Unit 1.1: Big data significance, complexity and data analysis

BIG DATA:

Big data is a term that describes the large volume of data – both structured and unstructured – that inundates a business on a day-to-day basis. But it's not the amount of data that's important. It's what organizations do with the data that matters. Big data can be analyzed for insights that lead to better decisions and strategic business moves.

IMPORTANCE/SIGNIFICANCE OF BIG DATA

- The use of Big Data is becoming common these days by the companies to outperform their peers.
- In most industries, existing competitors and new entrants alike will use the strategies resulting from the analyzed data to compete, innovate and capture value.
- Big Data helps the organizations to create new growth opportunities and entirely new categories of companies that can combine and analyze industry data.
- These companies have ample information about the products and services, buyers and suppliers, consumer preferences that can be captured and analyzed.

COMPLEXITY OF DATA AND DATA ANALYSIS

Complex data analytics refers to the use of advanced algorithmic methods to process large unstructured data sets effectively. Computers do analytical processing of data, in the past, this was largely about individual machines acting on individual well-defined data structures. We called this data analytics, which is the use of computers to analyze data and find meaningful patterns within it that can be used to make decisions. Today computing is evolving to cloud platforms, advanced algorithms, and big data and we can call this advanced analytics or complex analytics. Complex data analytics is the use of advanced algorithms to process big data structures.

With the convergence of cloud computing platforms, advances in algorithms, the growth of unlabeled big data sources and now the internet of things the revolution in information is entering a new stage, with the capacities of information technology greatly expanding. The creation of personal computing, the internet, and mobile devices has created a flood of new data sources.

In response, computing is moving up from individual machines with well-defined instructions acting on well defined individual data sets, to now running on clusters of machines, on massive amounts of unstructured data, using qualitatively different algorithms in the form of machine learning. In this process, we are collecting ever more data about ever more aspects of our world, we bring that into data centers and apply ever more sophisticated mathematical models and computer science methods to bear on building algorithms that allow us to look into this big data, to see what we have never seen before. A world that was previously only accessible through our imagination is being presented to us as real data and visualizations.

BIG DATA CHARACTERISTICS

1. Volume:

   Volume is the amount of data generated that must be understood to make data-based decisions. A text file is a few kilobytes, a sound file is a few megabytes while a full-length movie is a few gigabytes.

   Example:

   Amazon handles 15 million customer click stream user data per day to recommend products. Extremely **large volume of data** is a major characteristic of **big data online training**

2. Velocity:

   Velocity measures how fast data is produced and modified and the speed with which it needs to be processed. An increased number of data sources both machine and human generated drive velocity.

   Example:

   72 hours of video are uploaded to **YouTube** every minute this is the velocity. Extremely high velocity of data is another major big data characteristics

3. Variety:

   Variety defines data coming from new source both inside and outside of an enterprise It can be structured, semi-structured or unstructured.

   Structured data:

   It is typically found in tables with columns and rows of data. The intersection of the row and the column in a cell has a value and is given a “key,” which it can be referred to in queries. Because there is a direct relationship between the column and the row, these databases are commonly referred to as relational databases. A retail outlet that stores their sales data (name of person, product sold, amount) in an Excel spreadsheet or CSV file is an example of structured data.

   Example:

   A Product table in a database is an example of Structured Data

Product_id	Product_name	Product_price

1	Pen	$5.95
2	Paper	$8.95

Semi-structured data also has an organization, but the table structure is removed so the data can be more easily read and manipulated. XML files or an RSS feed for a webpage are examples of semi-structured data.

Example: XML file

Example:

