Midterm Review

ISOM3390: Business Programming in R

Format and Logistics of the Midterm Quiz

A close-note in-class midterm quiz on next Monday (Oct. 22nd)

- An appendix with all functions used in the exam paper will be given

Coverage: Topic 2 - Topic 5

Two sections of questions:

- MC questions (29 questions, in total 80 points)
    - the first 22 questions, each worth 3 points
    - the last 7 questions, each worth 2 points
- Short answer questions (4 questions, in total 20 points)
    - Evaluate R code and provide the results

R Programming in a Nutshell

Everything we'll do comes down to applying **functions** to **data**.

- **Data**: things like 7, "seven", 7.000, the matrix

$$\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$$

- **Functions**: things like log, + (taking two arguments), < (two), mod (two), mean (one)
    - A function is a machine which turns input objects (arguments) into an output object (return value), according to a definite rule
    - Machines are made out of machines; functions are made out of functions
        - E.g., `sum(c(a, b)^2)` for $f(a, b) = a^2 + b^2$

Before Functions, Data

At base level, all data can represented in binary format, by bits (i.e., TRUE/FALSE, YES/NO, 1/0).

Basic data types:

- Logicals: Direct binary values: `TRUE` or `FALSE`
- Integers: whole numbers (positive, negative or zero), represented by a fixed-length block of bits
- Characters: fixed-length blocks of bits, with special coding; and strings: sequences of characters
- Floating point numbers: an integer times a positive integer to the power of an integer, as in 3×10^6 or 1×3^{-1}
- Missing or ill-defined values: `NA`, `NaN`, etc.

Operators

- **Unary**: take just one argument. E.g., `-` for arithmetic negation, `!` for Boolean negation
- **Binary**: take two arguments. E.g., `+`, `-`, `*`, and `/`. Also, `%%` and `^`

```
-7

## [1] -7

7 + 5

## [1] 12

7 - 5

## [1] 2
```

```
7 * 5

## [1] 35

7^5

## [1] 16807

7/5

## [1] 1.4

7%%5

## [1] 2
```

Comparison Operators

These are also binary operators; they take two objects, and give back a logical value

```
7 > 5
```

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

```
7 < 5
```

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

```
7 >= 7
```

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

```
7 <= 5
```

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

```
7 == 5
```

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

```
7 != 5
```

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

Logical Operators

These basic ones are `&` (and) and `|` (or)

```
(5 > 7) & (6 * 7 == 42)
```

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

```
(5 > 7) | (6 * 7 == 42)
```

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

```
(5 > 7) | (6 * 7 == 42) & (0 != 0)
```

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

```
(5 > 7) | (6 * 7 == 42) & (0 != 0) | (9 - 8 >= 0)
```

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

Note: The double forms `&&` and `||` are different!

Data can Have Names

The assignment operator, `<-` or `=`, is used to bind an object to a name.

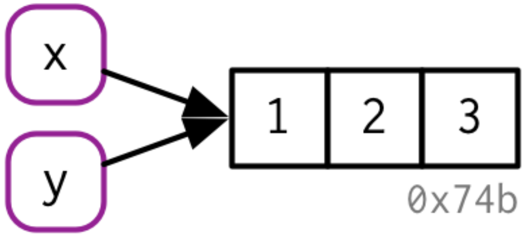
```
x <- c(1, 2, 3)
```

This above code does two things:

- Creating an object, a vector of values, `c(1, 2, 3)`.
- Binding the object to a name, `x`.

A name can be thought of as a reference to a value.

```
y <- x
```



The memory address of an object can be accessed by `lobstr::obj_addr()`:

```
lobstr::obj_addr(x)
```

```
## [1] "0x7f8550e43c80"
```

```
lobstr::obj_addr(y)
```

```
## [1] "0x7f8550e43c80"
```

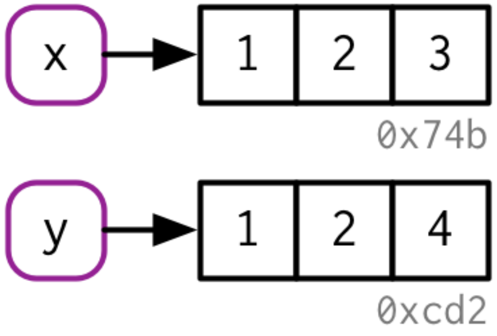
```
y[[3]] <- 4
```

R creates a new object with one value changed. Then it rebinds `y` to the new object.

```
lobstr::obj_addr(y)
```

```
## [1] "0x7f8551883c08"
```

Object that are not bound to names will get deleted by the garbage collector.



Naming Conventions

Object names must start with a letter, and can only contain:

- letters
- numbers
- the character `_`
- the character `.`

```r
a <- 0
first.variable <- 1
First.Variable <- 2
variable_2 <- 1 + first.variable
very_long_name.3 <- 4
```

Some words are reserved in R and cannot be used as object names:

- `Inf` and `-Inf` which respectively stand for positive and negative infinity.
- `NULL` denotes a null object.
- `NA` ("Not Available") represents a missing value.
- `NaN` means "Not a Number".

The Basic Data Structure: Vectors

A **data structure** allows us to group related values into an object.

A **vector** is an indexed sequence of values, all of the same type. It is the most basic data structure in R.

```
(dbl_var <- c(1, 2.5, 4.5))
```

```
## [1] 1.0 2.5 4.5
```

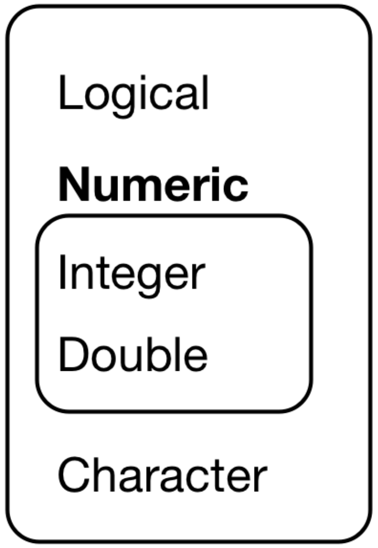
```
(log_var <- c(TRUE, FALSE, T, F))
```

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

```
(chr_var <- c("R programming", "Business Analytics"))
```

```
## [1] "R programming"      "Business Analytics"
```

Four common types of vectors: ***logical***; ***integer*** and ***double***, collectively known as ***numeric***; ***character***.



Two rare types: ***complex*** and ***raw***.

Type and Length of Vectors

Two intrinsic properties:

- Type, `typeof()`, what it is; it depends on the type of its elements.

```
## Numbers are generally treated as doubles (i.e. double precision real
## numbers)
typeof(c(1, 6, 10))
```

```
## [1] "double"
```

```
## If we explicitly want an integer, we need to specify the L suffix
typeof(c(1L, 6L, 10L))
```

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

- Length, `length()`, how many elements it contains.

```
# R has no scalar types. Individual numbers or strings are vectors of length
# 1.
length(7)
```

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

```
length(c(1, 6, 10))
```

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

Creating Vectors

Empty vectors of the given length and type can be created with the `vector()` function:

```
weekly.hours <- vector("integer", length = 5)
str(weekly.hours)
```

```
##  int [1:5] 0 0 0 0 0
```

```
typeof(weekly.hours)
```

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

```
length(weekly.hours)
```

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

```
weekly.hours[5] <- 8
```

Tests

Testing with the "`is`" functions:

- `is.character()`, `is.double()`, `is.integer()`, `is.logical()`;
- `is.vector()` tests for vectors with no attributes apart from names
- `is.atomic()` tests for atomic vectors or `NULL`
- `is.numeric()` tests for the numerical-ness of a vector, whether it's built on top of an integer or double.

```
is.vector(7)
```

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

```
is.atomic(c(1, 2.5, 4.5))
```

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

```
is.integer(c(2, 3, 4))
```

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

Implicit Coercion

When attempting to combine different types of vectors, they will be coerced to the most flexible type.

Types from least to most flexible are: logical, integer, double, and character.

```
str(c("a", 1))
```

```
##  chr [1:2] "a" "1"
```

```
# When a logical vector is coerced to an integer or double, TRUE becomes 1
# and FALSE becomes 0.
str(c(2, TRUE, F))
```

```
##  num [1:3] 2 1 0
```

```
# This is very useful in conjunction with sum() and mean(). E.g., the
# proportion that are TRUE can be given by
mean(as.numeric(c(FALSE, FALSE, TRUE)))
```

```
## [1] 0.3333333
```

Coercion often happens automatically.

```r
0 & TRUE  # logical operations (&, |, any, etc) will coerce to a logical
## [1] FALSE
TRUE + 2  # mathematical functions (+, log, abs, etc.) will coerce to a double or integer
## [1] 3
```

- Note: We will usually get a warning message if the coercion might lose information.

