Chapter 3: Objects, Data and Basic Operations

Data Types in R

- ✓ Integer
  - ➡ Any number that does not have a decimal place is an integer
  - ➡ Includes positive, negative numbers and 0
  - ➡ Examples: 1, -8, 9, 24

- ✓ Double
  - ➡ Used for numbers with decimals.
  - ➡ Default type for any numerical object.
  - ➡ Follows the IEEE 754 standard; uses 64 bits to store any number
  - ➡ Examples: 1.3, -8.5, 3.0, -98.2

- ✓ Complex
  - ➡ A square root of a negative number
  - ➡ Has the pattern: a + b*i, where a and b are doubles and i is the square root of -1
  - ➡ Examples: -2 - i, 3.4 + 23.1i, 4 + 2i

- ✓ Logical
  - ➡ Has two values: 0 or 1 produced as a result of a logical test

Data Types in R

- ✓ Date/Time
  - ➡ Used to store dates and timestamps in R
  - ➡ Can take multiple formats which can be controlled
  - ➡ Examples: Sep 1, 2012, 24/3/2004, 4/29/97, January 28 2005

- ✓ Character
  - ➡ All text data in R has this type
  - ➡ Examples: "John", "This is a car", "4.2"

Working with Data Types

- ✓ Checking data type of an object
  - ➡ Function *typeof*() may be used to check the data type of an object
    - Usage: Type *typeof*({object}) at the command line
    - Example: *typeof*(x) where x = 3, will return "integer"
    - Example: *typeof*(y) where y = -3 + 4i will return "complex"

- ✓ Checking data type of an object
  - ➡ Function *is.xxxx*() may be used to check if an object is of data type *xxxx*
    - Usage: Type *is.integer*({object}) at the command line
    - Example: *is.integer*(x) where x = 3, will return TRUE
    - Usage: Type *is.double*({object}) at the command line
    - Example: *is.double*(x) where x = 3, will return FALSE
    - Usage: Type *is.complex*({object}) at the command line
    - Example: *is.complex*(x) where x = 3, will return FALSE
    - Usage: Type *is.character*({object}) at the command line
    - Example: *is.character*(y) where y = "Hello", will return TRUE

- ✓ Some rules
  - ➡ Default type for all numbers: Double
  - ➡ An expression or function can contain mixed data types
    - If the types include double, integer or logical, the result is a double
    - If any of the types is complex, the result is complex
    - The largest integer in R is $2^{31} - 1$

Working with Data Types

- ✓ Creating an object of a specific data type
  - ➡ Function *as.xxxx*() is used to create an object as type xxxx
    - Usage: Type *as.integer*({object}) to create an integer object
    - Example: y <- *as.integer*(x) where x = 4.7, will assign a value of 4 to y
    - Usage: Type *as.character*({object}) to create a character object
    - Example: y <- *as.character*(x) where x = 4.7, will assign a value of "4.7" to y
    - Usage: Type *as.double*({object}) to convert to a double
    - Example: y <- *as.double*(x) where x = 3, will result in y = 3 where x is a double
      - The other way to do this is of course just set y <- 3
    - This function can be used to convert data to a different type. When converting from
      - Integer to Double or Complex, and back to Integer, the number is retained
      - Double to Integer, the decimal place is lost.
      - Double to Complex and back to Double, the number is retained
      - Complex to Double or Integer, the imaginary part is discarded
      - Double or Integer to Logical, the value TRUE is returned
      - Logical objects have two values: TRUE or FALSE corresponding to 1 or 0. Upon conversion to Double or Integer, these values are passed
      - If an Integer or Double has a value 0, if converted Logical, the value FALSE is assigned. Else, the value TRUE is assigned

Exercise

Object Types in R

- ✓ Variable
  - ➡ Is an object with a single value. It is the simplest object in R.
  - ➡ Can be assigned an expression involving other variables
  - ➡ Examples: x = 3, or x = b + c where b = 2 and c = 6

- ✓ Vector
  - ➡ Is a collection of elements of the same data type
  - ➡ Allows operations to be performed on every element
  - ➡ Examples: {1, 2, 3}, {"John", "Bethany", "Allison"}, {10.4, 7.8, 9.1}

