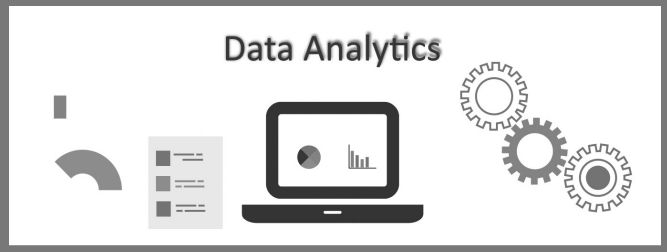
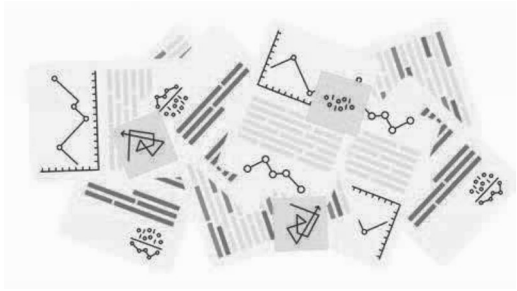
Introduction to Data Analytics



Module – 1 Syllabus

- Data-Information
- characteristics of data
- data munging
- Scraping
- Sampling
- Cleaning
- importance of data analytics

Data Analysis



- Data analysis is a process of obtaining raw data and converting it into information useful for decision-making by users.

Data Analytics

- Data Analytics the science of examining raw data with the purpose of drawing conclusions about that information

Applications of Analytics

- In commercial industries, to enable organizations to make more-informed business decisions and by scientists
- By researchers, to verify or disprove scientific models, theories and hypotheses.

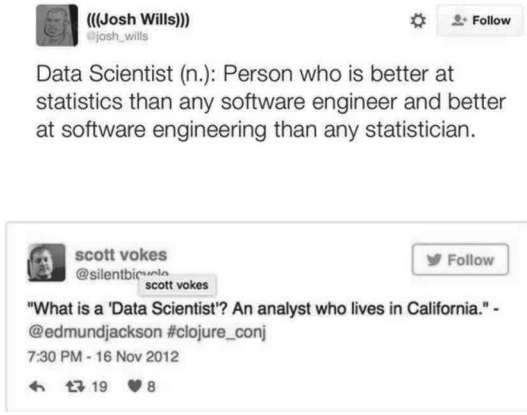
Analysis vs Analytics

- Data analytics is a broader term and includes data analysis as necessary subcomponent.
- Analytics defines the science behind the analysis.

Analysis vs Analytics

- The science means understanding the cognitive processes an analyst uses to understand problems and explore data in meaningful ways.
- Analytics also include data extract, transform, and load; specific tools, techniques, and methods; and how to successfully communicate results.

Data Science???



Data Science

- Dealing with unstructured and structured data, Data Science is a field that comprises of everything that related to data cleansing, preparation, and analysis.
- Involves in creation of new algorithms

Data vs Information

- Data are the facts or details from which information is derived.
- Individual pieces of data are rarely useful alone.
- For data to become information, data needs to be put into context.

Data vs Information

Comparison Chart

BASIS FOR COMPARISON	DATA	INFORMATION
Meaning	Data means raw facts gathered about someone or something, which is bare and random.	Facts, concerning a particular event or subject, which are refined by processing is called information.
What is it?	It is just text and numbers.	It is refined data.
Based on	Records and Observations	Analysis
Form	Unorganized	Organized
Useful	May or may not be useful.	Always
Specific	No	Yes
Dependency	Does not depend on information.	Without data, information cannot be processed.

Characteristics of data

The seven characteristics that define data quality are:

1. Accuracy and Precision
2. Legitimacy and Validity
3. Reliability and Consistency
4. Timeliness and Relevance
5. Completeness and Comprehensiveness
6. Availability and Accessibility
7. Granularity and Uniqueness

Characteristics of data

Accuracy and Precision: This characteristic refers to the exactness of the data.

Legitimacy and Validity: Requirements governing data set the boundaries of this characteristic.

Characteristics of data

Reliability and Consistency: Regardless of what source collected the data or where it resides, it cannot contradict a value residing in a different source or collected by a different system.

Timeliness and Relevance: Data collected too soon or too late could misrepresent a situation and drive inaccurate decisions.

Characteristics of data

Completeness and Comprehensiveness: Incomplete data is as dangerous as inaccurate data.

Availability and Accessibility: This presumes that the data exists and is available for access to be granted.

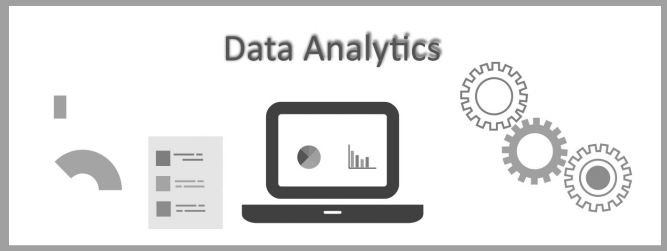
Characteristics of data

Completeness and Comprehensiveness: Incomplete data is as dangerous as inaccurate data.

Availability and Accessibility: This presumes that the data exists and is available for access to be granted.

Granularity and Uniqueness: The level of detail at which data is collected is important, because confusion and inaccurate decisions can otherwise occur.

Introduction to Data Analytics



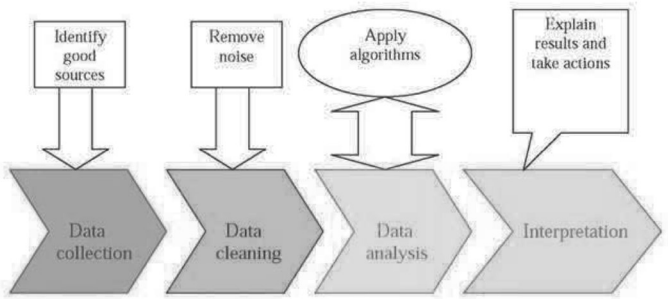
Topics of this session

- Data munging
- Scraping
- Sampling
- Cleaning

Recap of Measures of Data Quality

- **Accuracy:** correct or wrong, accurate or not
- **Completeness:** not recorded, unavailable
- **Consistency:** some modified but some not, dangling
- **Timeliness:** timely update?

Steps involved in Data Analysis



Data Munging

- Required for improving the quality of gathered data
- Involves **cleaning** and **transformation** of messy data available with us
- Also referred as **Data Wrangling**

Data Munging

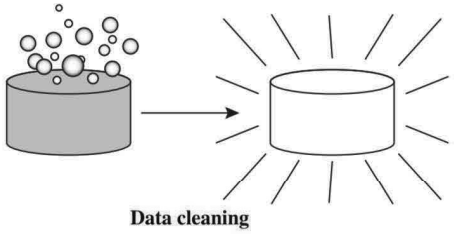
Before Data Munging

After Data Munging

Reason for noise in data

- Data in the Real World Is Dirty
- Lots of potentially incorrect data
  - Faulty instruments
  - Human or computer error
  - Transmission error

Data Cleaning



Data cleaning

Some examples for noisy data

- **Incomplete:** lacking attribute values, lacking certain attributes of interest, or containing only aggregate data

Ex: Occupation=“ ” (missing data)

- **Noisy:** containing noise, errors, or outliers

Ex: Salary=“−10” (an error)

Some examples for noisy data

- **Inconsistent:** containing discrepancies in codes or names, discrepancy between duplicate records

Ex:

1. Age=“42”, Birthday=“03/07/2010”
2. Was rating “1, 2, 3”, now rating “A, B, C”

- **Intentional:** disguised missing data

Ex: Jan. 1 as everyone’s birthday?