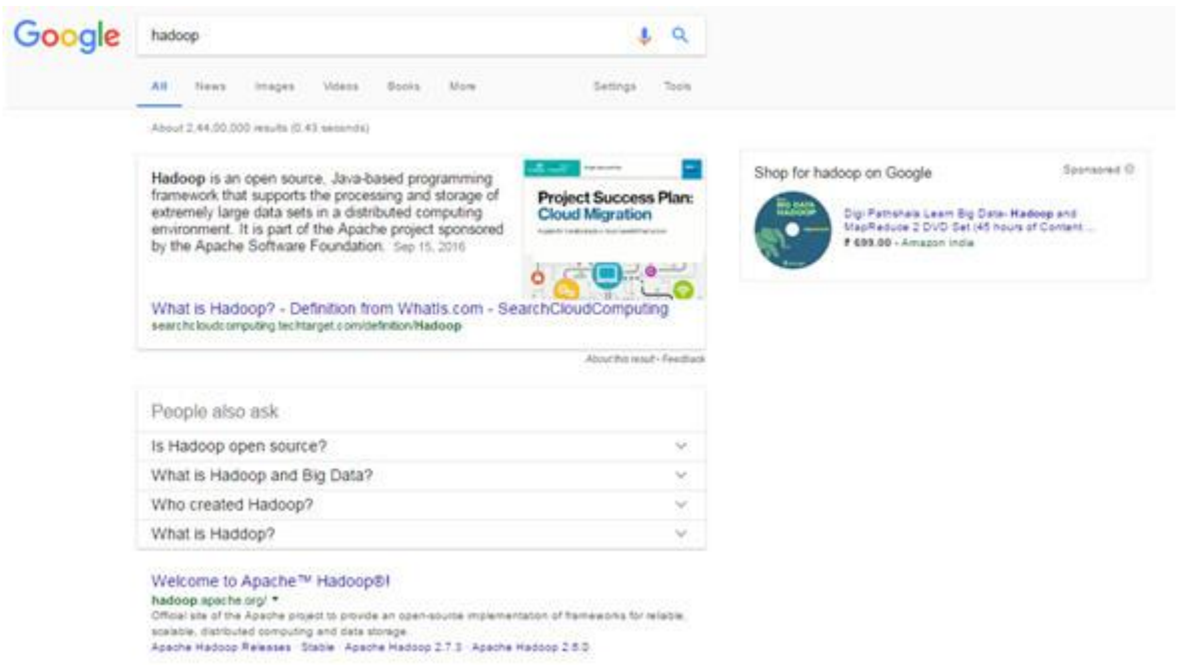
```
<product>

<name>Pen </name>

<price>$7.95</price>

</product>

<product>

<name>Paper </name>

<price>$8.95</price>

</product>
```

Unstructured data:

Unstructured data generally has no organizing structure, and Big Data technologies use different ways to add structure to this data. Typical example of unstructured data is, a heterogeneous data source containing a combination of simple text files, images, videos etc

Example:

Output returned by '**Google Search**'



About 2,44,00,000 results (0.43 seconds)

Hadoop is an open source, Java-based programming framework that supports the processing and storage of extremely large data sets in a distributed computing environment. It is part of the Apache project sponsored by the Apache Software Foundation. Sep 15, 2016

Project Success Plan: Cloud Migration

What is Hadoop? - Definition from WhatIs.com - SearchCloudComputing
searchcloudcomputing.techtarget.com/definition/**Hadoop**

About this result · Feedback

People also ask

Is Hadoop open source?

What is Hadoop and Big Data?

Who created Hadoop?

What is Haddop?

Welcome to Apache™ Hadoop®!
hadoop.apache.org/ ▾
Official site of the Apache project to provide an open-source implementation of frameworks for reliable, scalable, distributed computing and data storage.
Apache Hadoop Releases · Stable · Apache Hadoop 2.7.3 · Apache Hadoop 2.6.0

Shop for hadoop on Google Sponsored

Digi Pathshala Learn Big Data- **Hadoop** and MapReduce 2 DVD Set (45 hours of Content ...
₹ 699.00 - Amazon India

4. Variability

This refers to the inconsistency which can be shown by the data at times, thus hampering the process of being able to handle and manage the data effectively.

You can see that few values are missing in the below table

Department	Year	Minimum sales	Maximum sales
1	2010	?	1500
2	2011	10000	?

Data available can sometimes get messy and maybe difficult to trust. With wide variety in big data types generated, quality and accuracy are difficult to control.

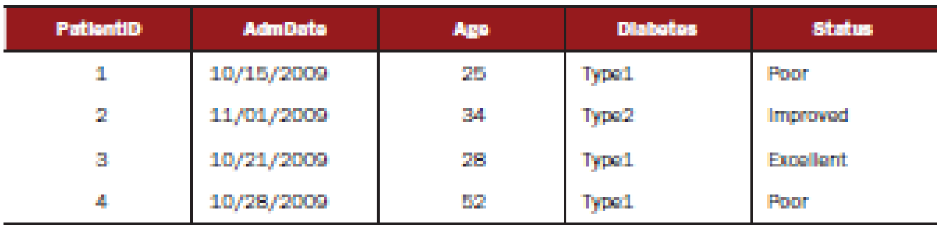
Example: A Twitter post has hashtags, typos and abbreviations

Unit 1.3: Basic of R

DATASET CREATION

A dataset is usually a rectangular array of data with rows representing observations and columns representing variables.