- ✓ Sequence
  - ➡ Is a vector made of numbers
  - ➡ Have a fixed pattern
  - ➡ Examples: {1, 4, 7}, {5.1, 6.2, 7.3}, {1 + 2i, 2 + 3i, 3 + 4i}

- ✓ Array
  - ➡ Is a multi-dimensional series of data
  - ➡ Each element has the same data type
  - ➡ Number of dimensions >= 1

Object Types in R

- ✓ Matrix
  - ➡ Is a two-dimensional array of the same data type
  - ➡ Allows operations to be performed on every object
  - ➡ Example:

3	5
1	7
2	4

- ✓ Factor
  - ➡ Is like a vector where distinct objects are stored as levels
  - ➡ The factor returns the objects as well as the levels
  - ➡ Can take any data type

- ✓ List
  - ➡ It is like a vector
  - ➡ Each component of a list can be a sequence
  - ➡ It can contain components of different data types, including a list

- ✓ Data Frame
  - ➡ Is a matrix like structure
  - ➡ Is a table of values of different data types

Object Types in R

- ✓ Table
  - ➡ Is a count of distinct levels of two or more factors

Intrinsic Attributes

- ✓ Atomic vs. recursive
  - ➡ Atomic structure: all components of the object has the same mode
    - Modes: numeric, complex, logical, character, or raw
  - ➡ Recursive structure: components can have different modes
    - Example: Lists. Different components may have different modes, including list
    - Other recursive structures: Functions and expressions

- ✓ Attributes
  - ➡ By definition, any object in R has two intrinsic attributes: mode and length
  - ➡ Function *mode*() is used to obtain the mode of an object
    - Usage: Type *mode*({object})
  - ➡ Function *length*() is used to obtain the length of an object
    - Usage: Type *length*({object})
  - ➡ Examples:

```
> x <- c(1:10)
> mode(x)
[1] "numeric"
> length(x)
[1] 10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Class of an object

- ✓ Working with classes
  - ➡ Every object in R is assigned a class
  - ➡ Functions in R perform different operations on an object based on its class
  - ➡ Function *class*() may be used to derive the class of an object
    - Usage: Type *class*({object})
    - Example:

```
> z <- list(Names=c("Megan","Jay"), DOB=c("Oct 7 1984","Sep 14 1976"), Age=c(27,36))
> typeof(z)
[1] "list"
> class(z)
[1] "list"
> phones <- read.csv("phones.csv",header=TRUE, sep=",", row.names=1)
> phones
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier
iPhone          Apple   399      USA 2687161         iOS 3000000     AT&T
Galaxy        Samsung   350    Korea  256121     Android 5716247  Verizon
Razr          Motorola  200      USA   26511     Android   12381   Sprint
Pearl       Blackberry  399   Canada  125819  Blackberry  123701   Rogers
Optimus One         LG  299    Korea  123291     Android   12312     AT&T
Lumia 800       Nokia   299  Finland   23432   Microsoft   87699  Verizon
> typeof(phones)
[1] "list"
> class(phones)
[1] "data.frame"
```

  - ➡ Function *unclass*() is used to temporarily remove the effects of an object's class
    - Usage: Type *class*({object})
    - Example: Covered in further detail in Chapter 5: Operations on Factors

Non- Intrinsic Attributes

- ✓ Assignment
  - ➡ Attributes can be assigned to any object in R
  - ➡ Function *attr*() is used for this purpose
    - Usage: Type *attr*({object}, {attribute})
    - Examples: Say vector x = a b c. To assign an description, type attr(x, "Descr") <- "Vector of alphabets"

- ✓ Retrieving attributes
  - ➡ Function *attributes*() is used for this purpose
    - Usage: Type *attributes*({object})
    - In the above example, attributes(x) will yield $Descr with value "Vector of alphabets"

- ✓ Multiple attributes
  - ➡ Function *structure*() is used for multiple assignment
    - Usage: Type *structure*({object}, attribute1 = "", attribute2 = "",…)
    - Example:

