Topic 2: Data Structures in R

ISOM3390: Business Programming in R

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).

Functions and Operators

Command	Description
+, -, *, /	add, subtract, multiply, divide
^	raise to the power of
%%	remainder after division
log(), exp()	logarithms and exponents
sqrt()	square root
round(), floor(), ceiling()	round to the nearest whole number, round down, or round up
>, <, ==, >=, <=, !=	comparisons
&, |, !	logical conjunction, disjunction, negation

The Basic Data Structure: Vectors

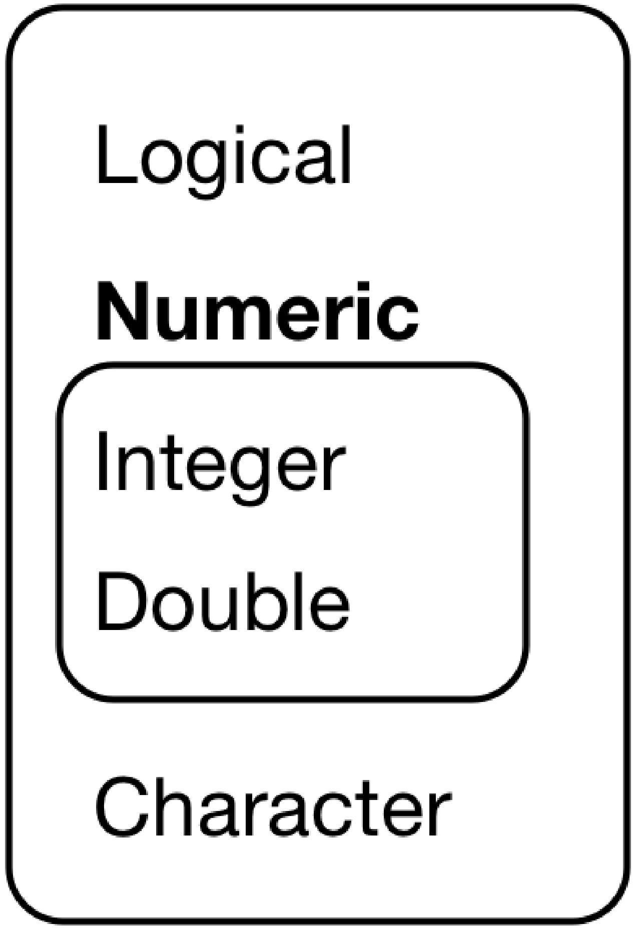
A **data structure** allows us to group related data.

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

Four common types of vectors:

- ***logical***;
- ***integer*** and ***double***, collectively known as ***numeric***;
- ***character***.

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



Creating Vectors

Vectors are usually created with `c()`, short for combine or concatenate:

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

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

```
# With the L suffix, we get an integer rather than a double
(int_var <- c(1L, 6L, 10L))
```

```
## [1]  1  6 10
```

```
# Use TRUE and FALSE (or T and F) to create logical vectors
(log_var <- c(TRUE, FALSE, T, F))
```

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

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

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

```
# str(), short for structure, can give a compact, human readable description of its data structure.
str(log_var)
```

```
##  logi [1:4] TRUE FALSE TRUE FALSE
```

All elements of a vector must be the same type.

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, Continued

Empty vectors of the given length and type can be created with the `vector()` function; helpful for filling things up later

```
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, not 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.

```
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"
```

Vectorization

Consider an example of vector arithmetic below:

```
v <- c(4, 7, 23.5, 76.2, 80)
2 * v
```

```
## [1]   8.0  14.0  47.0 152.4 160.0
```

Operators apply to vectors pairwise or elementwise, i.e. operate directly on each element.

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.

```
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
```

```
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
```

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
```

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.