Table 2.1 A patient dataset



PatientID	AdmDate	Age	Diabetes	Status
1	10/15/2009	25	Type1	Poor
2	11/01/2009	34	Type2	Improved
3	10/21/2009	28	Type1	Excellent
4	10/28/2009	52	Type1	Poor

Different traditions have different names for the rows and columns of a dataset. Statisticians refer to them as observations and variables, database analysts call them records and fields, and those from the data mining/machine learning disciplines call them examples and attributes.

R contains a wide variety of structures for holding data, including scalars, vectors, arrays, data frames, and lists. Table 2.1 corresponds to a data frame in R. This diversity of structures provides the R language with a great deal of flexibility in dealing with data. The data types or modes that R can handle include numeric, character, logical (TRUE/FALSE), complex (imaginary numbers), and raw (bytes). In R, PatientID, AdmDate, and Age would be numeric variables, whereas Diabetes and Status would be character variables.

DATA STRUCTURES

- Vector
- Matrix
- Array
- Dataframe
- List

2.2.1 Vectors

Vectors are one-dimensional arrays that can hold numeric data, character data, or logical data. The combine function c() is used to form the vector. Here are examples of each type of vector:

a <- c(1, 2, 5, 3, 6, -2, 4)

b <- c("one", "two", "three")

c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)

Here, a is numeric vector, b is a character vector, and c is a logical vector. Note that the data in a vector must only be one type or mode (numeric, character, or logical). You can't mix modes in the same vector.

NOTE Scalars are one-element vectors. Examples include f <- 3, g <- "US" and h <- TRUE. They're used to hold constants.

You can refer to elements of a vector using a numeric vector of positions within brackets. For example, a[c(2, 4)] refers to the 2nd and 4th element of vector a. Here are additional examples:

```
> a <- c(1, 2, 5, 3, 6, -2, 4)
> a[3]
[1] 5
> a[c(1, 3, 5)]
[1] 1 5 6
> a[2:6]
[1] 2 5 3 6 -2
```

The colon operator used in the last statement is used to generate a sequence of numbers.

For example, a <- c(2:6) is equivalent to a <- c(2, 3, 4, 5, 6).

2.2.2 Matrices

A matrix is a two-dimensional array where each element has the same mode (numeric, character, or logical). Matrices are created with the matrix function . The general format is

```
myymatrix <- matrix(vector, nrow=number_of_rows, ncol=number_of_columns,
byrow=logical_value, dimnames=list(
char_vector_rownames, char_vector_colnames))
```

where vector contains the elements for the matrix, nrow and ncol specify the row and column dimensions, and dimnames contains optional row and column labels stored in character vectors. The option byrow indicates whether the matrix should be filled in by row (byrow=TRUE) or by column (byrow=FALSE). The default is by column. Thefollowing listing demonstrates the matrix function .

Listing 2.1 Creating matrices

```
> y <- matrix(1:20, nrow=5, ncol=4) q
> y
[,1] [,2] [,3] [,4]
[1,] 1 6 11 16
[2,] 2 7 12 17
[3,] 3 8 13 18
[4,] 4 9 14 19
[5,] 5 10 15 20
> cells <- c(1,26,24,68)
> rnames <- c("R1", "R2")
> cnames <- c("C1", "C2") w
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,
dimnames=list(rnames, cnames))
> mymatrix
C1 C2
R1 1 26
R2 24 68
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=FALSE,
dimnames=list(rnames, cnames))
> mymatrix e
```

```
C1 C2
R1 1 24
R2 26 68
```

First, you create a 5x4 matrix q. Then you create a 2x2 matrix with labels and fill the matrix by rows w. Finally, you create a 2x2 matrix and fill the matrix by columns e. You can identify rows, columns, or elements of a matrix by using subscripts and brackets. X[i,] refers to the ith row of matrix X, X[,j] refers to jth column, and X[i, j] refers to the ijth element, respectively. The subscripts i and j can be numeric vectors in order to select multiple rows or columns, as shown in the following listing.

