Session Four - Data Types and Objects

By Hellen Gakuruh

May 23, 2016

Table of Contents

Session Goal

The main goal of this session is to enable you to work with R data structures or objects.

What we shall cover

By the end of this session you should:

- Be able know the basic data types in R
- Know the basic data structure in R and its types
- Be conversant with properties of R' data structures specifically type and attributes
- Be acquainted with R's data structures like arrays, matrices, data frames, and lists
- Know properties of an object in terms of how to get and set them

Prerequisite

To appreciate this session,

- you must have R and RStudio installed and running
- know how to use the console
- be conversant with RStudio panes like console, help and packages
- Be able to make function calls

Data Types

R recognizes only six data types, these are:

- **Logical**: True and False values mostly generated by conditions using binary operators (<, <=, >, >=, ==, !=, &, &&, |, ||)
- **Integer**: Whole number
- **Real/Double**: Continuous numbers e.g. 3.4, 2.9 e.t.c
- **String/character**: alphabetical letters and words/text denoted with single or double quotation
- **Complex**: unique numbers used mathematically for square rooting negative numbers and expressed as real and imaginary number (i) e.g. 3 + 2i
- **Raw**: data containing computer bytes or information on data storage units, these are more of computer language as opposed to human readable language

Logical vectors can sometimes be coerced into numeric (integer) values especially during analysis, in which case FALSE becomes 0 and TRUE becomes 1. These vectors also include **NA** which means **Not Available**, a marker for values that are missing. Vectors with any of the data type can have NA (missing data) but any operation carried out these vectors will result in NA since the value of NA is not known thus the whole operation is declared incomplete by R.

Complex and raw data types are rarely used as compared to logic, integer, real and character; however, it is good to know they exist in R.

There are different R' data structures that store one or more of these data types, the most basic being a **Vector**.

Vectors

A vector in R contains a series of data (though it can contain a single data point or no data); it can also be viewed as a variable. There are two types of vectors; **atomic** and **generic (lists)** vectors. These two vectors differ in the sense that atomic vectors can only contain one type of data while lists can have more than one type of data (and other objects).

To understand this, we need to look at some examples of vectors, we shall start with the atomic vector and it's different structures, but first we need to know how to create them.

Creating vectors

An atomic vector can be created by the function **c()** which means combine. Combine tells R to treat all the components as one unit.

As an example, let's create a vector of ages in a class room. The ages are; 23, 28, 19, 34, and 25.

```
# Creating a vector with the combine function
c(23, 28, 19, 34, 26)
## [1] 23 28 19 34 26
```

If you have a sequence of numbers (integers), then you do not need the c(), instead use **:**, or generate numeric values using the seq function.

```
# Creating numeric (integer) sequences
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
14:30
##  [1] 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

# Creating numeric (double) sequences
seq(from = 0, to = 1, length.out = 10)
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
##  [8] 0.7777778 0.8888889 1.0000000
```

To be able to use this vector or part of it, we need to store the values by assigning them to a name; this is done by **assignment operators**.

There are three ways to assign a vector to a name, they include the *arrow like* assignment operator, the **assign()** function and the **= (equals)** sign.

Of the three, the most commonly used is the *arrow like* operator composed of **< (greater than sign)** followed by a **- (minus sign)** without spaces i.e. **<-**. The assign() is mostly used in function definition (creating functions) and equals sign is used to pass values to argument.

```
# Using the "<-" operator
ages1 <- c(23, 28, 19, 34, 26)

# Using the assign function
assign("ages2", c(23, 28, 19, 34, 26))

# Compare if they are the same vectors
all.equal(ages1, ages2)
## [1] TRUE
```

* In this tutorial we shall use the **<-** when creating objects.

When you create a vector or any other object (like a function), it is temporary stored in the workspace. So we should be able to see two objects with the same data; ages1 and ages2. We can also use the function **ls/objects** to see the names of objects in our workspace.

```
ls()
## [1] "ages1" "ages2"
objects()
## [1] "ages1" "ages2"
```

Quick note, when you create an object, no output will be displayed on the console as R directly stores the object.