Data Cleaning

- Fill in missing values
- Smooth noisy data
- Identify or remove outliers
- Resolve inconsistencies

Data Cleaning (Dealing with Missing Values)

- Ignore the tuple
- Fill in the missing value manually
- Fill in it automatically with
  - a global constant
  - the attribute mean
  - the attribute mean for all samples belonging to the same class
  - the most probable value

Data Cleaning (Dealing with Noise)

- **Binning**
  - first sort data and partition into (equal-frequency) bins
  - then one can smooth by bin means, smooth by bin median, smooth by bin boundaries, etc.
- **Regression**
  - smooth by fitting the data into regression functions

Data Cleaning (Removing Outliers)

- **Clustering**
  - detect and remove outliers

Data Cleaning (Dealing with inconsistencies)

- **Combined computer and human inspection**
  - detect suspicious values and check by human (e.g., deal with possible outliers)

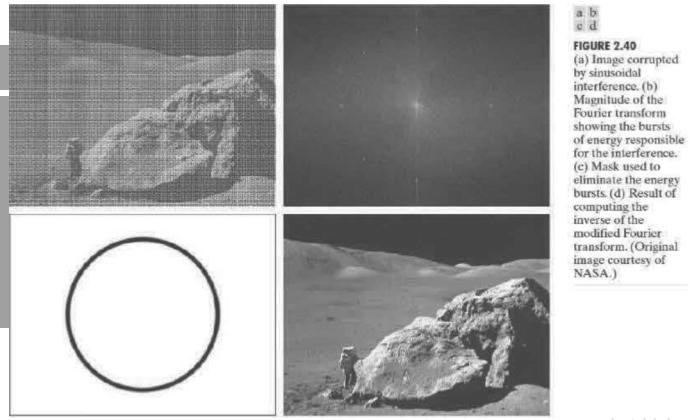
Data Transformation

- Transformation is mapping the Data to a new space

Ex:

- Fourier Transform
- Wavelet Transform

Data Transformation

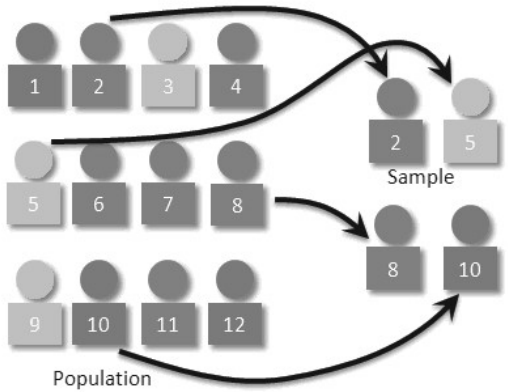


a b
c d

FIGURE 2.40
(a) Image corrupted by sinusoidal interference. (b) Magnitude of the Fourier transform showing the bursts of energy responsible for the interference. (c) Mask used to eliminate the energy bursts. (d) Result of computing the inverse of the modified Fourier transform. (Original image courtesy of NASA.)

Data Sampling

Sampling: obtaining a small sample *s* to represent the whole data set *N*

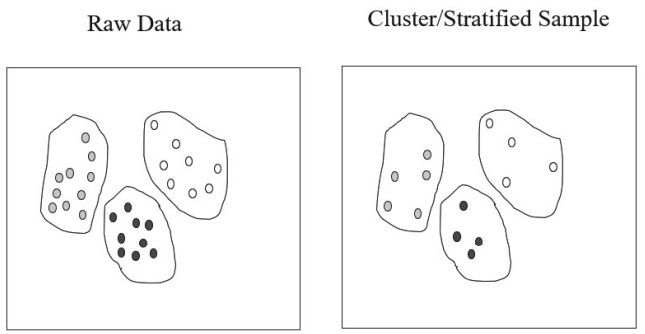


Types of Sampling

- **Simple random sampling:**
  - There is an equal probability of selecting any particular item
- **Stratified sampling:**
  - Partition the data set, and draw samples from each partition (proportionally, i.e., approximately the same percentage of the data)
  - Used in conjunction with skewed data

Types of Sampling

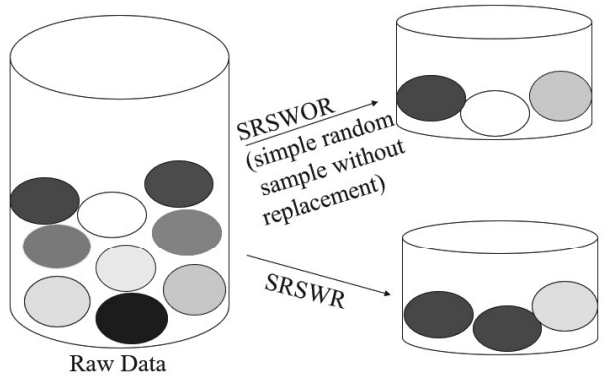
Stratified Sampling



Types of Sampling

- **Sampling without replacement**
  - Once an object is selected, it is removed from the population
- **Sampling with replacement**
  - A selected object is not removed from the population

Types of Sampling



Data Cleaning
Problem Solving

Data

Age	%Fat
23	9.5
23	26.5
27	7.8
27	17.8
39	31.4
41	25.9
47	27.4
49	27.2
50	31.2
52	34.6
54	42.5
54	28.8
56	33.4
57	30.2
58	34.1
58	32.9
60	41.2
61	35.7

Measures of Centrality

- Mean
- Median
- Mode
- Range
- Standard Deviation
- Variance

Find the above values for *age* and *%fat*

Mean, Median, Mode and Range www.cazoommaths.com

Mean
Add all the numbers then divide by the amount of numbers
9, 3, 1, 8, 3, 6
9 + 3 + 1 + 8 + 3 + 6 = 30
30 ÷ 6 = 5
The mean is 5

Median
Order the set of numbers, the median is the middle number
9, 3, 1, 8, 3, 6
1, 3, 3, 6, 8, 9
The median is 4.5

Mode
The most common number
9, 3, 1, 8, 3, 6
The mode is 3

Range
The difference between the highest number and lowest number
9, 3, 1, 8, 3, 6
9 - 1 = 8
The range is 8

mode

The value that occurs most often in a data set.

How to determine the mode in a set of scores.

Order the scores from least to greatest.
Locate the score that occurs the most.

3, 4, **5, 5, 5**, 6, 6, 7, 8, 8, 9

mode = 5

3, 4, **5, 5, 5, 6, 6, 6**, 8, 8, 9

modes = 5 and 6

two modes are called bimodal
more than two modes are called multimodal

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

there is no mode in this set of scores

one mode … unimodal
two modes … bimodal
three modes … trimodal
more than one mode … multimodal

Sample Variance

$$s^2 = \frac{\sum(x - \bar{x})^2}{n - 1}$$

Sample Standard Deviation

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}$$

$$Standard\ Deviation = \sqrt{Variance}$$

Answers

- For attribute ***age***

mean	46.44
median	51
modes	23,27,54,58
sd	13.21
variance	174.73
Range	38

- For attribute ***%fat***

mean	28.78
median	30.7
mode	No mode
sd	9.25
variance	85.64
Range	34.7

Smoothing Noisy Data

Binning

Sorted data for *price* (in dollars): 4, 8, 15, 21, 21, 24, 25, 28, 34

Partition into (equal-frequency) bins:
Bin 1: 4, 8, 15
Bin 2: 21, 21, 24
Bin 3: 25, 28, 34

Smoothing by bin means:
Bin 1: 9, 9, 9
Bin 2: 22, 22, 22
Bin 3: 29, 29, 29

Smoothing by bin boundaries:
Bin 1: 4, 4, 15
Bin 2: 21, 21, 24
Bin 3: 25, 25, 34

Exercise

Using the data for *age* given in Question 1, answer the following.

