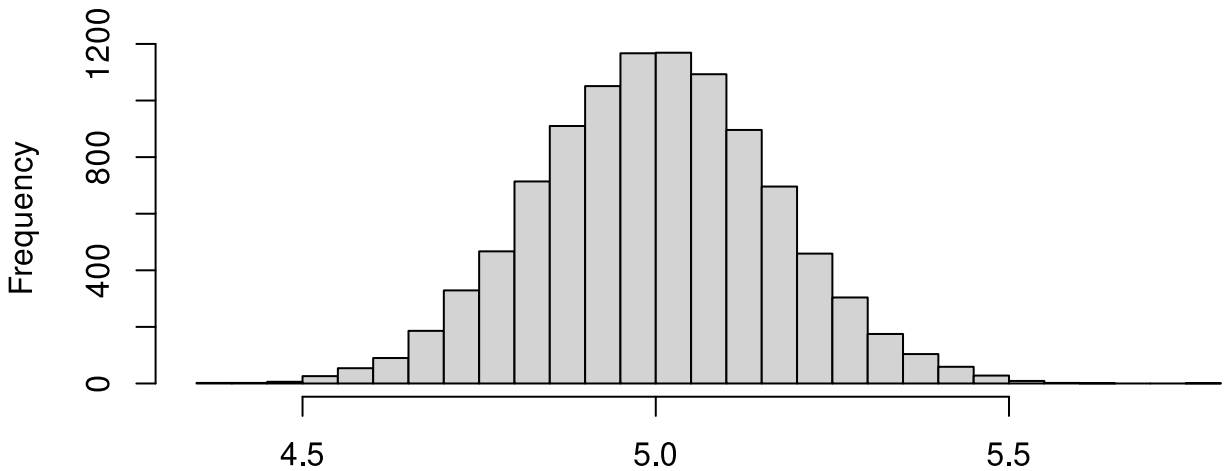
- `n` integer: the number of replications
- `expr`: the expression (i.e., R code) to evaluate *n* times
- `simplify` used to specify desired return value

`replicate()`: Example

`replicate()` is really good for conducting simulations because essentially all you are doing is repeatedly sampling from the *same* distrubtion

Let's simulate 10,000 samples of $n = 300$ from a uniform distribution (equal probability of all values, 0 to 10) and plot the means. **Why are these values normally distributed?**

```
hist(
  replicate(10000, mean(runif(300, 0, 10))),
  main = "", xlab = "", breaks = 25
)
```



`sweep()`

`sweep()` **sweeps out** a summary statistic from an input array (typically a matrix or dataframe)

```
sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
```

- `x`: an array (or matrix or dataframe)
- `MARGIN`: a vector of indices which correspond with `STATS` (this is typically columns [2] but can be rows [1] or both [c(1, 2)])
- `STATS`: the summary statistic to be swept out (typically a vector)
- `FUN`: the function to be used to carry out the sweep (default is to substract_)
- `check.margin`: if `TRUE` warn if `length(STATS)` doesn't match `length(x)`

`sweep()`: Example

It is often desired to *center* a variable prior to analysis. `sweep()` can be used to quickly center a bunch of columns in one call:

```
mtcars_c <- sweep(x = mtcars,
                  MARGIN = 2,
                  STATS = colMeans(mtcars))

head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
head(mtcars_c) |> round(1)
```

```
##                   mpg  cyl   disp    hp drat   wt qsec   vs   am gear carb
## Mazda RX4         0.9 -0.2  -70.7 -36.7  0.3 -0.6 -1.4 -0.4  0.6  0.3  1.2
## Mazda RX4 Wag     0.9 -0.2  -70.7 -36.7  0.3 -0.3 -0.8 -0.4  0.6  0.3  1.2
## Datsun 710        2.7 -2.2 -122.7 -53.7  0.3 -0.9  0.8  0.6  0.6  0.3 -1.8
## Hornet 4 Drive    1.3 -0.2   27.3 -36.7 -0.5  0.0  1.6  0.6 -0.4 -0.7 -1.8
## Hornet Sportabout -1.4  1.8  129.3  28.3 -0.4  0.2 -0.8 -0.4 -0.4 -0.7 -0.8
## Valiant           -2.0 -0.2   -5.7 -41.7 -0.8  0.2  2.4  0.6 -0.4 -0.7 -1.8
```

`tapply()`

`tapply()` is used to apply a function over descrete subsets of an array

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- `X`: an object that allows for subsetting (almost always a vector!)
- `INDEX`: a list of 1+ vectors (same length as `X`) that specify the groups to subset by
- `FUN`: the function to apply to each subset
- `...`: additional arguments to pass to `FUN`

`tapply()` is *very* useful for looking at descriptive statistics by group

Mean miles per gallon by automatic (0) or manual (1) transmission

```
tapply(mtcars$mpg, mtcars$am, mean)
```

```
##        0        1
## 17.14737 24.39231
```

Mean miles per gallon by automatic (0) or manual (1) transmission *and* number of cylinders (4, 6, or 8)

```
tapply(mtcars$mpg, list(mtcars$am, mtc
```

```
##        4        6     8
## 0 22.900 19.12500 15.05
## 1 28.075 20.56667 15.40
```