To see your values on the console, or what is referred to as **print to the console**, just type the name of your object (without quotation marks as it is an object and not a character) and you should see the values.

```
# Printing the first object
ages1
## [1] 23 28 19 34 26

# We can now remove the duplicate (second object)
rm(ages2)

# Rename the first object by reassigning name and removing original
ages <- ages1
rm(ages1)

# See workspace
ls()
## [1] "ages"
```

Great, we have created an atomic vector and printed it to the console. Now take a closer look at the values presented on the console, when you print "ages" vector. Do you see the number 1 in a square bracket i.e. **[1]**? The question is what is it, and what does it mean?

R produces outputs in a vector form and each element of the vector is indexed or given a position number in the vector, for example, 23 is the first value in the output vector, 28 is the second, 19 is the third, 34 is the fourth and 26 is the fifth.

You can now use this “ages” object to do computations without retyping the values as we did when using the console.

```
# Getting mean of class ages
mean(ages)
## [1] 26
```

Now let's create a character vector with the names of class students whose ages are stored in the ages object. Their names are, **HG**, **WN**, **TM**, **KL**, **DK**.

Character vectors are created the same way as numeric vector but this time with letters or text components enclosed with quotation marks (double **" "** or single **' '**). Quotation marks distinguishes a character with object names meaning R sees **ages** and **"ages"** differently; the first as an object and the second as a character.

```
# Creating a character object
nams <-  c("HG", "WN", "TM", "KL", "DK")

# Printing to the console
nams
## [1] "HG" "WN" "TM" "KL" "DK"
```

Now let's create an integer vector by converting the age data which is currently a numeric or double vector (we shall confirm this when discussing properties). Integer vectors are created by adding the suffix (qualifier) **L**.

```
# Creating an integer vector
agesInt <- c(23L, 28L, 19L, 34L, 26L)
```

A logical vector is created in the same way as the others, but they often result from checking conditions.

```
# Checking equality of two objects (returns one logical value)
all.equal(ages, agesInt)
## [1] TRUE
identical(ages, agesInt)
## [1] FALSE
```

Please read documentation on **all.equal** and **identical** to see why the two functions resulted in different outputs.

```
# Checking equality of element in two objects (element wise)
# Returns logical vector with the same number of elements as the vector
ages == agesInt
## [1] TRUE TRUE TRUE TRUE TRUE

# Note if the two vectors were not of the same length, the shorter vector
would have been recycled, read documentation on binary operator by typing
?"==")
```

Naming objects

A small note on naming your objects, you can basically name them anything you want, but it helps to have names that have some reference to the data. So you should assign names that are easy to recall and relate to.

However, there are certain naming exceptions and recommendations:

- R has some reserved words that should be avoided when naming objects, these are; if, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, NaN, NA, NA_integer_, NA_real_, NA_complex_, NA_character_, ..., etc.
- You cannot start the name of an object with a numeric value, but you can start it with an alphabet followed by a number.
- It can be as short as an alphabet e.g. x or a, and as long as you want, but it should not exceed 10,000 bytes. You can determine the number of bytes an object has with the function **object_size()** from the **pryr** package.
- You can use a period ".", or an underscore "_" within the name but periods are preferred in creating variable while underscore are preferred for naming files. Here are some examples:

```
age.mean <- mean(ages)
file.create("class_data.txt",sep = "\t")
```

So far we have discussed atomic vectors with one dimension. Think of dimensions as the number of indices to be used to locate an element in a vector, for example, a one dimension vector would require only one index like we saw for `ages` vector. A two dimensional vector would require two indices, that is the row and column. Dimensions in R are part of an object's attribute which we shall discuss later in this session.