23, 23, 27, 27, 39, 41, 47, 49, 50, 52, 54, 54, 56, 57, 58, 58, 60, 61

Perform smoothing operation using

- *smoothing by bin means*
- *smoothing by bin median*
- *smoothing by bin boundaries*

using a bin depth of 3.

Answers

- Divide the elements into bins of depth 3

Bin-1: 23, 23, 27
Bin-2: 27, 39, 41
Bin-3: 47, 49, 50
Bin-4: 52, 54, 54
Bin-5: 56, 57, 58
Bin-6: 58, 60, 61

Answers
Smoothing by
Bin Means

- **Smoothing by Bin Means**

Bin-1: 24, 24, 24
Bin-2: 36,36,36
Bin-3: 49,49,49
Bin-4: 53,53,53
Bin-5: 57,57,57
Bin-6: 60,60,60

Answers
Smoothing by
Bin Medians

- **Smoothing by Bin Medians**

Bin-1: 23, 23, 23
Bin-2: 39,39,39
Bin-3: 49,49,49
Bin-4: 54, 54, 54
Bin-5: 57, 57, 57
Bin-6: 60,60,60

Answers
Smoothing by
Bin Boundaries

- **Smoothing by Bin Boundaries**

Bin-1: 23, 23, 27
Bin-2: 27, 41, 41
Bin-3: 47, 50, 50
Bin-4: 52, 54, 54
Bin-5: 56, 58, 58 or 56,56,58
Bin-6: 58, 61, 61

North Central Cancer Treatment Group (NCCTG) provides the following dataset for lung cancer prediction.

Id no	inst	time	status	age	sex	ph. ecog	ph. karno	pat. karno	meal. cal	wt.loss
1	3	306	2	74	1	1	90	100	1175	
2	3	455	2	68	1	0	90	90	1225	15
3	3	1010	1	56	1	0	90	90		15
4	5	210		57	1	1	90	60	1150	11
5	1	883	2	60	1	0	100	90		0
6	12	1022	1		1	1	50	80	513	0
7	7	310	2	68	2	2	70	60	384	10
8	11		2	71	2	2	60	80	538	1
9	1	218	2	53	1	1	70	80	825	16
10	7	166	2	61		2	70	70	271	34
11	6	170	2	57	1	1	80	80	1025	27
12	16	654	2	68	2	2	70	70		23
13	11	728	2	68	2	1	90	90		5
14	21	71	2	60	1		60	70	1225	32
15	12	567	2	57	1	1	80	70	2600	60

Exercises

North Central Cancer Treatment Group (NCCTG) provides the following dataset for lung cancer prediction.

a) Fill the missing values present in this dataset. **(Filling missing values)**

b) Create a sampled dataset of size 5 using Random sampling with and without replacements. **(Data Sampling)**

Answers
Filling Missing Values

Missing values are replaced by appropriate central tendency measure

(Mean, median or mode)

Answers are marked RED

Id no	inst	time	status	age	sex	ph. ecog	ph. karno	pat. karno	meal. cal	wt.loss
1	3	306	2	74	1	1	90	100	1175	**15**
2	3	455	2	68	1	0	90	90	1225	15
3	3	1010	1	56	1	0	90	90	**994**	15
4	5	210	**2**	57	1	1	90	60	1150	11
5	1	883	2	60	1	0	100	90	**994**	0
6	12	1022	1	**63**	1	1	50	80	513	0
7	7	310	2	68	2	2	70	60	384	10
8	11	**484**	2	71	2	2	60	80	538	1
9	1	218	2	53	1	1	70	80	825	16
10	7	166	2	61	**1**	2	70	70	271	34
11	6	170	2	57	1	1	80	80	1025	27
12	16	654	2	68	2	2	70	70	**994**	23
13	11	728	2	68	2	1	90	90	**994**	5
14	21	71	2	60	1	**1**	60	70	1225	32
15	12	567	2	57	1	1	80	70	2600	60

Answers
Simple Random Sampling

Simple Random Sampling without Replacement

Id no	inst	time	status	age	sex	ph. ecog	ph. karno	pat. karno	meal. cal	wt.loss
1	3	306	2	74	1	1	90	100	1175	
5	1	883	2	60	1	0	100	90		0
6	12	1022	1		1	1	50	80	513	0
10	7	166	2	61		2	70	70	271	34
15	12	567	2	57	1	1	80	70	2600	60

Simple Random Sampling with Replacement

Id no	inst	time	status	age	sex	ph. ecog	ph. karno	pat. karno	meal. cal	wt.loss
1	3	306	2	74	1	1	90	100	1175	
9	1	218	2	53	1	1	70	80	825	16
9	1	218	2	53	1	1	70	80	825	16
13	11	728	2	68	2	1	90	90		5
14	21	71	2	60	1		60	70	1225	32

Data Cleaning
Problem Solving - 2

Data Normalization

- Min-max Normalization
- Z-Score Normalization
- Decimal Scale Normalization

Min-max Normalization

Min-max normalization performs a linear transformation on the original data. Suppose that min_A and max_A are the minimum and maximum values of an attribute, A. Min-max normalization maps a value, v_i, of A to v_i' in the range $[new\_min_A, new\_max_A]$ by computing

$$v_i' = \frac{v_i - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A. \quad (3.8)$$

Example

Min-max normalization. Suppose that the minimum and maximum values for the attribute *income* are \$12,000 and \$98,000, respectively. We would like to map *income* to the range $[0.0, 1.0]$. By min-max normalization, a value of \$73,600 for *income* is transformed to $\frac{73{,}600 - 12{,}000}{98{,}000 - 12{,}000}(1.0 - 0) + 0 = 0.716$. ■

Z-score Normalization

In ***z*-score normalization** (or *zero-mean normalization*), the values for an attribute, A, are normalized based on the mean (i.e., average) and standard deviation of A. A value, v_i, of A is normalized to v_i' by computing

$$v_i' = \frac{v_i - \bar{A}}{\sigma_A}, \quad (3.9)$$

where $\bar{A}$ and σ_A are the mean and standard deviation, respectively, of attribute A. The

Example

z-score normalization. Suppose that the mean and standard deviation of the values for the attribute *income* are \$54,000 and \$16,000, respectively. With z-score normalization, a value of \$73,600 for *income* is transformed to $\frac{73{,}600 - 54{,}000}{16{,}000} = 1.225$. ■

Decimal Scale Normalization

Normalization by decimal scaling normalizes by moving the decimal point of values of attribute A. The number of decimal points moved depends on the maximum absolute value of A. A value, v_i, of A is normalized to v_i' by computing

$$v_i' = \frac{v_i}{10^j}, \tag{3.12}$$

where j is the smallest integer such that $max(|v_i'|) < 1$.

