Writing Functions

Writing Functions

Next up we take on writing our own functions (we'll revisit this later on to go deeper). Knowing how to write functions vital to custom analyses!

- Function writing syntax

```
nameOfFunction <- function(input1, input2, ...) {
  #code
  #return something with return()
  #or returns last value
}
```

One nice thing is that you can generally look at the code for the functions you use by typing the function without `()` into the console.

```
var
```

```
function (x, y = NULL, na.rm = FALSE, use)
{
    if (missing(use))
        use <- if (na.rm)
            "na.or.complete"
        else "everything"
    na.method <- pmatch(use, c("all.obs", "complete.obs",
"pairwise.complete.obs",
        "everything", "na.or.complete"))
    if (is.na(na.method))
        stop("invalid 'use' argument")
    if (is.data.frame(x))
        x <- as.matrix(x)
    else stopifnot(is.atomic(x))
    if (is.data.frame(y))
        y <- as.matrix(y)
    else stopifnot(is.atomic(y))
    .Call(C_cov, x, y, na.method, FALSE)
}
<bytecode: 0x000000001f6cd810>
<environment: namespace:stats>
```

```
colMeans
```

```
function (x, na.rm = FALSE, dims = 1L)
{
    if (is.data.frame(x))
        x <- as.matrix(x)
    if (!is.array(x) || length(dn <- dim(x)) < 2L)
        stop("'x' must be an array of at least two dimensions")
    if (dims < 1L || dims > length(dn) - 1L)
```

```
    if (dims < 1L || dims > length(dn)) 1L)
        stop("invalid 'dims'")
    n <- prod(dn[id <- seq_len(dims)])
    dn <- dn[-id]
    z <- if (is.complex(x))
        .Internal(colMeans(Re(x), n, prod(dn), na.rm)) + (0+1i) *
            .Internal(colMeans(Im(x), n, prod(dn), na.rm))
    else .Internal(colMeans(x, n, prod(dn), na.rm))
    if (length(dn) > 1L) {
        dim(z) <- dn
        dimnames(z) <- dimnames(x)[-id]
    }
    else names(z) <- dimnames(x)[[dims + 1L]]
    z
}
<bytecode: 0x0000000020eb97c8>
<environment: namespace:base>
```

For some functions, they are generic and they won't show anything useful.

```
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x00000000160efed8>
<environment: namespace:base>
```

For those, you can pick a particular version of the function:

```
mean.default
```

```
function (x, trim = 0, na.rm = FALSE, ...)
{
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
        warning("argument is not numeric or logical: returning NA")
        return(NA_real_)
    }
    if (na.rm)
        x <- x[!is.na(x)]
    if (!is.numeric(trim) || length(trim) != 1L)
        stop("'trim' must be numeric of length one")
    n <- length(x)
    if (trim > 0 && n) {
        if (is.complex(x))
            stop("trimmed means are not defined for complex data")
        if (anyNA(x))
            return(NA_real_)
        if (trim >= 0.5)
            return(stats::median(x, na.rm = FALSE))
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
    }
    .Internal(mean(x))
}
```

```
<bytecode: 0x0000000016715b88>
<environment: namespace:base>
```

Ok, now you've seen some functions. Let's write our own!

Goal: Create a `standardize()` function (creating z-scores for a vector essentially)

- Take vector of values
    - subtract mean
    - divide by standard deviation
- Formula: For value i,

$$\frac{(value[i] - mean(value))}{sd(value)}$$

Let's take our generic syntax and apply it here.

```
nameOfFunction <- function(input1, input2, ...) {
  #code
  #return something with return()
  #or returns last value
}
```

```
standardize <- function(vector) {
  return((vector - mean(vector)) / sd(vector))
}
```

- Now use it! Create some data:

```
set.seed(10)
data <- runif(15)
data
```

```
 [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597 0.22543662
 [7] 0.27453052 0.27230507 0.61582931 0.42967153 0.65165567 0.56773775
[13] 0.11350898 0.59592531 0.35804998
```