Now let's look at higher dimensional (atomic) vectors, specifically **matrices** and **arrays**. Usual discourse in this area begins with array and then matrices as matrices are arrays with two dimensions, however, we will discuss matrices first as they are commonly used data objects and are easy to relate to.

Matrices

A matrix is a vector with two dimensions and in R; it is indexed by its row and column. There are various ways of creating matrices (other than importing them which we will cover in session five), these include:

- Creating from existing objects or vectors by using **cbind (column bind)** or **rbind (row bind)** functions.
- Using the **matrix** function or
- Coercing another data object into a matrix using **as.matrix** function

Please recall that atomic vectors of which matrix is one of them, can only contain one data type, if this is not the case, then some of the elements will be converted. The usual conversion logic is; logical < integer < double < complex < character. This means that,

- any other data type mix with character will result in a character vector,
- integer/double/logical mixed with complex becomes a complex vector,
- integer and/or logic mixed with double will result in a double vector,
- integer mixed with logic will result to an integer vector.

This will become clear when we discuss properties of an object and more specifically type of an object.

Creating matrices using cbind and rbind functions

As an example, let's use the vectors **ages** and **nams**, to create a matrix. First read up on the functions **cbind** and **rbind**, which between the two functions is suitable to form a data set with columns named nams and ages?

Try out both functions and see if it matches our requirement.

```
# rbinding the vectors
rbind(nams, ages)
##      [,1] [,2] [,3] [,4] [,5]
## nams "HG" "WN" "TM" "KL" "DK"
## ages "23" "28" "19" "34" "26"
# cbinding the vectors
cbind(nams, ages)
##      nams ages
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
```

Cbinding gave us the correct format. Now we can assign this object to the name "classData".

```
classData <- cbind(nams, ages)
classData
##      nams ages
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
```

Now we have a data set comprising student's names and their ages. We can easily see that the first student is named "HG" and is 23 years. Also note that the age vector is now a character vector.

Creating matrices using matrix function

Using the matrix function we can create two types of matrices given our two vectors ages and nams. We will let the first matrix to be a character vector and the second to be a numeric vector. This can be achieved by understanding matrix's arguments.

The most important argument in calling the matrix function is the **data** argument; however, it is vital to read how the other arguments affect creation of a matrix.

To create the first type of matrix (character vector), lets combine the two vectors into one by calling the c() function and passing it to the **data** argument. We can then specify the number of rows and columns as 5 and 2 respectively (though we can pass either the row or column alone).

```
# These are all the same
matrix(data = c(nams, ages), nrow = 5, ncol = 2)
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
matrix(data = c(nams, ages), nrow = 5)
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
matrix(data = c(nams, ages), ncol = 2)
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"

# Assigning a name
classData2 <- matrix(data = c(nams, ages), nrow = 5, ncol = 2)
#View the matrix on the console
classData2
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
```

Great, now to create the second matrix, we will put the student's names as row names thereby having a numeric vector with just the ages. Please read up on how the argument dimnames is used (it must be a list).

```
# Creating a numeric vector from character and numeric vectors
classData3 <- matrix(data = ages, nrow = 5, dimnames = list(c(nams),
c("StudentsAge")))
classData3
```

```
##    StudentsAge
## HG          23
## WN          28
## TM          19
## KL          34
## DK          26
```

Arrays

Arrays are multi-dimensional vectors. Multi-dimensional meaning they can have one to n dimensions, in this sense **matrices** are arrays with two dimensions. The most common type of array has three dimensions.

Arrays are created with the **array** function. But before we create an array, I would like to introduce you to two useful functions; **set.seed()** and **sample()**. set.seed function is used to ensure random numbers can be reproduced and sample() is used to draw random samples with a given probability. Read on "`?set.seed`" and "`?sample`".

```
set.seed(3885)
myArray <- array(sample(10:99, 8), dim = c(2, 2, 2), dimnames =
list(c("Exposed", "Not Exposed"), c("Yes", "No"), c("Urban", "Rural")))
myArray
## , , Urban
##
##             Yes No
## Exposed      50 48
## Not Exposed  70 43
##
## , , Rural
##
##             Yes No
## Exposed      49 29
## Not Exposed  44 39
```

Now let's discuss generic vectors, that is, lists (one dimensional) and data frames (two dimensional). As mentioned, they differ from atomic vectors in that they can store different data types.