Example

Decimal scaling. Suppose that the recorded values of A range from -986 to 917. The maximum absolute value of A is 986. To normalize by decimal scaling, we therefore divide each value by 1000 (i.e., $j = 3$) so that -986 normalizes to -0.986 and 917 normalizes to 0.917. ■

Exercise

Use these methods to normalize the following group of data:

200, 300, 400, 600,1000

(a) min-max normalization by setting min = 0 and max = 1

(b) z-score normalization

(c) normalization by decimal scaling

Answers Min-max Normalization

Given data is A = **200, 300, 400, 600,1000**

New_minA = 0
New_maxA = 1

minA = 200
maxA = 1000

Formula:

$$v_i' = \frac{v_i - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A$$

$V_i = 200$

$$V'_i = \frac{(200-200)}{(1000-200)} \times (1-0) + 0$$

Answers z-score Normalization

Given data is A = **200, 300, 400, 600,1000**

Formula:

$$v_i' = \frac{v_i - \bar{A}}{\sigma_A}$$

Mean ($\bar{A}$) = 500
Standard Deviation (σ_A) = 316.22
$V_i = 200$

$$V'_i = \frac{(200-500)}{(316.22)} = -0.94$$

Answers Decimal Scale Normalization

Given data is A = **200, 300, 400, 600,1000**

Formula: $v_i' = \frac{v_i}{10^j}$

$V_i = 200$

$V'_i = \frac{(200)}{(1000)} = 0.2$

Answers

Actual Value	Min-max Normalization	z-score Normalization	Decimal Scale Normalization
200	0.00	-0.94	0.2
300	0.13	-0.63	0.3
400	0.25	-0.31	0.4
600	0.50	0.31	0.6
1000	1.00	1.58	1

Histograms

- Histograms use **binning** to approximate data distributions
- a popular form of data reduction
- partitions the data distribution of an attribute A into disjoint subsets, referred to as **buckets** or **bins**.
- Buckets are 3 types
  1. Singleton
  2. Equal Width
  3. Equal frequency

Histograms. The following data are a list of *AllElectronics* prices for commonly sold items (rounded to the nearest dollar). The numbers have been sorted: 1, 1, 5, 5, 5, 5, 5, 8, 8, 10, 10, 10, 10, 12, 14, 14, 14, 15, 15, 15, 15, 15, 15, 18, 18, 18, 18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 25, 25, 25, 25, 25, 28, 28, 30, 30, 30.

Example

Divide the data into **Equal width** and

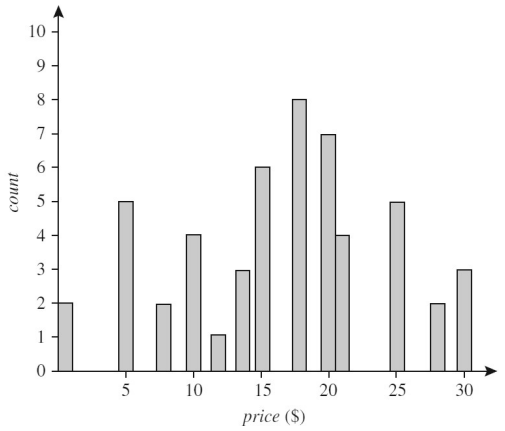
Equal Frequency Bins and

Draw the Histograms

Answers Singleton Histograms

Singleton Histogram

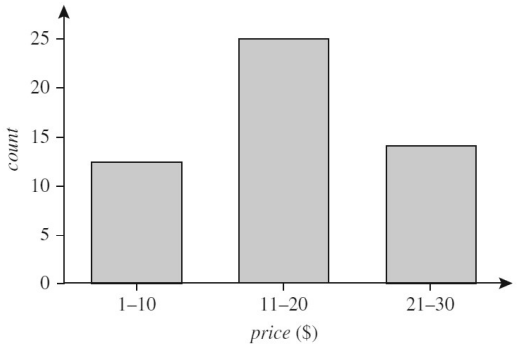
- Find the frequency of each value and draw the Histogram



Answers Equal Width Histograms

Equal-width Histogram

- Here the width of each bucket range is uniform (e.g., the width of $10 for the buckets in the below figure)



Answers Equal Frequency Histograms

Equal-frequency (or equal-depth) Histograms

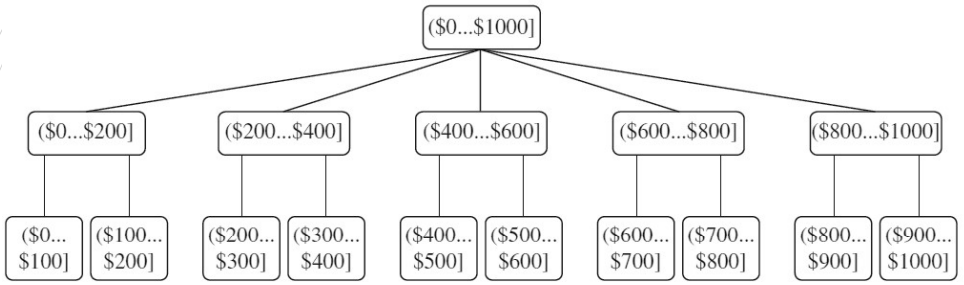
- Here, the buckets are created so that, roughly, the frequency (depth) of each bucket is **constant**
- Refer the Binning techniques for dividing the elements in to equal depth and applying smoothing techniques.
- After applying smoothing techniques, find frequency of each element and draw the histogram

Discretization

- Here the raw values of a numeric attribute are replaced by interval variables or conceptual labels
- Ex: *age values* can be replaced by
  interval labels 0–10, 11–20, etc.
  or
  conceptual labels (e.g., *youth, adult*, *senior*)

Concept Hierarchy Generation

- The labels of an attribute can be recursively organized into higher-level concepts, resulting in a ***concept hierarchy*** for the numeric attribute.
- Ex: Price values of various items can be organized into concept hierarchy for easy analysis



A concept hierarchy for the attribute *price*, where an interval ($X . . . $Y] denotes the range from $X (exclusive) to $Y (inclusive).

R-Programming

What is R

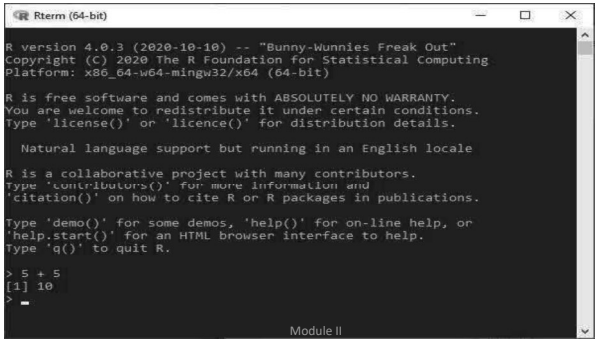
- R is a popular programming language used for statistical computing and graphical presentation.
- Its most common use is to analyze and visualize data.

Why Use R?

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has a large community support
- It has many packages (libraries of functions) that can be used to solve different problems

How to Install R

- To install R, go to https://cloud.r-project.org/ and download the latest version of R for Windows, Mac or Linux.
- When you have downloaded and installed R, you can run R on your computer.
- The screenshot below shows how it may look like when you run R on a Windows PC:



R Syntax

- To output text in R, use **single** or **double** quotes:
- **Example:**

```
"Hello World!"
```


R Print

- **Two-ways**
  - **Without Print**
    - Unlike many other programming languages, you can output code in R without using a print function:
    - **Example:**

```
"Hello World!“
```

  - **Using Print() function**
    - Like other programming languages, R does have a print() function available.
    - This might be useful if you are familiar with other programming languages.
    - **Example:**

```
print("Hello World!")
```


R Comments

- Comments can be used to explain R code, and to make it more readable.
- It can also be used to prevent execution when testing alternative code.
- Comments starts with a **#.**
- When executing the R-code, R will ignore anything that starts with **#.**
- **A comment before a line of code:**
  - Example:
    - **# This is a comment**

```
"Hello World!“
```

- **A comment at the end of a line of code:**
  - Example:

```
"Hello World!" # This is a comment
```


Contd…

- Comments does not have to be text to explain the code, it can also be used to prevent R from executing the code:
  - Example

```
# "Good morning!"
"Good night!"
```

- **Multiline Comments**
  - Unlike other programming languages, such as Java, there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments:
  - Example

```
# This is a comment
# written in
# more than just one line
"Hello World!"
```

R Variables

- Variables are containers for storing data values.
- R does not have a command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- To assign a value to a variable, use the **<-** sign.
- To output (or print) the variable value, just type the variable name:
  - Example

```
name <- "John“   # string
age <- 40          # number / integer

name   # output "John"
age    # output 40
```


Print / Output Variables

- Compared to many other programming languages, you do not have to use a function to print/output variables in R.
- Simply you can just type the name of the variable:
- Example
- name <- "John Doe"
  name # auto-print the value of the name variable
- R does have a print() function available if you want to use it.
- Example
- name <- "John Doe"

  print(name) # print the value of the name variable

Concatenate Elements

- You can also concatenate, or join, two or more elements, by using the paste() function.