```
> x <- c(1:6)
> x <- structure(x,Name = "Vector", Description = "Contains numbers 1 to 6")
> x
[1] 1 2 3 4 5 6
attr(,"Name")
[1] "Vector"
attr(,"Description")
[1] "Contains numbers 1 to 6"
```

Exercise

Vectors

- ✓ Creation/Assignment
  - ➡ Function *c*() is used to assign values to a vector and in the process creating it
    - Usage: Type *c*() to create a vector
    - Example: x <- *c*(1, 2, 4, 2, 5, 1, 6, 3) will result in x = 1 2 4 2 5 1 6 3
  - ➡ Functions *as.vector*() and *is.vector*() apply

- ✓ Arithmetic
  - ➡ Arithmetic operators listed in Chapter 2 can be used on vectors
    - Examples:
      - If x = 1 2 3 and y = 4 5 6, z <- x + y will yield z = 5 7 9
      - If x = 23.1 42.5 87.6 and y = 1.3 8.6 2.9, z <- x - y will yield z = 21.8 33.9 84.7

- ✓ Argument recycling
  - ➡ For expressions where vectors of unequal lengths are used, the arguments of the shorter vector are recycled
    - Examples:
      - If x = 1 2 3 and y = 4 5, z <- x + y will yield z = 5 7 7. Arguments of y will be recycled; for the first two arguments of x, the two arguments of y are used. For the third argument of x, because there is no corresponding argument for y, R recycles arguments and starts from the first.

Vectors

✓ Mathematical functions

➡ When a mathematical function is used on a vector, the result is also a vector

- Example:
  - xrad <- *x * pi /180* where x = 30 60 90, will result in xrad = 0.52, 1.05, 1.57
  - xsin <- sin(xrad) will result in xsin = 0.5, 0.87, 1.00
  - xlog <- log10(x) where x = 10 20 30 will result in xlog = 1.0 1.3 1.5

➡ Sample list of mathematical functions

Function	Description
sin, cos, tan	Sine, Cosine, Tangent
asin, acos, atan	Inverse of Sin, Cosine, Tangent
log	Logarithm of any base (default = e)
exp	Exponential
round	Rounding a number
sqrt	Square root
floor, ceiling, trunc	Creates integers from floating point numbers

Vectors

- ✓ Replication
  - ➡ Function *rep*() can be used to replicate a vector
    - Usage: Type *rep*({vector}, each =…, times = …) to replicate a vector
    - Examples:
      - y <- rep(3:5, 3) will yield y = 3 4 5 3 4 5 3 4 5
      - y <- rep(3:5, each = 2) will yield y = 3 3 4 4 5 5
      - y <- rep(3:5, each = 3, times = 2) will yield y = 3 3 3 4 4 4 5 5 5 3 3 3 4 4 4 5 5 5
      - y <- rep(3:5, c(2,2,2)) will yield y = 3 3 4 4 5 5

- ✓ Sorting
  - ➡ Function *sort*() is used to orders vector arguments
    - Usage: Type *sort*({vector}, decreasing = ...) to sort a vector
    - Example: y <- *sort(x)* where x = 1 2 4 2 1 4 7 2 1 will result in y = 1 1 1 2 2 2 4 7
    - Example: y <- *sort(x, decreasing* = TRUE) where x = 1 2 4 2 1 4 7 2 1 will result in y = 7 4 2 2 2 1 1 1

Vectors

- ✓ Logical vectors
  - ➡ Function *c*() can also be used to create a logical vector
    - Example: y <- x > 13 where x = 14 3 5 19 will yield y = TRUE FALSE FALSE TRUE
  - ➡ Logical vectors can be used in arithmetic functions. R will use a value of 1 for TRUE and 0 for FALSE

- ✓ Character vectors
  - ➡ Function *c*() can further be used to create a character vector
    - Example: y <- c("John", "Cara", "Brittany") will yield y = John Cara Brittany
  - ➡ Function *paste*() can be used to concatenate
    - Example: zcat <- paste(x, y, sep = "") where x = a b c and y = 1 2 3 will yield zcat = a1 b2 c3
    - Example: zcat <- paste(x, y, sep = "") where x = a b and y = 1 2 3 will yield zcat = a1 b2 a3 [arguments of x are recycled]

Sequences

- ✓ Creation
  - ➡ Function *seq*() can used to create a sequence
    - Usage: Type *seq*(from=.., to=.., by=.., length=..) to create a sequence
    - Examples:
      - y <- seq(-10, 10, by = 0.25) will yield y = -10 -9.75 -9.5 -9.25, ……, 9.5 9.75 10
      - y <- seq(from = 1, length = 10, by = 0.5) will yield y = 1 1.5 2 2.5 3 3.5 4 4.5 5
  - ➡ Function *c*() can also used to create a sequence
    - Usage: Type *c*(from:to) to create a sequence