- Apply the function:

```
result <- standardize(data)
result
```

```
 [1]  0.51053294 -0.52232963  0.09591275  1.46576309 -1.66286222
-0.94086777
 [7] -0.68822797 -0.69968029  1.06811337  0.11013572  1.25247769
0.82063172
[13] -1.51685322  0.96568634 -0.25843252
```

- Check result has mean 0 and sd 1

```
mean(result)
```

```
[1] 2.312784e-17
```

```
sd(result)
```

```
[1] 1
```

Goal: Add more inputs

- Make centering optional
- Make scaling optional

```
standardize <- function(vector, center, scale) {
  if (center) {
    vector <- vector - mean(vector)
    }
  if (scale) {
    vector <- vector / sd(vector)
    }
  return(vector)
  }
```

Here we've added arguments that should implicitly be `TRUE` or `FALSE` values (it would be better to give a default value so people using the function would know what is expected).

```
result <- standardize(data, center = TRUE, scale = TRUE)
result
```

```
 [1]  0.51053294 -0.52232963  0.09591275  1.46576309 -1.66286222
-0.94086777
 [7] -0.68822797 -0.69968029  1.06811337  0.11013572  1.25247769
0.82063172
[13] -1.51685322  0.96568634 -0.25843252
```

```
result <- standardize(data, center = FALSE, scale = TRUE)
result
```

```
 [1] 2.6115093 1.5786467 2.1968891 3.5667395 0.4381141 1.1601086 1.4127484
 [8] 1.4012961 3.1690897 2.2111121 3.3534540 2.9216081 0.5841231 3.0666627
[15] 1.8425438
```

- Give center and scale default arguments

```
standardize <- function(vector, center = TRUE, scale = TRUE) {
  if (center) {
    vector <- vector - mean(vector)
    }
  if (scale) {
    vector <- vector / sd(vector)
    }
  return(vector)
```

```
  }
```

- Apply it! The defaults will be used and aren't necessary if you don't want to change things.

```
standardize(data, center = TRUE, scale = TRUE)
```

```
 [1]  0.51053294 -0.52232963  0.09591275  1.46576309 -1.66286222
-0.94086777
 [7] -0.68822797 -0.69968029  1.06811337  0.11013572  1.25247769
0.82063172
[13] -1.51685322  0.96568634 -0.25843252
```

```
standardize(data)
```

```
 [1]  0.51053294 -0.52232963  0.09591275  1.46576309 -1.66286222
-0.94086777
 [7] -0.68822797 -0.69968029  1.06811337  0.11013572  1.25247769
0.82063172
[13] -1.51685322  0.96568634 -0.25843252
```

Goal: Also return

- `mean()` of original data
- `sd()` of original data

Return more than 1 object by returning a list (so we return one object, but a very flexible object that easily contains other objects!)

```
standardize <- function(vector, center = TRUE, scale = TRUE) {
  mean <- mean(vector) #save these so we can return them
  stdev <- sd(vector)
  if (center) {
    vector <- vector - mean
    }
  if (scale) {
    vector <- vector / stdev
    }
  return(list(vector, mean, stdev))
  }
```

- Apply it!

```
result <- standardize(data)
result
```

```
[[1]]
 [1]  0.51053294 -0.52232963  0.09591275  1.46576309 -1.66286222
-0.94086777
 [7] -0.68822797 -0.69968029  1.06811337  0.11013572  1.25247769
0.82063172
[13] -1.51685322  0.96568634 -0.25843252
```

```
[[2]]
[1] 0.4082695

[[3]]
[1] 0.1943237
```

```
result[[2]]
```

```
[1] 0.4082695
```

- We can fancy up what we return by giving names to the list elements!

```
standardize <- function(vector, center = TRUE, scale = TRUE) {
  mean <- mean(vector)
  stdev <- sd(vector)
  if (center) {
    vector <- vector - mean
    }
  if (scale) {
    vector <- vector / stdev
    }
  return(list(result = vector, mean = mean, sd = stdev))
  }
```