Explicit Coercion

If confusion is likely, objects can be explicitly coerced from one type to another using the "`as`" functions, if available.

```
x <- 0:6
typeof(x)
```

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

```
typeof(as.numeric(x))
```

```
## [1] "double"
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Nonsensical coercion results in `NA`s.

```
as.numeric(c("a", "b", "c"))
```

```
## Warning: NAs introduced by coercion
```

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

Attributes

For more complex R objects, e.g., named vectors, `attributes()` gives all additional (arbitrary) properties called attributes at once.

```
str(chr_var)
```

```
##  chr [1:2] "R programming" "Business Analytics"
```

```
attributes(chr_var)
```

```
## NULL
```

```
# The elements of a vector can be named through the function names().
names(chr_var) <- c("Course 1", "Course 2")
chr_var
```

```
##             Course 1             Course 2 
##      "R programming" "Business Analytics"
```

```
str(chr_var)
```

```
##  Named chr [1:2] "R programming" "Business Analytics"
##  - attr(*, "names")= chr [1:2] "Course 1" "Course 2"
```

```
attributes(chr_var)
```

```
## $names
## [1] "Course 1" "Course 2"
```

Subsetting a Vector

Subsetting allows us to pull out the pieces that we're interested in.

`[` is used to subset an vector.

```
x <- c(2.1, 4.2, 3.3, 5.4)
# Positive integers return elements at the specified positions
x[c(3, 1)]
```

```
## [1] 3.3 2.1
```

```
# Negative integers omit elements at the specified positions
x[-c(3, 1)]
```

```
## [1] 4.2 5.4
```

```
# Logical vectors select elements where the corresponding logical value is
# TRUE
x[c(TRUE, TRUE, FALSE, FALSE)]
```

```
## [1] 2.1 4.2
```

```
# If the vector is named, we can also use character vectors to return
# elements with matching names.
chr_var["Course 1"]
```

```
##        Course 1
## "R programming"
```

Vector Arithmetics

Arithmetic operators apply to vectors in an elementwise fashion:

```
v1 <- c(4, 6, 8, 24)
v2 <- c(34, 32.4, 12, 2.7)
v1 + v2
```

```
## [1] 38.0 38.4 20.0 26.7
```

The `+` operator connects the whole vectors directly as opposed to their individual elements (e.g., `v1[1] + v2[1]`; usually requires a loop that iterates over the elements).

In effect, R still performs the operation pairwise or elementwise.

Can also do pairwise comparisons:

```
v1 > 9
```

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

Logical operators work elementwise:

```
(v1 > 9) & (v1 < 20)
```

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

Recycling

When the vectors do not have the same length, a **recycling rule** by repeating the shorter vector till it fills in the size of the larger will be used.

```
v3 <- c(10, 2)
v1 + v3
```

```
## [1] 14  8 18 26
```

```
(z <- 2 * v1)  # actually it is the vector c(2)!
```

```
## [1]  8 12 16 48
```

Factors

An important type of augmented vectors (i.e., vectors with additional attributes) is factors, which are designed to handle categorical data.

```
d1 <- c("Male", "Female", "Male", "Female", "Female", "Female")
(d2 <- factor(d1))
```

```
## [1] Male   Female Male   Female Female Female
## Levels: Female Male
```

```
typeof(d2)
```

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

They are built on top of **integer** vectors and can only contain **pre-defined values**.

- A factor is stored internally as an integer vector with values `1`, `2`, …, `k`, where `k` is the number of levels of the factor.
- Using factors with labels (e.g., `"Male"` and `"Female"`) is better than using integers (e.g., `1` and `2`) because they are self-describing.

The two attributes, **class** and **levels**, make them behave differently from regular integer vectors.

```
attributes(d2)
```

```
## $levels
## [1] "Female" "Male"
##
## $class
## [1] "factor"
```

```
class(d2)
```

```
## [1] "factor"
```

```
levels(d2)
```

```
## [1] "Female" "Male"
```

The order of the levels can be set using the `levels` argument to the `factor()` function. This can be important in linear modelling because the first level is used as the baseline level.

```
(gender <- factor(d1, levels = c("Male", "Female")))
```

```
## [1] Male   Female Male   Female Female Female
## Levels: Male Female
```

```
gender[2] <- "Male"
gender
```

```
## [1] Male   Male   Male   Female Female Female
## Levels: Male Female
```

```
# By default, unused levels are not dropped.
gender[2]
```

```
## [1] Male
## Levels: Male Female
```

```
gender[2, drop = T]
```

```
## [1] Male
## Levels: Male
```

```
# But we can't use values that are not in the levels
gender[2] <- "Yes"
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = "Yes"): invalid factor level,
## NA generated
```

```
gender
```

```
## [1] Male   <NA>   Male   Female Female Female
## Levels: Male Female
```

Usage of Factors

One of the many things we can do with factors is to count the occurrence of each possible value.

```
table(gender)
```

```
## gender
##   Male Female
##      2      3
```

The `table()` function can also be used to obtain cross-tabulation of several factors.

```
age <- factor(c("adult", "adult", "juvenile", "juvenile", "adult", "adult"))
(t <- table(gender, age))
```

```
##         age
## gender   adult juvenile
##   Male       1        1
##   Female     2        1
```

Other functions that work with factors include `margin.table()`, `prop.table()`, etc.

Lists

Different from a **vector**, the elements of a **list** can be of any type.

Lists are constructed by using `list()`:

```
(mylst <- list(34453, "John", marks = c(14.3, 12, 15, 19)))
```

```
## [[1]]
## [1] 34453
##
## [[2]]
## [1] "John"
##
## $marks
## [1] 14.3 12.0 15.0 19.0
```

```
str(mylst)
```

```
## List of 3
##  $      : num 34453
##  $      : chr "John"
##  $ marks: num [1:4] 14.3 12 15 19
```

```
typeof(mylst)
```

```
## [1] "list"
```

The elements of a list are always numbered and may also have a name attached to them.

Lists Are Recursive

A list can contain other lists:

```
lists <- list(mylst, list(id = 34454, name = "Karen"))
str(lists)
```

```
## List of 2
##  $ :List of 3
##   ..$      : num 34453
##   ..$      : chr "John"
##   ..$ marks: num [1:4] 14.3 12 15 19
##  $ :List of 2
##   ..$ id  : num 34454
##   ..$ name: chr "Karen"
```

```
is.recursive(lists)
```

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

```
lists
```

```
## [[1]]
## [[1]][[1]]
## [1] 34453
##
## [[1]][[2]]
## [1] "John"
##
## [[1]]$marks
## [1] 14.3 12.0 15.0 19.0
##
##
## [[2]]
## [[2]]$id
## [1] 34454
##
## [[2]]$name
## [1] "Karen"
```

Subsetting a List

The operator `[` is used to extracts a sub-list of the original list.

```
lists[1]
```

```
## [[1]]
## [[1]][[1]]
## [1] 34453
##
## [[1]][[2]]
## [1] "John"
##
## [[1]]$marks
## [1] 14.3 12.0 15.0 19.0
```

```
str(lists[1])
```

```
## List of 1
##  $ :List of 3
##   ..$      : num 34453
##   ..$      : chr "John"
##   ..$ marks: num [1:4] 14.3 12 15 19
```

On the contrary, the operator `[[` and `$` (by name only) extract the value of a sub-list, which is not a list anymore

```
lists[[1]]
```

```
## [[1]]
## [1] 34453
##
## [[2]]
## [1] "John"
##
## $marks
## [1] 14.3 12.0 15.0 19.0
```

```
str(lists[[1]])
```

```
## List of 3
##  $      : num 34453
##  $      : chr "John"
##  $ marks: num [1:4] 14.3 12 15 19
```

```
lists[[1]][3]
```

```
## $marks
## [1] 14.3 12.0 15.0 19.0
```

```
typeof(lists[[1]][3])
```

```
## [1] "list"
```

```
lists[[1]][[3]]
```

```
## [1] 14.3 12.0 15.0 19.0
```

```
lists[[1]]$marks
```

```
## [1] 14.3 12.0 15.0 19.0
```

```
typeof(lists[[1]]$marks)
```

```
## [1] "double"
```

Attributes of a List

The names of the components are a list attribute.