- To combine both text and a variable, R uses comma (,):
- Example
- text <- "awesome"

  paste("R is", text)
- You can also use , to add a variable to another variable:

- Example
- text1 <- "R is"
- text2 <- "awesome"

- paste(text1, text2)

Multiple Variables

- R allows you to assign the same value to multiple variables in one line:
- Example
- # Assign the same value to multiple variables in one line
  var1 <- var2 <- var3 <- "Orange"

  # Print variable values
  var1
  var2
  var3

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for R variables are:

- A variable name must start with a letter and can be a combination of letters, digits, period(.)
  and underscore(_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

```
# Legal variable names:
myvar <- "John"
my_var <- "John"
myVar <- "John"
MYVAR <- "John"
myvar2 <- "John"
.myvar <- John
```

```
# Illegal variable names:
2myvar <- "John"
my-var <- "John"
my var <- "John"
_my_var <- "John"
my_v@ar <- "John"
TRUE <- "John"
```


Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- In R, variables do not need to be declared with any particular type, and can even change type after they have been set:
- Example
- my_var <- 30 # my_var is type of **numeric**
  my_var <- "Sally" # my_var is now of type **character** (aka string)

Basic Data Types

Basic data types in R can be divided into the following types:

•numeric - (10.5, 55, 787)

•integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)

•complex - (9 + 3i, where "i" is the imaginary part)

•character (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")

•logical (a.k.a. boolean) - (TRUE or FALSE)

We can use the class() function to check the data type of a variable:

Example

```
# numeric
x <- 10.5
class(x)

# integer
x <- 1000L
class(x)

# complex
x <- 9i + 3
class(x)

# character/string
x <- "R is exciting"
class(x)

# logical/boolean
x <- TRUE
class(x)
```


R Numbers

Numbers

There are three number types in R:

•numeric
•integer
•complex

Variables of number types are created when you assign a value to them:

Example

```
x <- 10.5    # numeric
y <- 10L     # integer
z <- 1i      # complex
```

Numeric

A `numeric` data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:

Example
```
x <- 10.5
y <- 55

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```


Integer

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an `integer` variable, you must use the letter `L` after the integer value:

Example
```
x <- 1000L
y <- 55L

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```


Complex

A `complex` number is written with an "`i`" as the imaginary part:

Example
```
x <- 3+5i
y <- 5i

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```


Type Conversion

You can convert from one type to another with the following functions:
- `as.numeric()`
- `as.integer()`
- `as.complex()`

Example
```
x <- 1L # integer
y <- 2 # numeric
# convert from integer to numeric:
a <- as.numeric(x)

# convert from numeric to integer:
b <- as.integer(y)

# print values of x and y
x
y
# print the class name of a and b
class(a)
class(b)
```

Exercises

Prog1 Check whether the person is eligible to vote or not

- Age>=18

Prog 2 Check whether the person is eligible to donate blood or not

- Age>=18 && weight > 50

Prog3Check the pass grade of a student

- Var1, var2, var3, var4, var5
- Sum of var1……5
- Avg= sum/5
- Avg>90 –A,, Avg>80 – B, avg>70----C, Avg>60 ---D

#If (age>= 18){

If age>=18:

R Math

- In R, you can use operators to perform common mathematical operations on numbers.
- The **+** operator is used to add together two values:
  - Example
    - 10 + 5
- And the - operator is used for subtraction:
  - Example
    - 10 - 5

Built-in Math Functions

- R also has many built-in math functions that allows you to perform mathematical tasks on numbers.
- For example, the **min()** and **max()** functions can be used to find the lowest or highest number in a set:
  - Example
    - max(5, 10, 15)
    - min(5, 10, 15)

- **sqrt()**

The sqrt() function returns the square root of a number:

Example
sqrt(16)

Contd…

- **abs()**
- The abs() function returns the absolute (positive) value of a number:
  - Example
    - abs(-4.7)
- ceiling() and floor()
- The **ceiling()** function rounds a number upwards to its nearest integer, and the **floor()** function rounds a number downwards to its nearest integer, and returns the result:
  - Example
    - ceiling(1.4)
    - floor(1.4)

R Strings

- A **character**, or **strings**, are used for storing **text**.
- A **string** is surrounded by either single **quotation marks**, or **double quotation marks.**
  - **Example**
    - "hello"
    - 'hello'

    "hello" is the same as **'hello'**

Assign a String to a Variable

- Assigning a string to a variable is done with the variable followed by the **<-** operator and the string:
- Example
  - str <- "Hello"
  - str # print the value of str

Multiline Strings

- **You can assign a multiline string to a variable like this:**
- **Example**
  - **str <- "Lorem ipsum dolor sit amet,**
    consectetur adipiscing elit,
    sed do eiusmod tempor incididunt
    ut labore et dolore magna aliqua."

    str # print the value of str

Exercises

1. Create a Vector with 5 random values in the range of -100 to +100.
2. Create two matrices by taking dimensions from the user and do addition, subtraction, Multiplication and Division.
3. Write a R Program to display 100 to 200 numbers. If a number is multiple of 3, print-"The number is multiple of 3". Similarly, also print the appropriate messages for 5 and for both 3 and 5.
4. Consider the following matrix and find the maximum of each row.

x <- 1:20
dim(x) <- c(5,4)

5. Assume you have created dataset using the following line of code:

Data <- data.frame(grade=c("A", "B", "A", "B", "C", "A", "C"), bar=1:7)

Write a R program to calculate the mean 'bar' value grouped by 'grade'.

R-Programming

What is R

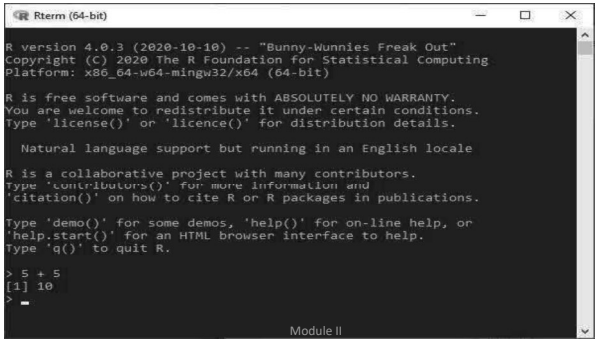
- R is a popular programming language used for statistical computing and graphical presentation.
- Its most common use is to analyze and visualize data.

Why Use R?

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has a large community support
- It has many packages (libraries of functions) that can be used to solve different problems

How to Install R

- To install R, go to https://cloud.r-project.org/ and download the latest version of R for Windows, Mac or Linux.
- When you have downloaded and installed R, you can run R on your computer.
- The screenshot below shows how it may look like when you run R on a Windows PC:



R Syntax

- To output text in R, use **single** or **double** quotes:
- **Example:**

```
"Hello World!"
```


R Print

- **Two-ways**
  - **Without Print**
    - Unlike many other programming languages, you can output code in R without using a print function:
    - **Example:**

```
"Hello World!“
```

  - **Using Print() function**
    - Like other programming languages, R does have a print() function available.
    - This might be useful if you are familiar with other programming languages.
    - **Example:**