    - Examples:
      - y <- c(1:5) will yield y = 1 2 3 4 5
      - y <- c(2*3:7 - 1) will yield y = 5 7 9 11 13
- ✓ In all other aspects, sequences behave the same as vectors of the respective data type

Exercise

Arrays

- ✓ Creation
  - ➡ Define a vector using function *c*() and then declare the vector as an array by including the function *dim*() [this assigns a dimension vector]
    - Usage: Assign dim({vector}) <- *c*()
    - Example:
      - x <- *c*(1:10) will yield a vector x = 1 2 3 4 5 6 7 8 9 10
      - dim(x) <- c(2,5) will result in x being assigned dimensions 2 x 5 turning it into an array
      - x now becomes 1 3 5 7 9
        2 4 6 8 10
  - ➡ Function *array*() can be used to perform both steps in a single assignment
    - Usage: Type *array*({vector}, dim = …, dimnames = …) to create an array
    - Examples:
      - xarr <- array(c(1:10), dim = c(2,5), dimnames = c("x","y")) will create the same array listed above and assign dimension names of x and y
  - ➡ Functions *as.array*() and *is.array*() apply
- ✓ Subsection of an array
  - ➡ Elements of an array can be retrieved by referencing its position, or the index
    - Example: For a 3 dimensional array a[3,4,2], a[1,3,1] with dimensions x, y and z, a[1,3,1] is the index of the 1st element in dimension x, 3rd element in y and 1st in z
  - ➡ Subsections of an array can be retrieved by referencing a subset of the dimensions
    - Example: In the above 3-d array, a[1, ,] will retrieve elements a[1,1,1], a[1,1,2], a[1,2,1], a[1,2,2], a[1,3,1], a[1,3,2], a[1,4,1] and a[1,4,2]

Arrays

- ✓ Arithmetic and mathematical operations
  - ➡ Arithmetic (Chapter 2) and mathematical (listed for vectors) operators can be used on arrays
    - Examples:
      - If x = $\begin{matrix} 3 & 5 \\ 1 & 7 \end{matrix}$ and y = $\begin{matrix} 2 & 4 \\ 8 & 1 \end{matrix}$, then x + y will yield $\begin{matrix} 5 & 9 \\ 9 & 8 \end{matrix}$
      - If x = $\begin{matrix} 30 & 90 \\ 60 & 120 \end{matrix}$ then sin(x * pi/180) will yield $\begin{matrix} 0.5 & 1 \\ 0.87 & 0.87 \end{matrix}$
  - ➡ For operations involving two arrays, both arrays must have the same dimension vector
  - ➡ For operations involving an array and a vector, the vector must have the same or fewer elements than the array
  - ➡ Using function *sort*() on an array will result in a vector

- ✓ Transposing an array
  - ➡ An array can be transposed by using function *aperm*()
    - Usage: Type *aperm*({array}, perm) to transpose an array, where perm is a vector that determines the order of the subscripts.
    - Example: To transpose a 2 x 3 x 4 array to 2 x 4 x 3, switch the y and z indices, i.e., make the dimension vector 1, 3, 2 instead of 1, 2, 3 . For doing this, type v aperm <- c(1, 3, 2))
    - Hint: aperm(w, c(1,2,3)) will yield a array that is the same as w.

Matrices

- ✓ Creation
  - ➡ Function *matrix*() can used to create a matrix
    - Usage: Type *matrix*(data=.., nrow=.., ncol=.., byrow=.., dimnames=...) to create a matrix
    - Examples:
      - y <- matrix(data=c(1:12), nrow=3, ncol=4, byrow=TRUE) will yield y =

1	2	3	4
5	6	7	8
9	10	11	12

      - The same can be achieved by y <- matrix(c(1:12), 3, 4, TRUE)
  - ➡ Functions *as.matrix*() and *is.matrix*() apply

- ✓ Transposing a matrix
  - ➡ Function *t*() can used to transpose a matrix
    - Usage: Type *t*({matrix}) to transpose a matrix
    - In the above example, t(y) will yield

1	5	9
2	6	10
3	7	11
4	8	12

- ✓ Adding a row
  - ➡ Function *rbind*() can used to add a row to a matrix or combine two vectors of the same length to a matrix
    - Usage: Type *rbind*({matrix},{vector}) to transpose a matrix
    - In the above example, rbind(y, c(20:23)) will yield

1	2	3	4
5	6	7	8
9	10	11	12
20	21	22	23

Matrices

✓ Adding a column