We shall begin with data frames which might be more recognizable as they resemble excel spreadsheets or SPSS and STATA data sets. Data frames are R's fundamental data structure.

Data.Frames

Data frames are similar to matrices with the exception that variables or column data can be of different type.

Data frames are created with the **data.frame()** or from coercing another vector like a matrix to a data.frame with **as.data.frame** function.

Following our earlier example on age and names, we will create a data frame but this time adding more detail; gender and residence.

```r
#Adding two new variables/vectors
set.seed(8392)
gender <- sample(c("Female", "Male"), size = 5, replace = TRUE)
set.seed(3489)
residence <- sample(c("Ubarn", "Rural"), size = 5, replace = TRUE)
#Creating a dataframe
class_df <- data.frame(nams, ages, gender, residence)
class_df
##   nams ages gender residence
## 1   HG   23 Female     Rural
## 2   WN   28   Male     Rural
## 3   TM   19 Female     Rural
## 4   KL   34 Female     Rural
## 5   DK   26 Female     Ubarn
```

Data frames can also contain unique data objects called **factors**. Factors are qualitative data often referred to as categorical data. Factors can either be **nominal** (categories have no ordering like gender and residence) or **ordinal** (categories have natural ordering like level of education, but the distance between each category cannot be measured). In R, factor vectors are created with the **factor** function where the argument **order** is used to indicate its ordinal.

Factor variables are internally stored as integer vector. These integers are the levels of the variable and do not have computational meaning. For example, we can convert the gender vector to a factor variable through coercion and see its levels.

```r
# Converting a character vector to a factor vector
gender2 <- as.factor(gender)
# Levels of factor vector
levels(gender2)
## [1] "Female" "Male"
# Inspecting internal levels stored as integers
as.integer(gender2)
## [1] 1 2 1 1 1
# Characters have no levels hence "Not Available (NA)" output
as.integer(gender)
## [1] NA NA NA NA NA
```

It is meaningless to use levels of a factor variable as though they were numerical values. So it is wrong to compute the mean or plot with graphs meant for numeric or interval values such scatter plots.

If you read documentation for data frames (?data.frame), you will see an argument called **stringsAsFactors**. By default this is set to **default.stringsAsFactors()** which in most cases would be TRUE. This means any character vector used to construct the data frame is coerced to a factor vector, if this is not what you what, then ensure you pass the correct value for that argument i.e. FALSE.

We will do a little bit more with data frames during the import and export session as well as the data transformation and manipulation session. And right before winding up this

session, we shall see how to determine if an object is a data frame as well the type of data its variables have. For now let's discuss **lists**.

Lists

Lists are interesting vectors not commonly seen in other statistical programs. Interesting because they behave like a **`carry all basket`** in the sense that they are one dimension but can contain numerous other objects not just data structures. They are excellent storage structures for different objects of the same project. You literally can throw in a one dimension vector, a matrix, a data frame, a sub-list, a function and any other project object. For example, you can have a list with a data set, a source script, outputs and plots.

Since they can contain other lists, they are also called **recursive vectors**.

Lists are created with the **list** function.

```
class_list <- list(matrix = classData2, dataframe = classData2)
## $matrix
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
##
## $dataframe
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
```

We can have a glimpse of the data contained in any data object with the function **str()**.

```
# Inspect the list
str(class_list)
## List of 2
##  $ matrix   : chr [1:5, 1:2] "HG" "WN" "TM" "KL" ...
##  $ dataframe: chr [1:5, 1:2] "HG" "WN" "TM" "KL" ...
```

When creating lists (and other data structures), it is good to provide names of their elements. For example in the list above we gave the name matrix and data frame. These names are used during indexing.