```
attributes(mylst)
```

```
## $names
## [1] ""      ""      "marks"
```

```
# Attributes of an object can be thought of as a named list used to attach
# metadata to this object.
str(attributes(mylst))
```

```
## List of 1
##  $ names: chr [1:3] "" "" "marks"
```

```
# names() gives the character vector of component names
names(mylst)
```

```
## [1] ""      ""      "marks"
```

```
names(mylst)[1:2] <- c("id", "name")
str(mylst)
```

```
## List of 3
##  $ id   : num 34453
##  $ name : chr "John"
##  $ marks: num [1:4] 14.3 12 15 19
```

List Flattening

A list can be flattened using the `unlist()` function.

It creates a vector with as many elements as there are values in a list.

- By default this will coerce different types to a common type.
- Each element of this vector will have a name generated from the name of the list component

```
unlist(mylst)
```

```
##      id     name   marks1   marks2   marks3   marks4
## "34453"   "John"   "14.3"     "12"     "15"     "19"
```

Matrices

Data elements can be stored in multi-dimensional data structures.

A special case of the multi-dimensional data structure is the matrix, which has 2 dimensions.

Matrices are created with `matrix()`:

```
# Two scalar arguments to specify rows and columns
matrix(1:6, nrow = 2, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
# Matrices can also by filled by rows by setting byrow argument to TRUE
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Dimensions of Matrices

Adding a `dim` attribute to a vector by using the assignment form of `dim()` allows it to behave like a 2-dimensional matrix or multi-dimensional array (later):

```
m <- 1:6
dim(m) <- c(3, 2)
m
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
attributes(m)
```

```
## $dim
## [1] 3 2
```

```
dim(m)
```

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

```
str(m)
```

```
##  int [1:3, 1:2] 1 2 3 4 5 6
```

Length and Names of Matrices

`length()` generalizes to `nrow()` and `ncol()` for matrices:

```
length(m)
```

```
## [1] 6
```

```
nrow(m)
```

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

```
ncol(m)
```

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

`names()` generalizes to `rownames()` and `colnames()`:

```
rownames(m) <- c("a", "b", "c")
colnames(m) <- c("A", "B")
m
```

```
##   A B
## a 1 4
## b 2 5
## c 3 6
```

```r
attributes(m)
```

```
## $dim
## [1] 3 2
##
## $dimnames
## $dimnames[[1]]
## [1] "a" "b" "c"
##
## $dimnames[[2]]
## [1] "A" "B"
```

```r
typeof(m)
```

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

```r
class(m)
```

```
## [1] "matrix"
```

Extracting Subsets

The elements of a matrix can be accessed through a similar indexing schema as in vectors, but this time with two indices (the dimensions of a matrix)

```
# Omitting any dimension returns full columns or rows
m[1, ]
```

```
## A B
## 1 4
```

```
# Extract elements by indicating both row and column indices
m[1, 2]
```

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

```
m[-c(1, 3), 2]
```

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

```
m["a", 2]
```

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

By default, when a single element or a single column/row of a matrix is retrieved, it is returned as a vector rather than a matrix. This behavior can be turned off by setting `drop = FALSE`.

```
m[c(1, 3), 2]
```

```
## a c
## 4 6
```

```
m[c(1, 3), 2, drop = FALSE]
```

```
##   B
## a 4
## c 6
```

Combining Matrices

`c()` generalizes to `cbind()` and `rbind()` for matrices. `cbind()` and `rbind()` are concatenation functions that respect `dim` attributes.

```
x <- 1:3
y <- 10:12
z <- 95:97
cbind(x, y, z)
```

```
##      x  y  z
## [1,] 1 10 95
## [2,] 2 11 96
## [3,] 3 12 97
```

```
cbind(cbind(x, y), z)
```

```
##      x  y  z
## [1,] 1 10 95
## [2,] 2 11 96
## [3,] 3 12 97
```

```
rbind(x, y, z)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
## z   95   96   97
```

List-Matrices

The dimension attribute can also be set on lists to make list-matrices:

```
l <- list(2:5, "a", TRUE, 1)
dim(l) <- c(2, 2)
l
```

```
##      [,1]      [,2]
## [1,] Integer,4 TRUE
## [2,] "a"       1
```

```
str(l)
```

```
## List of 4
##  $ : int [1:4] 2 3 4 5
##  $ : chr "a"
##  $ : logi TRUE
##  $ : num 1
##  - attr(*, "dim")= int [1:2] 2 2
```

```
class(l)
```

```
## [1] "matrix"
```

```
typeof(l)
```

```
## [1] "list"
```

Arrays

Arrays are extensions of matrices to more than 2 dimensions; used to represent multi-dimensional data.

Arrays are created with `array()` or by using the assignment form of `dim()`:

```
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))

# We can also modify an object in place by setting dim()
c <- 1:12
dim(c) <- c(3, 2, 2)
c
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
```

The assignment form of `dimnames()` is used to specify a list of character vectors as the names of entries in different dimensions.

```
dimnames(c) <- list(c("a", "b", "c"), c("one", "two"), c("A", "B"))
c
```

```
## , , A
##
##   one two
## a   1   4
## b   2   5
## c   3   6
##
## , , B
##
##   one two
## a   7  10
## b   8  11
## c   9  12
```

Array Flattening

```
as.vector(c)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

The `c()` function clears `dim` and `dimnames` attributes of arrays.

```
c(c)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
m
```

```
##   A B
## a 1 4
## b 2 5
## c 3 6
```

```
c(m)
```

```
## [1] 1 2 3 4 5 6
```

Data Frames

A data frame is the most common way of storing data in R.

A data frame is a list of equal-length vectors and has a 2-dimensional structure. So it shares properties of both the matrix and the list.

We can create a data frame using `data.frame()`, which takes named vectors as input:

```
default.data <- data.frame(income = c(1.5, 2.2, 2), gender = c("M", "F", "F"),
    default = c(T, T, F))
default.data
```

```
##   income gender default
## 1    1.5      M    TRUE
## 2    2.2      F    TRUE
## 3    2.0      F   FALSE
```

```
nrow(default.data)
```

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

```
length(default.data)
```

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

```
colnames(default.data)
```

```
## [1] "income"  "gender"  "default"
```

```
str(default.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ income : num  1.5 2.2 2
##  $ gender : Factor w/ 2 levels "F","M": 2 1 1
##  $ default: logi  TRUE TRUE FALSE
```

By default, `data.frame()` turns strings into factors. Use `stringsAsFactors = FALSE` to suppress this behavior:

```
default.data <- data.frame(income = c(1.5, 2.2, 2), gender = c("M", "F", "F"),
    default = c(T, T, F), stringsAsFactors = FALSE)
str(default.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ income : num  1.5 2.2 2
##  $ gender : chr  "M" "F" "F"
##  $ default: logi  TRUE TRUE FALSE
```

Attributes of a Data Frame

```
typeof(default.data)
```

```
## [1] "list"
```

```
attributes(default.data)
```

```
## $names
## [1] "income"  "gender"  "default"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "data.frame"
```

```
class(default.data)
```

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

```
row.names(default.data)
```

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

```
row.names(default.data) <- c("R1", "R2", "R3")
default.data
```

```
##    income gender default
## R1    1.5      M    TRUE
## R2    2.2      F    TRUE
## R3    2.0      F   FALSE
```

Combining Data Frames

We can combine data frames using `cbind()` and `rbind()`:

```
cbind(default.data, data.frame(balance = c(200, 150, 94)))
```

```
##    income gender default balance
## R1    1.5      M    TRUE     200
## R2    2.2      F    TRUE     150
## R3    2.0      F   FALSE      94
```

```
rbind(default.data, data.frame(income = 1.3, gender = "M", default = T))
```

```
##    income gender default
## R1    1.5      M    TRUE
## R2    2.2      F    TRUE
## R3    2.0      F   FALSE
## 1     1.3      M    TRUE
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number and names of columns must match.