➡ Function *cbind*() can used to add a column to a matrix or combine two vectors of the same length to a matrix (the vectors will form columns in the matrix)

- Usage: Type *cbind*({matrix}, {vector}) to add a column to a matrix
- In the previous example, cbind(t(y), c(20:23)) will yield

1	5	9	20
2	6	10	21
3	7	11	22
4	8	12	23

✓ Arithmetic and mathematical operations

➡ Arithmetic (Chapter 2) and mathematical (listed for vectors) operators can be used on matrices

- Examples:
  - If x = $\begin{matrix} 3 & 5 \\ 1 & 7 \end{matrix}$ and y = $\begin{matrix} 2 & 4 \\ 8 & 1 \end{matrix}$, then x + y will yield $\begin{matrix} 5 & 9 \\ 9 & 8 \end{matrix}$
  - If x = $\begin{matrix} 30 & 90 \\ 60 & 120 \end{matrix}$ then sin(x * pi/180) will yield $\begin{matrix} 0.5 & 1 \\ 0.87 & 0.87 \end{matrix}$

➡ For operations involving two matrices, both matrices must have the same dimension vector

➡ For operations involving a matrix and a vector, the vector must have the same or fewer elements than the matrix

➡ Using function *sort*() on an matrix will result in a vector

Matrices

- ✓ Number of rows/columns
  - ➡ Function *nrow*() is used to return the number of rows of a matrix
    - Usage: Type *nrow*({matrix})
    - In the previous example, nrows(y) will yield 3
  - ➡ Function *ncol*() is used to return the number of columns of a matrix
    - Usage: Type *ncol*({matrix})
    - In the previous example, ncol(y) will yield 4

- ✓ Row and column matrices
  - ➡ For any given matrix, the corresponding row matrix is a matrix with the same dimensions where every element is the row number
    - In the previous example, row(y) will yield

1	1	1	1
2	2	2	2
3	3	3	3

  - ➡ For any given matrix, the corresponding column matrix is a matrix with the same dimensions where every element is the column number
    - In the previous example, col(y) will yield

1	2	3	4
1	2	3	4
1	2	3	4

Exercise

Factor

- ✓ Creation
  - ➡ Function *factor*() is used to create a Factor
    - Usage: Type *factor*({vector},levels=…, labels=…)
    - Example: y <- factor(c(1,1,2,1,2),levels = c(1:3)) will yield the factor 1 1 2 1 2 where levels: 1 2 3
  - ➡ Functions *as.factor*() and *is.factor*() apply

- ✓ Levels
  - ➡ Function *levels*() is used to obtain the levels of a factor
    - Usage: Type *levels*({factor})
    - Example: x <- levels(c(1,1,2,1,2)) will yield levels: 1 2

- ✓ Ordered factor
  - ➡ To create a factor where the levels have an order, use function *ordered*()
    - Usage: Type *ordered*({vector},levels=…, labels=…)
    - Example: y <- ordered(c(1,1,2,1,2),levels = c(1:3)) will yield the factor 1 1 2 1 2 where levels: 1 < 2 < 3

List

- ✓ Creation
  - ➡ Function *list*() is used to create a List
    - Usage: Type *list*(name1={object1},name2={object2},...)
    - Example: y <- list(values=c(1:3),test=c("True","False","False")) will yield a list where $values = 1 2 3 and $test = True False False
  - ➡ Functions *as.list*() and *is.list*() apply

- ✓ Referencing values in a list
  - ➡ Use listname$objectname or listname[[{object #}]] to access a specific object in a list
    - From the above example, y$values will yield 1 2 3 and y[[2]] will yield True False False
  - ➡ Use objectname[{entry #}] to access a specific value of an object in a list
    - From the above example, y$values[2] will yield 2 and y[[2]][1] will yield True
  - ➡ Once referenced, they can be used in operations including assignment

- ✓ Changing names
  - ➡ Function *names*() can be used to access or change the names of objects in a list
    - Usage: Type *names*({object1})
    - In the above example, *names*(y) will yield "values" "test"
    - Further *names*(y) <- c("numbers","T_F") will change the object names. Objects y then turns into a list where $numbers = 1 2 3 and $T_F = True False False

List

- ✓ Adding values
  - ➡ Function *append*() is used to add values to a List
    - Usage: Type *append*({list}, values, after = ...)