The c() function can be used to combine different lists into one object.

```
class_list2 <- c(ListOne = class_list, ListTwo = list(myArray))

class_list2
## $ListOne.matrix
```

```
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
##
## $ListOne.dataframe
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
##
## $ListTwo
## , , Urban
##
##             Yes No
## Exposed      50 48
## Not Exposed  70 43
##
## , , Rural
##
##             Yes No
## Exposed      49 29
## Not Exposed  44 39
```

Properties of data objects (vector)

All vectors (atomic and generic) can be described by their **properties** which constitute their **type** and **attributes**.

Type of a vector

To determine the type of an object, call the function **typeof** (a complete list of the basic types is given in one of R's manuals called **R Language Definition** specifically section on `Objects`. There are also other specific functions that test for particular vector types, these functions begin with **is.**.

```
# What type of object is the vector ages?
typeof(ages)
## [1] "double"
# Is the vector a real number?
is.double(ages)
## [1] TRUE
# Is it a number?
is.numeric(ages)
## [1] TRUE
```

Note, numeric vectors are composed of both integers and double values, therefore `is.numeric()` tests if a vector is a number rather than its specific type (integer/double).

```
# What type of object is classData2
typeof(classData2)
## [1] "character"
# is it a matrix?
is.matrix(classData2)
## [1] TRUE

# What type of object is class_df
typeof(class_df)
## [1] "list"
# Is it a data frame?
is.data.frame(class_df)
## [1] TRUE

# What type of object is class_list
typeof(class_list)
## [1] "list"
# Is it a list?
is.list(class_list)
## [1] TRUE

# What type of object is gender2
typeof(gender2)
## [1] "integer"
# Is if a factor vector?
is.factor(gender2)
## [1] TRUE
```

From our exercise above, we can see that the type of an object (how R stores it internally) can differ from how it is presented to us. For example, matrices are internally stored by their atomic type but presented in a dimensional structure. Factors (which are not a data type) are stored internally as integer values or levels but presented as factor labels.

Surprisingly there is a function called **is.vector** which does not tell us if an object is a vector, instead it checks if an object has other attributes apart from its name(s). This function can therefore not be used to test vectorness in multi-dimension objects as they have dim attribute.

```
# Vectors ages and nams have no attributes
is.vector(ages)
## [1] TRUE

# But matrices and data frames have dim
is.vector(classData2)
## [1] FALSE
is.vector(class_df)
## [1] FALSE
```

```
# List has no attributes other than names
is.vector(class_list)
## [1] TRUE
```

To convert a list to an atomic vector, use **unlist** function.

```
unlist(class_list)
##     matrix1     matrix2     matrix3     matrix4     matrix5     matrix6
##        "HG"        "WN"        "TM"        "KL"        "DK"        "23"
##     matrix7     matrix8     matrix9    matrix10  dataframe1  dataframe2
##        "28"        "19"        "34"        "26"        "HG"        "WN"
##  dataframe3  dataframe4  dataframe5  dataframe6  dataframe7  dataframe8
##        "TM"        "KL"        "DK"        "23"        "28"        "19"
##  dataframe9 dataframe10
##        "34"        "26"
```

Attributes

Attributes of an object are additional information about the object or its metadata. Attributes are classified as either intrinsic or non-intrinsic.

Intrinsic Attribute

Intrinsic attributes are referred as intrinsic because they are fundamental constituents of an object and therefore all data objects would have it (you do not need to create them, but you can modify them).

There two types of intrinsic attributes, **mode** and **length**.

Mode (Basic type) of an object

Mode() is the S version of **typeof()**; it gives the internal basic object type. Therefore, if you call mode() on an object, it would return the basic data type as those returned with typeof() with the exception that integers and double are not differentiated and return a mode numeric.