Adding Columns to a Data Frame

Because data frames are speical lists, we can add new columns to a data frame in the same way we did with lists, with the restriction that new columns must have the same number of rows as the existing data frame:

```
default.data$education <- c("h", "l", "m")
default.data
```

```
##    income gender default education
## R1    1.5      M    TRUE         h
## R2    2.2      F    TRUE         l
## R3    2.0      F   FALSE         m
```

Subsetting and Querying

A column of a data frame can be extracted as a vector as in a list:

```
default.data$default
```

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

```
default.data[["default"]]  # alternatively, default.data[[3]]
```

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

We can perform some simple querying in a data frame by taking advantage of the indexing possibilities:

```
default.data[default.data$default == TRUE, ]
```

```
##    income gender default education
## R1    1.5      M    TRUE         h
## R2    2.2      F    TRUE         l
```

We can simplify the typing of these queries by using the function `attach()`; and we can use the function `detach()` to disables this facility.

```
attach(default.data)
default.data[default == TRUE & income >= 2, 1]
```

```
## [1] 2.2
```

Sorting a Data Frame by Selected Columns

Often data are better viewed when sorted. The `order()` function sorts a column and gives output that can sort the rows of a data frame.

```
order(default.data[, "income"])
```

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

```
(default.data.by.Age <- default.data[order(default.data[, "income"]), ])
```

```
##    income gender default education
## R1    1.5      M    TRUE         h
## R3    2.0      F   FALSE         m
## R2    2.2      F    TRUE         l
```

Summary of Data Structures in R

R's base data structures can be organized by their dimensionality and whether they're homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types):

	Homogeneous	Heterogeneous
1d	Vector	List
2d	Matrix	Data frame
nd	Array	

Functions

A function is a set of R statements that:

- takes input
- does something with that input
- and returns the result

While data structures tie related values into one object, functions tie related commands into one object.

In both cases: easier to understand, easier to work with, easier to build into larger things.

Defining a Function

Functions can be created using the `function` directive and are stored as objects of type **closure**.

Once a function is defined, it can be bound to a name with `<-` (the binding step is not compulsory; Anonymous functions).

```
fn <- function(x1, x2, ... ) {
                y1 <- code about x1, x2, ...
                y2 <- code about x1, x2, ...
                etc.
                output
}
```

- Here, `x1`, `x2`, etc. are the objects the function takes as input.
- The `output` line (without `<-`) contains the R object we want the function to return; Or use `return()` to explicitly specify the output to be returned.

Function Parameters

The **parameters** of a function are a special kind of variables that are used to refer to data provided as input to the function.

When a function is defined, an ordered list of parameters are specified within parentheses and separated by comma.

The `formals()` function or `str()` can be used to return a list of all the parameters of a function:

```
formals(sd)
```

```
## $x
##
##
## $na.rm
## [1] FALSE
```

```
str(sd)
```

```
## function (x, na.rm = FALSE)
```

Parameters potentially have **default values**.

Calling Functions

Arguments are the actual input (values, variables or expressions thereof) passed/supplied to a function when a function is called/invoked.

In a function call/invocation, the arguments for that call are evaluated, and the resulting values are assigned/bound to the corresponding parameters.

```r
f <- function(a, b = 1, c = NULL) {
    if (is.null(c))
        a^2 + b
}

# Calling the function in an interactive context causes the result to be
# automatically printed.  Function arguments can be missing for parameters
# having default values.
f(2)
```

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

```r
f <- function(a, b) {
    a^2
}
# Arguments for a call are evaluated lazily; that is, they are evaluated
# only as needed.
f(2)
```

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

Mapping Arguments to Parameters

When calling a function, arguments can be specified by position or by name.

```
mydata <- rnorm(100)
sd(mydata)
```

```
## [1] 1.076858
```

```
sd(x = mydata)
```

```
## [1] 1.076858
```

```
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 1.076858
```

```
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 1.076858
```

```
# matching by partial name
sd(na = FALSE, x = mydata)
```

```
## [1] 1.076858
```

When positional matching is mixed with matching by name, arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

More on Default Parameter Values

In addition to just constant values, the default value can be defined in terms of other parameters:

```r
normalize <- function(x, m = mean(x), s = sd(x)) {
    (x - m)/s
}
z <- c(1, 3, 6, 10, NA)
normalize(z)
```

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

```r
normalize <- function(x, m = mean(x, na.rm = y), s = sd(x, na.rm = y), y = FALSE) {
    (x - m)/s
}
normalize(z, y = TRUE)
```

```
## [1] -1.0215078 -0.5107539  0.2553770  1.2768848         NA
```

```r
# This works, but the syntax is a little clunky.
```

The `...` Construct

`...` (called an ellipsis or three dots) captures any number of arguments that aren't otherwise matched, and can be easily passed on to other functions.

There are essentially two situations when we want to use `...`:

- When the number of arguments passed to the function can't be known in advance:

```
str(c)
```

```
## function (...)
```

```
str(paste)
```

```
## function (..., sep = " ", collapse = NULL)
```

```
# Any arguments that appear after ... on the argument list must be fully
# named.
paste("a", "b", sep = ":")
```

```
## [1] "a:b"
```

- When other functions are called within a function and these functions can have a variable number of arguments:

```
normalize <- function(x, m = mean(x, ...), s = sd(x, ...), ...) {
    (x - m)/s
}
normalize(z, na.rm = TRUE)
```

```
## [1] -1.0215078 -0.5107539  0.2553770  1.2768848         NA
```

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
```

```
## [1] "a, b, c, d, e, f, g, h, i, j"
```

Explicit `return` Statements

The value returned by the function is usually the last statement it evaluates. But we can choose to return early by using `return()`:

```r
f <- function(x) {
    if (!x) {
        return(something_short)
    }
    # Do something that takes many lines to express
}
```

Everything that Happens is a Function Call

This includes

- infix operators like `+`:

```
x <- 8:10; y <- 3:5
x + y
```

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

```
# ` allows us to refer to functions or variables that have otherwise reserved or illegal names
`+`(x, y)
```

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

- flow control operators like `for`, `if`, and `while`:

```
for (i in 1:2) print(i)
```

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

```
`for`(i, 1:2, print(i))
```

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

```
if (i == 1) print("yes!") else print("no.")
```

```
## [1] "no."
```

```
`if`(i == 1, print("yes!"), print("no."))
```

```
## [1] "no."
```

- subsetting operators like `[` and `$`:

```
x[3]
```

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

```
`[`(x,3)
```

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

- and even the curly brace `{`.

```
{ print(1); print(2); print(3) }
```

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

```
`{`(print(1), print(2), print(3))
```

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

Organizing R Code

Another means to create organization is putting the code for support functions into their own script files and loading them when needed using the `source` command:

In the `functions.R` file:

```r
add <- function(a, b) {
    a + b
}
```

In the `script.R` file:

```r
source("functions.R")
add(2, 3)
```

How to Bind a Value to a Name?

```r
a <- 1
b <- 2
f <- function(x) {
    a * x + b
}

g <- function(x) {
    a <- 2
    b <- 1
    f(x)
}
```

What is the return value of calling `g(2)`?

Symbol-Value Bindings

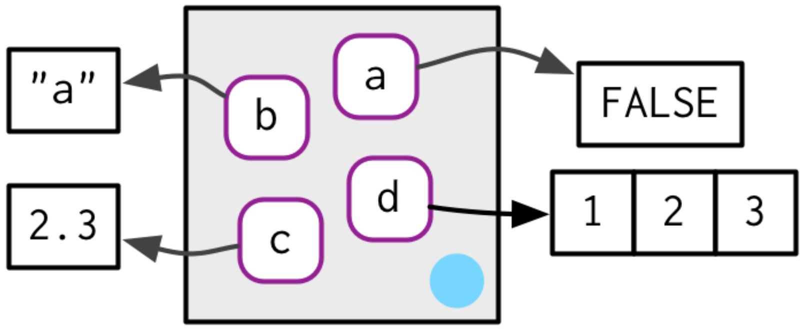
- The symbols which occur in a function can be divided into three classes:
  - **Formal parameters**: those occurring in the parameter list of the function.Their values are determined by the process of binding the actual arguments to the formal parameters.
  - **Local variables**: those whose values are determined by the evaluation of expressions in the body of the functions.
  - **Free variables**: not formal parameters or local variables
- How does R determine values for free variables?

The **scoping rules** of a language determine how a value is associated with a free variable in a function.

Environments: A Data Structure Powering Scoping

The job of an environment is to bind or associate a set of names to a set of values.

We can think of an environment as a bag of names each pointing to an object stored somewhere in memory.



Operations on Environments

Environments can be created with the `new.env()` function.