Listing 2.2 Using matrix subscripts

```
> x <- matrix(1:10, nrow=2)
> x
[,1] [,2] [,3] [,4] [,5]
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
> x[2,]
[1] 2 4 6 8 10
> x[,2]
[1] 3 4
> x[1,4]
[1] 7
> x[1, c(4,5)]
[1] 7 9
```

Create a 5x4 matrix
2x2 matrix filled
by rows
2x2 matrix filled
by columns

First a 2 x 5 matrix is created containing numbers 1 to 10. By default, the matrix is filled by column. Then the elements in the 2nd row are selected, followed by the elements
in the 2nd column. Next, the element in the 1st row and 4th column is selected.
Finally, the elements in the 1st row and the 4th and 5th columns are selected.
Matrices are two-dimensional and, like vectors, can contain only one data type.
When there are more than two dimensions, you'll use arrays (section 2.2.3). When there are multiple modes of data, you'll use data frames (section 2.2.4).

2.2.3 Arrays

Arrays are similar to matrices but can have more than two dimensions. They're created
with an array function of the following form:

```
myarray <- array(vector, dimensions, dimnames)
```

where vector contains the data for the array, dimensions is a numeric vector giving the maximal index for each dimension, and dimnames is an optional list of dimension labels. The following listing gives an example of creating a three-dimensional (2x3x4) array of numbers.

Listing 2.3 Creating an array

```
> dim1 <- c("A1", "A2")
> dim2 <- c("B1", "B2", "B3")
> dim3 <- c("C1", "C2", "C3", "C4")
> z <- array(1:24, c(2, 3, 4), dimnames=list(dim1, dim2, dim3))
> z
, , C1
B1 B2 B3
A1 1 3 5
A2 2 4 6
, , C2
B1 B2 B3
A1 7 9 11
A2 8 10 12
, , C3
B1 B2 B3
A1 13 15 17
A2 14 16 18
, , C4
B1 B2 B3
A1 19 21 23
A2 20 22 24
```

As you can see, arrays are a natural extension of matrices. They can be useful in programming new statistical methods. Like matrices, they must be a single mode. Identifying elements follows what you've seen for matrices. In the previous example, the z[1,2,3] element is 15.

2.2.4 Data frames

A data frame is more general than a matrix in that different columns can contain different modes of data (numeric, character, etc.). It's similar to the datasets you'd typically see in SAS, SPSS, and Stata. Data frames are the most common data structure
you'll deal with in R.

The patient dataset in table 2.1 consists of numeric and character data. Because there are multiple modes of data, you can't contain this data in a matrix. In this case, a data frame would be the structure of choice.

A data frame is created with the data.frame() function :

```
mydata <- data.frame(col1, col2, col3,…)
```

where col1, col2, col3, … are column vectors of any type (such as character, numeric, or logical). Names for each column can be provided with the names function.

The following listing makes this clear.

Listing 2.4 Creating a data frame

```
> patientID <- c(1, 2, 3, 4)
> age <- c(25, 34, 28, 52)
> diabetes <- c("Type1", "Type2", "Type1", "Type1")
> status <- c("Poor", "Improved", "Excellent", "Poor")
> patientdata <- data.frame(patientID, age, diabetes, status)
> patientdata
patientID age diabetes status
1 1 25 Type1 Poor
2 2 34 Type2 Improved
3 3 28 Type1 Excellent
4 4 52 Type1 Poor
```

Each column must have only one mode, but you can put columns of different modes together to form the data frame. Because data frames are close to what analysts typically think of as datasets, we'll use the terms columns and variables interchangeably when discussing data frames.

There are several ways to identify the elements of a data frame. You can use the subscript notation you used before (for example, with matrices) or you can specify column names. Using the patientdata data frame created earlier, the following listing demonstrates these approaches.

Listing 2.5 Specifying elements of a data frame

```
> patientdata[1:2]
patientID age
1 1 25
2 2 34
3 3 28
4 4 52
> patientdata[c("diabetes", "status")]
diabetes status
1 Type1 Poor
2 Type2 Improved
3 Type1 Excellent q
4 Type1 Poor
> patientdata$age
[1] 25 34 28 52
```

The $ notation in the third example is new q. It's used to indicate a particular variable from a given data frame. For example, if you want to cross tabulate diabetes type by status, you could use the following code:

```
> table(patientdata$diabetes, patientdata$status)
Excellent Improved Poor
Type1 1 0 2
Type2 0 1 0
```

It can get tiresome typing patientdata$ at the beginning of every variable name, so shortcuts are available. You can use either the attach() and detach() or with() functions to simplify your code.