Atomic vectors as earlier noted must have all their elements of the same type and hence the same mode. List (generic vector) has the mode **list** but its elements can be of any mode.

Mode of an object can be changed through coercion with the **as.something()** group of function.

Length

All data objects have a length property which is the number of components or elements in an object. For one dimensional objects, the function **length** is used, for two dimensional objects like data frames and matrices **ncol()** and **nrow()** are used. **Dim** function is used for higher dimensional vectors (including 2 dimensional objects) like arrays and tables.

```
# Length of a one dimensional atomic vector
length(ages)
## [1] 5

# Length of a matrix
ncol(classData2)
## [1] 2
nrow(classData2)
## [1] 5
dim(classData2)
## [1] 5 2

# Length of an array
dim(myArray)
## [1] 2 2 2

# Length of a data frame
ncol(class_df)
## [1] 4
nrow(class_df)
## [1] 5
dim(class_df)
## [1] 5 4

# Length of a list (one dimension)
length(class_list)
## [1] 2
```

You can remove components/elements of a one dimensional object by assigning value to length.

```
# Reducing length
ages2 <- ages
length(ages2)
## [1] 5
length(ages2) <- 4
ages2
## [1] 23 28 19 34

class_list3 <- class_list2
length(class_list3) <- 2
class_list3
## $ListOne.matrix
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
##
```

```
## $ListOne.dataframe
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
```

We can also use the length of an object to add elements to the object.

```
# Adding to an atomic vector (1d)
ages2[length(ages2) + 1] <- 44
ages2
## [1] 23 28 19 34 44

# Adding an element to a list
class_list3[[length(class_list3) + 1]] <- myArray
class_list3
## $ListOne.matrix
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
##
## $ListOne.dataframe
##      [,1] [,2]
## [1,] "HG" "23"
## [2,] "WN" "28"
## [3,] "TM" "19"
## [4,] "KL" "34"
## [5,] "DK" "26"
##
## [[3]]
## , , Urban
##
##              Yes No
## Exposed       50 48
## Not Exposed   70 43
##
## , , Rural
##
##              Yes No
## Exposed       49 29
## Not Exposed   44 39
```

To add an element we had to use either single or double square brackets, this is referred to as indexing which we shall explore during our sixth session on transforming and manipulating data objects.

Non-intrinsic Attributes

There are a number of non-intrinsic attributes that an object can have, these include names, dimensions, dimnames, classes, ts (time series) and copying attributes.

Function **attributes()** can be used to establish non-intrinsic attributes of an object. For pure base type, there will be no non-intrinsic attributes.

```
attributes(ages)
## NULL
attributes(classData)
## $dim
## [1] 5 2
##
## $dimnames
## $dimnames[[1]]
## NULL
##
## $dimnames[[2]]
## [1] "nams" "ages"
attributes(myArray)
## $dim
## [1] 2 2 2
##
## $dimnames
## $dimnames[[1]]
## [1] "Exposed"     "Not Exposed"
##
## $dimnames[[2]]
## [1] "Yes" "No"
##
## $dimnames[[3]]
## [1] "Urban" "Rural"
attributes(class_df)
## $names
## [1] "nams"      "ages"      "gender"    "residence"
##
## $row.names
## [1] 1 2 3 4 5
##
## $class
## [1] "data.frame"
attributes(class_list)
## $names
## [1] "matrix"    "dataframe"
```

The important attributes that we shall discuss are **names**, **dimensions** and **classes**.