```
(e1 <- new.env())
```

```
## <environment: 0x7f85520aa0b0>
```

```
# Printing an evironment just displays its memory address.
```

We can get and set elements of an environment with `$` and `[[` in the same way as with a list or using `get()` and `assign()`:

```
e1[["a"]] <- FALSE
e1$b <- "a"
e1[["c"]] <- 2.3
e1$d <- c(1, 2, 3)
e1[["d"]]
```

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

```
assign("f", c("business programming", "business analytics"), envir = e1)
get("f", envir = e1)
```

```
## [1] "business programming" "business analytics"
```

Most of the syntax for lists also works for environments.

We can coerce a list to be an environment and vice versa:

```
(a_list <- as.list(e1))
```

```
## $a
## [1] FALSE
##
## $b
## [1] "a"
##
## $c
## [1] 2.3
##
## $d
## [1] 1 2 3
##
## $f
## [1] "business programming" "business analytics"
```

```
# ...and back again.
as.environment(a_list)
```

```
## <environment: 0x7f855078d978>
```

The `ls()` and `ls.str()` functions take an environment argument (defaults to the current environment), allowing us to list its content.

```
ls(envir = e1)
```

```
## [1] "a" "b" "c" "d" "f"
```

```
ls.str(envir = e1)
```

```
## a :  logi FALSE
## b :  chr "a"
## c :  num 2.3
## d :  num [1:3] 1 2 3
## f :  chr [1:2] "business programming" "business analytics"
```

We can test to see if a variable exists in an environment using the `exists()` function:

```
exists("a", e1)
```

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

```
exists("non-existing", e1)
```

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

`rm()` can be used to remove objects in an environment.

```
rm(a, b, envir = e1)
rm(list = c("c", "d"), envir = e1)
as.list(e1)
```

```
## $f
## [1] "business programming" "business analytics"
```

The following code can be used to delete all objects in the workspace:

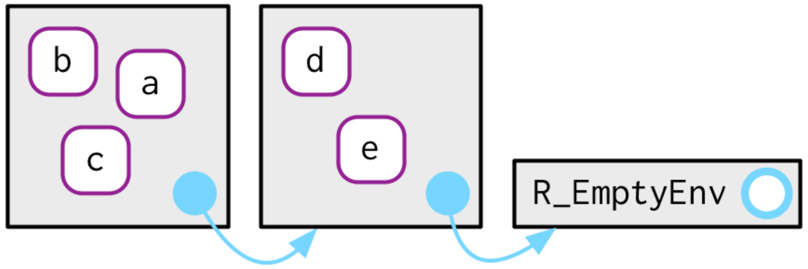
```
rm(list = ls())
```

Parents

Each environment has a parent except the **empty environment**.

Environments are chained/nested. And the ancestors of an environment include all parent environments up to the **empty environment**.

```
(e2a <- new.env(parent = emptyenv()))
## <environment: 0x7f85520a8bf8>
(e2b <- new.env(parent = e2a))
## <environment: 0x7f8552156580>
parent.env(e2b)
## <environment: 0x7f85520a8bf8>
parent.env(e2a)
## <environment: R_EmptyEnv>
```



Special Environments

- The **current environment**, accessed with `environment()` (with no arguments provided), is the environment in which code is currently executing.
- The **global environment**, accessed with `globalenv()` and printed as `R_GlobalEnv` or `.GlobalEnv`, is also called the **"workspace"**. It's where all interactive (i.e. outside of a function) computation takes place.
- The **empty environment**, accessed with `emptyenv()` and printed as `R_EmptyEnv`, is the ultimate ancestor of all environments and the only environment without a parent.
- The **base environment**, accessed with`baseenv()`, is the environment of the base package. Its parent is the empty environment.

Function Environments

Most environments are not created with `new.env()` but are created as a consequence of using functions.

There're 4 types of environments associated with a function:

- Binding a function to a name with `<-` defines a **binding environment**. It determines how we find the function.
- The **enclosing environment** is the environment where the function is defined. It determines how the function finds values as we will see.

```
environment()
```

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

```
e1$g <- function(x) x + 10
environment(e1$g)  # The enclosing environment is the current environment
```

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

```
exists("g", envir = environment())  # But the current environment doesn't contain g.
```

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

- Calling a function creates an **execution environment** that holds the the execution and stores variables created during execution.

```
simple_function <- function() {
    print(environment())  # Returns the current environment without arguments being specified
}
simple_function()
```

```
## <environment: 0x7f855070ab90>
```

```
environment(simple_function)  # Returns the enclosing environment of the function when it takes a function
```

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

- Every execution environment is associated with a **calling environment** from which is the function was called.

```
# define g inside f
y <- 3
f <- function(x) {
    y <- 1
    g <- function(x) {
        (x + y)/2
    }
    g(x)
}
```

```
# define g outside f
y <- 3
f <- function(x) {
    y <- 1
    g(x)
}
g <- function(x) {
    (x + y)/2
}
```

Lexical (Static) Scoping

How to resolve the free viariable bindings (i.e., find a value for a symbol) in R?

- R walks up the **chain of environments** and uses the first binding for a symbol it finds.

Then the question goes to how R determines the order to search the environments for bindings.

- The parent of the **execution environment** is the **enclosing environment** (as opposed to **calling environment**) of the function.
- The values of free variables encountered during execution are searched for in the environment in which the function is defined.
- The scoping rule is called **lexical scoping** or **static scoping**.
    - It's the code's static structure, rather than the dynamic relationship manifested through the code execution, determines where to search for bindings.

"First-Class Functions"

In R, functions are objects in their own right.

By **"first-class"**, we mean that functions can be computed, passed, stored, etc. wherever other objects can be computed, passed, stored, etc.

For example, a function is able to return another function as its value.

```
a <- 1
b <- 2
f <- function(a, b) {
    function(x) a * x + b  # An anonymous function
}
g <- f(2, 1)
```

What is the object referenced by `g`? What returns by calling `g(2)`?

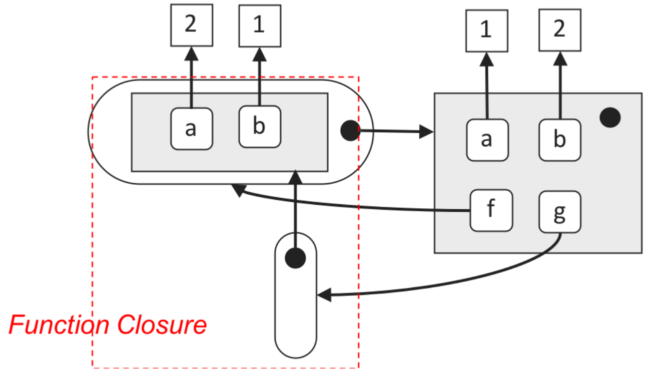
Function Closures

A **function closure** or **closure** is a function together with an environment that provides value bindings for free varialbes.

A function call uses both parts of the closure because the code of the function is evaluated using variables encapsulated in the environment part.

Almost all functions in R are closures as they remember the environment where they are created.

```r
a <- 1
b <- 2
f <- function(a, b) {
    function(x) {
        a * x + b
    }
}
g <- f(2, 1)
```



When creating a function inside another function, the enclosing environment of the child function is the execution environment of the parent, which is no longer ephemeral.

Example: Generating a Family of Power Functions

```r
power <- function(exponent) {
    print(environment())
    function(x) x^exponent
}
```

Use `power()` to make two new functions:

```r
square <- power(2)
```

```
## <environment: 0x7f85520b5f08>
```

```r
square
```

```
## function(x) x^exponent
## <environment: 0x7f85520b5f08>
```

```r
cube <- power(3)
```

```
## <environment: 0x7f8550aa07f0>
```

```r
cube
```

```
## function(x) x^exponent
## <bytecode: 0x7f855097c808>
## <environment: 0x7f8550aa07f0>
```

The two closures differ in their enclosing environments, which bind distinct values to the symbol `exponent`.

We can convert an environment to a list to see its content:

```
as.list(environment(square))
```

```
## $exponent
## [1] 2
```

```
as.list(environment(cube))
```

```
## $exponent
## [1] 3
```

A function that makes new functions is called a **function factory**.

Super/Deep Assignment: `<<-`

The change made by the regular assignment, `<-`, within the function are local and temporary.