List Extension

Lists can be concatenated using the `c()` function.

```
mylst<-c(mylst, list(gender="male",age=19) )
str(mylst)
```

```
## List of 5
##  $       : num 34453
##  $       : chr "John"
##  $ marks : num [1:4] 14.3 12 15 19
##  $ gender: chr "male"
##  $ age   : num 19
```

If given a combination of vectors and lists, `c()` will coerce the vectors to lists before combining them.

```
str(c(mylst, c("Ana", "Mike")))
```

```
## List of 7
##  $       : num 34453
##  $       : chr "John"
##  $ marks : num [1:4] 14.3 12 15 19
##  $ gender: chr "male"
##  $ age   : num 19
##  $       : chr "Ana"
##  $       : chr "Mike"
```

Lists Are Recursive

A list can contain other lists:

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

```
## List of 2
##  $ :List of 5
##   ..$       : num 34453
##   ..$       : chr "John"
##   ..$ marks : num [1:4] 14.3 12 15 19
##   ..$ gender: chr "male"
##   ..$ age   : num 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]]$marks
## [1] 14.3 12.0 15.0 19.0
##
## [[1]]$gender
## [1] "male"
##
## [[1]]$age
## [1] 19
##
##
## [[2]]
## [[2]]$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]]$marks
## [1] 14.3 12.0 15.0 19.0
##
## [[1]]$gender
## [1] "male"
##
## [[1]]$age
## [1] 19
```

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

```
## List of 1
##  $ :List of 5
##   ..$       : num 34453
##   ..$       : chr "John"
##   ..$ marks : num [1:4] 14.3 12 15 19
##   ..$ gender: chr "male"
##   ..$ age   : num 19
```

On the contrary, the operator `[[` (by position) and `$` (by name) 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
##
## $gender
## [1] "male"
##
## $age
## [1] 19
```

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

```
## List of 5
##  $       : num 34453
##  $       : chr "John"
##  $ marks : num [1:4] 14.3 12 15 19
##  $ gender: chr "male"
##  $ age   : num 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"
```

List Extension, Continued

Lists can also be extended by adding further components:

```r
mylst$parents <- c("Ana", "Mike")
mylst[["parents"]] <- c("Ana", "Mike")
mylst
```

```
## [[1]]
## [1] 34453
##
## [[2]]
## [1] "John"
##
## $marks
## [1] 14.3 12.0 15.0 19.0
##
## $gender
## [1] "male"
##
## $age
## [1] 19
##
## $parents
## [1] "Ana"  "Mike"
```

Attributes of a List

The names of the components are a list attribute.

```
attributes(mylst)
```

```
## $names
## [1] ""         ""         "marks"    "gender"   "age"      "parents"
```

```
# 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 [1:6] "" "" "marks" "gender" ...
```

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

```
## [1] ""         ""         "marks"    "gender"   "age"      "parents"
```

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

```
## List of 6
##  $ id     : num 34453
##  $ name   : chr "John"
##  $ marks  : num [1:4] 14.3 12 15 19
##  $ gender : chr "male"
##  $ age    : num 19
##  $ parents: chr [1:2] "Ana" "Mike"
```

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   gender      age 
##  "34453"   "John"   "14.3"     "12"     "15"     "19"   "male"     "19" 
## parents1 parents2 
##    "Ana"   "Mike" 
```

Factors

A factor is a vector that can contain only **pre-defined values**, and is used to store 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"
```

Factors are built on top of **integer** vectors using two attributes: the **class**, `"factor"`, which makes them behave differently from regular integer vectors, and the **levels**, which defines the set of allowed values.

```
attributes(d2)
```

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

```
class(d2)
```

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

```
levels(d2)
```

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

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 factors are self-describing.

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
```

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

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

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

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

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

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

```r
# 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
```

```r
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
```

Sometimes we wish to calculate the marginal and relative frequencies for this type of tables

```
margin.table(t,1)
```

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

```
margin.table(t,2)
```

```
## age
##    adult juvenile
##        3        2
```

Relative frequencies with respect to each margin and overall are given by

```
prop.table(t,1)              ## P(y|a)
```

```
##         age
## gender       adult  juvenile
##   Male   0.5000000 0.5000000
##   Female 0.6666667 0.3333333
```

```
prop.table(t,2)              ## P(a|y)
```

```
##         age
## gender       adult  juvenile
##   Male   0.3333333 0.5000000
##   Female 0.6666667 0.5000000
```

```
prop.table(t)                ## P(a,y)
```

```
##         age
## gender   adult juvenile
##   Male     0.2      0.2
##   Female   0.4      0.2
```

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 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
```

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
```

```
attributes(m)
```

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

```
typeof(m)
```

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

```
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, 1.0)
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
```

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). This gives rise to the five data types most often used in data analysis as shown in the table below:

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