Name attribute

Elements in a vector can be named or not. These names can be assigned when creating a vector or modification after creation.

```
# Naming vector components during creation
(agesNams <- c(HG = 23, WN = 28, TM = 19, KL = 34, DK = 26))
## HG WN TM KL DK
## 23 28 19 34 26

# Adding names after vector creation
names(ages2) <- nams
ages2
## HG WN TM KL DK
## 23 28 19 34 44

# Creating a modified copy of a vector
agesNams2 <- setNames(c(23, 28, 19, 34, 26), nm = c("HG", "WN", "TM", "KL",
"DK"))
agesNams2
## HG WN TM KL DK
## 23 28 19 34 26
```

A vector can have some named elements and unnamed elements. When **names()** is called in such a circumstance, R will output an empty string (" ")in place of the missing name.

```
names(c(a = 1, b = 2, 3, d = 4, 5))
## [1] "a" "b" ""  "d" ""
```

When a vector has no names and names() is called, R will output a reserved word (object) **NULL** which means an undefined value.

```
names(1:5)
## NULL
```

Names are especially handy when dealing with higher dimensional objects, we shall see this when discussing subsetting.

Dimensions

Dimension is not a new term by now as we have used it to establish length. In simple terms dimension is the extent of a multi-dimensional object such as matrices, arrays and data frames. This is established with the **dim()** function.

Since we covered this during our discussion on length attribute, we shall leave it at that.

Classes

Classes in R are attributes used to facilitate implementation of **object oriented programming (OOP)**. They often determine how an object will be handled by special functions called **generic functions**.

In summary, generic functions store other unique functions called **methods** each of which act differently given a certain class of object. A typical example is the **plot** function. plot() is a generic function with 29 methods. When you call plot(), it will first determine the class of the object and then look for a method that can act on that classed object. You can see all the methods available for the plot function with **methods(plot)**. Notice these methods have a **plot.something**, the "something" is the class for which that method will act on. If you are keen enough, you are bound to see a method for data frame which means plot() can read a data frame and produce a graph by calling the method **plot.data.frame**. Also, do you notice there are no methods for the base type? This is because generic functions like plot() are not base R objects but S3 objects (S4 has generic functions but it is a recent addition so they have not been implemented in R). If an object of base type is passed in a call to a generic and a method is not explicitly defined for it, the default method (e.g. plot.default()) will be called. So in essence, a class of a vector is used to determine the method to be called by a generic function. Please read more on this from Object Oriented Programming in R.

All objects have a class attribute, at the very least it would have its basic type (those produced by typeof or mode). To determine the class of an object, **class()** is used.

```
# 1 dimension atomic vectors
class(ages)
## [1] "numeric"
class(nams)
## [1] "character"

# 2 dimension atomic vector
class(classData2)
## [1] "matrix"
class(class_df)
## [1] "data.frame"

# 1 dimension generic vector
class(class_list)
## [1] "list"
```

An object can have more than one class thereby having a vector of classes. A class can be added either directly with the **class() <-** assignment, through the **attr()** function or creating an object with **structure()** function. Note, when you add a class attribute to an object it ceases to be a base object and becomes one of the other OO systems in R. In our case they will be s3 objects as the classes are not defined. To establish this, before we add classes, let's get to know what type of objects we have. A package called **pryr** has some efficient functions to accomplish this task.

```
# We will use the pryr package (install it first)
library(pryr)

# class = numeric
otype(ages)
## [1] "base"
# typeof = character
```

```
otype(nams)
## [1] "base"
# class = matrix
otype(classData2)
## [1] "base"
# class = data.frame
otype(class_df)
## [1] "S3"
# class = list
otype(class_list)
## [1] "base"
```

Numeric, character, matrix and lists are base objects and surprisingly only data frame is an s3 object. We can now add a class to the base object and have the data frame to have a class vector.

```
# Simple atomic vector
class(ages) <- "myNumeric"
class(ages)
## [1] "myNumeric"

# Matrix
class(classData2) <- "myMatrix"
class(classData2)
## [1] "myMatrix"

# data frame
class(class_df) <- c("ts", "glm")
class(class_df)
## [1] "ts"  "glm"

# List
class(class_list) <- "myList"
class(class_list)
## [1] "myList"
```