```
print("Hello World!")
```


R Comments

- Comments can be used to explain R code, and to make it more readable.
- It can also be used to prevent execution when testing alternative code.
- Comments starts with a **#.**
- When executing the R-code, R will ignore anything that starts with **#.**
- **A comment before a line of code:**
  - Example:
    - **# This is a comment**

```
"Hello World!“
```

- **A comment at the end of a line of code:**
  - Example:

```
"Hello World!" # This is a comment
```


Contd…

- Comments does not have to be text to explain the code, it can also be used to prevent R from executing the code:
  - Example

```
# "Good morning!"
"Good night!"
```

- **Multiline Comments**
  - Unlike other programming languages, such as Java, there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments:
  - Example

```
# This is a comment
# written in
# more than just one line
"Hello World!"
```

R Variables

- Variables are containers for storing data values.
- R does not have a command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- To assign a value to a variable, use the **<-** sign.
- To output (or print) the variable value, just type the variable name:
  - Example

```
name <- "John“   # string
age <- 40          # number / integer

name   # output "John"
age    # output 40
```


Print / Output Variables

- Compared to many other programming languages, you do not have to use a function to print/output variables in R.
- Simply you can just type the name of the variable:
- Example
- name <- "John Doe"
  name # auto-print the value of the name variable
- R does have a print() function available if you want to use it.
- Example
- name <- "John Doe"

  print(name) # print the value of the name variable

Concatenate Elements

- You can also concatenate, or join, two or more elements, by using the paste() function.

- To combine both text and a variable, R uses comma (,):
- Example
- text <- "awesome"

  paste("R is", text)
- You can also use , to add a variable to another variable:

- Example
- text1 <- "R is"
- text2 <- "awesome"

- paste(text1, text2)

Multiple Variables

- R allows you to assign the same value to multiple variables in one line:
- Example
- # Assign the same value to multiple variables in one line
  var1 <- var2 <- var3 <- "Orange"

  # Print variable values
  var1
  var2
  var3

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for R variables are:

- A variable name must start with a letter and can be a combination of letters, digits, period(.)
  and underscore(_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

```
# Legal variable names:
myvar <- "John"
my_var <- "John"
myVar <- "John"
MYVAR <- "John"
myvar2 <- "John"
.myvar <- John
```

```
# Illegal variable names:
2myvar <- "John"
my-var <- "John"
my var <- "John"
_my_var <- "John"
my_v@ar <- "John"
TRUE <- "John"
```


Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- In R, variables do not need to be declared with any particular type, and can even change type after they have been set:
- Example
- my_var <- 30 # my_var is type of **numeric**
  my_var <- "Sally" # my_var is now of type **character** (aka string)

Basic Data Types

Basic data types in R can be divided into the following types:

•numeric - (10.5, 55, 787)

•integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)

•complex - (9 + 3i, where "i" is the imaginary part)

•character (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")

•logical (a.k.a. boolean) - (TRUE or FALSE)

We can use the class() function to check the data type of a variable:

Example

```
# numeric
x <- 10.5
class(x)

# integer
x <- 1000L
class(x)

# complex
x <- 9i + 3
class(x)

# character/string
x <- "R is exciting"
class(x)

# logical/boolean
x <- TRUE
class(x)
```


R Numbers

Numbers

There are three number types in R:

•numeric
•integer
•complex

Variables of number types are created when you assign a value to them:

Example

```
x <- 10.5    # numeric
y <- 10L     # integer
z <- 1i      # complex
```

Numeric

A `numeric` data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:

Example
```
x <- 10.5
y <- 55

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```


Integer

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an `integer` variable, you must use the letter `L` after the integer value:

Example
```
x <- 1000L
y <- 55L

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```


Complex

A `complex` number is written with an "`i`" as the imaginary part:

Example
```
x <- 3+5i
y <- 5i

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```


Type Conversion

You can convert from one type to another with the following functions:
- `as.numeric()`
- `as.integer()`
- `as.complex()`

Example
```
x <- 1L # integer
y <- 2 # numeric
# convert from integer to numeric:
a <- as.numeric(x)

# convert from numeric to integer:
b <- as.integer(y)

# print values of x and y
x
y
# print the class name of a and b
class(a)
class(b)
```

Exercises

Prog1 Check whether the person is eligible to vote or not

- Age>=18

Prog 2 Check whether the person is eligible to donate blood or not

- Age>=18 && weight > 50

Prog3Check the pass grade of a student

- Var1, var2, var3, var4, var5
- Sum of var1……5
- Avg= sum/5
- Avg>90 –A,, Avg>80 – B, avg>70----C, Avg>60 ---D

#If (age>= 18){

If age>=18:

R Math

- In R, you can use operators to perform common mathematical operations on numbers.
- The **+** operator is used to add together two values:
  - Example
    - 10 + 5
- And the - operator is used for subtraction:
  - Example
    - 10 - 5

Built-in Math Functions

- R also has many built-in math functions that allows you to perform mathematical tasks on numbers.
- For example, the **min()** and **max()** functions can be used to find the lowest or highest number in a set:
  - Example
    - max(5, 10, 15)
    - min(5, 10, 15)

- **sqrt()**

The sqrt() function returns the square root of a number:

Example
sqrt(16)

Contd…

- **abs()**
- The abs() function returns the absolute (positive) value of a number:
  - Example
    - abs(-4.7)
- ceiling() and floor()
- The **ceiling()** function rounds a number upwards to its nearest integer, and the **floor()** function rounds a number downwards to its nearest integer, and returns the result:
  - Example
    - ceiling(1.4)
    - floor(1.4)

R Strings

- A **character**, or **strings**, are used for storing **text**.
- A **string** is surrounded by either single **quotation marks**, or **double quotation marks.**
  - **Example**
    - "hello"
    - 'hello'

    "hello" is the same as **'hello'**

Assign a String to a Variable

- Assigning a string to a variable is done with the variable followed by the **<-** operator and the string:
- Example
  - str <- "Hello"
  - str # print the value of str

Multiline Strings

- **You can assign a multiline string to a variable like this:**
- **Example**
  - **str <- "Lorem ipsum dolor sit amet,**
    consectetur adipiscing elit,
    sed do eiusmod tempor incididunt
    ut labore et dolore magna aliqua."

    str # print the value of str

Exercises

1. Create a Vector with 5 random values in the range of -100 to +100.
2. Create two matrices by taking dimensions from the user and do addition, subtraction, Multiplication and Division.
3. Write a R Program to display 100 to 200 numbers. If a number is multiple of 3, print-"The number is multiple of 3". Similarly, also print the appropriate messages for 5 and for both 3 and 5.
4. Consider the following matrix and find the maximum of each row.

x <- 1:20
dim(x) <- c(5,4)

5. Assume you have created dataset using the following line of code:

Data <- data.frame(grade=c("A", "B", "A", "B", "C", "A", "C"), bar=1:7)

Write a R program to calculate the mean 'bar' value grouped by 'grade'.