```r
a <- 1
f <- function(x) {
    a <- a + x
    a
}
f(1)
```

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

What change has been made to `a`?

```r
a
```

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

Instead, the super/deep assignment arrow, `<<-`, modifies an existing variable found by walking up the chains of environments.

```
a <- 1
g <- function(x) {
    a <<- a + x
    a
}
g(1)
```

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

```
a
```

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

If `<<-` doesn't find an existing variable, it creates one in the global environment.

Package Environments

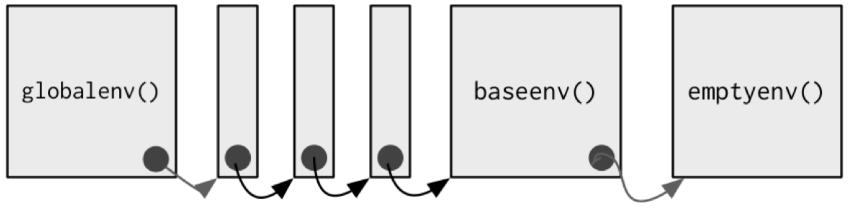
In R, the use of packages gives another way to group R code and create organization on it.

```
search()  # It gives the packages that are currently loaded
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Each entry in the list represents a package environment that contains every publicly accessible function defined in the corresponding package.

They make up the search path (starting with the global environment) that R walks up to find the first binding for a symbol.



The search path

We can access any environment on the search list using `as.environment()`:

```
as.environment("package:stats")
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "/Library/Frameworks/R.framework/Versions/3.4/Resources/library/stats"
```

The following code prints all publicly accessible functions in the `package:stats` environment:

```
ls(envir = as.environment("package:stats"))
```

Each time a new package is loaded with `library()`, it is inserted between the **global environment** and the package that was previously at the top of the search path.

```
library("tidyr")
search()[1:3]
```

```
## [1] ".GlobalEnv"    "package:tidyr" "package:stats"
```

```
z <- list(mean = "fluffernutter", median = 3.4)
attach(z)
mean
```

```
## [1] "fluffernutter"
```

```
search()[1:4]
```

```
## [1] ".GlobalEnv"    "z"             "package:tidyr" "package:stats"
```

`attach()` creates an environment, slots it into the list right after the global environment and populates it with the objects we're attaching.

```
mean <- function(x) x + 1
# The mean() function in the package:base environment is masked by the
# user-defined function
mean(1:6)
```

```
## [1] 2 3 4 5 6 7
```

```
# The :: operator lets us specify which mean() function we want
base::mean(1:6)
```

```
## [1] 3.5
```

Overview of Control Structures

- Control structures in R allow us to control the flow of execution of a series of R expressions, depending on runtime conditions. Common structures include
  - `if-else`: test a condition
  - `for`: execute a loop a fixed number of times
  - `while`: execute a loop while a condition is true
  - `repeat`: execute an infinite loop
  - `break`: break the execution of a loop
  - `next`: skip an iteration of a loop
- Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

Conditionals: `if` and `else`

```
if (<condition_1>) {
            # do something
} else if (<condition_2>) {
            # do something different
} else {
            # do something different
}
```

- Conditions need to give **one** `TRUE` or `FALSE` value
- The `else if` clause and `else` clause are not necessary.
- Can be written in a variety of forms, e.g., one-line actions don't need braces

```
if (x > 3) y <- 10 else y <- 0
```

```
y <- if (x > 3) 10 else 0
```

- Can nest arbitrarily deeply.

Combining Logicals

- Conditional execution requires a **single** logical value.
- Unlike `&` and `|` that combine logical values element-wise, `&&` and `||` give one logical value, **lazily**:

```
(0 > 0) && (z <- (9 > 4))
```

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

```
exists("z")
```

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

```
(1 > 0) && (z <- (9 > 4))
```

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

```
exists("z")
```

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

Simplify `if-else` with Multiple Selection: `switch()`

```
cars$speed
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
## [24] 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24
## [47] 24 24 24 25
```

```
if (type.of.summary == "mean") {
    mean(cars$speed)
} else if (type.of.summary == "median") {
    median(cars$speed)
} else if (type.of.summary == "histogram") {
    hist(cars$speed)
} else "I don't understand"
```

```
# A variable is given to select on; then a value is assigned to each op
switch(type.of.summary, mean = mean(cars$speed), median = median(cars$s
    histogram = hist(cars$speed), "I don't understand")
```

Loops: `for`

A `for` loop takes an iterator variable (a counter) and assign it successive values from a sequence or a vector; most commonly used for iterating over the elements of an object (a list, a vector, etc.)

```r
x <- c("a", "b", "c", "d")
for (i in 1:4) print(x[i])
for (i in seq_along(x)) print(x[i])
for (letter in x) print(letter)
```

- `for` is used when the number of times to repeat (values to iterate over) is clear in advance.
- `for` loops can be nested arbitrarily deeply.

Loops: `while`

- A `while` loop begins by testing a condition. If it's `TRUE`, then it executes the loop body. Once the loop body is executed, the condition is tested again, and so on.
- Conditions used by `while` must be a **single** logical value (like `if`)

```r
count <- 0
while (count < 5) {
    print(count)
    count <- count + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

- `while` is used when we can recognize when to stop once we're there, even if we can't guess it to begin with.

Unconditional Loops: `repeat`

- `repeat` initiates an infinite loop. It will execute until we press `Escape` or quit R, whichever happens soonest.
- The infinite loop can be broken out of by introducing a `break` statement via a conditional.

```
x0 <- 1
tol <- 1e-08
repeat {
    x1 <- computeEstimate()
    if (abs(x1 - x0) < tol)
        break else x0 <- x1
}
```

- A `next` statement skips the rest of the current iteration and starts the next iteration.

```
for (i in 1:100) {
    if (i <= 20)
        next  # Skip the first 20 iterations
    # Do something here
}
```

Vectorized Operations

Consider implementing a vector addition $a + b$ using loops:

```
c <- vector("numeric", length(a))
for (i in seq_along(a)) c[i] <- a[i] + b[i]
```

How R adds 2 vectors and does matrix multiplication:

```
a + b
A %*% B
```

Vectorization is the operation of converting repeated operations on simple values ("scalars") into a single operation on whole objects (vectors, matrices, etc.).

It eliminates many loops.

Vectorized Conditionals: `ifelse()`

The 1st argument takes a logical vector.

Depending on `TRUE` or `FALSE`, it picks elements for the return value from either the 2nd or 3rd arguments:

```
(x <- runif(6, 0, 2))
```

```
## [1] 1.4612081 1.2541968 1.4633669 0.7601322 1.8024570 1.6452845
```

```
ifelse(x^2 > 1, 2 * abs(x) - 1, x^2)
```

```
## [1] 1.9224161 1.5083936 1.9267338 0.5778009 2.6049140 2.2905691
```

```
# it can also accept vectors in the second and third arguments.  The
# recycling rule applies when the vectors aren't the same size.
ifelse(x^2 > 1, 1:3, -1:-6)
```

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

Vectorizing Functions Using `lapply()`

```
df <- data.frame(a = rnorm(10), b = runif(10, 0, 20), c = rnorm(10, 4, 8))
```

`lapply()` (short for "list apply") takes a function, applies it to each element in a vector or a list (can also accept vector inputs), and returns the results in the form of a list.

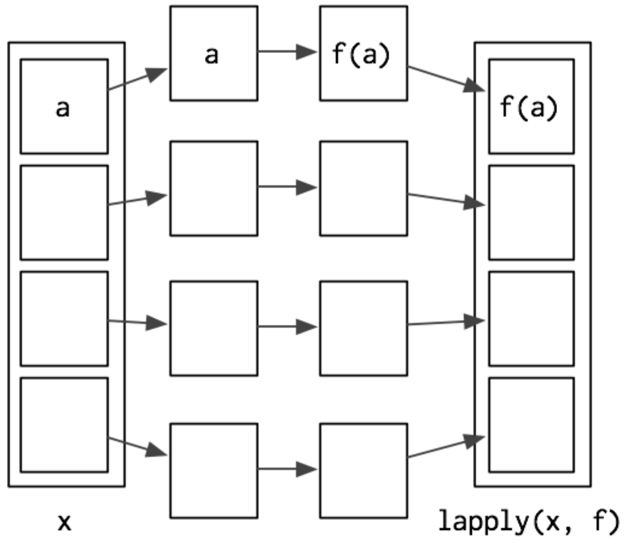
```
str(lapply)

## function (X, FUN, ...)

lapply(df, mean)

## $a
## [1] 0.248928
##
## $b
## [1] 9.789091
##
## $c
## [1] 4.300721

# The names of the original list is preserved in the output.
```



Specifying Other Arguments for the Function to be Applied

The elements of the 1st argument to `lapply()` are always supplied as the 1st argument of the function we are applying.