- Apply it!

```
result <- standardize(data, center = TRUE, scale = TRUE)
result
```

```
$result
 [1]  0.51053294 -0.52232963  0.09591275  1.46576309 -1.66286222
-0.94086777
 [7] -0.68822797 -0.69968029  1.06811337  0.11013572  1.25247769
0.82063172
[13] -1.51685322  0.96568634 -0.25843252

$mean
[1] 0.4082695

$sd
[1] 0.1943237
```

```
result$sd
```

```
[1] 0.1943237
```

`stop()` and `switch()`

Often you want to check on inputs to make sure they are of the right form (that's good practice if you are going to share your code). You can use `if()` or `switch()` to do this check.

Here we'll write a function to create a summary (mean, median, or trimmed mean).

- First we check the input to make sure it is a numeric vector.
- Then we use `stop()` to jump out if that condition isn't met.
- If the condition is met, we use `switch()` an alternative to `if/then/else` to pick which function to apply.

```
summarizer <- function(vec, type, trim = 0.05) {
  if(!is.vector(vec) | !is.numeric(vec)){
    stop("Not a vector or not numeric my friend.")
  }
  switch(type,
         mean = mean(vec),
         median = median(vec),
         trimmed = mean(vec, trim),
         stop("Mistake!")
         )
}
summarizer(letters, "mean")
```

```
Error in summarizer(letters, "mean"): Not a vector or not numeric my
friend.
```

```
summarizer(c(1,1,1,6,10), "mean")
```

```
[1] 3.8
```

```
summarizer(c(1,1,1,6,10), "trimmed", 0.2)
```

```
[1] 2.666667
```

```
summarizer(c(1,1,1,6,10), "means")
```

```
Error in summarizer(c(1, 1, 1, 6, 10), "means"): Mistake!
```

Naming conventions

That's the basics of function writing. Let's talk about a framework to make coherent code!

Use of consistent naming schemes is important!

Generally, naming objects must:

- start with a letter
- only have letters, numbers, `_`, and `.`

When we write functions and create objects we should try to follow this advice:

- Functions named using verbs
    - `standardize()` or `find_mean()` or `renderDataTable()`

- Data objects named using nouns
    - `my_df` or `weather_df`

Naming things is actually really tough... You should try to follow a common naming scheme:

- snake_case_used
- camelCaseUsed
- UpperCamelCase
- use.of.periods

You'll also need to name inputs to your functions. Try to stick to these when possible:

- x, y, z: vectors
- w: a vector of weights
- df: a data frame
- i, j: numeric indices (typically rows and columns)
- n: length, or number of rows
- p: number of columns

Otherwise, consider matching names of arguments in existing R functions. For example, use `na.rm` to determine if missing values should be removed.

There are some readings on this available in the weekly overview!

Input Matching

You might wonder why sometimes we name our arguments when we call our functions and sometimes we don't. Generally, we don't name the first 2-3 arguments but name ones after that. However, that is just convention. In R, you can use positional matching for everything or name each input, or combine the two ideas!

Let's look at some examples. Consider the inputs of the `cor()` function

```
function (x, y = NULL, use = "everything", method = c("pearson",
    "kendall", "spearman"))
```

- Apply it to `iris` data using positional matching (first argument to `x` second to `y`):

```
cor(iris$Sepal.Length, iris$Sepal.Width)
```

```
[1] -0.1175698
```

- R will use positional matching for all inputs not explicitly named. Here it applies `iris$Sepal.Width` to the first input of the function that wasn't specified, here `y`.

```
cor(x = iris$Sepal.Length, method = "spearman", iris$Sepal.Width)
```

```
cor(x = iris$Sepal.Length, method = "spearman", iris$Sepal.Width)
```

```
[1] -0.1667777
```