    - Example: y <- list(values=c(1:3),test=c("True","False","False")) will yield a list where $values = 1 2 3 and $test = True False False
    - In the above example, y <- append(y, as.integer(25), after = 1) will insert 25 into the second place

- ✓ Function *append*() applies to other objects: vectors, matrices, factors etc

Exercise

Data Frame

- ✓ Creation
  - ➡ Function *data.frame*() is used to create a Data Frame
    - Usage: Type *data.frame*(name1={object1},name2={object2},...)
    - Example: y <- data.frame(Var1=c(1:3),test=c("True","False","False")) will yield a data frame with two columns: Var1 and test with values 1 2 3 and True False False respectively
  - ➡ Functions *as.data.frame*() and *is.data.frame*() apply
- ✓ Referencing values in a data frame
  - ➡ Use dataframename$variablename to access a specific variable in a data frame
    - From the above example, y$Var1 will yield 1 2 3
  - ➡ Use objectname[{entry #}] to access a specific value of an object
    - From the above example, y$test[2] will yield False
  - ➡ Once referenced, they can be used in operations including assignment
- ✓ Changing names
  - ➡ Function *names*() can be used to access or change the names of objects in a data frame
    - Usage: Type *names*({object1})
    - In the above example, *names*(y) will yield "Var1" "test"
    - Further *names*(y) <- c("Var2","test1") will change the object names. Objects y then turns into a data frame where $Var2 = 1 2 3 and $test1 = True False False

Data Frame

- ✓ Working with data frames
  - ➡ When a data frame is created (or loaded from a package such as "Datasets"), the variables in the data frame can be only be accessed by referencing the data frame
  - ➡ Function *attach*() allows variables in a data frame to be referenced independently of the data frame. This function creates a copy of the data frame in position 2 in the search path.
    - Usage: Type *attach*({data frame}) attaches the data frame
    - Example: attach(women) attaches data frame "women" from "Datasets"
  - ➡ Variables of the attached data frame may be independently referenced and used
  - ➡ Any assignments made to the variables will create a copy of the variable, with the same name, and will be placed in position 1 of the search path
  - ➡ To modify the data frame itself,
    - The data frame needs to be referenced along with the variable
      - Example: women$height <- height + 10. This adds 10 to the variable height in data frame women
    - The data frame needs to be detached and then re-attached. Function *detach*() may be used to remove the data frame from the search path
      - Usage: Type *detach*({data frame}) attaches the data frame
      - Example: detach(women) attaches data frame "women"

NOTE: *This approach works for most object types, including tables and lists*

Table

- ✓ Creation
  - ➡ Function *table*() can used to create a table of cross-classifying counts from two underlying factors
    - Usage: Type *table*(..., exclude=…, dnn=.., deparse.level =...) to create a table
    - Examples:
      - If x = a b c and y = 1 2 3, then z <- table(x,y,) will yield z =

	y		
x	1	0	0
	0	1	0
	0	0	1

      - Row and column names can be changed by using *dnn*
  - ➡ Functions *as.table*() and *is.table*() apply

- ✓ Basic operations
  - ➡ Any value in the table can be referenced by using the index position in square brackets
    - In the above example, y[1] = 1, y[4] = 0 and y[5] = 1
  - ➡ Once created, the table can be treated like a matrix and is subject to all matrix operations

- ✓ Underlying function
  - ➡ *tabulate*() is the underlying function and is used to count the number of occurrences of an integer in a vector
    - Usage: Type *tabulate*({vector},nbins=...) to tabulate an integer vector
    - Example: If x = 1 1 2 2 1 1 3, tabulate(x,nbins = 4) will yield 4 2 1 0

Exercise