NC STATE UNIVERSITY

R Functions

Justin Post

What do we want to be able to do?

The course provides a brief overview of R data structures followed by the following topics:

- Loops in R
- Vectorized functions (apply family of functions)
- How R functions work
- Function writing

Writing Functions

- 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
}
```

Writing Functions

- Can look at code for functions

```
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: 0x0000000019558700>
## <environment: namespace:stats>
```

Writing Functions

- Can look at code for functions

```
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)
##         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
```

Writing Functions

- Can look at code for functions

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000017e89428>
## <environment: namespace:base>
```

Writing Functions

- Can look at code for functions

```
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
```

Writing Functions

- Goal: Create a `standardize()` function
- Take vector of values
  - subtract mean
  - divide by standard deviation
- z-score idea
- Formula: For value i,

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

Writing Functions

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

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

Writing Functions

- Now use it!

```
data <- runif(5)
data
```

```
## [1] 0.1985510 0.9055358 0.1268306 0.4333906 0.1967265
```

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

```
## [1] -0.5428453  1.6671771 -0.7670422  0.1912591 -0.5485487
```

Writing Functions

- Check result has mean 0 and sd 1

```
mean(result)
```

```
## [1] 2.218278e-17
```

```
sd(result)
```

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

Writing Functions

- 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)
}
```

Writing Functions

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

```
## [1] -0.5428453  1.6671771 -0.7670422  0.1912591 -0.5485487
```

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

```
## [1] 0.6206673 2.8306897 0.3964704 1.3547717 0.6149638
```

Writing Functions

- 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)
}
```

Writing Functions

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

```
## [1] -0.5428453  1.6671771 -0.7670422  0.1912591 -0.5485487
```

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

```
## [1] -0.5428453  1.6671771 -0.7670422  0.1912591 -0.5485487
```

Writing Functions

- Return more than 1 object by returning a list
- Goal: Also return
  - `mean()` of original data
  - `sd()` of original data

Writing Functions

```
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(vector, mean, stdev))
}
```

Writing Functions

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

```
## [[1]]
## [1] -0.5428453  1.6671771 -0.7670422  0.1912591 -0.5485487
##
## [[2]]
## [1] 0.3722069
##
## [[3]]
## [1] 0.3198993
```

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

```
## [1] 0.3722069
```

Writing Functions

- Fancy up what we return by giving names

```
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))
}
```

Writing Functions

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

```
## $result
## [1] -0.5428453  1.6671771 -0.7670422  0.1912591 -0.5485487
##
## $mean
## [1] 0.3722069
##
## $sd
## [1] 0.3198993
```

```r
result$sd
```

```
## [1] 0.3198993
```

Quick Examples

- Go to the course files page and try Exercise 4 - Basic Functions

Writing Functions

- Can bring in unnamed arguments
- Arguments that can be used by functions **inside** your function

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000017e89428>
## <environment: namespace:base>
```

```
x <- c(rnorm(5), NA)
mean(x)
```

```
## [1] NA
```

```
mean(x, na.rm = TRUE)
```

```
## [1] 0.4093447
```

Writing Functions

- Can bring in unnamed arguments
- Arguments that can be used by functions **inside** your function

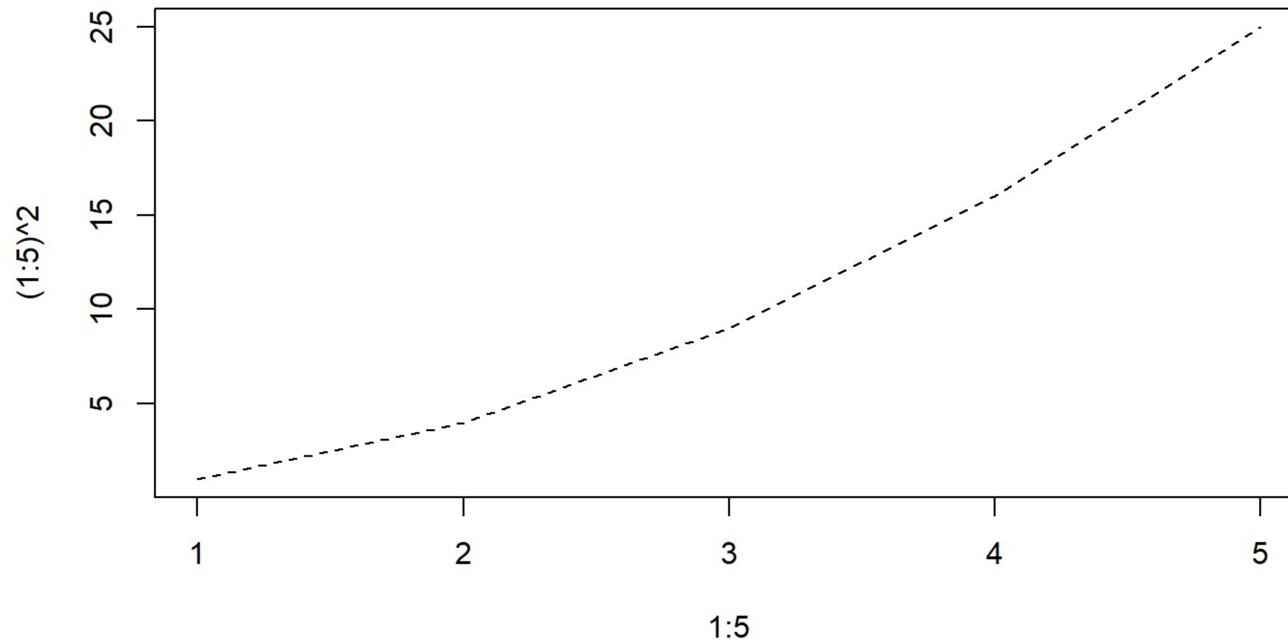
```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x00000000161dd960>
## <environment: namespace:graphics>
```