R Data Structures

- **Vectors**
- **Lists**
- **Matrices**
- **Arrays**
- **Data Frames**
- **Factors**

VECTORS

- A vector is simply a **list of items** that are of the **same type**.
- To combine the list of items to a vector, use the **c()** function and **separate the items by a comma.**
- Example

```
# Vector of strings
fruits <- c("banana", "apple", "orange")

# Print fruits
fruits
```


Numeric Vector

- A vector that combines numerical values:
- Example

```
# Vector of numerical values
numbers <- c(1, 2, 3)

# Print numbers
numbers
```

- To create a vector with numerical values in a sequence, use the **: operator**
- Example

```
# Vector with numerical values in a sequence
numbers <- 1:10

numbers
```


Numeric Vectors

- You can also create **numerical values** with decimals in a **sequence**, but note that if the last element does not belong to the sequence, it is not used:
- Example

```
# Vector with numerical decimals in a sequence
numbers1 <- 1.5:6.5
numbers1

# Vector with numerical decimals in a sequence where the last element is not used
numbers2 <- 1.5:6.3
numbers2
```

Logical Vectors

- A vector of logical values:
- Example

```
# Vector of logical values
log_values <- c(TRUE, FALSE, TRUE, FALSE)

log_values
```


Vector Length

- To find out how many items a vector has, use the length() function:
- Example
  - fruits <- c("banana", "apple", "orange")
  - **length(**fruits**)**

Sort a Vector

- To sort items in a vector alphabetically or **numerically,** use the **sort()** function:
- Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
numbers <- c(13, 3, 5, 7, 20, 2)
# sorting
sort(fruits)  # Sort a string
sort(numbers) # Sort numbers
```


Access Vectors

- You can access the **vector items** by referring to its **index number** inside brackets **[ ]**. The first item has index 1, the second item has index 2, and so on:
- Example

```
fruits <- c("banana", "apple", "orange")
# Access the first item (banana)
fruits[1]
```

Access Vectors

- You can also access multiple elements by referring to different index positions with the c() function:
- Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
# Access the first and third item (banana and orange)
fruits[c(1, 3)]
```

- You can also use negative index numbers to access all items except the ones specified:
- Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
# Access all items except for the first item
fruits[c(-1)]
```


Change an Item

- To change the value of a specific item, refer to the index number:
- Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")

# Change "banana" to "pear"
fruits[1] <- "pear"

# Print fruits
fruits
```


Repeat Vectors

- To repeat vectors, use the **rep()** function:
- Example

```
#Repeat each value:
repeat_each <- rep(c(1,2,3), each = 3)
repeat_each
```


Repeat Vectors

- Example

```
#Repeat the sequence of the vector:
Repeat_times <- rep(c(1,2,3), times = 3)
repeat_times
```

- Example

```
#Repeat each value independently:
repeat_indepent <- rep(c(1,2,3), times = c(5,2,1))
repeat_indepent
```

Sequenced Vectors

- Generating Sequenced Vectors
- One of the examples on top, showed you how to create a vector with numerical values in a sequence with the : operator

- Example

```
numbers <- 1:10
numbers
```


Seq() function

- To make bigger or smaller steps in a sequence, use the seq() function:

- Example
- numbers <- seq(from = 0, to = 100, by = 20)

- Numbers
- The seq() function has three parameters: from is where the sequence starts, to is where the sequence stops, and by is the interval of the sequence.

Programs on Vectors

1. Write a R program to create a vector of a specified type and length. Create vector of numeric, complex, logical and character types of length 6
2. Write a R program to add two vectors of integers type and length 3.
3. Write a R program to append value to a given empty vector
4. Write a R program to multiply two vectors of integers type and length 3
5. Write a R program to divide two vectors of integers type and length 3
6. Write a R program to find Sum, Mean and Product of a Vector.
7. Write a R program to find Sum, Mean and Product of a Vector, ignore element like NA or NaN.
8. Write a R program to find the minimum and the maximum of a Vector.
9. Write a R program to sort a Vector in ascending and descending order.
10. Write a R program to add 3 to each element in a given vector. Print the original and new vector
11. Write a R program to combines two given vectors by columns, rows
12. Write a R program to find the elements of a given vector that are not in another given vector.

R-Programming

R Data Structures

- Vectors
- **Lists**
- Matrices
- Arrays
- Data Frames
- Factors

R Lists

- A list in R can contain many different data types inside it.
- A list is a collection of data which is ordered and changeable.
- To create a list, use the **list()** function:
- Example
- # List of strings
  fruits <- **list(**"apple", "banana", "cherry"**)**

  # Print the list
  fruits

Access Lists

- You can access the list items by referring to its **index number**, inside brackets. The first item has index 1, the second item has index 2, and so on:
- Example

```
fruits <- list("apple", "banana", "cherry")

fruits[1]
```

Change Item Value

- To change the value of a specific item, refer to the **index number**:
- Example

```
fruits <- list("apple", "banana", "cherry")
fruits[1] <- "blackcurrant"

# Print the updated list
fruits
```


List Length

- To find out how many items a list has, use the **length()** function:
- Example

```
fruits <- list("apple", "banana", "cherry")

length(fruits)
```


Check if Item Exists

- To find out if a specified item is present in a list, use the **%in%** operator:
- Example

```
#Check if "apple" is present in the list:

fruits <- list("apple", "banana", "cherry")

"apple" %in% fruits
```


Add List Items

- To add an item to the end of the list, use the **append()** function:
- Example

```
Add "orange" to the list:
fruits <- list("apple", "banana", "cherry")
append(fruits, "orange")
```

- To add an item to the right of a specified index, add **"after=index number"** in the append() function:

Example

```
#Add "orange" to the list after "banana" (index 2):
fruits <- list("apple", "banana", "cherry")
append(fruits, "orange", after = 2)
```

Remove List Items

- You can also remove list items. The following example creates a new, updated list without an "apple" item:
- Example

  Remove "apple" from the list:

```
fruits <- list("apple", "banana", "cherry")

newlist <- fruits[-1]

# Print the new list
newlist
```


Range of Indexes

- You can specify a range of indexes by specifying **where to start and where to end the range**, by using the **:** operator:
- Example

```
#Return the second, third, fourth and fifth item:

fruits <- list("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

(fruits)[2:5]
```


Loop Through a List

- You can loop through the list items by using a **for** loop:
- Example

```
# Print all items in the list, one by one:

fruits <- list("apple", "banana", "cherry")

for (x in fruits) {
  print(x)
}
```


Join Two Lists

- There are several ways to **join**, or **concatenate**, two or more lists in R.
- The most common way is to use the **c() function**, which combines two elements together:
- Example

```
list1 <- list("a", "b", "c")
list2 <- list(1,2,3)
list3 <- c(list1,list2)

list3  # joins two lists
```

R Data Structures

- Vectors
- Lists
- **Matrices**
- Arrays
- Data Frames
- Factors

R Matrices

Matrices

- A matrix is a two dimensional data set with columns and rows.
- A column is a vertical representation of data, while a row is a horizontal representation of data.
- A matrix can be created with the **matrix()** function. Specify the ***nrow*** and ***ncol*** parameters to get the amount of rows and columns:
- Example

```
# Create a matrix
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)
# Print the matrix
thismatrix
```


Contd....