ATTACH, DETACH, AND WITH

The attach() function adds the data frame to the R search path. When a variable name is encountered, data frames in the search path are checked in order to locate the variable. Using the mtcars data frame from chapter 1 as an example, you could use the following code to obtain summary statistics for automobile mileage (mpg), and plot this variable against engine displacement (disp), and weight (wt):

```
summary(mtcars$mpg)
plot(mtcars$mpg, mtcars$disp)
plot(mtcars$mpg, mtcars$wt)
```

This could also be written as

```
attach(mtcars)
summary(mpg)
plot(mpg, disp)
plot(mpg, wt)
detach(mtcars)
```

The detach() function removes the data frame from the search path. Note that detach() does nothing to the data frame itself. The statement is optional but is good programming practice and should be included routinely. (I'll sometimes ignore this sage advice in later chapters in order to keep code fragments simple and short.)

The limitations with this approach are evident when more than one object can have the same name. Consider the following code:

```
> mpg <- c(25, 36, 47)
> attach(mtcars)
The following object(s) are masked _by_ '.GlobalEnv': mpg
```

Indicates age variable in patient data frame

```
> plot(mpg, wt)
Error in xy.coords(x, y, xlabel, ylabel, log) :
'x' and 'y' lengths differ
> mpg
[1] 25 36 47
```

Here we already have an object named mpg in our environment when the mtcars data frame is attached. In such cases, the original object takes precedence, which isn't what you want. The plot statement fails because mpg has 3 elements and disp has 32 elements.

The attach() and detach() functions are best used when you're analyzing a single data frame and you're unlikely to have multiple objects with the same name. In any case, be vigilant for warnings that say that objects are being masked. An alternative approach is to use the with() function. You could write the previous example as

```
with(mtcars, {
summary(mpg, disp, wt)
plot(mpg, disp)
plot(mpg, wt)
})
```

In this case, the statements within the {} brackets are evaluated with reference to the mtcars data frame. You don’t have to worry about name conflicts here. If there’s only one statement (for example, summary(mpg)), the {} brackets are optional. The limitation of the with() function is that assignments will only exist within the function brackets. Consider the following:

```
> with(mtcars, {
stats <- summary(mpg)
stats
})
Min. 1st Qu. Median Mean 3rd Qu. Max.
10.40 15.43 19.20 20.09 22.80 33.90
> stats
Error: object ‘stats’ not found
```

If you need to create objects that will exist outside of the with() construct, use the special assignment operator <<- instead of the standard one (<-). It will save the object to the global environment outside of the with() call. This can be demonstrated with the following code:

```
> with(mtcars, {
nokeepstats <- summary(mpg)
keepstats <<- summary(mpg)
})
> nokeepstats
Error: object ‘nokeepstats’ not found
> keepstats
Min. 1st Qu. Median Mean 3rd Qu. Max.
10.40 15.43 19.20 20.09 22.80 33.90
```

Most books on R recommend using with() over attach(). I think that ultimately the choice is a matter of preference and should be based on what you’re trying to achieve and your understanding of the implications. We’ll use both in this book.

CASE IDENTIFIERS

In the patient data example, patientID is used to identify individuals in the dataset. In R, case identifiers can be specified with a rowname option in the data frame function.

For example, the statement

```
patientdata <- data.frame(patientID, age, diabetes, status,
row.names=patientID)
```

specifies patientID as the variable to use in labeling cases on various printouts and graphs produced by R.

2.2.5 Factors

As you’ve seen, variables can be described as nominal, ordinal, or continuous. Nominal variables are categorical, without an implied order. Diabetes (Type1, Type2) is an example of a nominal variable. Even if Type1 is coded as a 1 and Type2 is coded as a 2 in the data, no order is implied. Ordinal variables imply order but not amount.

Status (poor, improved, excellent) is a good example of an ordinal variable. You know that a patient with a poor status isn't doing as well as a patient with an improved status, but not by how much.

Continuous variables can take on any value within some range, and both order and amount are implied. Age in years is a continuous variable and can take on values such as 14.5 or 22.8 and any value in between. You know that someone who is 15 is one year older than someone who is 14.

Categorical (nominal) and ordered categorical (ordinal) variables in R are called factors. Factors are crucial in R because they determine how data will be analyzed and presented visually. You'll see examples of this throughout the book. The function factor() stores the categorical values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

For example, assume that you have the vector

diabetes <- c("Type1", "Type2", "Type1", "Type1")

The statement *diabetes <- factor(diabetes)* stores this vector as (1, 2, 1, 1) and associates it with 1=Type1 and 2=Type2 internally (the assignment is alphabetical). Any analyses performed on the vector diabetes will treat the variable as nominal and select the statistical methods appropriate for this level of measurement. For vectors representing ordinal variables, you add the parameter ordered=TRUE to
the factor() function . **Given the vector**

status <- c("Poor", "Improved", "Excellent", "Poor")

the statement status <- factor(status, ordered=TRUE) will encode the vector as (3, 2, 1, 3) and associate these values internally as 1=Excellent, 2=Improved, and
3=Poor.