```
x <- 1:3
str(runif)
```

```
## function (n, min = 0, max = 1)
```

```
lapply(x, runif)
```

```
## [[1]]
## [1] 0.009455037
##
## [[2]]
## [1] 0.6893039 0.5705788
##
## [[3]]
## [1] 0.5838207 0.4276002 0.3029788
```

The `...` construct takes the remaining arguments and passes them down to the function being applied to the elements of the list.

```
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 1.73348
##
## [[2]]
## [1] 7.094676 6.786334
##
## [[3]]
## [1] 0.7943403 0.2046941 1.9012485
```

What if the elements of the list is passed down to the function as an argument other than the 1st one?

Use of Anonymouse Functions

If we want to vary a different argument, we can use an **anonymous function**.

```
str(rep.int)
```

```
## function (x, times)
```

```
lapply(c(2, 3, 6), function(x) rep.int(3, times = x))
```

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

`lapply()` and other members in the `apply` family of functions make heavy use of anonymous functions.

Applying Related Functions

Because functions are treated as **"first-class citizens"** in R, `lapply()` can work with groups of related functions as with lists of any other type of objects.

```
compute_mean <- list(base = function(x) mean(x),
                     sum = function(x) sum(x) / length(x),
                     manual = function(x) {
                        total <- 0
                        n <- length(x)
                        for (i in seq_along(x)) total <- total + x[i] / n
                        total
                        }
                     )

x <- runif(100)

unlist(lapply(compute_mean, function(f) f(x)))
```

```
##      base       sum    manual
## 0.5034359 0.5034359 0.5034359
```

Other Variations of `lapply()`

Other variations of `lapply()` simply use different types of input or output:

	Descriptions
sapply()	produces a result of the simplest type possible, such as vectors, matrices, and lists, instead of lists only.
mapply()	applies a function in parallel over a set of arguments.
apply()	evaluates a function over the margins of an array.
tapply()	groups the elements of a vector and applies a function over the resulted subsets.

Vector Output: `sapply()`

`sapply()` (short for "simplified [l]apply") behaves similarly to `lapply()` except that it tries to simplify the results if possible.

```
lapply(df, mean)
```

```
## $a
## [1] 0.248928
##
## $b
## [1] 9.789091
##
## $c
## [1] 4.300721
```

```
sapply(df, mean)
```

```
##        a        b        c
## 0.248928 9.789091 4.300721
```

Essentially, `sapply()` calls `lapply()` on its input and then applies the following simplifying algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

```
sapply
```

```
## function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
## {
##     FUN <- match.fun(FUN)
##     answer <- lapply(X = X, FUN = FUN, ...)
##     if (USE.NAMES && is.character(X) && is.null(names(answer)))
##         names(answer) <- X
##     if (!identical(simplify, FALSE) && length(answer))
##         simplify2array(answer, higher = (simplify == "array"))
##     else answer
## }
## <bytecode: 0x7f854ec76a40>
## <environment: namespace:base>
```

Multiple Inputs: `mapply()`

`mapply()` (short for "multivariate apply") is a multivariate version of `sapply()` which applies a function in parallel over a set of arguments.

```
str(mapply)
```

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

```
str(rnorm)
```

```
## function (n, mean = 0, sd = 1)
```

```
mapply(rnorm, 1:4, 1:4)
```

```
## [[1]]
## [1] 2.780725
##
## [[2]]
## [1] 2.919915 1.027530
##
## [[3]]
## [1] 4.243916 1.510048 4.000466
##
## [[4]]
## [1] 2.685595 3.688621 2.967808 4.878526
```

```
# The 3rd argument is re-cycled to the length of the longest
mapply(rnorm, 1:4, 1:4, 2)
```

```
## [[1]]
## [1] 3.525168
##
## [[2]]
## [1]  3.0320692 -0.9520316
##
## [[3]]
## [1] 4.419708 4.577541 5.031956
##
## [[4]]
## [1] 2.784419 5.447405 4.971411 2.063029
```

Vectorizing a Function

In many statistical applications, we want to calculate the **sum of squares** $\Sigma_{i=1}^{n} \frac{(x_i - \mu)^2}{\sigma^2}$. Implement it with an R function:

```r
sumsq <- function(mu, sigma, x) sum(((x - mu)/sigma)^2)
```

Generate some data to investigate its effect:

```r
x <- rnorm(100)
sumsq(1, 1, x)
```

```
## [1] 198.6083
```

Suppose we want to evaluate or plot it for 10 different choices of `mu` or `sigma`?

```r
sumsq(1:10, 1:10, x)  # Does it print 10 values?
```

When **vectorizing** a function, `mapply()` allows multiple arguments to the function to vary.

How about we changing the code to the following:

```
mapply(sumsq, 1:10, 1:10, x)  # Does it print 10 values?
```

`Vectorize()`

The `MoreArgs` argument of `mapply()` takes a list of arguments that will be supplied as constant inputs to each call.

```
mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

```
##  [1] 198.6083 123.9161 110.3023 105.6110 103.4733 102.3302 101.6519
##  [8] 101.2188 100.9266 100.7211
```

`Vectorize()` is a function wrapper for `mapply()` and provides more convenient interface for use.

```
vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
vsumsq(1:10, 1:10, x)
```

```
##  [1] 198.6083 123.9161 110.3023 105.6110 103.4733 102.3302 101.6519
##  [8] 101.2188 100.9266 100.7211
```

Array Input: `apply()`

```
(m <- matrix(rnorm(12), 3, 4))
```

```
##             [,1]        [,2]       [,3]       [,4]
## [1,] -0.08498165  1.32318690 -1.4180000 -0.4359164
## [2,]  1.20083308 -1.10762489 -0.1381998 -0.3375154
## [3,]  1.67867619 -0.08624604  0.8048340  0.5933868
```

```
sapply(m, mean)
```

```
##  [1] -0.08498165  1.20083308  1.67867619  1.32318690 -1.10762489
##  [6] -0.08624604 -1.41799998 -0.13819981  0.80483403 -0.43591644
## [11] -0.33751539  0.59338679
```

`lapply()` and `sapply()` treat the matrices and arrays as though they were vectors, whereas `apply()` (short for "array apply") evaluates a function over the **margins of an array**.

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

```
apply(m, 2, mean)  # column mean
```

```
## [1]  0.93150921  0.04310532 -0.25045525 -0.06001501
```

```
apply(m, 1, sum)  # row sum
```

```
## [1] -0.6157112 -0.3825070  2.9906510
```

The `MARGIN` argument essentially indicates which dimension of the array we want to preserve or retain.

Col/Row Sums and Means

Shortcut functions perform column/row sums and means. They are much faster and more descriptive:

`rowSums(m) <=> apply(m, 1, sum)`

`rowMeans(m) <=> apply(m, 1, mean)`

`colSums(m) <=> apply(m, 2, sum)`

`colMeans(m) <=> apply(m, 2, mean)`

```
apply(m, 2, mean)  ## column mean
```

```
## [1]  0.93150921  0.04310532 -0.25045525 -0.06001501
```

```
apply(m, 1, sum)  ## row sum
```

```
## [1] -0.6157112 -0.3825070  2.9906510
```

```
colMeans(m)  ## column mean
```

```
## [1]  0.93150921  0.04310532 -0.25045525 -0.06001501
```

```
rowSums(m)  ## row sum
```

```
## [1] -0.6157112 -0.3825070  2.9906510
```

Other Ways to Apply

- Pass the optional arguments to `FUN` via the `...` construct:

```
apply(m, 1, quantile, probs = 0.5)  # row median
```

```
## [1] -0.2604490 -0.2378576  0.6991104
```

- Preserve more than 1 dimension:

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a, c(1, 2), mean)
```

```
##             [,1]       [,2]
## [1,] -0.08009211  0.4794365
## [2,] -0.65264409 -0.3383999
```

A Common Problem

```r
(frogger_scores <- data.frame(player = rep(c("Nick", "Charles", "Samuel"), times = c(4,
    3, 5)), score = round(rlnorm(12, 8), -1)))
```