```
plot(x = 1:5, y = (1:5)^2, type = "l", lty = "dashed")
```

Writing Functions

```
plot(x = 1:5, y = (1:5)^2, type = "l", lty = "dashed")
```



Writing Functions

```
plot.default
```

```
## function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
##     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
##     ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
##     panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,
##     ...)
## {
##     localAxis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)
##     localBox <- function(..., col, bg, pch, cex, lty, lwd) box(...)
##     localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)
##     localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)
##     xlabel <- if (!missing(x))
##         deparse(substitute(x))
##     ylabel <- if (!missing(y))
##         deparse(substitute(y))
##     xy <- xy.coords(x, y, xlabel, ylabel, log)
##     xlab <- if (is.null(xlab))
##         xy$xlab
##     else xlab
##     ylab <- if (is.null(ylab))
##         xy$ylab
##     else ylab
##     xlim <- if (is.null(xlim))
```

Writing Functions

- Add unnamed arguments to our function for use with `sd()` and `mean()`

```
sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x0000000018230908>
## <environment: namespace:stats>
```

```
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)
```

Writing Functions

- Add unnamed arguments to our function for use with `sd()` and `mean()`

```
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))
}
```

Writing Functions

```
sData <- standardize(airquality$Ozone, na.rm = TRUE)
sData$sd
```

```
## [1] 32.98788
```

```
sData$result
```

```
##   [1] -0.03423409 -0.18580489 -0.91334473 -0.73145977          NA -0.42831817
##   [7] -0.57988897 -0.70114561 -1.03460136          NA -1.06491552 -0.79208809
##  [13] -0.94365889 -0.85271641 -0.73145977 -0.85271641 -0.24643321 -1.09522968
##  [19] -0.36768985 -0.94365889 -1.24680048 -0.94365889 -1.15585800 -0.30706153
##  [25]          NA          NA          NA -0.57988897  0.08702254  2.20901373
##  [31] -0.15549073          NA          NA          NA          NA          NA
##  [37]          NA -0.39800401          NA  0.87519070 -0.09486241          NA
##  [43]          NA -0.57988897          NA          NA -0.64051729 -0.15549073
##  [49] -0.67083145 -0.91334473 -0.88303057          NA          NA          NA
##  [55]          NA          NA          NA          NA          NA          NA
##  [61]          NA  2.81529692  0.20827918 -0.30706153          NA  0.66299158
##  [67] -0.06454825  1.05707566  1.66335885  1.66335885  1.29958893          NA
##  [73] -0.97397305 -0.45863233          NA -1.06491552  0.17796502 -0.21611905
##  [79]  0.57204910  1.11770398  0.63267742 -0.79208809          NA          NA
##  [85]  1.14801813  1.99681461 -0.67083145  0.29922166  1.20864645  0.23859334
##  [91]  0.66299158  0.51142078 -0.09486241 -1.00428721 -0.79208809  1.08738982
##  [97] -0.21611905  0.72361990  2.42121284  1.42084557  2.05744293          NA
```

Writing Functions

- Note: You can get at the unnamed arguments with `list(...)`