Now let's confirm they are all s3 type due to addition of a class attribute.

```
otype(ages)
## [1] "S3"
otype(classData2)
## [1] "S3"
otype(class_df)
## [1] "S3"
otype(class_list)
## [1] "S3"
```

Classes can be removed with **unclass()** function, however, this is highly discouraged as there are no checking mechanism for content conformity (applies to S3 classes) hence have implications on method dispatch.

```r
# Unclassing a data frame
aq <- unclass(airquality)
class(aq)
## [1] "list"

# Unclassing a time series data
am <- unclass(airquality)
class(am)
## [1] "list"
```

Unclassing these two objects (data frame and time series) has lead to a list, if they are passed to a generic function, the generic will search for methods that work with lists rather than data frames or time series.

Other ways of adding S3 classes

```r
# Using the attri function
attr(aq, "class") <- "data.frame"
class(aq)
## [1] "data.frame"

# By creating a new object with structure function
alph <- structure(letters[1:10], class = "myFactor")
```

Gauge yourself

Have you mastered R's data types and Objects?

1. List R's basic data types
2. What data type do you expect from these vectors (attempt to answer them before running the code) a. c("a", 2i, 4L, TRUE) a. c(2, FALSE) a. c(2.5, 3L) a. c(2, 1, NA, 6) a. factor(c("Yes", "No"))
3. Create a logical vector that says TRUE if the vector **ages** is equal or above the mean and FALSE if it is below the mean. Tip, read documentation on comparison operators **?Comparison**
4. What do NA and NULL mean and how do they differ
5. What is a vector? How many types of vectors are there?
6. How are basic atomic vectors created?
7. How do you assign names to a vector?
8. List R's reserved words
9. Which of these object names would raise an error
   a. oneOfMyVector
   b. 1ofmyvectors
   c. one.of.my.vectors
10. What are matrices and array,
11. What are data frames and lists
12. List the properties of a data object

13. What is the difference between typeof() and mode()
14. How can you determine the length of one dimensional object and of multi-dimensional objects
15. Which are the key non-intrinsic attributes
16. Why are classes important in R?

You are ready for the fifth session if you know these

1. There are six basic data types in R. These are logical, integers, double/real, characters, complex and raw
2. Expected data type a. c("a", 2i, 4L, TRUE) = Character a. c(2, FALSE) = Double a. c(2.5, 3L) = Double a. c(2, 1, NA, 6) = Double a. factor(c("Yes", "No")) = Integer
3. Code to create a logical vector

```
m <- mean(ages)
ages >= m
## [1] FALSE  TRUE FALSE  TRUE  TRUE
```

4. NA is a logical constant that means **Not Available**, it is used as a marker for missing value. NULL is an object that signifies unidentified value
5. A vector is R's basic data structure. It is created with combine function **c()**
6. Assignment can be done in three ways, using: a. "arrow" **<-** (short version of assign function and highly recommended) a. assignment function **assign()** (useful in function definition) a. Equals sign **=** (though not recommended)
7. R's reserved words include; if, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, NaN, NA, NA_integer_, NA_real_, NA_complex_, NA_character_, ... etc.
8. Object name that would raise an error is b) 1ofmyvectors because a name cannot start with a number
9. A matrix is a two dimensional vector. Arrays are multi-dimensional vectors of which matrices is one of them. They are atomic vectors meaning they can only have one type of data.
10. Data frames are two dimensional generic vectors while lists are one dimensional generic vectors. Data frames and lists can contain different types of data
11. Objects in R can have two properties, its type and attributes or additional information about the object
12. Mode is an **S** implementation of typeof(). Both calls would output the same data type with the exception of integer and double which have a mode numeric
13. You can get the length of a one dimensional object with the function **length()** and **dim()** for multi-dimensional objects
14. The three most important non-intrinsic attribute are **name**, **dimension** and **class**, they are preserved during object transformation
15. Class of an object is important in R as it is used to implement Object Oriented Programming (R)