- R will also do partial matching but you should avoid this generally.

```
cor(iris$Sepal.Length, iris$Sepal.Width, met = "spearman")
```

Infix functions

Lastly, let's take up the idea of an **infix** function. An infix function is a function that goes between arguments (as opposed to prefix that goes prior to the arguments - what we usually do).

```
mean(3:5) #prefix
```

```
[1] 4
```

```
3 + 5 #+ is infix
```

```
[1] 8
```

```
`+`(3, 5) #used as a prefix function
```

```
[1] 8
```

Common *built-in* infix functions include:

- `::` (look directly in a package for a function)
- `$` (grab a column)
- `^`
- `*`
- `/`
- `+`
- `-`
- `>`
- `>=`
- `<`
- `<=`
- `==`
- `!=`
- `&` (and)
- `|` (or)
- `<-` (storage arrow)
- `|>` (pipe!)

Others infix operators use `%symbol%` syntax:

- `%*%` (matrix multiplication)
- `%in%` (check if LHS value(s) is(are) *in* RHS value(s)

We can call infix functions like prefix functions if we need to using the backtick symbol ``` (top left of the keyboard usually)

```
cars <- as.matrix(cars)
t(cars) %*% cars
```

```
      speed   dist
speed 13228  38482
dist  38482 124903
```

```
`%*%`(t(cars), cars)
```

```
      speed   dist
speed 13228  38482
dist  38482 124903
```

You can also write your own infix function!

```
`%+%` <- function(a, b) paste0(a, b)
"new" %+% " string"
```

```
[1] "new string"
```

R actually allows you to overwrite `+` and other operators: just don't do that... that wouldn't be good

With infix functions we can use precedence rules to save typing:

```
x <- y <- 2
`<-`(x, `<-`(y, 2)) #interpretation of above code!

x <- y = 2# error! <- has higher precedence
`=`(`<-`(x, y), 2) #interpretation of above code!

x = y <- 2 # this will work!
`=`(x, `<-`(y, 2)) #interpretation of above code!
```

This is one of the major differences between `=` and `<-` usage. You can't do

```
x = y = 2
```

but can do it with the storage arrow.

There is a weird difference between how infix functions are evaluated. For user defined infix functions, they evaluate left to right. For built-in ones, they evaluate right to left!

- User defined example:

```
`%-%` <- function(a, b) {
    paste0("(", a, " %-% ", b, ")")
}
```

```
"a" %-% "b" %-% "c" #user defined infix are evaluated left to right!
```

```
[1] "((a %-% b) %-% c)"
```

```
`%-%`(`%-%`("a", "b"), "c")  #interpretation of above code!
```

```
[1] "((a %-% b) %-% c)"
```

- Built-in example:

```
x <- y <- 2
`<-`(x, `<-`(y, 2)) #interpretation of above code!
```

Base R Pipe

This one deserves its own section! The pipe operator (`%>%`) was made popular by the `tidyverse` and the `magrittr` package. You would need to read in `dplyr` (part of the tidyverse) or `magrittr` to have access to the pipe.

Due to the popularity, R created a baseR pipe (`|>`). The idea of the pipe is to make code more readable! Essentially, you can read code left to right when using a pipe instead of inside out.

Consider the code below:

```
library(dplyr)
```

```
Warning: package 'dplyr' was built under R version 4.1.3


Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
arrange(select(filter(as_tibble(Lahman::Batting), teamID == "PIT"), playerI
```

```
# A tibble: 4,920 x 3
   playerID      G   X2B
   <chr>     <int> <int>
 1 wanerpa01   154    62
 2 wanerpa01   148    53
 3 sanchfr01   157    53
 4 wanerpa01   152    50
 5 comorad01   152    47
 6 mclouna01   152    46
 7 wagneho01   135    45
```

```
 8 parkeda01   158    45
 9 vanslan01   154    45
10 wagneho01   132    44
# i 4,910 more rows
```