```
f <- function(x, ...){
    unnamed <- names(list(...))
    unnamedVals <- list(...)
    modifyX <- x^2
    return(list(newX = modifyX, elipNames = unnamed, elipValues = unnamedVals))
}
```

Writing Functions

```
f(x = 10, a = 1, b = "hey there", num = 1:3)
```

```
## $newX
## [1] 100
##
## $elipNames
## [1] "a"   "b"   "num"
##
## $elipValues
## $elipValues$a
## [1] 1
##
## $elipValues$b
## [1] "hey there"
##
## $elipValues$num
## [1] 1 2 3
```

Recap

- Function writing opens R up!
- Syntax

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

- Can set defaults in function definition
- Can return a named list
- Can give unnamed arguments for use

Naming conventions and input matching

- Use of consistent naming schemes is important!
- Objects
  - must start with a letter
  - can only have letters, numbers, _, and .

Naming conventions and input matching

- Use of consistent naming schemes is important!
- Objects
  - must start with a letter
  - can only have letters, numbers, _, and .
- Functions usually verbs, data objects usually nouns
- Naming schemes
  - snake_case_used
  - camelCaseUsed
  - UpperCamelCase
  - use.of.periods

Naming conventions and input matching

- Also need to name inputs! (From R for Data Science)
  - 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.

Naming conventions and input matching

- Consider the inputs of the `cor()` function

```
cor
```

```
## function (x, y = NULL, use = "everything", method = c("pearson",
##     "kendall", "spearman"))
## {
##     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")
##     method <- match.arg(method)
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     if (!is.matrix(x) && is.null(y))
##         stop("supply both 'x' and 'y' or a matrix-like 'x'")
##     if (!(is.numeric(x) || is.logical(x)))
##         stop("'x' must be numeric")
##     stopifnot(is.atomic(x))
##     if (!is.null(y)) {
##         if (!(is.numeric(y) || is.logical(y)))
##             stop("'y' must be numeric")
```

Naming conventions and input matching

- Consider the inputs of the `cor()` function
- Apply it to `iris` data…

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

```
## [1] -0.1175698
```

- Notice R doesn't require names: here it is using positional matching

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

Naming conventions and input matching

- Consider the inputs of the `cor()` function
- Apply it to `iris` data…

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

```
## [1] -0.1667777
```

- Positional match for inputs not explicitly called

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

Naming conventions and input matching

- Consider the inputs of the `cor()` function
- Apply it to `iris` data…

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

```
## [1] -0.1667777
```

- Partial matching is used if not exactly met!

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

stop() and switch()

To kick out of a function, you can use `stop()`

```
transposeDF <- function(df) {
    if(!is.data.frame(df)){
        stop("I want a data frame only!")
    }
    t(df)
}
transposeDF(iris)
```

```
##              [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
## Sepal.Length "5.1"    "4.9"    "4.7"    "4.6"    "5.0"    "5.4"    "4.6"
## Sepal.Width  "3.5"    "3.0"    "3.2"    "3.1"    "3.6"    "3.9"    "3.4"
## Petal.Length "1.4"    "1.4"    "1.3"    "1.5"    "1.4"    "1.7"    "1.4"
## Petal.Width  "0.2"    "0.2"    "0.2"    "0.2"    "0.2"    "0.4"    "0.3"
## Species      "setosa" "setosa" "setosa" "setosa" "setosa" "setosa" "setosa"
##              [,8]     [,9]     [,10]    [,11]    [,12]    [,13]    [,14]
## Sepal.Length "5.0"    "4.4"    "4.9"    "5.4"    "4.8"    "4.8"    "4.3"
## Sepal.Width  "3.4"    "2.9"    "3.1"    "3.7"    "3.4"    "3.0"    "3.0"
## Petal.Length "1.5"    "1.4"    "1.5"    "1.5"    "1.6"    "1.4"    "1.1"
## Petal.Width  "0.2"    "0.2"    "0.1"    "0.2"    "0.2"    "0.1"    "0.1"
## Species      "setosa" "setosa" "setosa" "setosa" "setosa" "setosa" "setosa"
##              [,15]    [,16]    [,17]    [,18]    [,19]    [,20]    [,21]
## Sepal.Length "5.8"    "5.7"    "5.4"    "5.1"    "5.7"    "5.1"    "5.4"
```

`stop()` and `switch()`

To kick out of a function, you can use `stop()`

```
transposeDF <- function(df) {
    if(!is.data.frame(df)){
        stop("I want a data frame only!")
    }
    t(df)
}
transposeDF(as.matrix(iris))
```