- You can also create a matrix with strings:
- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

thismatrix
```


Access Matrix Items

- You can access the items by using **[ ]** brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

thismatrix[1, 2]
```

Contd…

- The **whole row** can be accessed if you specify **a comma after the number** in the bracket:
- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
thismatrix[2,]
```

- The **whole column** can be accessed if you specify a comma before the number in the bracket:
- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

thismatrix[,2]
```


Access More Than One Row

- More than one row can be accessed if you use the **c()** function:
- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

thismatrix[c(1,2),]
```


Access More Than One Column

- More than one column can be accessed if you use the **c()** function:
- Example

```
thismatrix <- matrix( c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

thismatrix[, c(1,2)]
```


Add Rows and Columns

- Use the **cbind()** function to add additional columns in a Matrix:
- Example

```
thismatrix <- matrix( c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

newmatrix <- cbind( thismatrix, c("strawberry", "blueberry", "raspberry") )

# Print the new matrix

newmatrix
```

Contd…

- Use the **rbind()** function to add additional rows in a Matrix:

- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

newmatrix <- rbind(thismatrix, c("strawberry", "blueberry", "raspberry"))

# Print the new matrix
newmatrix
```


Remove Rows and Columns

- Use the **c()** function to remove rows and columns in a Matrix:

- Example

```
thismatrix <- matrix( c("apple", "banana", "cherry", "orange", "mango", "pineapple"), nrow = 3, ncol =2 )

#Remove the first row and the first column
thismatrix <- thismatrix[-c(1), -c(1)]

thismatrix
```


Check if an Item Exists

- To find out if a specified item is present in a matrix, use the **%in%** operator:

- Example

```
# Check if "apple" is present in the matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

"apple" %in% thismatrix
```


Amount of Rows and Columns

- Use the **dim()** function to find the amount of rows and columns in a Matrix:

- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

dim(thismatrix)
```

Matrix Length

- Use the **length()** function to find the dimension of a Matrix:

- Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

length( thismatrix )
```


Loop Through a Matrix

- You can loop through a Matrix using a **for** loop.
- The loop will start at the first row, moving right:

- Example

Loop through the matrix items and print them:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

for (rows in 1:nrow(thismatrix)) {
  for (columns in 1:ncol(thismatrix)) {
    print(thismatrix[rows, columns])
  }
}
```


Combine two Matrices

- Again, you can use the **rbind()** or **cbind()** function to combine two or more matrices together:

- Example

```
# Combine matrices
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol = 2)
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2, ncol = 2)

# Adding it as a rows
Matrix_Combined <- rbind(Matrix1, Matrix2)
Matrix_Combined

# Adding it as a columns
Matrix_Combined <- cbind(Matrix1, Matrix2)
Matrix_Combined
```


Matrices Exercises

1. Write a R program to create a blank matrix.
2. Write a R program to create a matrix with 10 rows and 5 columns for the first 50 numeric values.
3. Write a R program to access the element at 3rd column and 2nd row, only the 3rd row and only the 4th column of a given matrix.
4. Write a R program to create two 4x3 matrix and add, subtract, multiply and divide the matrixes
5. Write a R program to convert a matrix to a 1 dimensional array.
6. Write a R program to extract the submatrix whose rows have column value > 7 from a given matrix.
7. Write a R program to concatenate two given matrixes of same column but different rows. Also display the dimensions of the resultant matrix

R Data Structures

- Vectors
- Lists
- Matrices
- **Arrays**
- Data Frames
- Factors

R Arrays

Arrays

- Compared to matrices, arrays can have more than **two dimensions**.
- You can use the **array()** function to create an array, and the **dim parameter** to specify the dimensions:
- Example

```
# An array with one dimension with values ranging from 1 to 24
thisarray <- c(1:24)
thisarray

# An array with more than one dimension
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray
```


Access Array Items

- You can access the array elements by referring to the index position. You can use the **[ ]** brackets to access the desired elements from an array:

- Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

multiarray[2, 3, 2]
```


Contd…

- The syntax is as follow: **array[row position, column position, matrix level]**
- You can also access the whole row or column from a matrix in an array, by using the c() function:
- Example

```
thisarray <- c(1:24)

# Access all the items from the first row from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[c(1),,1]

# Access all the items from the first column from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[,c(1),1]
```

- A comma (,) **before c()** means that we want to access the **column.**
- A comma (,) **after c()** means that we want to access the **row.**

Check if an Item Exists

- To find out if a specified item is present in an array, use the **%in%** operator:

- Example

```
#Check if the value "2" is present in the array:

thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

2 %in% multiarray
```


Amount of Rows and Columns

- Use the **dim()** function to find the **amount of rows and columns** in an array:

- Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

dim(multiarray)
```


Array Length

- Use the **length()** function to find the dimension of an array:

- Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

length(multiarray)
```


Loop Through an Array

- You can loop through the array items by using a for loop:

- Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

for(x in multiarray){
  print(x)
}
```

Arrays Exercises

1. Write a R program to convert a given matrix to a 1 dimensional array.
2. Write a R program to create an array of two 3x3 matrices each with 3 rows and 3 columns from two given two vectors.
3. Write a R program to create an 3 dimensional array of 24 elements using the dim() function.
4. Write a R program to create an array of two 3x3 matrices each with 3 rows and 3 columns from two given two vectors. Print the second row of the second matrix of the array and the element in the 3rd row and 3rd column of the 1st matrix.
5. Write a R program to combine three arrays so that the first row of the first array is followed by the first row of the second array and then first row of the third array.
6. Write a R program to create a two-dimensional 5x3 array of sequence of even integers greater than 50
7. Write a R program to create an array using four given columns, three given rows, and two given tables and display the content of the array.

R-Data Frames

R Data Structures

- Vectors
- Lists
- Matrices
- Arrays
- **Data Frames**
- Factors

R Data Frames

- Data Frames are data displayed in a format as **a table.**
- Data Frames can have different types of data inside it. While the first column can be **character**, the **second and third can be numeric or logical**. However, **each column should have the same type of data**.
- Use the **data.frame()** function to create a data frame:
- Example

```
# Create a data frame
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),     # Character data
  Pulse = c(100, 150, 120),                         # Numeric
  Duration = c(60, 30, 45)                          # Numeric
)
# Print the data frame
Data_Frame
```


Summarize the Data

- Use the **summary()** function to summarize the data from a Data Frame:
- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame

summary(Data_Frame)
```

Access Items

- You can use single brackets **[ ]**, double brackets **[[ ]]** or **$** to access columns from a data frame:
- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)
Data_Frame[1]                    # Index

Data_Frame[["Training"]]         # double brackets

Data_Frame$Training              # using dollar sign
```


Add Rows

- Use the **rbind()** function to add new rows in a Data Frame:
- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)
# Add a new row
New_row_DF <- rbind(Data_Frame, c("BMI", 108, 110))
# Print the new row
New_row_DF
```


Add Columns

- Use the **cbind()** function to add new columns in a Data Frame:
- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new column
New_col_DF <- cbind(Data_Frame, Steps = c(1000, 6000, 2000))

# Print the new column
New_col_DF
```


Remove Rows and Columns

- Use the **c()** function to remove rows and columns in a Data Frame:
- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Remove the first row and column
Data_Frame_New <- Data_Frame[-c(1), -c(1)]

# Print the new data frame
Data_Frame_New
```

Amount of Rows and Columns

- Use the **dim()** function to find the amount of rows and columns in a Data Frame:

- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

dim(Data_Frame)
```


Contd.....

- You can also use the **ncol()** function to find the number of columns and **nrow()** to find the number of rows:
- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

ncol(Data_Frame)
nrow(Data_Frame)
```


Data Frame Length

- Use the **length()** function to find the number of columns in a Data Frame (similar to ncol()):

- Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

length(Data_Frame)
```


Combining Data Frames

- Use the **rbind()** function to combine two or more data frames in R **vertically**:
- Example

```
# Data Frame 1
Data_Frame1 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)
# Data Frame 2
Data_Frame2 <- data.frame (
  Training = c("Stamina", "Stamina", "Strength"),
  Pulse = c(140, 150, 160),
  Duration = c(30, 30, 20)
)
New_Data_Frame <- rbind(Data_Frame1, Data_Frame2) # Data Frames Combining Vertically
New_Data_Frame
```

Contd….

- And use the **cbind()** function to combine two or more data frames in R horizontally:
- Example

```
# Data Frame1
Data_Frame3 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)
# Data Frame 2
Data_Frame4 <- data.frame (
  Steps = c(3000, 6000, 2000),
  Calories = c(300, 400, 300)
)
New_Data_Frame1 <- cbind(Data_Frame3, Data_Frame4) # Data Frames Combining Vertically
New_Data_Frame1
```