- Forget what the functions do for a minute. To parse this we need to start on the inside.
    - The first function is `as_tibble(Lahman::Batting)`
    - The result of that is then the first argument to `filter()`
    - The result of this is then the first argument to `select()`
    - The result of that is then the first argument to `arrange()`
- Yikes. Piping makes thigns way easier to read!

```
Lahman::Batting |> #read the pipe as "then"
  as_tibble() |>
  filter(teamID == "PIT") |>
  select(playerID, G, X2B) |>
  arrange(desc(X2B))
```

```
# A tibble: 4,920 x 3
   playerID      G   X2B
   <chr>     <int> <int>
 1 wanerpa01   154    62
 2 wanerpa01   148    53
 3 sanchfr01   157    53
 4 wanerpa01   152    50
 5 comorad01   152    47
 6 mclouna01   152    46
 7 wagneho01   135    45
 8 parkeda01   158    45
 9 vanslan01   154    45
10 wagneho01   132    44
# i 4,910 more rows
```

- This is easy to parse!
    - First take the Batting dataset and turn it into a tibble (special data frame)
    - Then filter it
    - Then select from that
    - Then arrange that

Generically, `|>` does the following

- `x |> f(y)` turns into `f(x,y)`
- `x |> f(y) |> g(z)` turns into `g(f(x, y), z)`

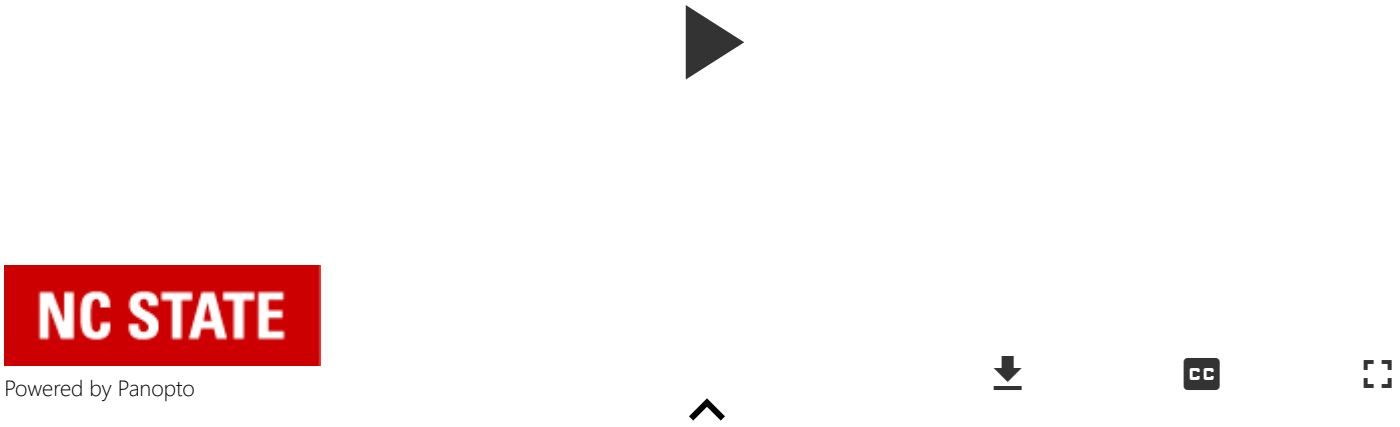
We'll be using this a lot from here on out!

Quick R Video

Please pop this video out and watch it in the full panopto player!

14- Writing Functions

[Auto-generated transcript. Edits may have been applied for clarity.]



Recap!

Functions allow you to customize your code

- Can specify default values and return multiple objects using a named list
- Much more to know!
    - Unnamed arguments
    - Input matching, environments, and lazy evaluation
    - Writing pipeable functions & side-effect functions
    - Infix functions
    - Helper functions and function writing strategy
- Naming conventions and input matching
- `stop()` and `switch()`
- infix functions