```
## Error in transposeDF(as.matrix(iris)): I want a data frame only!
```

`stop()` and `switch()`

Often you want to check on inputs, can use `if()` or `switch()`

```
center <- function(vec, type, ...) {
  if(!is.vector(vec)){
    stop("Not a vector my friend.")
  }
    switch(type,
                  mean = vec - mean(vec),
                  median = vec - median(vec),
                  trimmed = vec - mean(vec, ...),
                  stop("Mistake!")
    )
}
center(c(1,1,1,6,10), "mean")
```

```
## [1] -2.8 -2.8 -2.8  2.2  6.2
```

`stop()` and `switch()`

Often you want to check on inputs, can use `if()` or `switch()`

```
center <- function(vec, type, ...) {
  if(!is.vector(vec)){
    stop("Not a vector my friend.")
  }
    switch(type,
                 mean = vec - mean(vec),
                 median = vec - median(vec),
                 trimmed = vec - mean(vec, ...),
                 stop("Mistake!")
    )
}
center(c(1,1,1,6,10), "median")
```

```
## [1] 0 0 0 5 9
```

`stop()` and `switch()`

Often you want to check on inputs, can use `if()` or `switch()`

```
center <- function(vec, type, ...) {
  if(!is.vector(vec)){
    stop("Not a vector my friend.")
  }
    switch(type,
                 mean = vec - mean(vec),
                 median = vec - median(vec),
                 trimmed = vec - mean(vec, ...),
                 stop("Mistake!")
    )
}
center(c(1,1,1,6,10), "trimmed", trim = 0.2)
```

```
## [1] -1.666667 -1.666667 -1.666667  3.333333  7.333333
```

`stop()` and `switch()`

Often you want to check on inputs, can use `if()` or `switch()`

```
center <- function(vec, type, ...) {
    if(!is.vector(vec)){
        stop("Not a vector my friend.")
    }
    if(type == "mean"){
        vec - mean(vec)
    } else if (type =="median"){
        vec - median(vec)
    } else if (type =="trimmed") {
        vec - mean(vec, ...)
    } else {
        stop("Mistake!")
    }
}
center(c(1,1,1,6,10), "trimmed", trim = 0.2)
```

```
## [1] -1.666667 -1.666667 -1.666667  3.333333  7.333333
```

Quick Examples

- Go to the course files page and try Exercise 5 - More Functions

Writing Pipeable functions

- Two types of pipeable functions:

  1. **transformations**
  2. **side-effects**

Pipeable functions

- Two types of pipeable functions:
  1. **transformations**
  2. **side-effects**
- transformations naturally return the modified argument (df)
- side-effects don't - usually a plot, saving a file, etc.
- can silently return df with `invisible()`

Pipeable functions

- Two types of pipeable functions:
    1. **transformations**
    2. **side-effects**

```
printNumObs <- function(df) {
    cat("The number of observations in the data set is ", nrow(df), "\n", sep = "")
}
iris %>% printNumObs %>% summarize(mean = mean(Sepal.Length))
```

```
## The number of observations in the data set is 150
```

```
## Error in UseMethod("summarise"): no applicable method for 'summarise' applied to an object c
```

Pipeable functions

- Two types of pipeable functions:
    1. **transformations**
    2. **side-effects**

```
printNumObs <- function(df) {
    cat("The number of observations in the data set is ", nrow(df), "\n", sep = "")
    invisible(df)
}
iris %>% printNumObs %>% summarize(mean = mean(Sepal.Length))
```

```
## The number of observations in the data set is 150
```

```
##       mean
## 1 5.843333
```

Pipeable functions

```
printNumObs <- function(df) {
    cat("The number of observations in the data set is ", nrow(df), "\n", sep = "")
    invisible(df)
}
temp <- printNumObs(iris)
```

```
## The number of observations in the data set is 150
```

```
str(temp)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Infix functions

- Speaking of pipes…
- Infix function - a function that goes between arguments (as opposed to prefix)

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

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

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

```
## [1] 8
```

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

```
## [1] 8
```

Infix functions

- Infix function - a function that goes between arguments (as opposed to prefix)
- Common *built-in* infix functions include: `::`, `$`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&`, `|`, `<-`
- Others use `%symbol%` syntax: `%*%` (matrix multiplication), `%in%` (check if LHS value(s) is(are) *in* RHS value(s)
- Can call like prefix functions

```
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
```

Infix functions

- Infix function - a function that goes between arguments (as opposed to prefix)
- Can write your own!