Additionally, any analyses performed on this vector will treat the variable as ordinal and select the statistical methods appropriately. By default, factor levels for character vectors are created in alphabetical order. This worked for the status factor, because the order "Excellent," "Improved," "Poor" made sense. There would have been a problem if "Poor" had been coded as "Ailing" instead, because the order would be "Ailing," "Excellent," "Improved." A similar problem exists if the desired order was "Poor," "Improved," "Excellent." For ordered factors, the alphabetical default is rarely sufficient.

You can override the default by specifying a levels option. **For example**,

status <- factor(status, order=TRUE,
levels=c("Poor", "Improved", "Excellent"))

would assign the levels as 1=Poor, 2=Improved, 3=Excellent. Be sure that the specified levels match your actual data values. Any data values not in the list will be set to missing. The following listing demonstrates how specifying factors and ordered factors impact data analyses.

Listing 2.6 Using factors

> patientID <- c(1, 2, 3, 4) q

> age <- c(25, 34, 28, 52)

> diabetes <- c("Type1", "Type2", "Type1", "Type1")

```
> status <- c("Poor", "Improved", "Excellent", "Poor")
> diabetes <- factor(diabetes)
> status <- factor(status, order=TRUE)
> patientdata <- data.frame(patientID, age, diabetes, status)
> str(patientdata)
'data.frame': 4 obs. of 4 variables:
$ patientID: num 1 2 3 4 w
$ age : num 25 34 28 52
$ diabetes : Factor w/ 2 levels "Type1","Type2": 1 2 1 1
$ status : Ord.factor w/ 3 levels "Excellent"<"Improved"<..: 3 2 1 3
> summary(patientdata)
patientID age diabetes status
Min. :1.00 Min. :25.00 Type1:3 Excellent:1 e
1st Qu.:1.75 1st Qu.:27.25 Type2:1 Improved :1
Median :2.50 Median :31.00 Poor :2
Mean :2.50 Mean :34.75
3rd Qu.:3.25 3rd Qu.:38.50
Max. :4.00 Max. :52.00
```

First, you enter the data as vectors q. Then you specify that diabetes is a factor and status is an ordered factor. Finally, you combine the data into a data frame. The function str(object) provides information on an object in R (the data frame in this case) w. It clearly shows that diabetes is a factor and status is an ordered factor, along with how it's coded internally. Note that the summary() function treats the variables differently e. It provides the minimum, maximum, mean, and quartiles for the continuous variable age, and frequency counts for the categorical variables diabetes and status.

Enter data as vectors
Display object structure
Display object summary

2.2.6 Lists

Lists are the most complex of the R data types. Basically, a list is an ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name. For example, a list may contain a combination of vectors, matrices, data frames, and even other lists. You create a list using the list() function :

```
mylist <- list(object1, object2, ...)
```

where the objects are any of the structures seen so far. Optionally, you can name the objects in a list:

```
mylist <- list(name1=object1, name2=object2, ...)
```

The following listing shows an example.

Listing 2.7 Creating a list

```
> g <- "My First List"
> h <- c(25, 26, 18, 39)
> j <- matrix(1:10, nrow=5)
> k <- c("one", "two", "three")
> mylist <- list(title=g, ages=h, j, k)
> mylist
$title
[1] "My First List"
$ages
[1] 25 26 18 39
[[3]]
[,1] [,2]
[1,] 1 6
[2,] 2 7
[3,] 3 8
[4,] 4 9
[5,] 5 10
[[4]]
[1] "one" "two" "three"
> mylist[[2]]
[1] 25 26 18 39
> mylist[["ages"]]
[[1] 25 26 18 39
```

In this example, you create a list with four components: a string, a numeric vector, a matrix, and a character vector. You can combine any number of objects and save them as a list.

You can also specify elements of the list by indicating a component number or a name within double brackets. In this example, mylist[[2]] and mylist[["ages"]] both refer to the same four-element numeric vector. Lists are important R structures

Create list

Print entire list

Print second
component