```
##     player score
## 1     Nick  4790
## 2     Nick  2750
## 3     Nick  6290
## 4     Nick  4910
## 5  Charles  1070
## 6  Charles  1980
## 7  Charles  2250
## 8   Samuel  5610
## 9   Samuel   820
## 10  Samuel  3720
## 11  Samuel  4680
## 12  Samuel  2470
```

How can we calculate some statistic on a variable (`score`) that has been split into groups (defined by `player`)?

Splitting a Data Frame: `split()`

`split()` takes a vector or a data frame and splits it into groups determined by a factor or a list of factors:

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

First, split the datas by `player`:

```
(scores_by_player <- split(frogger_scores$score, frogger_scores$player))
```

```
## $Charles
## [1] 1070 1980 2250
##
## $Nick
## [1] 4790 2750 6290 4910
##
## $Samuel
## [1] 5610  820 3720 4680 2470
```

The Apply and Combine Steps

Next, apply the (`mean()`) function to each element:

```
(list_of_means_by_player <- lapply(scores_by_player, mean))
```

```
## $Charles
## [1] 1766.667
##
## $Nick
## [1] 4685
##
## $Samuel
## [1] 3460
```

Finally, combine the result into a single vector:

```
(mean_by_player <- unlist(list_of_means_by_player))
```

```
##  Charles     Nick   Samuel
## 1766.667 4685.000 3460.000
```

The **apply** and **combine** steps can be condensed into one by using `sapply()`.

Alternatively:

```
(frogger_by_player <- split(frogger_scores, frogger_scores$player))
```

```
## $Charles
##    player score
## 5 Charles  1070
## 6 Charles  1980
## 7 Charles  2250
##
## $Nick
##   player score
## 1   Nick  4790
## 2   Nick  2750
## 3   Nick  6290
## 4   Nick  4910
##
## $Samuel
##    player score
## 8  Samuel  5610
## 9  Samuel   820
## 10 Samuel  3720
## 11 Samuel  4680
## 12 Samuel  2470
```

```
(sapply(frogger_by_player, function(x) mean(x$score)))
```

```
##  Charles     Nick   Samuel
## 1766.667 4685.000 3460.000
```

A More Complex Problem: Enron Emails

```
head(small_corpus <- read.csv("small_corpus.csv", colClasses = c("character",
    "factor", "factor", "integer")), n = 10)
```

```
##          time from  to n
## 1  1999-01-04  114  65 2
## 2  1999-01-04  114 169 2
## 3  1999-01-07  114 110 4
## 4  1999-01-07  114 112 4
## 5  1999-01-07  114 169 2
## 6  1999-01-08  114 145 2
## 7  1999-01-08  114 169 8
## 8  1999-01-12  114 169 4
## 9  1999-01-13  114  22 2
## 10 1999-01-13  114  29 2
```

```
length(levels(small_corpus$from))
```

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

```
length(levels(small_corpus$to))
```

```
## [1] 21
```

Suppose that we want to quantify the intensity of email exchange between a pair of employees during this period?

```
small_corpus_by_pair <- split(small_corpus$n, list(small_corpus$from, small_corpus$to))
str(small_corpus_by_pair, list.len = 15)
```

```
## List of 231
##  $ 107.107: int(0)
##  $ 112.107: int 2
##  $ 114.107: int [1:5] 2 2 4 12 4
##  $ 145.107: int(0)
##  $ 155.107: int(0)
##  $ 160.107: int(0)
##  $ 169.107: int(0)
##  $ 22.107 : int(0)
##  $ 38.107 : int(0)
##  $ 50.107 : int(0)
##  $ 65.107 : int(0)
##  $ 107.11 : int(0)
##  $ 112.11 : int(0)
##  $ 114.11 : int(0)
##  $ 145.11 : int(0)
##   [list output truncated]
```

- With multiple factors and many levels, creating an interaction can result in many levels that are empty.

- We can drop empty levels when calling the `split()` function:

```
small_corpus_by_pair <- split(small_corpus$n, list(small_corpus$from, small_corpus$to),
    drop = TRUE)
str(small_corpus_by_pair, list.len = 15)
```

```
## List of 41
##  $ 112.107: int 2
##  $ 114.107: int [1:5] 2 2 4 12 4
##  $ 38.11  : int [1:5] 2 2 4 2 2
##  $ 114.110: int [1:6] 4 2 2 2 4 2
##  $ 155.110: int [1:6] 4 2 2 4 2 2
##  $ 169.110: int [1:6] 2 2 4 2 2 2
##  $ 114.112: int [1:4] 4 2 2 4
##  $ 38.112 : int 2
##  $ 107.114: int [1:4] 2 4 6 2
##  $ 112.114: int 2
##  $ 155.114: int [1:6] 6 2 2 2 2 2
##  $ 169.114: int [1:15] 4 8 2 4 4 2 2 2 6 2 ...
##  $ 38.114 : int 2
##  $ 114.123: int 2
##  $ 145.128: int 2
##   [list output truncated]
```

The "Split-Apply-Combine" Paradigm

The basic principle is as follows:

- **Split** a data set into (manageable) piece (e.g., according to some factor)
- **Apply** a function to each piece (e.g., `mean()`)
- **Combine** all the pieces into a single output (e.g., an array)

However, **"split-apply-combine"** is such a common data analysis paradigm for which we need something easier.

Group Apply: `tapply()`

`tapply()` groups the elements of a vector and applies a function over the resulted subsets.

```
str(tapply)
```

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

```
tapply(frogger_scores$score, frogger_scores$player, mean)
```

```
##  Charles     Nick   Samuel 
## 1766.667 4685.000 3460.000
```

`tapply()` can be thought of as a combination of `split()` and `sapply()` for **vectors only**.

```
tapply2 <- function(x, group, f, ..., simplify = TRUE) {
    pieces <- split(x, group)
    sapply(pieces, f, ..., simplify = simplify)
}
tapply2(frogger_scores$score, frogger_scores$player, mean)
```

```
##  Charles     Nick   Samuel 
## 1766.667 4685.000 3460.000
```

```
tapply(small_corpus$n, list(small_corpus$from, small_corpus$to), sum)
```

```
##     107 11 110 112 114 123 128 145 155 160 165 167 169 22 29 38 46 50  6
## 107  NA NA  NA  NA  14  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
## 112   2 NA  NA  NA   2  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
## 114  24 NA  16  12  NA   2  NA   8  56   2  10  NA  60  4  2 14 NA  2 NA
## 145  NA NA  NA  NA  NA  NA   2  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
## 155  NA NA  16  NA  16  NA  NA   2  NA  NA  NA  NA   4 NA NA NA NA NA NA
## 160  NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  5 NA NA NA NA NA
## 169  NA NA  14  NA  50  NA  NA  NA   8  NA  14  NA  NA NA NA NA  2 NA  2
## 22   NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
## 38   NA 12  NA   2   2  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
## 50   NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  16  NA NA NA NA NA  2 NA
## 65   NA NA  NA  NA  NA  NA  NA  NA  NA   2  NA  NA  NA  2 NA  2 NA NA NA
##     65 96
## 107 NA NA
## 112  2 NA
## 114 10 NA
## 145 NA NA
## 155 NA NA
## 160 NA NA
## 169 NA NA
## 22   2 NA
## 38   4  2
## 50  NA NA
## 65  NA NA
```

Higer-Order Functions and Functionals

A **higher-order function** is a function that does at least one of the following:

- returns a function as its result
  - closures, functions returned by another function
- takes one or more functions as arguments (i.e. procedural parameters)
  - **functionals**

"To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs."

— Bjarne Stroustrup

Functionals implemented in base R are efficient and less error prone by better communicating intent.

Why Use `apply` Functions Instead of `for` Loops?

- The code is cleaner (once we're familiar with the concept). The code can be easier to code and read, and less error prone because we don't have to deal with subsetting and saving the results.
- `apply` functions can be faster than `for` loops, sometimes dramatically.

Advantages of Vectorization

In R, **vectorization** means operations or functions operate on all elements of a vector without needing to loop through and act on each element one at a time.

- Clarity: the syntax is about **what** we're doing
- Abstraction: the syntax hides **how** the computer does it
- Concision: we write less
- Generality: same syntax works for numbers, vectors, arrays, …
- Speed: e.g., modifying big vectors over and over is slow in R; work gets done by optimized low-level code