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

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

- Can overwrite + and other operators: just don't do that…

Infix functions

- Infix function - a function that goes between arguments (as opposed to prefix)
- Can use precendence 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!
```

Infix functions

- Infix function - a function that goes between arguments (as opposed to prefix)
- Can use precendence rules to save typing

```
`%-%` <- 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!
```

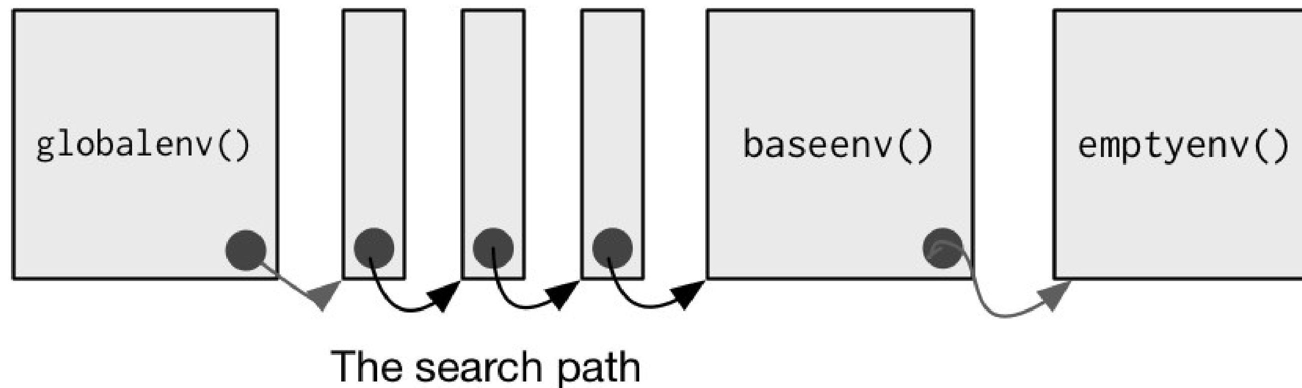
```
## [1] "((a %-% b) %-% c)"
```

Environments and Lexical Scoping

- R objects live in an environment
- You can think of it as a “bag of names” that point to things in memory
- Like a list but with no ordering (and other things)

Environments and Lexical Scoping

- Environments have ‘parents’ and ‘children’
  - Global environment is where our created function objects live
  - Search path has all packages loaded in (most recent package is the parent of the global environment)
  - base environment is the child of the ultimate ancestor, the empty environment



Environments and Lexical Scoping

- Environments have 'parents' and 'children'
  - Global environment is where our created function objects live
  - Search path has all packages loaded in (most recent package is the parent of the global environment)
  - base environment is the child of the ultimate ancestor, the empty environment

```
##  [1] ".GlobalEnv"         "package:knitr"     "package:forcats"
##  [4] "package:stringr"   "package:dplyr"     "package:purrr"
##  [7] "package:readr"     "package:tidyr"     "package:tibble"
## [10] "package:ggplot2"   "package:tidyverse" "package:stats"
## [13] "package:graphics"  "package:grDevices" "package:utils"
## [16] "package:datasets"  "package:methods"   "Autoloads"
## [19] "package:base"
```

Environments and Lexical Scoping

- Don't need to fully understand them… a few important things to know

```
library(pryr) #install if needed
x <- "hey"
where("x")
```

```
## <environment: R_GlobalEnv>
```

```
where("mean")
```

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

Environments and Lexical Scoping

- When you call a function, it creates temporary function environments
- This is why variables in functions don't overwrite things (mean still exists as is!)

```
f <- function(x){
    mean <- paste0(x, " is a value")
    mean
}
f(1:3)
```

```
## [1] "1 is a value" "2 is a value" "3 is a value"
```

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000017e89428>
## <environment: namespace:base>
```

Environments and Lexical Scoping

- When you call a function, it creates temporary function environments

```
g <- function(x) {
  if (!exists("a", inherits = FALSE)) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
```

```
#Running the function doesn't create
#the a object in our global environment!
g(10)
```

```
## Defining a
```

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

```
g(10)
```

```
## Defining a
```

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

Environments and Lexical Scoping

- When you call a function, it creates temporary function environments
- This is why variables can have the same name in a function and in your global environment

```
y <- 10
f <- function(x){
    y <- 1
    x + y
}
f(15)
```

```
## [1] 16
```

Environments and Lexical Scoping

- If R doesn't find an object in the current environment, it will search up the path

```
y <- 1
f <- function(x){
    x + y
}
f(10)
```

```
## [1] 11
```

- Much more to it, but this should give you a strong foundation

Quick Examples

- Go to the course files page and try Exercise 6 - Last Functions